

pRISM+ User's Guide

pRISM+ Version 2.0 for MIPS



000-5444-001



Copyright © 1999 Integrated Systems, Inc. All rights reserved. Printed in U.S.A.

Document Title: **pRISM+ User's Guide, pRISM+ Version 2.0 for MIPS**

Part Number: **000-5444-001**

Revision Date: **May 1999**

Integrated Systems, Inc. • 201 Moffett Park Drive • Sunnyvale, CA 94089-1322

	Corporate	pSOS or pRISM+ Support	MATRIX _x Support
Phone	408-542-1500	1-800-458-7767, 408-542-1925	1-800-958-8885, 408-542-1930
Fax	408-542-1950	408-542-1966	408-542-1951
E-mail	ideas@isi.com	psos_support@isi.com	mx_support@isi.com
Home Page	http://www.isi.com		

LICENSED SOFTWARE - CONFIDENTIAL/PROPRIETARY

This document and the associated software contain information proprietary to Integrated Systems, Inc., or its licensors and may be used only in accordance with the Integrated Systems license agreement under which this package is provided. No part of this document may be copied, reproduced, transmitted, translated, or reduced to any electronic medium or machine-readable form without the prior written consent of Integrated Systems.

Integrated Systems makes no representation with respect to the contents, and assumes no responsibility for any errors that might appear in this document. Integrated Systems specifically disclaims any implied warranties of merchantability or fitness for a particular purpose. This publication and the contents hereof are subject to change without notice.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS252.227-7013 or its equivalent. Unpublished rights reserved under the copyright laws of the United States.

TRADEMARKS

AutoCode, ES_p, MATRIX_x, pRISM, pRISM+, pSOS, SpOTLIGHT, and Xmath are registered trademarks of Integrated Systems, Inc. BetterState, BetterState Lite, BetterState Pro, DocumentIt, Epilogue, HyperBuild, OpEN, OpTIC, pHILE+, pLUG&SIM, pNA+, pREPC+, pROBE+, pRPC+, pSET, pSOS+, pSOS+m, pSOSim, pSOSystem, pX11+, RealSim, SystemBuild, and ZeroCopy are trademarks of Integrated Systems, Inc.

ARM is a trademark of Advanced RISC Machines Limited. Diab Data and Diab Data in combination with D-AS, D-C++, D-CC, D-F77, and D-LD are trademarks of Diab Data, Inc. ELANIX, Signal Analysis Module, and SAM are trademarks of ELANIX, Inc. SingleStep is a trademark of Software Development Systems, Inc. SNiFF+ is a trademark of TakeFive Software GmbH, Austria, a wholly-owned subsidiary of Integrated Systems, Inc.

All other products mentioned are the trademarks, service marks, or registered trademarks of their respective holders.

Contents

Using This Manual	xix
Organization	xix
Conventions	xxi
Font Conventions	xxi
Symbol Conventions	xxii
Mouse Conventions	xxii
Note, Caution, and Warning Conventions	xxii
Format Conventions	xxiii
Commonly Used Terms and Acronyms	xxiii
Related Publications	xxiv
Support	xxv
Contacting Integrated Systems Support	xxvi
 1	
Overview of the pRISM+ Environment	
1.1 What is pRISM+?	1-1
1.2 pSOSystem ADE	1-3
1.3 pRISM+ Host Tools	1-4
1.3.1 pRISM+ Manager	1-4
1.3.2 pRISM+ Configuration Wizard	1-5

1.3.3	pRISM+ Editor	1-6
1.3.4	pRISM+ Source Code Engineering Tool - SNiFF+ [Optional]	1-6
1.3.5	Object Browser	1-7
1.3.6	Embedded System Profiler (ESp) [Optional]	1-7
1.3.7	pRISM+ Shell	1-8
1.3.8	pRISM+ Cross-Compiler Suite	1-8
1.3.9	pRISM+ Source-Level Debugger — SearchLight	1-8
1.3.10	SingleStep Debugger [Optional]	1-9
1.3.11	Run-time Analysis (RTA) Suite [Optional]	1-9
1.4	pRISM+ Communications Infrastructure	1-10
1.4.1	Communication Server	1-10
1.4.2	Debug Server	1-10
1.5	Getting More Information About pRISM+	1-10
1.5.1	pRISM+ Documentation	1-10
1.5.2	Documentation Roadmap	1-11

2 Understanding pSOSystem

2.1	What Is pSOSystem?	2-1
2.2	System Architecture	2-1
2.2.1	Target Architecture	2-4
2.2.2	Host Development System Layout	2-5
2.2.3	Sample Applications	2-11
2.3	System Configurations	2-12
2.3.1	Host System Configuration	2-12
2.3.1	pSOSystem System Library	2-14
2.4	Where to Go From Here?	2-14

3 Quick Start with a Tutorial

3.1	Before You Begin	3-2
3.2	Launch pRISM+	3-3
3.3	Start A New Project with pRISM+	3-4
3.4	Choosing a Project Tool	3-4
3.5	Using pRISM+ Editor	3-5
3.5.1	Choosing a pSOSystem Sample Application As a Starting Point	3-5
3.5.2	Setting Up a New Project	3-6
3.5.3	Getting Acquainted with pRISM+ Editor	3-6
3.6	Using SNIFF+	3-12
3.6.1	Choosing a pSOSystem Sample Application As a Starting Point	3-12
3.6.2	Setting Up a New Project	3-12
3.6.3	Getting Acquainted with SNIFF+	3-13
3.7	Configuring the Target Board	3-19
3.7.1	Connecting the Target Board to the Host Machine	3-19
3.7.2	Starting the Terminal Emulation Program on a Windows Platform	3-19
3.7.3	Starting the Terminal Emulation Program on a UNIX Platform	3-20
3.8	Configuring the Target Communications Parameters	3-20
3.9	Adding a Target Board to the pRISM+ Target List	3-23
3.10	Downloading and Debugging with SingleStep Source-Level Debugger	3-26
3.11	Downloading/Debugging with SearchLight Source-Level Debugger	3-29
3.12	Using Object Browser	3-32
3.12.1	About Object Browser	3-32

3.13	Using ESp.	3-36
3.13.1	Configuring an Experiment	3-36
3.13.2	Starting a Data Collection	3-39
3.13.3	Analyzing the Data	3-39

4 Understanding the pRISM+ Manager

4.1	The pRISM+ Development Environment	4-1
4.1.1	Overview.	4-1
4.1.2	pRISM+ Manager and the pRISMSpace	4-3
4.1.3	The Tool Manager	4-7
4.1.4	The Target Manager.	4-9
4.1.5	After Downloading the Application	4-11

5 pRISM+ Editor

5.1	Makefile Browser	5-1
5.1.1	Makefile View	5-3
5.1.2	Source View.	5-3
5.1.3	Additional Makefiles	5-3
5.1.4	Current Project and Current Target	5-4
5.2	Program Editor	5-4
5.3	Message View	5-5
5.4	Using the pRISM+ Editor.	5-5
5.4.1	Creating New Source Files	5-5
5.4.2	Saving New Source Files	5-6
5.4.3	Copying an Existing Source Files	5-6
5.4.4	Adding Source Files to Your Project	5-7
5.4.5	Error Checking Your Files	5-8

5.4.6	Introducing an Error	5-9
5.4.7	Profiling Your Project	5-10
5.4.8	Accessing the Link Map Analyzer Tool	5-10
5.4.9	Including Custom Libraries	5-10
5.4.10	Adding a Makefile.	5-10
5.4.11	Adding a BSP Makefile	5-11
5.4.12	Removing a Makefile.	5-11
5.4.13	Using the Buffer List	5-11

6 Using SNIFF+ in the pRISM+ Environment

6.1	Overview	6-1
6.2	Key Features of pRISM+ Application Development Framework	6-2
6.2.1	Source Code Comprehension	6-2
6.2.2	Team Development.	6-2
6.2.3	Mixed-Platform Development	6-3
6.2.4	Integrated Make Support	6-3
6.2.5	Flexible Application Development Framework.	6-4
6.3	Key SNIFF+ Concepts	6-4
6.3.1	Code Comprehension and Browsing	6-4
6.3.2	Source Code Parsing	6-4
6.3.3	Projects	6-5
6.3.4	Workspaces	6-10
6.3.5	Working Environments.	6-11
6.3.6	How File Sharing Works	6-14
6.3.7	SNIFF+ Build and Make Support	6-17
6.3.8	Building Targets When Using Team Working Environments	6-18

6.4	Using the pRISM+ Application Development Framework	6-18
6.4.1	Team Development Support.	6-18
6.4.2	pRISM+ Default Working Environments Settings	6-19
6.4.3	Restoring the Default Working Environment Settings.	6-22
6.4.4	What Can You Do with pRISM+ Team Support?	6-24
6.5	pSOSystem Source Projects	6-26
6.5.1	File and Directory View of a pSOSystem Sample Application . . .	6-26
6.5.2	pSOSystem Projects.	6-28
6.5.3	Browse View Versus Build View of pSOSystem Source Projects .	6-34
6.5.4	Browsing pSOSystem.	6-35
6.5.5	Utilities Programs	6-35
6.6	pRISM+ Make Support	6-35
6.6.1	pRISM+ Make Options at a Glance.	6-36
6.6.2	pSOSystem Application Make Structure.	6-36
6.6.3	Make Attributes of pSOSystem Source Projects	6-39
6.6.4	Making a pSOSystem Target Executable	6-42
6.6.5	Using pSOSystem Makefiles.	6-42
6.6.6	Using the SNiFF+ Makefile-Generation Feature	6-43
6.6.7	Generating Makefiles for Your Project	6-45
6.6.8	Hybrid Make Model	6-46
6.6.9	Doing Team-Based Builds	6-49
6.6.10	Building from the Command Line	6-50
6.7	Using the pRISM+ Application Development Framework with SNiFF+ .	6-50
6.7.1	Starting a New Project with pRISM+.	6-51
6.7.2	Starting a Project from Your Existing Code Base	6-63
6.7.3	Working with Multiple Source Trees.	6-80
6.7.4	Integrating a Custom Board Support Package into pRISM+	6-82

6.7.5	Converting a Project Made with pRISM+ Editor	6-87
6.7.6	Starting with an Existing Application for a Previous Version of pRISM+/pSOSystem	6-87

7 pRISM+ Configuration Wizard

7.1	pRISM+ Wizard Features	7-2
7.2	pRISM+ Wizard Interface and Modes	7-2
7.2.1	pRISM+ Wizard Interface	7-2
7.2.2	pRISM+ Wizard Modes	7-4
7.2.3	Error Checking.	7-6
7.2.4	Upgrading a Configuration File.	7-6

8 The SearchLight Debugger - A Tutorial

8.1	What is SearchLight Debugger?	8-1
8.2	Starting SearchLight Debugger and Downloading an Application	8-2
8.2.1	Accessing SearchLight Debugger	8-2
8.2.2	Downloading an Application	8-2
8.3	Debugging in System Debug Mode	8-4
8.3.1	Step, Stepi, Next and Nexti Commands and Code Views	8-4
8.3.2	Setting and Removing an OS Breakpoint	8-9
8.3.3	Viewing Memory Variables	8-12
8.3.4	Viewing Registers	8-14
8.3.5	Navigating Through the Files Window	8-15
8.3.6	Using Find to Locate a Text String and Set a Breakpoint	8-17
8.3.7	Examining the Call Stack	8-20
8.3.8	Examining System Objects	8-22

8.4	Debugging in Task Debug Mode	8-25
8.4.1	Accessing Task Debug Mode	8-25
8.4.2	Setting Breakpoints in TDM.	8-28
8.4.3	Removing Tasks from Task Debug Mode	8-29
8.4.4	Exiting Task Debug Mode	8-30
8.4.5	Conclusion	8-31

9 The SingleStep Debugger - A Tutorial

9.1	What is SingleStep Debugger?.....	9-1
9.2	Using SingleStep Debugger	9-2
9.2.1	Before You Begin	9-2
9.2.2	Starting SingleStep Debugger for pSOSystem.	9-2
9.2.3	The Toolbar and Source Windows	9-6
9.2.4	Invoking the Command Window	9-7
9.2.5	Running the System Debug Tutorial	9-7
9.2.6	Source, Mixed, and Disassembly Display Modes	9-9

10 ES_p

10.1	ES _p Prerequisites	10-2
10.2	Placing User-Defined Event in the Application.	10-2
10.3	Refining Data Collection Needs	10-3
10.3.1	Buffer Management	10-3
10.3.2	Event Specification	10-4
10.4	Tailoring the Configuration Table	10-5
10.5	Tailoring the Application's Stacks	10-5
10.6	Post-Mortem Analysis in ES _p	10-6

11 Object Browser

11.1	Monitoring for Stack Problems	11-4
11.1.1	Stack Problem Setup	11-4
11.1.2	Understanding Your Stack Graphics Data	11-4
11.2	Finding Memory Leaks	11-4
11.3	Checking for Deadlocks and Priority Inversion	11-5
11.4	Logging Data in the CSV Files	11-7
11.5	Selective Logging of Data in Graph Frame	11-7

12 Run-Time Analysis (RTA) Suite

12.1	Overview	12-1
12.1.1	Run-Time Error Checker	12-1
12.1.2	Visual Interactive Profiler	12-1
12.1.3	Link Map Analyzer	12-1
12.1.4	Stack Use Analyzer	12-2

13 pRISM+ Shell

13.1	Using Interactive pSOS-Aware Commands	13-2
13.1.1	Obtaining Status of a pSOS Object	13-3
13.1.2	Modifying Communication Timeouts	13-3
13.1.3	Downloading a pSOS+ Executable	13-5
13.1.4	Using pRISM+ Shell with SearchLight Debugger	13-5
13.2	Using and Invoking a pRISM+ Shell Tcl Script	13-8
13.2.1	Using an Existing Tcl Script for Testing	13-9
13.2.2	pRISM+ Shell Script Example	13-11
13.3	Using Low-Level TCL/CORBA Services	13-14
13.4	Customizing the pRISM+ Shell	13-14

14 pRISM+ Target Agents

14.1	pMONT+ Target Agent	14-1
14.1.1	Target Requirements for Monitoring an Application	14-2
14.1.2	Configuring pMONT+	14-2
14.1.3	pMONT+ Driver Usage	14-4
14.1.4	pMONT+ Behavior on the Target	14-5
14.1.5	log_event() System Call	14-6
14.1.6	Memory Usage	14-7
14.2	pROBE+ Target Agent	14-7
14.2.1	pROBE+ Behavior on the Target	14-8
14.2.2	Configuring pROBE+	14-8

15 Customize the pRISM+ Tools/Environment

15.1	Customizing Your pRISM+ Tools	15-1
15.1.1	Customizing Your Toolbar	15-1
15.1.2	Incorporating a Custom BSP for pSOSystem	15-3
15.2	Customizing Your pRISM+ Environment	15-5
15.2.1	Multiple pRISM+ Installations	15-5
15.2.2	Multiple-users Configuration (UNIX Only)	15-7
15.2.3	Mixed-Platform Development for Solaris and Windows	15-8
15.2.4	Redefining Your Environment Variables	15-12
15.2.5	Redefining Your Color Settings (Solaris and HP-UX)	15-13
15.2.6	Setting a Printer for On-line Help (Solaris and HP-UX)	15-13

A Board-Support Package Information

A.1	pSOSystem/MIPS Operating Mode	A-2
A.2	IDT 79S465 Evaluation Board	A-3
A.2.1	Hardware Setup	A-3
A.2.2	pSOSystem Boot Configuration	A-5
A.2.3	Building pSOSystem Boot ROMs	A-7
A.2.4	Memory Layout and Usage	A-8
A.2.5	Devices Supported for the IDT 79465 Evaluation Board	A-10
A.2.6	Miscellaneous	A-10
A.3	IDT79S440 Board	A-10
A.3.1	Hardware Setup	A-11
A.4	IDT79S500 Board	A-14
A.4.1	Hardware Setup	A-14
A.5	LSI4101 Board	A-16
A.5.1	Hardware Setup	A-16
A.5.2	pSOSystem Boot Configuration	A-19
A.5.3	Building pSOSystem Boot ROMs	A-20
A.5.4	Memory Layout and Usage	A-21
A.5.5	Devices Supported for the MiniRISC and TinyRISC Evaluation Boards	A-23
A.5.6	MIPS16 Support	A-23
A.5.7	Miscellaneous	A-24

B pRISM+ Environment Variables

B.1	pRISM+ Variables for the Windows Environment	B-1
B.2	pRISM+ Variables for the UNIX Environment	B-5

C pRISM+ Supported Host/Target Connections

C.1	Using a Serial Connection	C-1
C.1.1	Building a pSOSystem Application	C-2
C.1.2	Configuring Target Environment	C-2
C.1.3	Configuring Target Communications Parameters	C-3
C.1.4	Configuring Host Tools Connection with the Target	C-3
C.1.5	Using pRISM+ Tools	C-4
C.2	Using an Ethernet Connection	C-4
C.2.1	Building a pSOSystem Application	C-4
C.2.2	Configuring Target Environment	C-5
C.2.3	Booting pSOSystem	C-5
C.2.4	Configuring Host Tools Connection with the Target	C-6
C.2.5	Using pRISM+ Tools	C-6
C.3	Using a Communication Server Remotely	C-7
C.3.1	Building a pSOSystem Application	C-7
C.3.2	Configuring Target Environment	C-7
C.3.3	Booting pSOSystem	C-9
C.3.4	Using pRISM+ Tools	C-9
C.4	Using the TFTP Server	C-9
C.4.1	Building a pSOSystem Application	C-10
C.4.2	Sys_conf.h Settings	C-10
C.4.3	Configuring Target Environment	C-10
C.4.4	Configuring Host Environment	C-11
C.4.5	Using the TFTP Server Connection	C-13

D pRISM+ Shell Commands

D.1	Overview	D-1
D.2	Communication Server- and Debug Server-Based Commands	D-2
	boot	D-4
	breakpoint	D-5
	cb	D-8
	cn	D-9
	comm	D-10
	condvar	D-11
	connect	D-12
	csabout	D-13
	db	D-14
	dcn	D-16
	debugger	D-17
	di	D-19
	disassemble	D-20
	disconnect	D-21
	dl	D-22
	dm	D-23
	dr	D-24
	dssession	D-25
	ev	D-27
	evaluate	D-28
	evt	D-29
	fl	D-30
	fm	D-31

go	D-32
halt	D-33
he	D-34
help	D-35
il	D-36
init	D-37
initialize	D-38
lb	D-39
log	D-40
memory	D-41
mod	D-43
mutex	D-44
osbreakpoint	D-45
partition	D-50
pm	D-51
pr	D-52
probe	D-53
psos	D-55
q*	D-59
queue	D-63
quit	D-65
region	D-66
register	D-68
sc	D-70
semaphore	D-71
session	D-72
sf	D-76

	<code>stackfrm</code>	D-77
	<code>t*</code>	D-78
	<code>target</code>	D-80
	<code>task</code>	D-82
	<code>tsd</code>	D-84
	<code>version</code>	D-85
D.3	Comparison of pROBE+ and pRISM+ Shell Commands	D-86
D.4	TCL Commands	D-88
	<code>type</code>	D-89
	<code>vinfo</code>	D-90
	<code>bind</code>	D-91
	<code>set</code>	D-92
	<code>new</code>	D-93
	<code>delete</code>	D-94
	<code>toString</code>	D-95
	<code>invoke</code>	D-96
	<code>slength</code>	D-97

E **pSOSystem Source Projects**

E.1	Generic pSOSystem Projects	E-1
E.2	Drivers Project	E-1
E.3	Bsp Projects	E-2
E.4	Sample Application Projects	E-2
E.5	Sample Application Projects	E-2
E.6	VPATH	E-2
E.6.1	gnu gmake and VPATH	E-2
E.6.2	\$< Macro	E-3

E.6.3	Compiler Option -o:	E-3
E.6.4	Compiler Option -I@:	E-3
E.6.5	Use of Relative Path for Overriding.	E-4
E.6.6	Generating Include and Link Paths	E-4
E.6.7	Object and .opt files Overriding	E-4
E.6.8	With or Without SNIFF+.	E-4
E.6.9	macros.incl File	E-5
E.6.10	Problems Using Recursive Make	E-5
E.6.11	Check_vpath Target.	E-5
E.6.12	Gnu Make	E-5
E.7	pLUGINS+ Scripts.	E-6
E.7.1	Scripts to Create SNIFF+ Projects for pSOSystem+.	E-6
E.7.2	Integration scripts:	E-10

Glossary **gloss-1**

Index **index-1**

Using This Manual

This manual describes the pRISM+ Development Suite for real-time embedded applications — a solution from Integrated Systems, Inc. that includes products produced by both ISI and third parties. The pRISM+ tools cover the life cycle of real-time embedded applications development.

pRISM+ is the only development suite working with pSOSystem, the industry's leading real-time embedded operating system.

Organization

This document is organized as follows:

- [Chapter 1, *Overview of the pRISM+ Environment*](#), provides information on the advanced features of customizing your pRISM+ Environment.
- [Chapter 2, *Understanding pSOSystem*](#), provides an introduction to pSOSystem, the scalable operating system that is incorporated into your pRISM+ application. It also provides instructions on how to incorporate a custom BSP and how to create a custom pSOS+ application.
- [Chapter 3, *Quick Start with a Tutorial*](#), provides a tutorial of how to use the pRISM+ for pSOSystem tools to create, compile, build, download, and test your pSOS+ application. It highlights how to use the pRISM+ Editor, SearchLight debugger, pRISM+ Shell, ESsp, and Object Browser.
- [Chapter 4, *Understanding the pRISM+ Manager*](#), provides an overview of the pRISM+ Manager.
- [Chapter 5, *pRISM+ Editor*](#), provides an overview of the pRISM+ Editor the pRISM+ project editor.

- [Chapter 6, *Using SNIFF+ in the pRISM+ Environment*](#), provides an overview of the SNIFF+ tools, a component of pRISM+.
- [Chapter 7, *pRISM+ Configuration Wizard*](#), provides an overview of the pRISM+ Configuration Wizard.
- [Chapter 8, *The SearchLight Debugger - A Tutorial*](#), provides a tutorial illustrating the capabilities of the SearchLight debugger, using the pSOSystem sample application `pdemo`.
- [Chapter 9, *The SingleStep Debugger - A Tutorial*](#), provides a tutorial illustrating the capabilities of the SingleStep debugger using the pSOSystem sample application `pdemo`.
- [Chapter 10, *ESp*](#), provides additional informations when using ESp.
- [Chapter 11, *Object Browser*](#), provides additional information when using Object Browser.
- [Chapter 12, *Run-Time Analysis \(RTA\) Suite*](#), provides a brief overview of the Run-Time Analysis Suite.
- [Chapter 13, *pRISM+ Shell*](#), provides detailed information on how to use the pRISM+ Shell to modify your communication timeouts, create testing Tcl scripts, and use debugger type commands.
- [Chapter 14, *pRISM+ Target Agents*](#), provides a information on how to use the pSOSystem target agents.
- [Chapter 15, *Customize the pRISM+ Tools/Environment*](#), provides information on how to customize the pRISM+ tools environment.
- [Appendix A, *Board-Support Package Information*](#), provides board-specific information.
- [Appendix B, *pRISM+ Environment Variables*](#), provides a list of special pSOSystem and pRISM+ environment variables you can use.
- [Appendix C, *pRISM+ Supported Host/Target Connections*](#), provides special information on pRISM+ connections.
- [Appendix D, *pRISM+ Shell Commands*](#), provides a list of the supported pRISM+ Shell commands.
- [Appendix E, *pSOSystem Source Projects*](#), provides a description of the source projects included with the pSOSystem.

- The [Glossary](#) defines terms relevant to the pRISM+ and pSOSystem development environment.

Conventions

This section describes the conventions used in this document.

Font Conventions

This sentence is set in the default text font, Bookman Light. Bookman Light is used for general text, menu selections, window names, and program names. Fonts other than the standard text default have the following significance:

Courier: *Courier* is used for command and function names, file names, directory paths, environment variables, messages and other system output, code and program examples, system calls, prompt responses, and syntax examples.

bold Courier: **Courier** is used for user input (anything you are expected to type in).

italic Courier: *Courier* is used for command and function names, file names, directory paths, environment variables, messages and other system output, code and program examples, system calls, prompt responses, and syntax examples.

bold italic Courier: **italic Courier** is used for user input (anything you are expected to type in).

italic: *Italics* are used in conjunction with the default font for emphasis, first instances of terms defined in the glossary, and publication titles.

Bold Helvetica narrow: **Bold Helvetica narrow** font is used for buttons, fields, and icons in a graphical user interface. Keyboard keys are also set in this font.

Sample Input/Output

In the following example, user input is shown in **bold Courier**, and system response is shown in *Courier*.

commstats

```
Number of total packets sent          160
Number of acknowledgment timeouts    0
```

Number of response timeouts	0
Number of retries	0
Number of corrupted packets received	0
Number of duplicate packets received	0
Number of communication breaks with target	0

Symbol Conventions

This section describes symbol conventions used in this document.

- []** Brackets indicate that the enclosed information is optional. The brackets are generally not typed when the information is entered.
- |** A vertical bar separating two text items indicates that either item can be entered as a value.
- ˘** The breve symbol indicates a required space (for example, in user input).
- %** The percent sign indicates the UNIX operating system prompt for C shell.
- \$** The dollar sign indicates the UNIX operating system prompt for Bourne and Korn shells.

Mouse Conventions

This document assumes you have a standard, right-handed three-button mouse. From left to right, the buttons are referred to as MB1, MB2, and MB3. All instructions assume MB1 unless otherwise noted.

- click** Press and quickly release a mouse button. MB1 is assumed if “click” is used without a button designation. For example, “click the root window.”
- double-click** Click MB1 twice in quick succession.
- drag** Place the cursor over an object, then hold down MB1 while moving the mouse. Release the button when the object arrives at the desired location on the screen.

Note, Caution, and Warning Conventions

Within the text of this manual, you may find notes, cautions, and warnings. These statements are used for the purposes described below.

NOTE: Notes provide special considerations or details that are important to the procedures or explanations presented.

- CAUTION:** Cautions indicate actions that may result in possible loss of work performed and associated data. An example might be a system crash that results in the loss of data for that given session.

WARNING: Warnings indicate actions or circumstances that may result in file corruption, irrecoverable data loss, data security risk, or damage to hardware.
-

Format Conventions

The reference section in this manual adheres to a standard format. The name of the command, a brief description, and its syntax appear at the top of the first page. The remaining information about the command appears below the syntax and is organized under the following headings:

Description

Provides a description of the command.

Usage

Provides detailed usage information for the item being described.

See Also

Lists the location of other relevant information.

Commonly Used Terms and Acronyms

The following terms and acronyms are commonly associated with pSOSystem and appear in this manual.

ASR	See asynchronous signal routine.
asynchronous signal routine	A function within an application that executes in response to an asynchronous signal.
callout	A function that a device driver uses to notify a pSOSystem component of an interrupt event. A callout is called from an ISR.
FD	File descriptor.
FLIST	A contiguous sequence of blocks used to hold file descriptors on a pHILE+ formatted volume.
ISR	See interrupt service routine.

interrupt service routine	A function within an application or device driver that takes control of the system when the CPU has been triggered with an exception from an external source.
KI	See kernel interface.
kernel interface	A user-provided communication layer between nodes in a multi-processing environment (pSOS+m).
NFS	Network file system.
NI	Network interface.
RSC	See remote service call.
remote service call	A service call made from one node to another in a multiprocessing environment (pSOS+m).
ROOTBLOCK	The root block on a pHILE+ formatted volume, which contains all information needed by pHILE+ to locate other vital information on the volume.
socket	The endpoint for communication across a network.
task	The smallest unit of execution in a system designed with pSOSystem that can compete on its own for system resources.
TCP/IP	Transport Control Protocol/Internet Protocol, a software protocol for communications between computers.
UDP	User Datagram Protocol.

Related Publications

As you read this manual, you may also want to refer to the other manuals in the standard documentation set for more detailed descriptions:

- *pSOSystem System Concepts*: provides theoretical information about the operation of pSOSystem.
- *pSOSystem System Calls*: describes the system calls and C language interface to pSOS+, pHILE+, pREPC+, pNA+, and pRPC+.
- *pROBE+ User's Guide*: describes how to use the pROBE+ System Debugger/Analyzer.
- *pSOSystem Advanced Topics*: contains information on how to customize your usage of your pSOSystem. It contains sections on using and creating BSPs and Assembly Language information.

- *pSOSystem Application Examples*: describes the application examples that are provided for you and tutorials on how to use these examples.

Based on the options you have purchased, you might also need to reference one or more of the following manuals:

- *Routing Architecture User's Guide*: describes the pSOSystem Routing Architecture for OpEN Shortest Path First (OSPF), Routing Information Protocol (RIP), and other related routing protocols.
- *RIP Version 2 User's Guide*: describes how to use the pSOSystem RIP protocol.
- *C++ Support Package User's Guide*: describes how to implement C++ applications in a pSOSystem environment.
- *SNMP User's Guide*: describes the internal structure and operation of SNMP, Integrated System's Simple Network Management Protocol product. This manual also describes how to install and use the SNMP MIB (Management Information Base) Compiler.
- *OpEN User's Guide*: describes how to install and use the pSOSystem OpEN (OpEN Protocol Embedded Networking) product.
- *OSPF User's Guide*: describes the Open Shortest Path First (OSPF) pSOSystem protocol driver
- *TCP/IP for OpEN User's Guide*: describes how to use the pSOSystem Streams-based TCP/IP for OpEN (OpEN Protocol Embedded Networking) product.

Support

Customers in the United States can contact Integrated Systems Technical Support as described below.

International customers can contact:

- The local Integrated Systems branch office.
- The local pSOSystem distributor.
- Integrated Systems Technical Support as described below.

Before contacting Integrated Systems Technical Support, please gather the following information available:

- Your customer ID and complete company address.
- Your phone and fax numbers and e-mail address.
- Your product name, including components, and the following information:
 - The version number of the product.
 - The host and target systems.
 - The type of communication used (Ethernet, serial).
- Your problem (a brief description) and the impact to you.

In addition, please gather the following information:

- The procedure you followed to build the code. Include components used by the application.
- A complete record of any error messages as seen on the screen (useful for tracking problems by error code).
- A complete test case, if applicable. Attach all include or startup files, as well as a sequence of commands that will reproduce the problem.

Contacting Integrated Systems Support

To contact Integrated Systems Technical Support, use one of the following methods:

- Call 408-542-1925 (U.S. and international countries).
- Call 1-800-458-7767 (1-800-458-pSOS) (U.S. and international countries with 1-800 support).
- Send a FAX to 408-542-1966.
- Send e-mail to psos_support@isi.com.
- Access our web site: <http://customer.isi.com>.

Integrated Systems actively seeks suggestions and comments about our software, documentation, customer support, and training. Please send your comments by e-mail to ideas@isi.com or submit a Problem Report form via the internet (<http://customer.isi.com/report.shtml>).

Overview of the pRISM+ Environment

This chapter provides a brief overview of the pRISM+® for pSOSystem™ architecture and components. It describes how to use pRISM+ to create a pSOSystem-based embedded system application. A documentation roadmap, located at the end of the chapter, will further assist you in finding more information about pRISM+ or any of the products mentioned in this chapter.

1.1 What is pRISM+?

pRISM+ is an integrated development environment provided by Integrated Systems Inc. for building embedded systems. It combines pSOSystem, a scalable, high performance real-time operating system with a set of development, debugging and profiling tools into one powerful environment to deliver run-time performance for embedded systems and higher productivity to developers.

pRISM+ offers the embedded industry's most comprehensive set of tools. pRISM+ includes tools for every step of the embedded development process. From team development and source code engineering tools to application building tools, and run-time target debugging and profiling tools. pRISM+ offers more tools than ever before for embedded developers. Based on the industry standard CORBA framework, pRISM+ also provides an open interface for integration of third-party tools.

Available for Windows 95, Windows 98, Windows NT, Solaris, and HP-UX, pRISM+ offers native look and feel on each platform it supports. In addition, pRISM+ offers extensive on-line documentation and context sensitive help for tools and operating system components.

The pRISM+ Development Environment includes the following tools as part of this pRISM+ release. Any optional pRISM+ products are noted.

- pSOSystem — Integrated Systems' family of scalable, multitasking, real-time operating system and networking products. pSOSystem includes Board Support Packages for many off-the-shelf CPU boards in source form, as well as target agents necessary to support pRISM+ host-based tools.
- pRISM+ Host Tools
 - pRISM+ Manager ([page 1-4](#))
 - pRISM+ Configuration Wizard ([page 1-5](#))
 - pRISM+ Editor ([page 1-6](#))
 - pRISM+ Source Code Engineering Tool — SNIFF+™ (an optional product) ([page 1-6](#))
 - pRISM+ Cross Compiler Suite — Diab Data™ Compilers ([page 1-8](#))
 - Run-Time Analysis Tool Suite (an optional product) ([page 1-9](#))
 - Object Browser ([page 1-7](#))
 - pRISM+ Source-Level Debugger — SearchLight ([page 1-8](#))
 - SingleStep™ Source-Level Debugger — SDS (an optional product for PowerPC and 68K) ([page 1-9](#))
 - Embedded System Profiler (ESp®) (an optional product) ([page 1-7](#))
 - pRISM+ Shell ([page 1-8](#))
- pRISM+ Communications Infrastructure
 - Communication Server ([page 1-10](#))
 - Debug Server ([page 1-10](#))
- pRISM+ for pSOSystem Documentation
 - pSOSystem CD-ROM Documentation Set ([page 1-11](#))
 - pRISM+ on-line tutorials and interactive help ([page 1-10](#))

[Figure 1-1 on page 1-3](#) shows the pRISM+ architecture and communication.

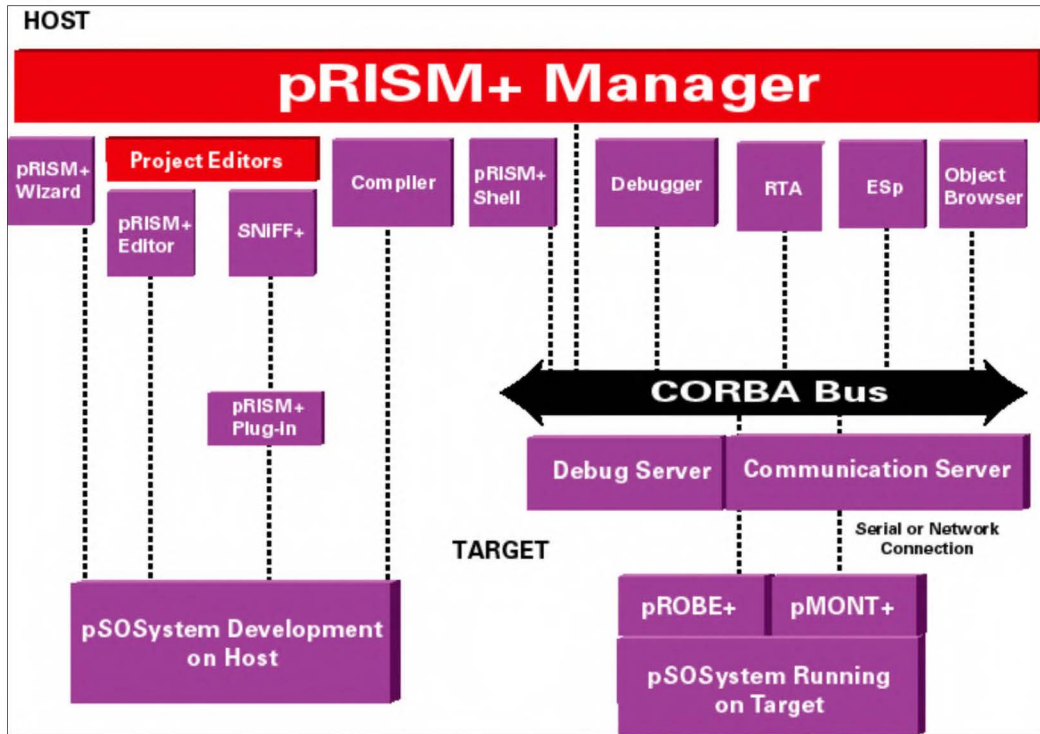


FIGURE 1-1 pRISM+ for pSOSystem Architecture

1.2 pSOSystem ADE

pRISM+ is fully integrated with pSOSystem, Integrated Systems, Inc. family of real-time embedded components including the pSOS+ kernel, the industry's leading embedded RTOS. pSOSystem scales to match price/performance requirements of the entire range of embedded applications from simple, stand-alone devices to complex, networked, multiprocessing systems.

The pSOSystem family of run-time options includes the fast, efficient pSOS+ single-processor and pSOS+m multiprocessing real-time kernels. Extremely compact, the multitasking pSOS+ kernel supports both standard and custom hardware applications. The pSOS+m kernel allows you to create multiprocessor-based applications with few code changes.

The pSOSystem networking suite offers the industry's most advanced networking capabilities. It includes support for all major industry-standard, UNIX-based TCP/UDP/IP communications protocols, SNMP network management, OpEN (a STREAMS-based networking framework and networking enabler), and STREAMS-compliant protocols such as TCP/IP and X.25.

Other elements within pSOSystem include:

- the pROBE+ debugger (a target agent supporting the source level debugger).
- pMONT (the target agent supporting the Embedded System Profiling and Object Browser tools).
- pREPC+ (a re-entrant ANSI C library),.
- pHILE+ (the file system manager, supports a variety of standard as well as optimized file formats).

Two other essential components of an embedded target image are the application code, written by the user, and a board support package (BSP). BSPs are provided in source form as part of pSOSystem and give the developer with the interface software that allows pSOSystem to operate on a particular hardware platform. In particular, the BSPs include drivers for a host of common I/O devices, including serial, Ethernet, SCSI, and timers.

NOTE: For more information on pSOSystem, refer to [Chapter 2](#) of this manual and the documentation set that accompanied your pRISM+ selection.

1.3 pRISM+ Host Tools

This section provides a brief description of the pRISM+ tools. To learn more about the pRISM+ tools, complete the pRISM+ Getting Started tutorial in [Chapter 3](#).

1.3.1 pRISM+ Manager

The pRISM+ Manager is the central launch point for all the pRISM+ tools. From pRISM+ Manager, you can:

- Access pRISM+ Editor, Makefile Browser, and Source Editor designed by Integrated Systems, Inc.
- Access pRISM+ Source Code Engineering Tool — SNIFF+. Note that SNIFF+ is an optional product.
- Configure pSOSystem using the pRISM+ Configuration Wizard.

- Compile an application using the build button.
- Configure target communication settings for each target using the Target Menu.
- Specify a target board selection using the Target Selection Window.
- Reset the Communication Server using the Reset button.
- Download the executable code to the target using the download button.
- Control target execution using the Go and the Halt buttons.
- Access the SearchLight Source-Level Debugger.
- Access the SingleStep Source-Level Debugger from SDS. SingleStep is an optional product and it is only available for 68K and PPC processor families.
- Access the Object Browser to get Snapshots of the run-time target system.
- Access ES_p to get event-by-event profile information of the run-time target system.
- Access pRISM+ Shell, a Tcl shell with CORBA IDL extensions for direct pRISM+ server access.

The pRISM+ Manager stores information for each pRISM+ session, such as the choice of target board, target communication settings, location of project source files. This information is entered once by you and then shared by all pRISM+ tools.

1.3.2 pRISM+ Configuration Wizard



The pRISM+ Configuration Wizard helps users to configure pSOSystem for each embedded application. pSOSystem is a highly scalable and configurable operating system. Users can custom fit pSOSystem for each application with a single header file, `sys_conf.h`. In its most simple mode, pRISM+ Configuration Wizard is a graphical editor of this pSOSystem configuration header file (`sys_conf.h`). The “Wizard” function of pRISM+ Configuration Wizard provides a “guided tour” to the configuration process. The Wizard can take users through the necessary configuration parameters based on the type of application and operating system components included in the application. Users can also get on-line help for each configuration parameter. Furthermore, pRISM+ Configuration Wizard also provides error checking on the value of configuration parameters and possible erroneous combinations of parameters.

1.3.3 pRISM+ Editor



The pRISM+ Editor is a flexible and easy-to-use, scalable, cross-platform project and code development editor that allows you to create projects in C, C++, and other languages. It is comprised of a Makefile Browser, a programmer oriented text editor, and a Message View for tracking compiler error messages.

The Source editor allows you to create and modify application source files and integrate source files into your pRISM+ project. Once you have modified your project, you can use the Message View to find common mistakes, locate and correct the mistakes, and recompile your project. The Makefile Browser displays the targets of one or more of your project's makefiles. The pRISM+ Editor is equipped with context-sensitive help that will provide you insight to the major features of the pRISM+ Editor. For additional information, refer to the pRISM+ on-line help or [Chapter 5](#).

1.3.4 pRISM+ Source Code Engineering Tool - SNiFF+ [Optional]



SNiFF+ offers an extensive and powerful **set of source code engineering tools for source code comprehension, project management, team-based** development, interface to CMVC tools, automated build support, and integrated documentation generation. SNiFF+ also provides integration with a wide range of source editors to support software development.

SNiFF+ provides the most advanced browsing and cross-referencing features to help users to gain rapid source code comprehension. Powerful filtering and visualization techniques can work with large projects with many thousands of files, tens of thousands of symbols, and millions of lines of code. SNiFF+'s powerful parsers can extract the symbolic information from a code base even before the code is compiled.

SNiFF+'s project and workspace concepts support effective team development by allowing a team to develop against a common code base. Seamless integrated with CMVC tools such as ClearCase, SNiFF+ can be adapted to any organization with big projects thus supporting effective cooperation between teams.

Based on the project and workspace concept, SNiFF+ facilitates the build process with a set of make support files that are automatically generated and managed by SNiFF+. The building framework supports complex projects organized in multiple teams that are working on multiple concurrent configurations on multiple platforms.

SNiFF+ also provides flexibility in the choice of tools for software development. Users can choose from the default SNiFF+ Source Editor or a wide range of other source editors such as Emacs, vi or CodeWright. All changes in the code are immediately reflected in all the browsing tools; no compilation is necessary.

Once SNIFF+ has parsed a code base, it can generate documentation automatically, as well as keep the documentation up-to-date based on software changes.

1.3.5 Object Browser



Object Browser is a run-time analysis tool. It monitors target behavior by taking periodic snapshots of the operating system objects on the target while the target system is running. Information on OS objects such as tasks, message queues, semaphores, and other critical information such as stack and memory usage can be displayed graphically. This gives a sampled view of the target run-time behavior over time. Two intuitive graphical display modes exist. The Snapshot View is best suited for displaying run-time attributes of system objects, for example, run-time status and configuration parameters of a task. The alternative, Graph View, is best used to display the level of usage, for an example, each task's stack usage as a percentage of its own maximum allowed stack size. From these intuitive graphical displays, users can easily spot problems such as stack overflow or memory leak over time. Each collection of data obtained from the running target system can either be stored in Object Browser and compared with past or future samples or exported to standard desktop tools such as Microsoft Excel for documentation purposes.

1.3.6 Embedded System Profiler (ESp) [Optional]



Like Object Browser, ESp is also a run-time analysis tool. However, unlike the Object Browser's sampled view of the target run-time system, ESp offers a time-continuous, event-by-event view of target run-time system. ESp gives you the data between samples offered by Object Browser displaying a more complete picture of the behavior of the run-time system.

ESp acts as a logic analyzer for software. Between user-defined (trigger and detri-
gger) points, ESp can log every event that takes place on the target. These events may be operating system calls, context switches, or even user-defined events. Each event is individually time-stamped and mapped to the task or the ISR which executed it, and displayed in a time-indexed graph. These actions allow the developer to follow the context switch history, task state transitions, interrupts, system calls, and all other activities on the target. ESp is, therefore, an essential tool for studying scheduling behavior, task synchronization and timing to identify problems such as priority inversion, deadlock, and starvation. ESp can be configured to gather post-processing information on a system enabling you to identify the events that led up to a crash.

ESp can also tally CPU usage by each task and ISR. This can help developers to identify performance bottlenecks in the system. For additional information, refer to ESp chapter.

1.3.7 pRISM+ Shell



The pRISM+ Shell is a Tcl shell with pRISM+ specific extensions. These extensions provide several levels of services to pRISM+ users:

- pSOS+ queries (a quick access to your application)
- Tcl scripting (assistance in testing your applications)
- TCL/CORBA services (allows any CORBA service to be called from TCL)

The most basic level of service pRISM+ Shell allows you to issue interactively to get and set information about the operating system components running on the target to obtain run-status. This can be used to augment source-level debugging which may not have full OS-query services.

pRISM+ Shell also provides other pROBE+ commands such as making pSOS+ system calls and breakpoint services. This gives you a way to query the target by accessing the pRISM+ Communications Server and Debug Services just like other pRISM+ client tools. The Tcl shell scripting language allows you to script a debug session and execute it from the command line.

For additional information, refer to [Chapter 13, pRISM+ Shell](#).

1.3.8 pRISM+ Cross-Compiler Suite

For PowerPC, MIPS and 68K processors, pRISM+ is integrated with the D-CC and D-C++ compiler suites provided by Diab Data, an Integrated Systems subsidiary. Each suite is comprised of a C/C++ cross compiler, a program profiler, assembler, linker, archiver, and ANSI compliant libraries. Each compiler is specifically created for the CPU architecture it supports and each provides CPU-optimized code to ensure maximum performance from the CPU being used. The pRISM+ Cross Compiler Suite also provides special support for embedded development such as flexible control over location of code and data segments in memory, ability to mix assembler and C functions, and support of position-independent code and data.

1.3.9 pRISM+ Source-Level Debugger — SearchLight



SearchLight is a source-level debugger for advanced C, C++ and assembly level debugging. It is available for PowerPC, 68K, and MIPS processors.

SearchLight features a simple point-and-click graphical interface. This provides fast and easy access to target debugging information. Using this simple and intuitive interface, you can control program execution, perform sophisticated breakpoint operations, display and modify variables, and traverse complex data structures. In

addition, you can instantly access files, functions, stacks, and local and global variables.

SearchLight supports debugging over serial and Ethernet target interfaces. For more information on SearchLight, refer to the SearchLight on-line help or [Chapter 8](#).

1.3.10 SingleStep Debugger [Optional]



For PowerPC and 68K processors, the SingleStep source-level debugger from Software Development Systems (SDS) is another debugger option. Users of SingleStep can get full access to pSOSystem operating system information such as state of tasks, queues, and semaphores as well as component configuration information.

SingleStep features a simple point-and-click graphical user interface, proving fast, easy access to all debugging information. SingleStep supports advanced C, C++, and assembly level debugging features, including program execution control, sophisticated breakpoint definitions, multiple variable viewing and automated traversal of complex data structures.

You can debug with SingleStep over serial interface, Ethernet interface, BDM/JTAG interface, as well as many in-circuit emulators.

You can debug with SingleStep over a serial interface or an Ethernet interface, as well as many in-circuit emulators.

1.3.11 Run-time Analysis (RTA) Suite [Optional]

The RTA Suite is a powerful combination of software tools, including a Visual Interactive Profiler, a Run-Time Error Checker and a Visual Link Map Analyzer. These tools draw on information from Diab Data's D-CC and D-C++ compiler suites and the target application. This information provides the critical insight needed by each developer to improve program performance, reliability, and memory usage in advanced 32-bit applications.

The Visual Interactive Profiler provides users with code coverage information for a run-time system. With VIP, users can easily identify hot spots and dead code in an application. Tightly integrated with pSOSystem's memory allocation and deallocation algorithms, the Run-Time error checker can help customers to find elusive run-time memory errors in their pSOSystem-based embedded applications. The Visual Link Map Analyzer provides an intuitive, graphical interface to configure optimum memory layouts. With the RTA Suite, pRISM+ users can profile and analyze the

underlying run-time behavior of their code and then invoke profile-guided optimizations for improved performance.

1.4 pRISM+ Communications Infrastructure

1.4.1 Communication Server

The Communication Server is responsible for target services such as reading memory, execution control, and pSOSystem-awareness. It is also responsible for communication with the target. It accepts requests from clients and interacts with the target to satisfy these requests.

1.4.2 Debug Server

The Debug Server is responsible for the core debugging services such as symbol management, breakpoint handling, and execution processing. It accepts requests from the debugger client (for example: SearchLight) and interacts with the Communication Server to satisfy these requests.

1.5 Getting More Information About pRISM+

This section contains a documentation road map and a list of manuals providing additional information about pRISM+.

1.5.1 pRISM+ Documentation

pRISM+ documentation is available in both printed and on-line CD-ROM formats. A complete list of all documents supporting pRISM+ is located in the *pRISM+ Release Notes*.

Release Information

- *pRISM+ CD-ROM Installation for Windows* booklet
- *pRISM+ CD-ROM Installation for UNIX* booklet
- *System Administration Guide: License Manager for pRISM+*
- *pRISM+ Release Notes*
- *pRISM+ Upgrade Guide* (for existing customers)

On-line Documentation

- On-line tutorials and on-line help.
- On-line Manuals.

Host Tools Documentations

- pRISM+ User's Guide (this book)

1.5.2 Documentation Roadmap

[Figure 1-2 on page 1-12](#) provides a roadmap of pRISM+ for pSOSystem and supported host tools documentation.

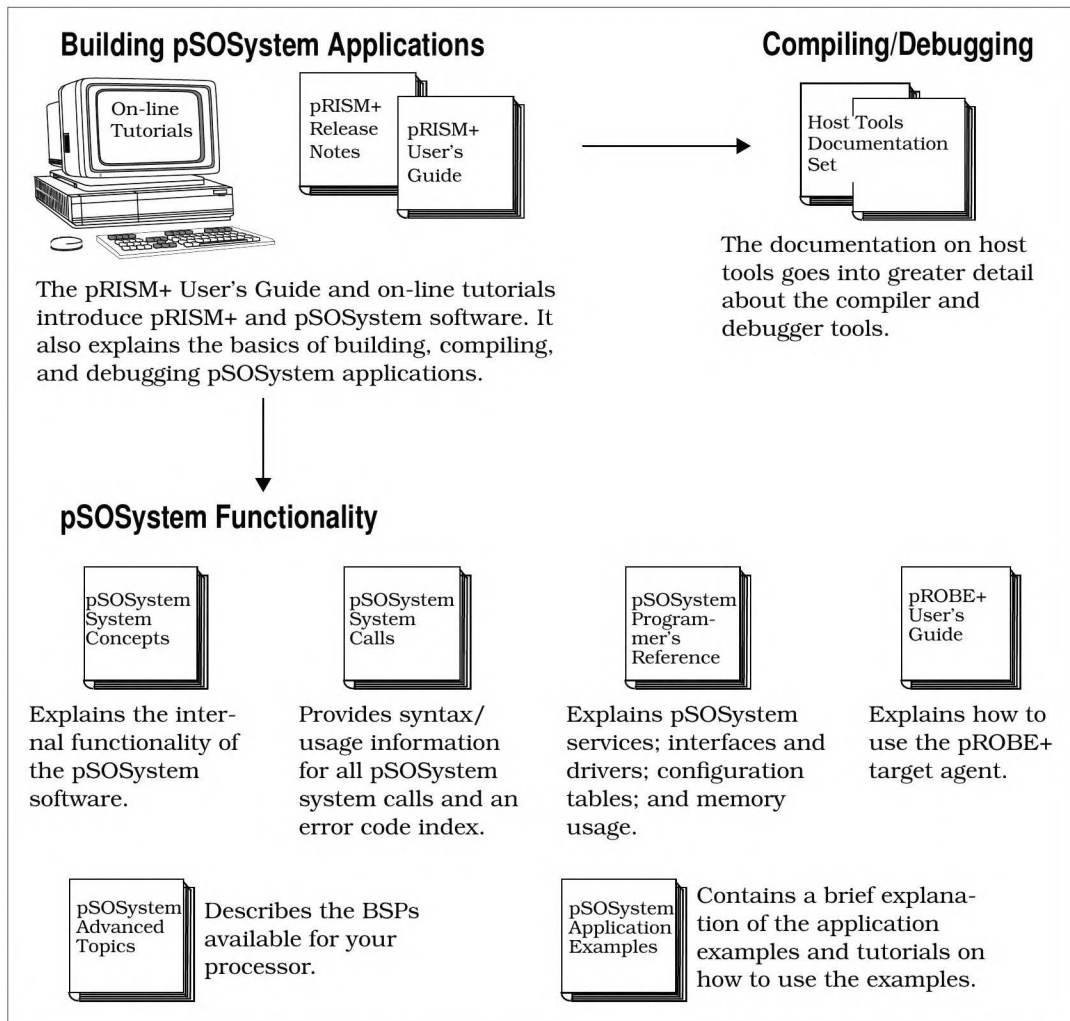


FIGURE 1-2 pRISM+ for pSOSystem Documentation Roadmap

This chapter introduces the internal organization and operating theory of the pSOSystem environment.

2.1 What Is pSOSystem?

pSOSystem is a modular, high-performance real-time operating system designed specifically for embedded microprocessors. It provides a complete multitasking environment based on open systems standards.

pSOSystem is designed to meet three overriding objectives:

- Performance
- Reliability
- Ease-of-Use

The result is a fast, deterministic, yet accessible system software solution. Accessible in this case translates to a minimal learning curve. pSOSystem is designed for quick start-up on both custom and commercial hardware.

The pSOSystem software is supported by an integrated set of cross development tools that can reside on UNIX or Windows based computers. These tools can communicate with a target over a serial or TCP/IP network connection.

2.2 System Architecture

The pSOSystem software employs a modular architecture. It is built around the pSOS+ real-time multi-tasking kernel and a collection of companion software com-

ponents. Software components are standard building blocks delivered as absolute position-independent code modules. They are standard parts in the sense that they are unchanged from one application to another. This black box technique eliminates maintenance by the user and assures reliability, because hundreds of applications execute the same, identical code.

Unlike most system software, a software component is not wired down to a piece of hardware. It makes no assumptions about the execution (target) environment. Each software component utilizes a user-supplied configuration table that contains application and hardware related parameters to configure itself at start-up.

Every component implements a logical collection of system calls. To the application developer, system calls appear as re-entrant C functions callable from an application. Any combination of components can be incorporated into a system to match your real-time design requirements. pSOSystem includes the following components:

NOTE: Certain components may not yet be available on all target processors. Check the release notes to see which pSOSystem components are available on your target.

- **pSOS+ Real-time Multitasking Kernel:** A field-proven, multitasking kernel that provides a responsive, efficient mechanism for coordinating the activities of your real-time system.
- **pSOS+m Multiprocessor Multitasking Kernel:** Extends the pSOS+ feature set to operate seamless across multiple, tightly-coupled or distributed processors.
- **pNA+ TCP/ IP Network Manager:** A complete TCP/IP implementation including gateway routing, UDP, ARP, and ICMP protocols; uses a standard socket interface that includes stream, datagram, and raw sockets.
- **pRPC+ Remote Procedure Call Library:** Offers SUN compatible RPC and XDR services; allows you to build distributed applications using the familiar C procedure paradigm.
- **pHILE+ File System Manager:** Gives efficient access to mass storage devices, both local and on a network. Includes support for CD-ROM devices, MS-DOS compatible floppy disks, and a high-speed proprietary file system. When used in conjunction with the pNA+ component and the pRPC+ subcomponent, offers client-side NFS services.
- **pREPC+ ANSI C Standard Library:** Provides familiar ANSI C run-time functions such as `printf()`, `scanf()`, and so forth, in the target environment.

- **pROBE+**: The pROBE+ debugger is a comprehensive system debugger and analyzer for the pSOSystem environment.

Figure 2-1 illustrates the pSOSystem environment.

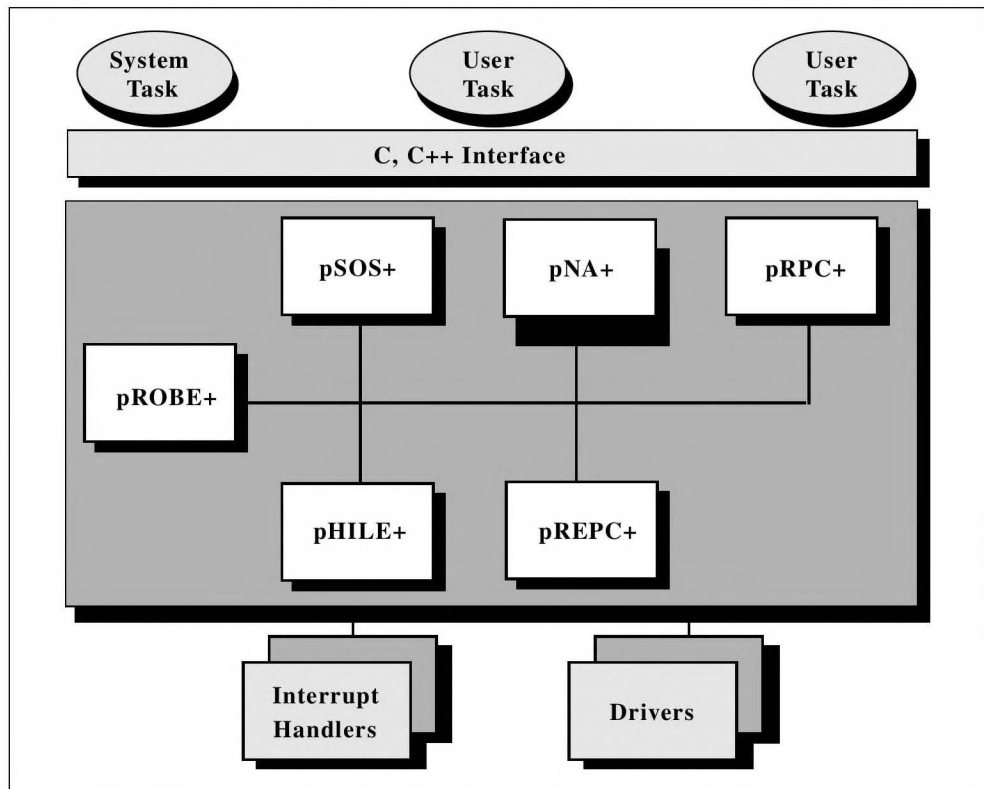


FIGURE 2-1 The pSOSystem Environment

In addition to these core components, pSOSystem includes the following:

- Networking protocols including SNMP, FTP, Telnet, TFTP, NFS, MLPP, X.25, ISDN, and STREAMS.
- Run-time loader.
- User application shell.
- Support for C++ applications.
- Boot ROMs.

- Pre-configured versions of pSOSystem for popular commercial hardware.
- pSOSystem templates for custom configurations.
- Chip-level device drivers.
- Sample applications.

This manual describes how to get started with pSOSystem. This includes building and debugging pSOSystem applications.

2.2.1 Target Architecture

This section introduces the internal organization and operating theory of the pSOSystem environment.

The purpose of the pSOSystem environment is to help you developing an application on a host system, then download, and run the application on an embedded computer. The embedded computer is called the *target system*. The description of the pSOSystem environment begins with the target system architecture. The description of the host system starts in [Section 2.2.2, Host Development System Layout](#). For an illustration of the relationship between the host and the target system, see [Figure 2-2](#).

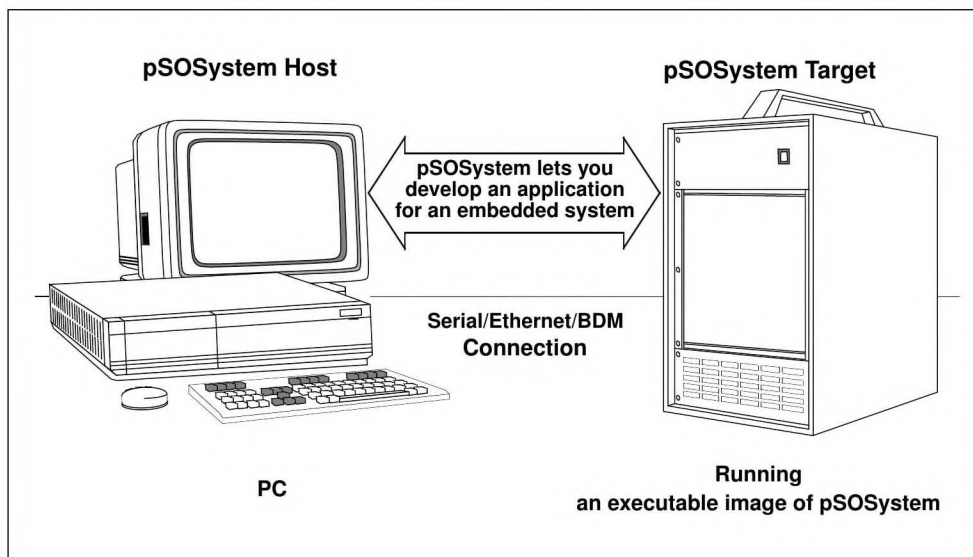


FIGURE 2-2 Architecture of pSOSystem

In a pSOSystem environment model, the target system software is usually an application that you develop on the host, as shown in [Figure 2-2](#). Two major software elements run on the target hardware: the *pSOSystem* software and the *application code*. You link these elements together on the host system and download this combination to the target. The downloaded software is called an *executable image*.

pSOSystem Software

The pSOSystem software provides a standard set of services for the application code and debugging tools. It almost always contains the pSOS+ real-time kernel and frequently contains the following companion software elements:

- pREPC+, pROBE+, pNA+, and pHILE+ components.
- Device drivers and interrupt handlers for the target hardware.
- Configuration tables used to customize the operating system for a particular target system.

The pSOSystem software is a combination of standard components, system configuration code, and hardware-specific environment code. The hardware-specific code is known as a *pSOSystem Board-Support Package*, or BSP. Integrated Systems provides BSPs for several target boards. If you are using one of the boards, you can begin developing pSOSystem application code immediately. If you are using unsupported or custom hardware, you must develop a board-support package for the target system.

Application Code

The application code is what makes one target system different from another. It implements the functional behavior of the target system. Normally, application code is very specialized and contains few standard software elements, if any. It is usually developed from scratch, although you can utilize code fragments from the sample applications that come with the pSOSystem software.

2.2.2 Host Development System Layout

pSOSystem code consists of read-only *object* libraries, *include* files, and *source* files. The code can be kept in a central location on the host system so that multiple users can have access to it. The directory tree that contains this shared code is the *pSOSystem directory tree*, and its root directory is the *pSOSystem root directory*. Within pSOSystem source files, path names generally begin with `PSS_ROOT`. The environment variable `PSS_ROOT` is set to the path name of the pSOSystem root directory by the installation procedure.

You can create a pSOSystem executable image from any directory in the host system, not just within the pSOSystem root directory tree. A directory where you create an executable image is called a *working directory*. For information on the contents of this directory, see section *Working Directory* on page 2-7.

Configuration Files

Source files that control the configuration of the pSOSystem environment are called configuration files. Configuration files exist for all systems built with the pSOSystem software, and these files are compiled and linked into the executable image. A set of the common configuration files resides in the `PSS_ROOT\configs\std` directory. You should not need to make changes to these common files.

The configuration files contain a variety of parameters that control the behavior of the pSOSystem software. Examples of these parameters are the baud rate for the serial channels and IP addresses in network systems. You can change these parameters either when you build the pSOSystem environment or through an interactive start-up dialog during run-time start-up.

Configuration parameters are normally specified at system build time by the values you supply in the system configuration file `sys_conf.h`. This system configuration file resides in the *working directory*. An option in `sys_conf.h` allows you to specify that the operating system should try to locate saved versions of these parameters in the target board's nonvolatile storage area. This is useful when you are using the pSOSystem Boot ROMs because the executable image you then download can use the same parameter values you give to the Boot ROMs. You can also enable a special start-up dialog that allows you to change the parameters at run time start-up through an RS-232 connection. Both of these options are enabled by definitions in `sys_conf.h`.

The C source files in `PSS_ROOT\configs\std` contain numerous conditional compilation statements controlled by the contents of `sys_conf.h`. The `dialog.c` file contains the source code for the optional system start-up dialog. Most of the other files contain the start-up code that builds the configuration tables for the various operating system components. These files are provided as read-only source files; you should not need to modify them.

In addition to the source files, `PSS_ROOT\configs\std` contains a file called `config.mk`, which the application's makefile must include. The directives in `config.mk` compile the files in the `std` directory.

Board-Support Package

Directory `PSS_ROOT\bsps` contains the software for several board support packages (BSPs). Details on these BSPs are provided in the *pSOSystem Advanced Topics* manual.

System Library

The `libsys.a` file in the `PSS_ROOT\sys\os` directory is the system library. It contains the various operating system components and the run time bindings that the application uses to make system calls to these components. The system library is usually built once as part of the pSOSystem host installation. It needs to be rebuilt only when you receive new or updated software components.

Working Directory

The pSOSystem executable image is built from within a working directory. The working directory contains the application code. Its location does not depend on the location of the pSOSystem root directory. A working directory must contain the following:

- A system configuration file (`sys_conf.h`)
- A `makefile`
- A driver configuration file (`drv_conf.c`)
- Application code

System Configuration File

The system configuration file `sys_conf.h` is a C include file that must reside in the working directory. The `sys_conf.h` file has many elements and affects many aspects of the pSOSystem environment. The following list illustrates the range of items that `sys_conf.h` controls, namely:

- Which pSOSystem components are built into the executable image
- Which peripheral devices are enabled
- Whether a start-up dialog is included
- Various entries in the individual component configuration tables, such as the numbers of tasks, queues, and other objects, for the pSOS+ environment.

makefile

This section describes the rules for writing a `makefile` to build pSOSystem applications.

NOTE: The following examples use the UNIX slash type `/`. On Windows systems the slash type `\` should be substituted. Refer to the sample applications installed on your system for the appropriate slash type to use. The sample applications are located in the directory `$PSS_ROOT/apps` (UNIX) or `$PSS_ROOT\apps` (Windows).

The first items in the `makefile` are the following macro definitions:

<code>PSS_BSP</code>	Supplies the path name of the pSOSystem board-support package you use to build the executable image. This is usually one of the subdirectories of <code>PSS_ROOT/bsps</code> .
<code>PSS_DRVOBJS</code>	Defines the set of object files and libraries for drivers that you have added to the pSOSystem environment. It must include at least <code>drv_conf.o</code> .
<code>PSS_APPOBJS</code>	Defines the set of all the object files and object libraries that make up the application.

After the preceding macro definitions, the `makefile` must have the following lines:

```
PSS_CONFIG=$(PSS_ROOT)/configs/std

#-----
# $(SNIFF_MAKE_CMD).mk implements the SNIFF+ workspace over-riding
# and should be included before any other file. For non SNIFF+ build
# it has no effect.
#-----
include $(PSS_CONFIG)/$(SNIFF_MAKE_CMD).mk
include $(PSS_BSP)/bsp.mk
include $(PSS_CONFIG)/config.mk
```

The remainder of the `makefile` contains the rules that define how to build application modules. The `*.mk` files that you include define several macros. These macros are used in the following `makefile` commands:

CC	Invokes the C compiler
COPTS	Specifies options for the C compiler that are appropriate for building an executable image
AS	Invokes the assembler
AOPTS	Specifies options for the assembler that are appropriate for building an executable image

The following is an example `makefile` for building an application that contains one object module, `demo.o`.

```
PSS_DROBJs=drv_conf.o
PSS_APOBJs= demo.o

PSS_CONFIG=$(PSS_ROOT)/configs/std

#-----
# $(SNIFF_MAKE_CMD).mk implements the SNIFF+ workspace over-
# riding and should be included before any other file.
# For non SNIFF+ build it has no effect.
#-----
include $(PSS_CONFIG)/$(SNIFF_MAKE_CMD).mk
include $(PSS_BSP)/bsp.mk
include $(PSS_CONFIG)/config.mk

clean:
    @rm -f ram.coff *.cfe *.cof *.elf ram.* rom.* *.a
    @rm -f *.o *.map *.hex *.x *.opt *.L app.* qpsos.tmp
    @rm -f ram.dld rom.dld os.dld app.dld lib.dld driver.dld
    @rm -f *.db *.db2 *.oul *.blk

drv_conf.o: drv_conf.c \
    makefile sys_conf.h $(PSS_BSP)/bsp.h
    $(CC) $(COPTS) -o drv_conf.o $<

demo.o: demo.c \
    makefile demo.h sys_conf.h
    $(CC) $(COPTS) -Xno-optimized-debug -o demo.o $<
```


When you invoke `make` to build the pSOSystem executable image you can specify, as a parameter, one of the output targets listed in the following table:

TABLE 2-1 Output Target File Options

Output Parameter	Description
<code>ram.hex</code>	An executable image in S-record format for Motorola processors or Intel Extended Hexadecimal format for Intel processors, suitable to download to the target board's RAM.
<code>ram.elf</code>	An executable image in ELF format, suitable for conversion to a <code>ram.hex</code> file.
<code>rom.hex</code>	An executable image in S-record format for Motorola processors or Intel Extended Hexadecimal format for Intel processors, suitable for placement in ROM.
<code>rom.elf</code>	An executable image in ELF format, suitable for placement in ROM. (It is seldom useful for producing ROMs unless the PROM programmer accepts ELF formatted input files.)
<code>os.hex</code>	An executable image of the pSOSystem software in S-record format for Motorola processors or Intel Extended Hexadecimal format for Intel processors without the application.
<code>os.elf</code>	An executable image of the pSOSystem software in ELF format, without the application.
<code>app.hex</code>	An executable image of the application (without the operating system) in S-record format for Motorola processors or Intel Extended Hexadecimal format for Intel processors.
<code>app.elf</code>	An executable image of the application (without the operating system) in ELF format.

If you do not specify a target, the first target found in the makefile of the application is built.

To build a system for downloading to the RAM of the target board using a source level debugger such as SearchLight, for example, you would build the `ram.elf` target.

There are many build options for pRISM+ which depend on the project editor you use, and also on whether you are compiling from the pRISM+ IDE or the command line. Please refer to the chapters on pRISM+ Editor and SNiFF+ for specific information regarding how to build an executable in either environment.

The pSOSystem build process also produces an ASCII map file. The map file contains a load map and cross-reference listing of symbols. Its name is `ram.map`, `rom.map`, `os.map`, or `app.map`, depending on the output target you specify.

Driver Configuration File

The driver configuration file `drv_conf.c` contains two routines that are called during system start-up to install pSOSystem drivers in the appropriate pSOS I/O Jump tables. You can find more information about the pSOS I/O Jump tables in the *pSOSystem System Concepts* manual. Each of the pSOSystem sample applications includes an example driver configuration file. Normally, you do not need to edit this file unless you are adding special or custom drivers to the pSOSystem environment.

The driver configuration file `drv_conf.c` contains three routines that are called during system start-up. The routines are:

<code>SetUpDrivers()</code>	This function installs pSOSystem drivers in the appropriate slot in the pSOS I/O jump table. You can find more information about the pSOS I/O jump in the pSOSystem System concepts manual.
<code>DrvSysStartCO()</code>	This function is invoked during pSOS initialization. It sets up the Initial Device Name Table for all devices referred by the pSOS I/O jump table. This function also cleans up the driver specific data area when pSOSystem is re-initialized.
<code>SetUpNI()</code>	This routine sets up the Network Interfaces for pNA+.

Each of the pSOSystem sample applications include an example driver configuration file. Normally, you do not need to edit this file unless you are adding special or custom drivers to the pSOSystem environment.

2.2.3 Sample Applications

Directory `PSS_ROOT\apps` contains several subdirectories, each of which contains a pSOSystem sample application. If you use a supported target platform, the sample applications allow you to build, download, and run an executable image without writing a single line of code.

Each sample directory contains source code, a makefile, and a `README` file for the application. You can use the source code for each sample as a starting point for an application or as a learning tool. For example, the following two sample applications are recommended starting points:

<code>hello</code>	This simple one-task application displays the message "Hello, world" to the target's serial port (system console).
<code>pdemo</code>	A simple application that uses most of the pSOS+ services.

The `pdemo` sample application is also used by the tutorials found in subsequent chapters of this manual.

2.3 System Configurations

The pRISM+ installation procedure configures your system's environment for using the development tools. This section describes the system configurations made by the installation procedure and, should you wish to examine them, contains instructions on how to view the changes made to your system's configuration.

2.3.1 Host System Configuration

During the pRISM+ installation, the following environment variables are set:

<code>PSS_ROOT</code>	The pSOSystem root directory.
<code>PSS_BSP</code>	Path to BSP directory.
<code>DIABLIB</code>	Installation directory of the Diab Data compiler.

In addition, the installation procedure sets the system path specification to include the host utilities directory and the compiler executables directory.

To view the setting of a specific environment variable on UNIX systems use the command:

```
echo $env_varname
```

where `env_varname` is the environment variable name.

On Windows systems, the environment variable settings can be displayed using the MS-DOS `SET` command.

Table 2-2 lists the environment variables set by the installation procedure and the value to which each should be set.

TABLE 2-2 Environment Variables Set By Install Procedure

OS	Environment Variable	Value
		Comments
UNIX C or Bourne shell	PSS_ROOT	/usr/isi<target>
	PSS_BSP	\$PSS_ROOT/bsps/targ targ specifies directory of target board's BSPs. For example, if target is Motorola FADS, value of targ is ads8xx.
	DIABLIB	/isi<target>/diab_ver ver is version number of compiler. For example, if compiler version is 4.2b.
Windows	PSS_ROOT	drive:\isi<target> drive: is install to drive.
	PSS_BSP	\$PSS_ROOT\bsps\targ targ specifies directory of target board's BSPs. For example, if target is Motorola FADS, value of targ is ads8xx.
	DIABLIB	drive:\isi<target>\diab_ver ver is version number of compiler. For example, if compiler version is 4.2b.
NOTE: The <target> portion of each value is a variable, to be replaced with ppc, 68k, or mip as appropriate for your target processor.		

To verify that the system path variable includes the host utilities directory and the compiler executables directory on UNIX systems, enter the command:

```
echo $PATH
```

To view the path on Windows systems use the MS-DOS command:

```
path
```

The host utilities directory settings are shown in the following table:

TABLE 2-3 Host Utility Directories

Path	OS
PSS_ROOT/bin/hpux	HPUX
PSS_ROOT/bin/solaris	Solaris
PSS_ROOT\bin\win32	Windows

The compiler executables directory is specified by the `DIABLIB` environment variable (see [Table 2-2](#)) and should appear in the system path specification.

If any of the host system configuration settings are not correct, the installation process probably did not complete successfully. Refer to the *pRISM+ Installation* guide for more information on how to fix the configuration settings.

2.3.1 pSOSystem System Library

During the pRISM+ installation process the pSOSystem system library is built. If this step completely successfully you will find the `libsys.a` file contained within the following directories:

<code>\$PSS_ROOT/sys/os</code>	(UNIX systems)
<code>\$PSS_ROOT\sys\os</code>	(Windows systems)

If you are using the C++ compiler you will also find the file, `libsysxx.a` in the directories listed above.

2.4 Where to Go From Here?

You can continue onto the tutorial chapters which describe building the `hello` and `pdemo` sample applications and using the SearchLight or SingleStep debugger.

The tutorial chapters include:

- [Chapter 3, *Quick Start with a Tutorial*](#)
- [Chapter 8, *The SearchLight Debugger - A Tutorial*](#)
- [Chapter 9, *The SingleStep Debugger - A Tutorial*](#)

Along with source code for the application, each sample application directory includes the necessary build files and pSOSystem configuration files. For additional information about sample program files, refer to the *pSOSystem Application Examples* manual.

This chapter introduces the pRISM+ development environment by walking you through an edit, compile, and debug cycle with a pSOSystem sample application. It is intended as your first introduction to most of the tools in pRISM+ and the sequence to using these tools or [Appendix A](#).

It is strongly recommended that you go through this tutorial with a standard off-the-shelf target board supported by this release of pRISM+. For a list of supported target boards, refer to the pRISM+ Release Notes.

In this tutorial you will learn how to:

- Complete prerequisites before starting the tutorial. Refer to [Section 3.1, Before You Begin](#) on page 3-2.
- [Launch pRISM+](#) on page 3-3.
- [Start A New Project with pRISM+](#) on page 3-4.
- Select a project tool to use as the basis of your development environment, in [Section 3.4, Choosing a Project Tool](#) on page 3-4.
- Select a pSOSystem sample application as a starting point on [page 3-5](#).
- Use the pRISM+ Editor as a development tool on [page 3-5](#).
- Use SNIFF+ to perform some basic development tasks (if you purchased this optional product) on [page 3-12](#).
- Build the pSOSystem sample application `pdemo` to generate a target executable image.
- Configure both the target connection and communication parameters.

- Configure pRISM+ host tools to connect to the target.
- Download the target executable image using a source-level debugger.
- Use Object Browser to take snapshots of the target run-time behavior.
- Use ES_p to profile the target run-time behavior.

3.1 Before You Begin

In order to run this tutorial, you must first complete a list of prerequisites. This section goes over the prerequisites that are required to use the pRISM+ Tutorial.

Install pRISM+

Install pRISM+ if you have not already done so. To install pRISM+, follow the instructions provided in the installation booklet included in the pRISM+ CD-ROM jewel case.

License pRISM+

If you have installed pRISM+ using the start-up key, you will be able to run pRISM+ for 60 days. After the 60 day period, you will need another license file. Please take some time out NOW to apply for your permanent license. To apply for your permanent license, fill out the license request form and send to Integrated Systems, Inc.

After you receive your permanent license file, install it by following the directions given in the Administration Guide: License Manager manual.

Read release information

For information such as new features in this release, bug fixes since last release and other release information, refer to the pRISM+ release notes.

Set up a Target Board to Run the Tutorial

The pRISM+ tutorial will take you through a typical edit-compile-debug cycle using pRISM+. This tutorial assumes that:

- A pSOSystem Boot ROM or flash is installed on the board.
- Your target board has a serial port, which a terminal emulation program will use to communicate with the Boot ROM/flash on the target board.
- Your target board has an Ethernet port and your development host is connected to your target board via this Ethernet interface.

pRISM+ supports a number of **off-the-shelf** single board computers. We strongly recommend that you use one of the supported boards for the tutorial. A list of supported boards in this release is offered in release notes and [Appendix A](#). Refer to [Appendix A](#) for information on how to set up one of the supported boards.

For instructions on how to use pRISM+ over a BDM/JTAG connection or serial connection, refer to [Appendix C](#), *pRISM+ Supported Host/Target Connections*.

Connect the Target Board to the Development Host

[Figure 3-1](#) shows the development configuration needed for the pRISM+ Tutorial.

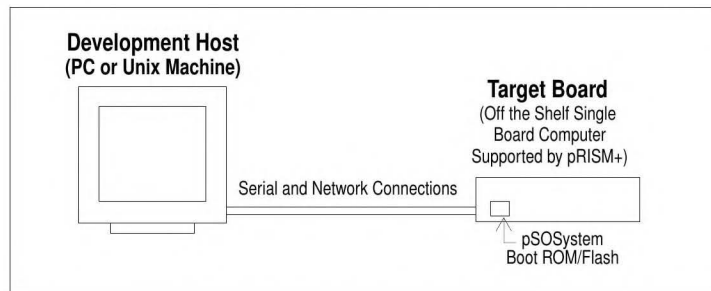


FIGURE 3-1 Host Target Hardware Connection for Tutorial

Upon completing all the prerequisites, proceed to the next section to begin the pRISM+ tutorial.

3.2 Launch pRISM+

Use this procedure to launch pRISM+ on both Windows and UNIX platforms.

For Windows

1. To start the Orbix Daemon, select **Start** → **Programs** → **pRISM+ 2.0** *target_name* → **Orbix Daemon**.

This launches the Orbix Daemon needed by pRISM+ tools to communicate. Since the Orbix Daemon will not be used directly, you can choose to iconize the **Orbix Daemon** window.

2. To launch pRISM+, select **Start** → **Programs** → **pRISM+ 2.0** *target_name* → **pRISM+ *target_name***.

For UNIX

1. To start the Orbix Daemon, type **orbixd &** from the command line.

This launches the Orbix Daemon needed by pRISM+ tools to communicate.

2. To launch pRISM+, type **prismplus20 &** from the command line.

3.3 Start A New Project with pRISM+

You start using pRISM+ by starting a new pRISMSpace. A pRISMSpace holds information about your project such as location of project source files and your choice of a board support package (BSP). Once you set up a pRISMSpace, the information gathered on your project is stored in a pRISMSpace file in your project directory. A pRISMSpace file has the .psp extension.

1. To start a new pRISMSpace, select **File→New** from the pRISM+ Manager as shown in [Figure 3-2](#).

This starts the pRISMSpace Wizard to guide you through the rest of the steps in setting up a new pRISMSpace for your new project.

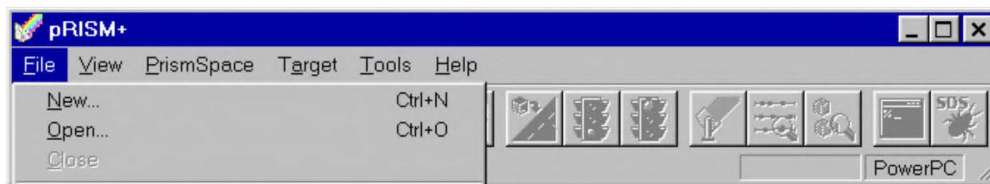


FIGURE 3-2 Selecting **File → New** from the pRISM+ Manager

3.4 Choosing a Project Tool

The first thing pRISMSpace Wizard will prompt you for is your choice of a development tool. pRISM+ offers two choices for development tools: pRISM+ Editor and SNiFF+.

NOTE: You will not see this window if you have not purchased the SNiFF+ option.

pRISM+ Editor is an easy-to-use, fast-start editor specifically designed for BSP developers and other small project teams. The pRISM+ Editor provides a simple environment for embedded developers. By default, pRISM+ Editor is included in every pRISM+ development environment.

SNiFF+ is a sophisticated software engineering tool which brings tremendous benefits to developers who work with large amounts of source code. SNiFF+ offers powerful browsers for source code comprehension, automated makefile generation, automatic documentation generation, interface to CMVC tools and other source code engineering functions. SNiFF+ is an optional package to pRISM+. It is available only if you have purchased this add-on option.

Depending upon your choice of development tool, choose one of the following steps, then proceed to that section:

1. If your choice is to use pRISM+ Editor, proceed to [Section 3.5, Using pRISM+ Editor](#) section.
2. If your choice is to use SNiFF+, proceed to [Section 3.6, Using SNiFF+](#).

3.5 Using pRISM+ Editor

This section will show you how to use pRISM+ Editor to perform several basic development tasks. You can:

- Select pRISM+ Editor as your development tool of choice.
- Choose a pSOSystem sample application as a starting point.
- Get acquainted with pRISM+ Editor.
- Build the sample application to produce a target executable.

From pRISMSpace Wizard, select pRISM+ Editor as your development tool then choose **Next**.

3.5.1 Choosing a pSOSystem Sample Application As a Starting Point

This tutorial will use a pSOSystem sample application, `pdemo`, to show you how to use pRISM+ tools.

1. From pRISMSpace Wizard, choose **Start with a pSOSystem example application**, then click **Next**.
2. From the list of pSOSystem sample applications, select `pdemo` and click **Next**.

3.5.2 Setting Up a New Project

You are prompted to name your pRISMSpace.

1. Enter `proj1` in the **pRISMSpace name** field.

pRISM+ Editor will place a copy of the `pdemo` sample application in the pRISMSpace directory.

2. Click **Finish** to exit pRISMSpace Wizard.

This starts the pRISM+ Editor.

This completes the steps of configuring your first pRISMSpace.

3.5.3 Getting Acquainted with pRISM+ Editor

Figure 3-3 on page 3-7 shows the pRISM+ Editor. These are **points of interest in** the pRISM+ Editor:

- Name and definition of the project. In the tutorial example, the project name is `proj1` and it is defined by a makefile. Double clicking on the application `proj1` folder gives the makefile for the `pdemo` project.
- Name of the project, `proj1`, which is the same as the pRISMSpace name specified earlier.
- The default target, `ram.elf`, which is the default target to be built.
- All the object files that make up the `ram.elf` target, such as `begin.o`, `bspcfg.o`, and so on.
- The default BSP to be linked with this application.
- Other libraries to be linked with this application.

Extensive on-line help is available for pRISM+ Editor. To access a functional introduction to pRISM+ Editor, select **Help** → **Welcome**.

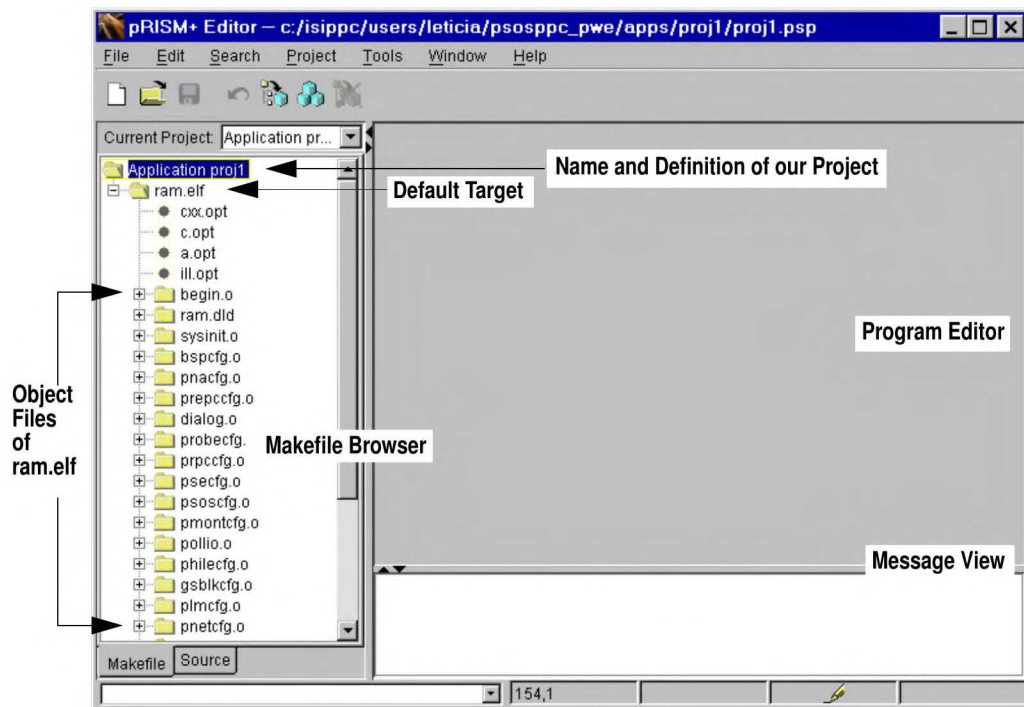


FIGURE 3-3 pRISM+ Editor Window

Viewing Default Project Settings

To view the default project settings for `proj1`, select **PrismSpace** → **Settings** from pRISM+ Manager. Figure 3-4 displays the **Project Settings** dialog box.

The following are the choices for the project settings:

- **pSOSystem Configuration File** — Associated with each pSOSystem sample application is a pSOSystem configuration file called `sys_conf.h`. This file is used to specify which OS components are to be included in an application and how these components are configured.
- **Board Support Package** — This specifies which Board Support Package is to be linked with `pdemo`. The current default value was set at installation time. Ensure that the setting matches the target board you are using with the tutorial. If the value does not match the target board you are using, change it now before continuing with this tutorial.

- **Build Make Target** — pSOSystem makefiles define a number of Build/Make targets to support embedded requirements. For example, the same target executable can be build for RAM or ROM. This field specifies the type of target executable you would like to build by default. For a description of what each of these targets are, refer to [Table 2-1, Output Target File Options](#).

The target you need for this tutorial is `ram.elf`, which contains the application as well as the necessary OS components need by `pdemo`. It is intended to be downloaded to the target with either a source-level debugger or the **Loader** button in pRISM+ Manager.

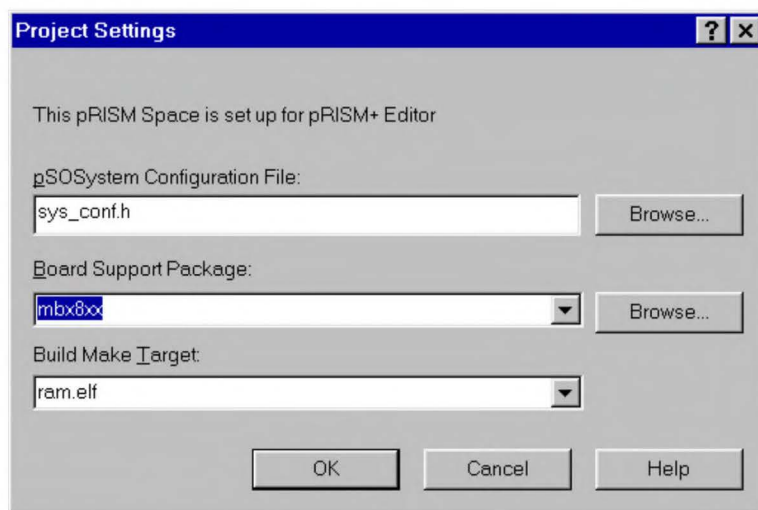


FIGURE 3-4 Project Settings Dialog Box

Project Makefile

pRISM+ Editor's concept of a project is a set of files associated with a build and make target defined in a makefile. When you told the pRISMSpace Wizard that you wanted to begin with a pSOSystem sample application `pdemo` using the pRISM+ Editor, pRISM+ Editor started by parsing the makefile of the `pdemo` sample application and found the `ram.elf` target as part of the `all` rule. In essence, pRISM+ Editor's projects are makefile defined.

To examine the project makefile for `pdemo`, double-click on the name of the project **Application proj1**. In the makefile, you see the rules to compile the files that are part of the `pdemo` application, `drv.conf.c` and `demo.c`, and other pSOSystem makefiles included by this makefile to generate the final executable `ram.elf` (see [Figure 3-5](#)).

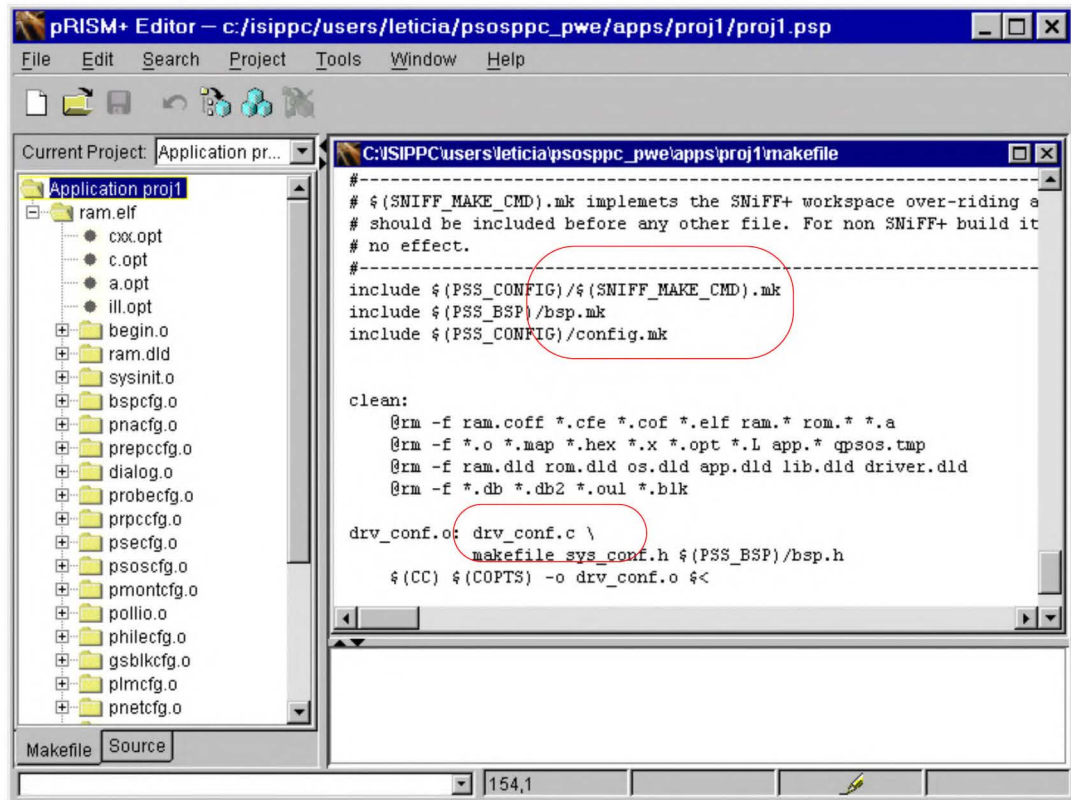


FIGURE 3-5 Makefile Example

Accessing Source Files

The pRISM+ Editor offers two views to each project's source files. You can select a view with the tabs labeled **Makefile** or **Source**. To edit any source file, double click on the file name in either view. Note that you can open multiple source files with the pRISM+ Editor.

Viewing Board Support Package Source Files

In a previous section you learned how to determine which BSP is by default linked to your application, this section shows you how to view your BSP source files.

To view the source file associated with the default BSP to be linked to `pdemo`, open the BSP project by opening the makefile for the BSP project.

1. To open the BSP project, select **Project → Add BSP Makefile**.

This command opens the **Add BSP makefile** dialog box, with your default BSP already highlighted. After adding your BSP makefile to the desktop, you will be able to peruse the source files associated with the BSP.

2. Click **OK** to add your BSP project into the Makefile window. As shown in [Figure 3-6](#), you can now access your BSP source files.

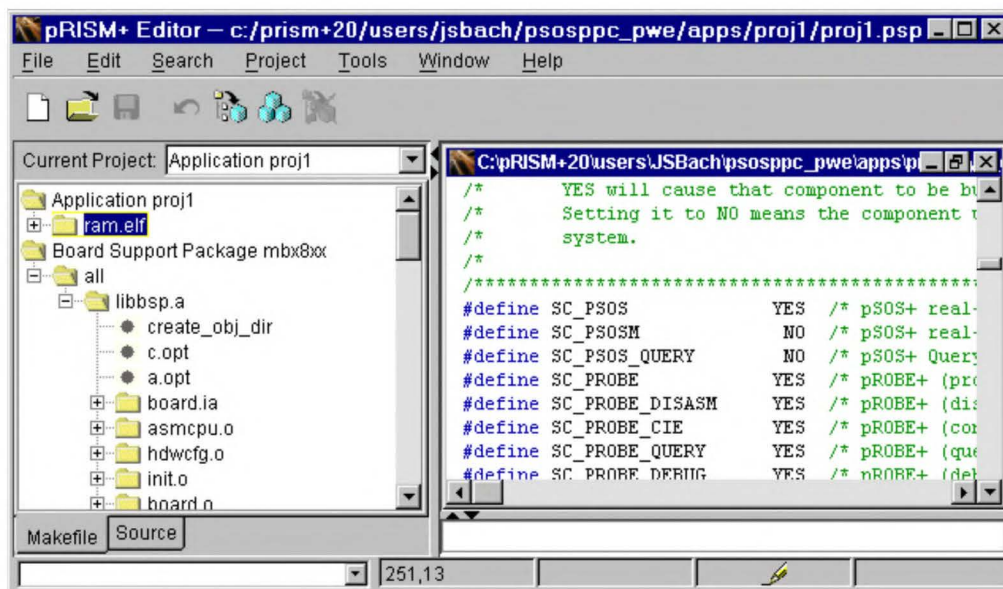


FIGURE 3-6 Accessing BSP Source Files in the Makefile Window

It is a common practice among experienced BSP developers to develop custom BSPs from existing BSPs. If you are working with several BSPs at the same time it is helpful to have multiple BSPs open. Note that you can open multiple BSPs by adding multiple BSP makefiles to the desktop.

Building ram.elf

Use this procedure to build the target executable `ram.elf` to continue with this tutorial.

1. Select `ram.elf` by highlighting it as shown in [Figure 3-7](#).
2. Click **Make the project** to complete the build. You can also select **Project → Make ram.elf** from the pRISM+ Editor menu. This will build `ram.elf`.

If you have not modified any of the `demo.c` code, you should not experience any problems during the compile. Should any problems arise, error messages will be displayed in the Message View. Double-click on a compilation error message to locate the line in a file where the error occurred.

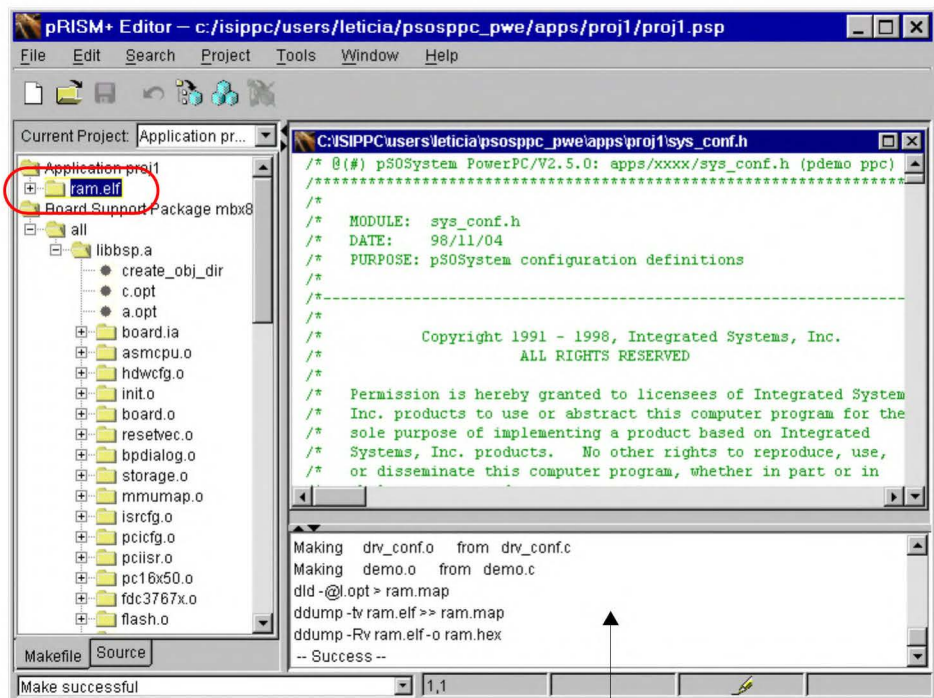


FIGURE 3-7 Building `ram.elf`

This concludes the pRISM+ Editor tutorial. For more information on the pRISM+ Editor, refer to [Chapter 5, pRISM+ Editor](#).

Now you're ready to proceed to downloading the target. To download the executable you have just built to the target, continue to [Section 3.7, Configuring the Target Board](#).

3.6 Using SNIFF+

In this section, you will learn how to use SNIFF+ to perform some basic development tasks. You can:

- Select SNIFF+ as your development tool of choice.
- Choose a pSOSystem sample application as a starting point.
- Get acquainted with SNIFF+.
- Build the sample application to produce a target executable.

From pRISMSpace Wizard, select SNIFF+ as your development tool of choice.

3.6.1 Choosing a pSOSystem Sample Application As a Starting Point

This tutorial will use a pSOSystem sample application, `pdemo`, to show you how to use pRISM+ tools. The steps are as follows:

1. From pRISMSpace Wizard, choose **Start with a pSOSystem example application**, then click **Next**.
2. From the list of sample applications, select `pdemo`, then click **Next**.

3.6.2 Setting Up a New Project

NOTE: When you start with a pSOSystem sample application, the pRISMSpace name is, by default, the same as the name of the pSOSystem sample application you're using. It is not user modifiable. You will also notice that the pRISMSpace directory is not user modifiable. The reason for this is explained in the exploring SNIFF+ section.

Click **Finish** to exit pRISMSpace Wizard. pRISM+ will place your pRISMSpace file in the pRISMSpace directory. This will also start SNIFF+.

pRISM+ Manager will start SNIFF+ and open a shared version of the sample application `pdemo` in your private workspace.

This completes the steps of configuring your first pRISMSpace.

3.6.3 Getting Acquainted with SNIFF+

pRISMSpace Wizard will start SNIFF+ for you and open the `pdemo` application. When this is successful, you will see the SNIFF+ Project Editor window showing the `pdemo` project as in [Figure 3-9 on page 3-14](#).

NOTE: Extensive on-line help is available for SNIFF+. To access on-line help, click the **?** in the SNIFF+ menu bar as shown in [Figure 3-8](#).

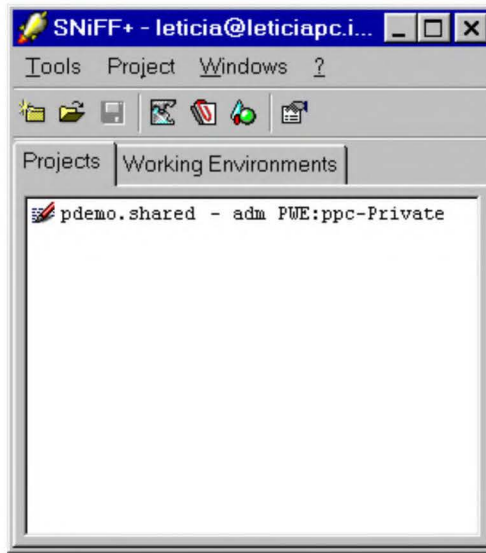


FIGURE 3-8 SNIFF+ Window

Looking At `pdemo` with SNIFF+ Project Editor

Now let's use SNIFF+ to look at the `pdemo` sample application (see [Figure 3-9 on page 3-14](#)). SNIFF+ offers a hierarchical project view. You can see the individual source files as well as the overall project structure. Some points of interest are:

- ① Source files of `pdemo.shared`. They're visible because of the check mark
- ② Check mark to decide if you want to display source files of a project
- ③ Project Hierarchy

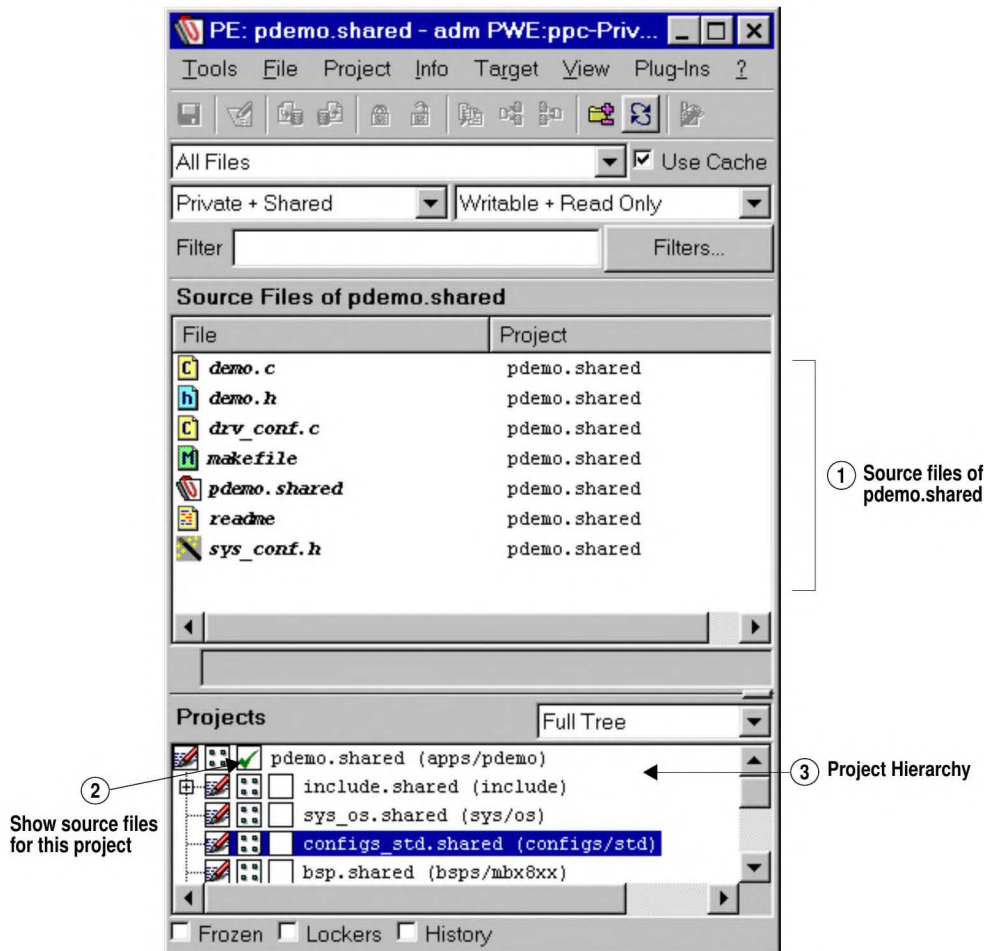


FIGURE 3-9 PE Window

Note the project structure that contains other parts of pSOSystem needed by pdemo in the **Projects** area. This source project is made up of pSOSystem include files (include.shared), operating system components files (os.shared), system configuration (configs.std.shared), and the BSP project (bsp.shared and bsp-src.shared).

NOTE: Ensure that the BSP shown in the Project Editor window matches the board you are using.

NOTE: If the BSP shown does not match what you plan to use for the tutorial, select **PrismSpace** → **Settings** from pRISM+ Manager to change the BSP setting as shown in [Figure 3-10](#).

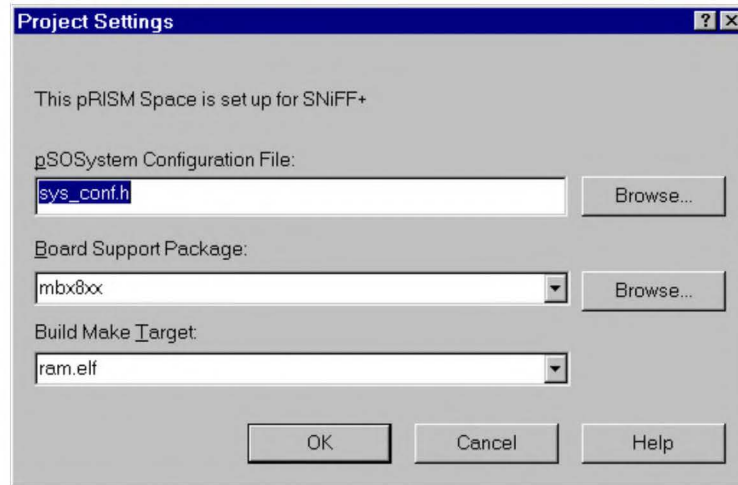


FIGURE 3-10 Project Settings Dialog Box

Editing a File with SNIFF+

Use this procedure to use SNIFF+ to edit a file.

1. To edit any file, double click on the name of the file in the Project Editor window.

NOTE: The pSOSystem SNIFF+ integration implements file sharing out-of-the-box. In order not to corrupt shared files, refer to [Chapter 6](#) on how to use SNIFF+ in your development environment after you complete this tutorial.

2. Make a local copy of a file before you make any changes to the file. Otherwise, you can corrupt the only version of this file in pSOSystem.
3. To make a local copy of any file, right click on the file and choose **Make Local Copy** from the menu.

NOTE: You do not need to change anything in the pdemo sample application to be able to continue with the tutorial.

Configuring pSOSystem for Your Application

Associated with each pSOSystem sample application is a pSOSystem configuration file called `sys_conf.h`. This file is used to specify which OS components are to be included in an application and how these components are configured.

Use this procedure to configure your application for pSOSystem.

1. To see the default setup for `pdemo` sample application, double click on `sys_conf.h` in the file list to bring up the pRISM+ Configuration Wizard.

NOTE: The default editor for `sys_conf.h`, the pSOSystem configuration file, is the pRISM+ Wizard, and not the default SNIFF+ Source Editor tool.

2. If you choose to use SNIFF+ Source Editor, right click on `sys_conf.h` and choose the **Edit** option.

NOTE: You do not need to change anything to continue with the tutorial.

Browsing pSOSystem with SNIFF

We will use some simple examples to see how SNIFF+ can help you understand pSOSystem code by allowing you to browse and navigate the pSOSystem source tree.

First look at how the configuration parameters in `sys_conf.h` file are used to configure pSOSystem. Specifically, let's see what happens in pSOSystem when you select the pSOS+ component. To do so, follow these steps.

1. Right click on `sys_conf.h` in the PE window.
2. Choose the **Edit** option from the pop-up menu to open `sys_conf.h` in a source editor window. From the source editor window you will see that pSOS+ is selected for `pdemo`.
3. Locate and highlight `SC_PSOS` and select **Info** → **Retrieve SC_PSOS from all source projects**.

You will see a list of every instance in pSOSystem where this parameter is used (see [Figure 3-11 on page 3-17](#)). You can then navigate to a pSOSystem configuration file `sysinit.c` where all the OS component initialization routines are called if the component is enabled in `sys_conf.h` file.

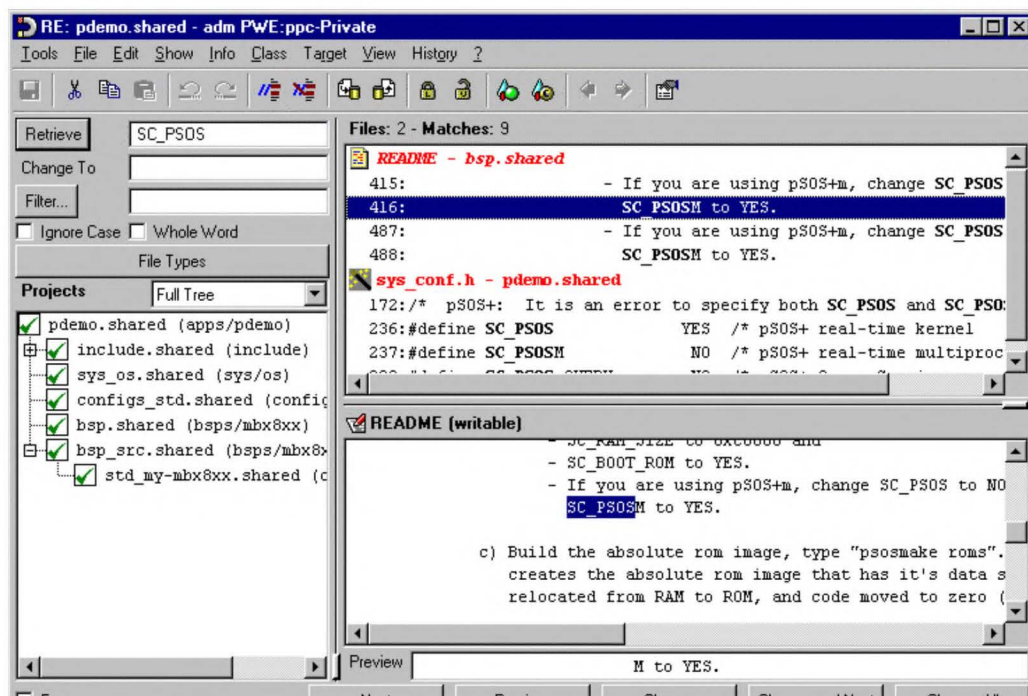


FIGURE 3-11 SNIFF+ Retriever Tool

The following steps show how the SNIFF+ Symbol Browser (“SB”) can help you to quickly access any function of interest, and how the SNIFF+ Cross Referencer can help you to see call relationships between functions.

1. From the PE window, select **Tools** → **Symbol Browser** to bring up the SNIFF+ Symbol Browser tool.
2. Use SB to look for a function called `PsosSetup` as shown in [Figure 3-12 on page 3-18](#). If **Select from All Projects** has not already been selected, you must do the following:
 - a. Select `pdemo.shared`; right-click it.
 - b. Select **Select from All Projects**.
 - c. In the **Filter** tab, type in `PsosSetup`.
 - d. Press **Enter** to search for the function `PsosSetup` in the `pdemo` project.

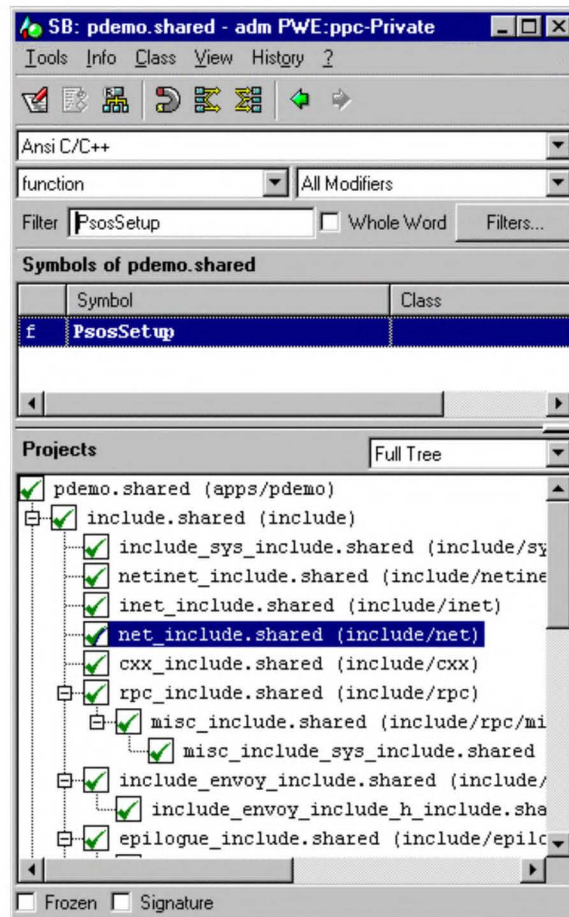


FIGURE 3-12 Symbol Browser (SB) Window

3. If **Select from All Projects** has already been selected, double-click on **PsosSetup** to take you to the source file where this function is implemented.
4. To see all the other functions **PsosSetup** refers to, from the SE window choose **Info** → **PSOSSetup Refers-To**. This will bring up the SNIFF+ Cross Referencer.

You will see a call tree that consists of **PsosSetup** and all the functions that it refers to. From here, you can see exactly how pSOS+ is configured.

Building a Target Executable

Return to the Project Editor (PE) window to build and compile your sample application to produce a target executable. In the PE window, highlight `pdemo.shared` and select **Target Make ram.elf**.

This concludes the brief SNIFF+ tutorial. For more information on using SNIFF+, see [Chapter 6, Using SNIFF+ in the pRISM+ Environment](#).

To download the executable you have just built to the target, see [Section 3.8, Configuring the Target Communications Parameters](#).

3.7 Configuring the Target Board

This section shows how to **configure your target board for communication with pRISM+ host tools**. You will:

- Connect your board through a serial connection to the host.
- Start a terminal emulation program on your development host.
- Use the terminal emulation program to communicate with the pSOSystem Boot ROM or flash on the target.
- **Configure the target to wait for a connection request from a host-based source-level debugger over the Ethernet.**

NOTE: Make sure that you have completed the steps described in [Section 3.1](#) and [Appendix A](#) to install a pSOSystem Boot ROM or flash on your board.

3.7.1 Connecting the Target Board to the Host Machine

You need to connect your target board using a serial cable to your host machine. This connection is needed by pSOSystem Boot ROM/flash to communicate to host-based terminal emulation program. You may need a null-modem cable.

3.7.2 Starting the Terminal Emulation Program on a Windows Platform

From the **Start** menu, select **Programs→pRISM+ 2.0<target_name>→pPROBE+ Console (COM1 or COM2)** to start a HyperTerminal session that is pre-configured to support the pSOSystem Boot ROM or flash.

This HyperTerminal is used to communicate with the pSOSystem Boot ROM or flash on your target board in order to set up your board to communicate with other pRISM+ host-based tools.

Proceed to [Section 3.8, Configuring the Target Communications Parameters](#).

3.7.3 Starting the Terminal Emulation Program on a UNIX Platform

You need to configure a terminal emulation program on your UNIX host to have the following settings:

9600 baud 8 data bits 1 stop bit no parity

Proceed to [Section 3.8, Configuring the Target Communications Parameters](#).

3.8 Configuring the Target Communications Parameters

Use this procedure to configure the target communications parameters.

1. Power up or reset your target board. The pSOSystem Boot ROM or flash should display a screen similar to the one in [Figure 3-13 on page 3-21](#).
2. To modify the default communication parameters, press any key within 60 seconds.

The objective is to set up your target board to wait for the host debugger through a network connection on the next reset or start-up. If your board contains nonvolatile storage to store these communication parameters, the settings that you set will be intact even after a power-down.

3. When prompted to <M>odify or <C>ontinue, enter **M** to begin modifying the default parameters stored in the pSOSystem Boot ROM or flash.
4. When prompted with `How should the board boot?`, configure the board to wait for the host debugger through a network connection on the next power up or reset by choosing Option 3 as shown in [Figure 3-14 on page 3-21](#).

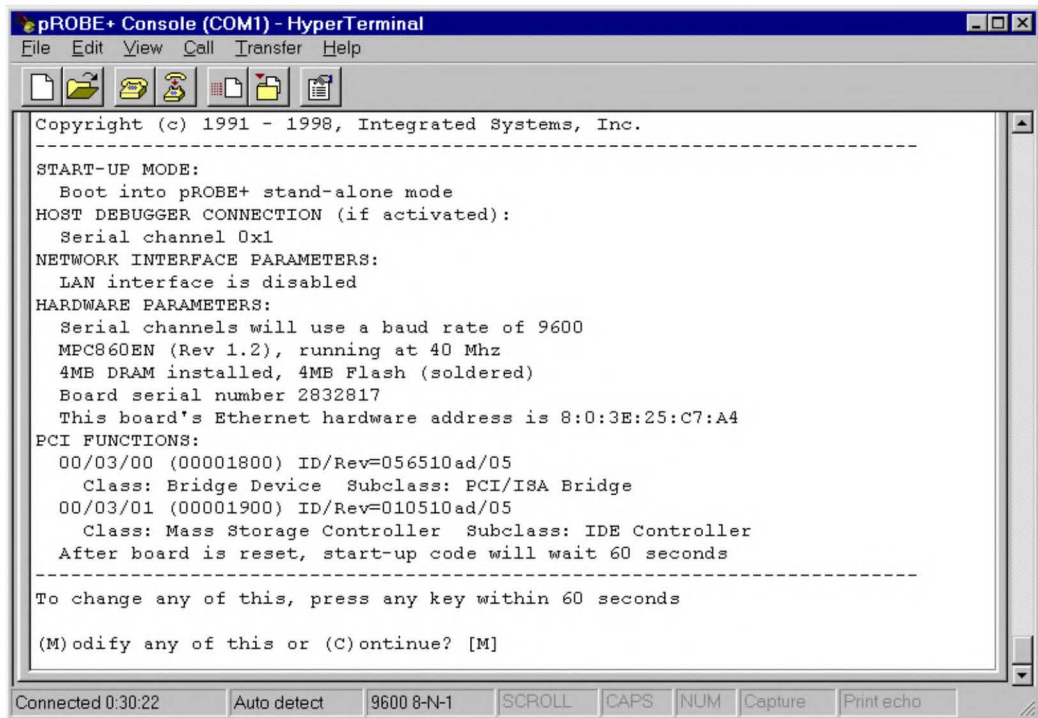


FIGURE 3-13 pROBE+ Console (COM1) — HyperTerminal Window

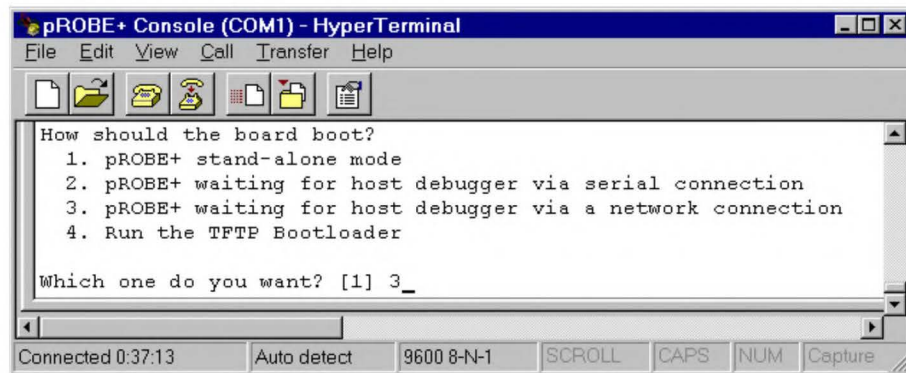


FIGURE 3-14 Board Start-Up Mode

- When prompted for NETWORK INTERFACE PARAMETERS, as shown in Figure 3-15, set your target board's network interface parameters to valid values for your network. You must use a valid IP address.

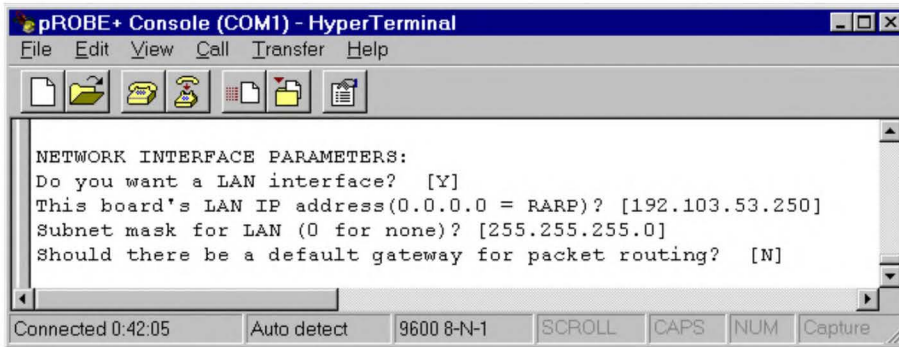


FIGURE 3-15 Network Interface Parameters Settings

- When prompted for HARDWARE PARAMETERS, as shown in Figure 3-16, you do not need to **change the default** baud rate used by the pSOSystem Boot ROM or flash to communicate with the host for this tutorial.

NOTE: If you do change the default baud rate, the pre-configured HyperTerminal settings on the Windows host will need to be changed accordingly. The target board you are using might not support a baud rate that you selected. See the pSOSystem BSP to see if a particular baud rate is supported before you change this setting.

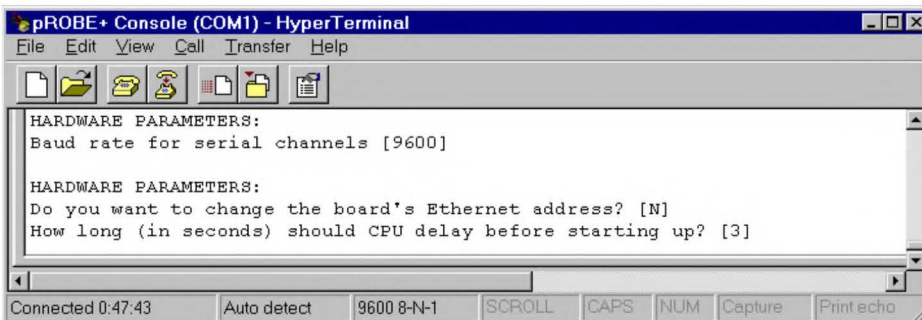


FIGURE 3-16 Default Baud Rate and CPU Delay Setting

In most cases, you do not need to change the board's Ethernet address.

7. Shorten the default CPU delay before starting up the setting, as illustrated in [Figure 3-16](#).

This parameter determines how many seconds after a reset your board will be ready to respond to a connection request from a host based debugger over the Ethernet. It will be set to 3 for this tutorial.

8. Enter **C** after you finish setting the parameters.

In this scenario, your target is ready to be connected to a source level debugger.

9. Check to see if your target board can respond to a ping. If your target board does not respond to a ping, check your network parameters. Make sure you can ping your board before attempting to connect it to any pRISM+ host tools.

3.9 Adding a Target Board to the pRISM+ Target List

pRISM+ keeps a target list for all the target boards you use. All the pRISM+ tools use this target list to get information on how to access target boards. Once you've registered a target board with pRISM+ by adding it to the target list, this target is accessible by all pRISM+ tools. You only need to enter board information once for each board.

In [Section 3.8, *Configuring the Target Communications Parameters*](#), you learned to set up your board to communicate over the Ethernet with pRISM+ host tools.

Use this procedure to add your board to the pRISM+ Target List so pRISM+ host tools can connect to it.

1. To add a board to the pRISM+ Target List, select **Target → List** from pRISM+ Manager to bring up the **Target List** dialog box.
2. Click **Add** to add a board to the list; this action opens the **Add Target** dialog box.
3. In the **Add Target** dialog box, enter the name of the board you would like to use to identify your board.

NOTE: This does not have to be the name of your board as used by DNS.

4. Click **OK** to open the **Properties for Target board_ID board** dialog box (see [Figure 3-17](#)).

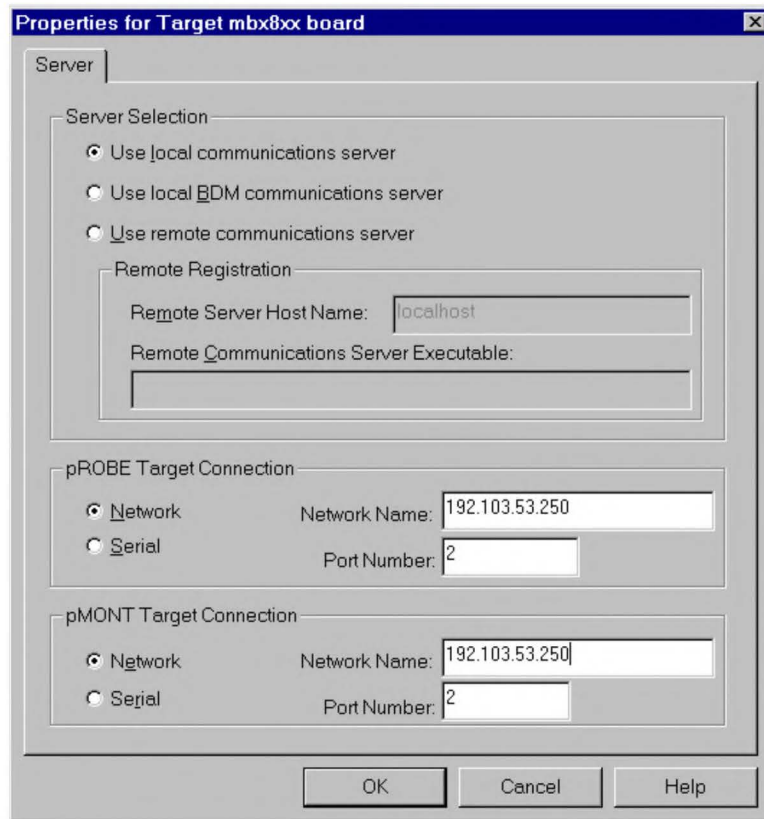


FIGURE 3-17 Properties for Target Dialog Box

5. As shown in [Figure 3-17](#), make sure you set the following in the **Properties for Target board_ID board** dialog box:
 - a. In the **Server Selection** area, choose **Use local communication server**. This is the default setting that tells pRISM+ to start a communication server on your local host machine to handle host to target communications.
 - b. In the **pROBE Target Connection** area, choose **Network as the** connection type. This setting tells pRISM+ to connect to target agent pROBE+ using the network connection. This means you will be using your source level debugger over the Ethernet to debug your application running on the target.

- c. In the **pPROBE Target Connection** area, set **Network Name** to the IP address of your target or to the name of your target in the hosts file that DNS uses. This is the name TCP/IP will use to find your target hardware.

NOTE: This name does not have to match the name of your target entered in the last section.

- d. There is usually no need to change the **Port Number** setting in the **pPROBE Target Connection** area.
- e. In the **pMONT Target Connection** area, choose **Network** as the connection type to tell pRISM+ to connect to target agent pMONT using the network connection. This means that you will be using Esp and Object Browser over the Ethernet to analyze your target's run-time behavior.
- f. In the **pMONT Target Connection** area, set **Network Name** to the IP address of your target or name of your target in the hosts file which DNS uses. This is the name TCP/IP will use to find your target hardware.

NOTE: This name does not have to match the name of your target entered in the last section.

- g. There is usually no need to change the **Port Number** setting in the **pMONT Target Connection** area.

NOTE: You can configure your application so that pPROBE+ and pMONT+ use different types of connections. For this tutorial, you will use the Ethernet connection for all the tools.

6. Click **OK** to save your settings.
7. Click **Select** to select your target board as shown in [Figure 3-18 on page 3-26](#).

This tells pRISM+ to connect host tools to this target by default when host tools are invoked.

Once a target has been selected, you will see its name displayed in the pRISM+ Target window as shown in [Figure 3-19 on page 3-26](#).

The target board and the host tools have been configured for communication. You are ready to download, debug, and profile the sample application pdemo.

8. Click **Close**.

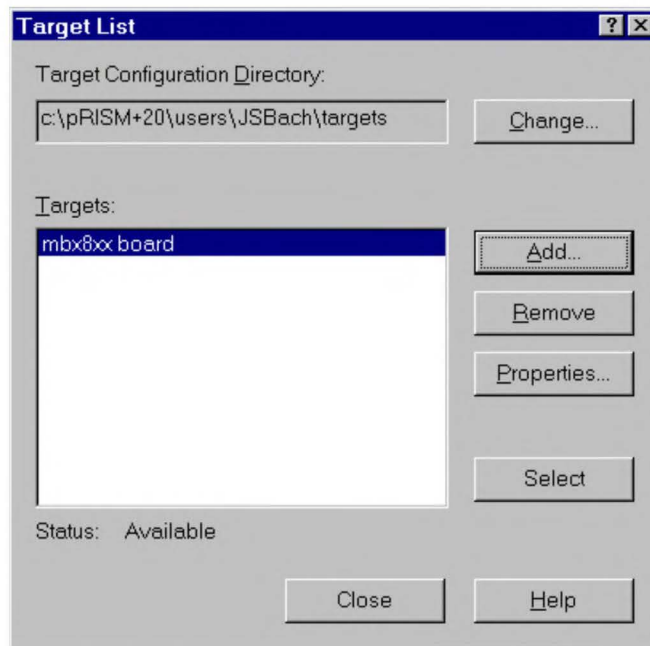


FIGURE 3-18 Target List Dialog Box



FIGURE 3-19 pRISM+ Target Window

3.10 Downloading and Debugging with SingleStep Source-Level Debugger

This section shows how to use SingleStep to download to the target the `ram.elf` file built earlier. You will run `pdemo` and make sure the application is running on your target successfully by examining the pSOS-specific information with the debugger.

NOTE: This section is not intended to be a debugger tutorial. For an in-depth tutorial on SingleStep, see [Chapter 9, *The SingleStep Debugger - A Tutorial*](#) and [the *SingleStep User's Guide* from SDS](#).

Use this procedure to download and debug with the SingleStep source level debugger.

1. Click the **SDS** button in pRISM+ Manager to start SingleStep. The SingleStep **Debug** dialog box appears (see [Figure 3-20](#)).

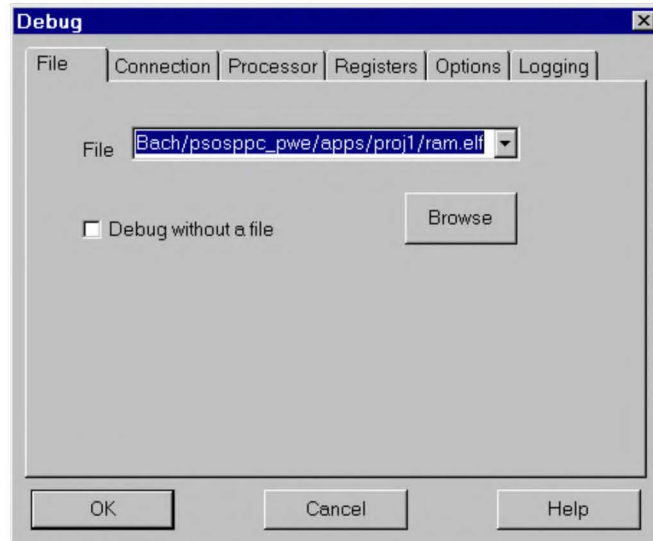


FIGURE 3-20 SingleStep **Debug** Dialog Box

pRISM+ has passed on the location of `ram.elf` and the target connection information to SingleStep. You can accept the default settings to begin the download.

Upon a successful connection of SingleStep to the target and a successful download, the SingleStep window appears as illustrated in [Figure 3-21 on page 3-28](#).

2. Click **Close** on the Debug Status window.
The execution is halted at the first line of the root task.
3. To run pdemo, click the **Go** button.
4. Wait for a few seconds. Click the **Stop** button.
5. To use the pSOS-awareness of SingleStep to verify that pSOS+ is up and running on the target board, select **Data** → **Kernel Objects** to open the **pSOS+ Kernel Objects and Configuration** window (see [Figure 3-22 on page 3-29](#)).

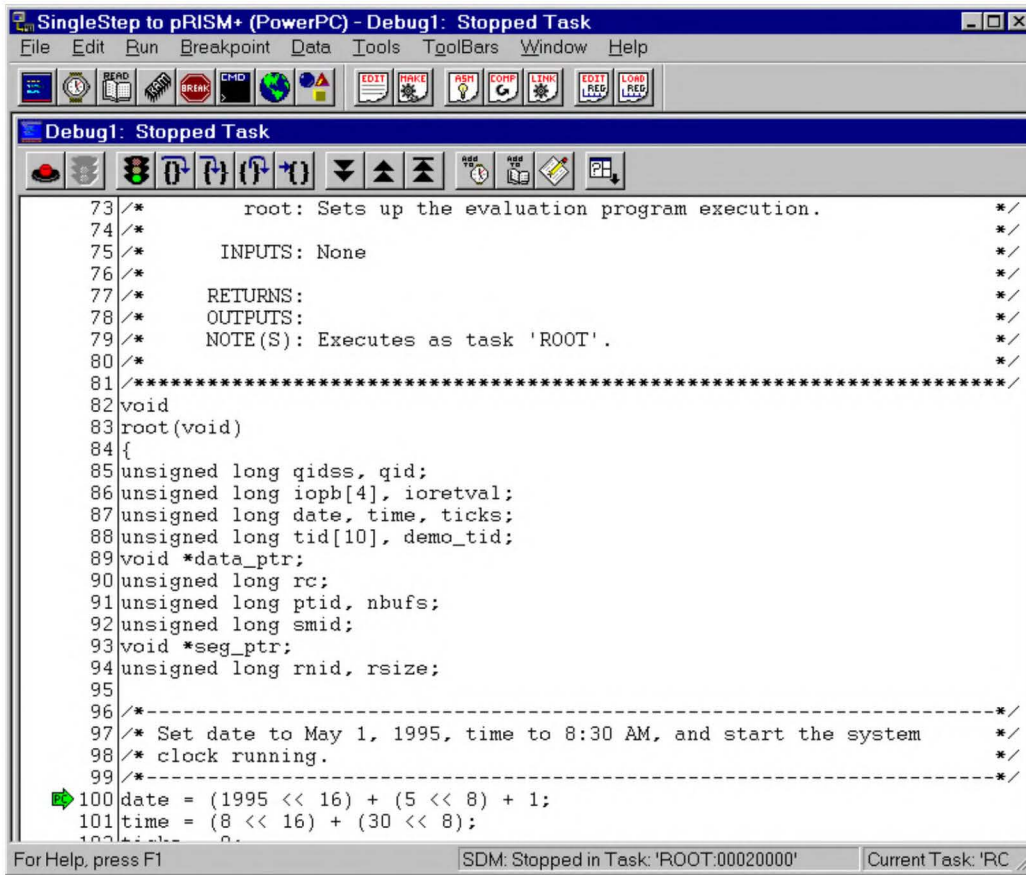


FIGURE 3-21 SingleStep Window

You can browse pSOSystem objects and configuration information.

6. To continue with the tutorial, you need to leave the target running. Click the **Go** button again to tell the target to run, and minimize SingleStep.

This concludes the SingleStep Debugger tutorial section. For more information about using SingleStep, refer to [Chapter 9](#). For a look at other tools proceed to [Section 3.12](#).

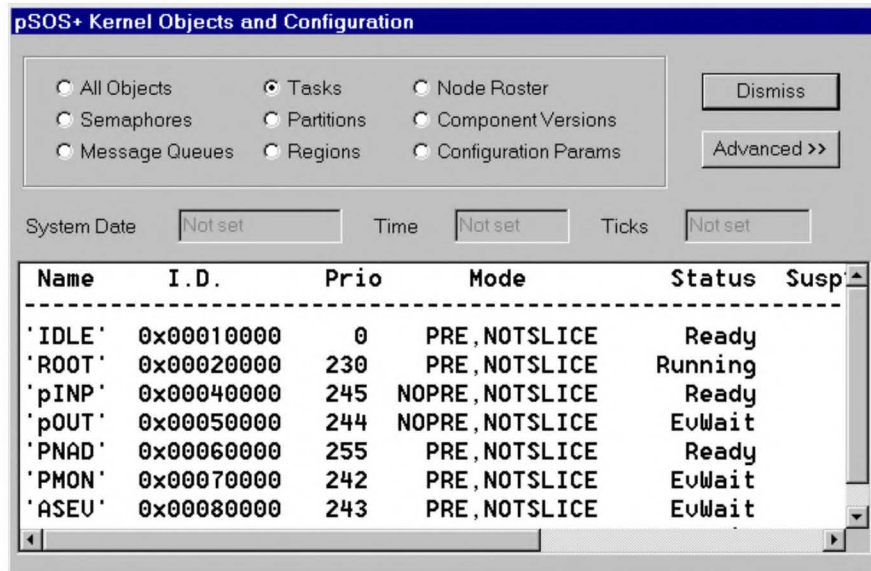


FIGURE 3-22 pSOS+ Kernel Objects and Configuration Window

3.11 Downloading/Debugging with SearchLight Source-Level Debugger

You will learn how to use SearchLight to download to the target the `ram.elf` file that you built earlier. You need to run `pdemo`, and make sure that the application is running on your target successfully by examining the pSOS-specific information with the debugger.

NOTE: This section is not intended to be a debugger tutorial. For an in-depth tutorial on SearchLight, see [Chapter 8](#).

Follow this procedure to download and debug with the SearchLight source-level debugger.

1. Click the **SearchLight** button in pRISM+ Manager to open a SearchLight window. Choose **File** → **Load** to open the **Load** dialog box. Accept the default settings in the **Load** dialog box to begin the download.

pRISM+ has passed on the location of `ram.elf` and the target connection information to SearchLight.

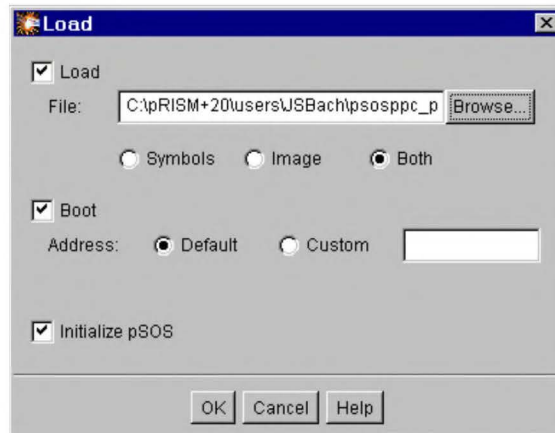


FIGURE 3-23 Load Dialog Box

Upon a successful connection of SearchLight to the target and a successful download, you will see the SearchLight window as shown in [Figure 3-24](#).

The execution is stopped at the first line of the root task.

2. To run `pdemo`, click the **Run** button. Wait for a few seconds.
3. Click the **Stop** button.
4. To use the pSOS-awareness of SearchLight to verify that pSOS+ is up and running on the target board, select **View** → **Tasks** to open the Tasks window.

You can see that all the tasks in `pdemo` had been started.

5. To continue with the tutorial, you must leave the target running.

Click the **Run** button again to tell the target to run and minimize SearchLight Debugger.

This concludes the SearchLight tutorial. For more information on using SearchLight, see [Chapter 8](#).

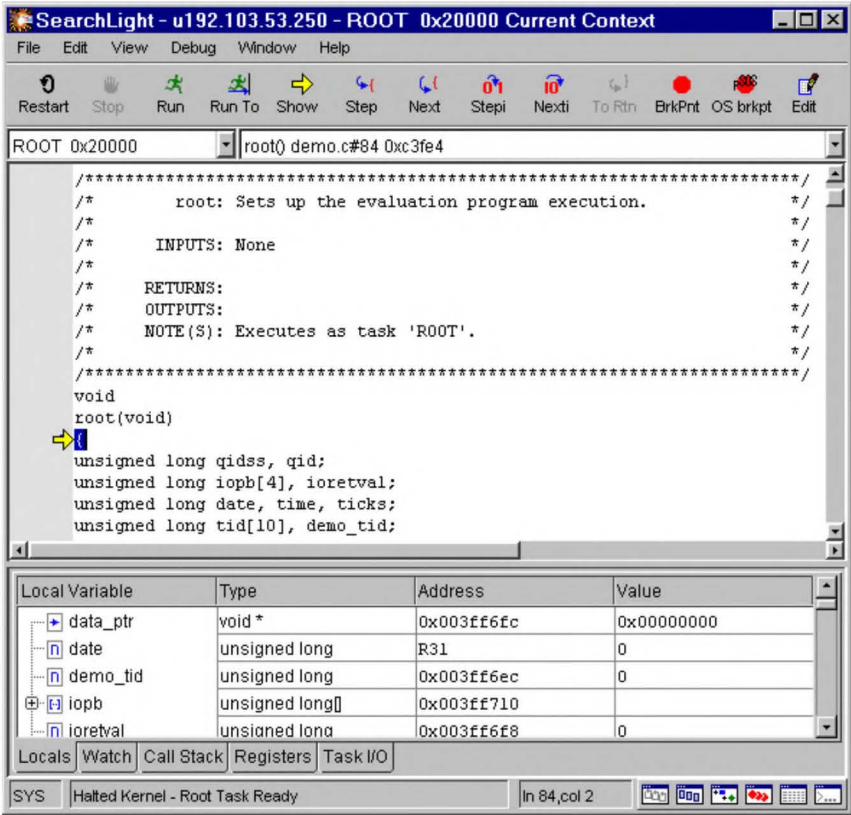


FIGURE 3-24 SearchLight Window

3.12 Using Object Browser

3.12.1 About Object Browser

Object Browser helps you to understand run-time behavior of a target system by taking periodic snapshots of operating system objects at user-defined intervals while your system is running.

The host-based Object Browser uses the pMONT target agent to obtain target information. Since pMONT runs as a set of tasks on the target, your target application must include pMONT and must be running for Object Browser to work.

NOTE: Make sure that your target is running prior to invoking Object Browser.

To invoke Object Browser, click the button in the pRISM+ tool bar.






FIGURE 3-25 Object Browser Window

There are two kinds of graphical representations used by Object Browser: Snapshot View and Graph View.

Graph View

The Graph View is used to display run-time information for the following objects (see [Figure 3-26](#)):

- Stack 
- Region 
- Message Queue 

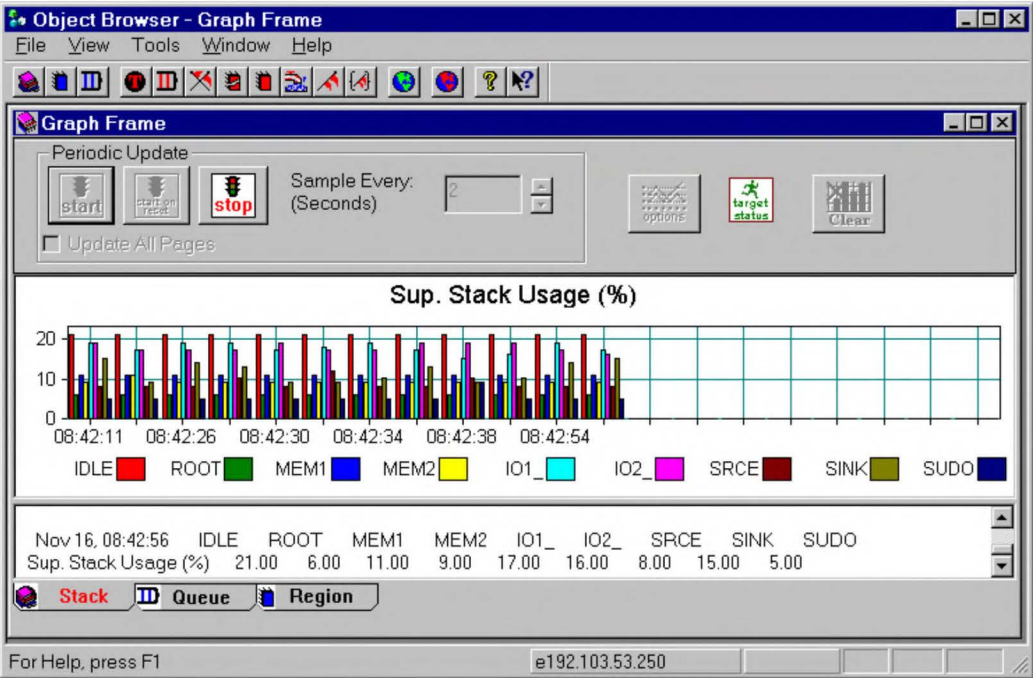










FIGURE 3-26 Graph Frame Window

Snapshot View

The Snapshot View is used to display run-time information for the following objects:

- Task 
- Queue 
- Semaphore 
- Partition 
- Region 
- Stack Problem 
- Mutex 
- Conditional Variables 

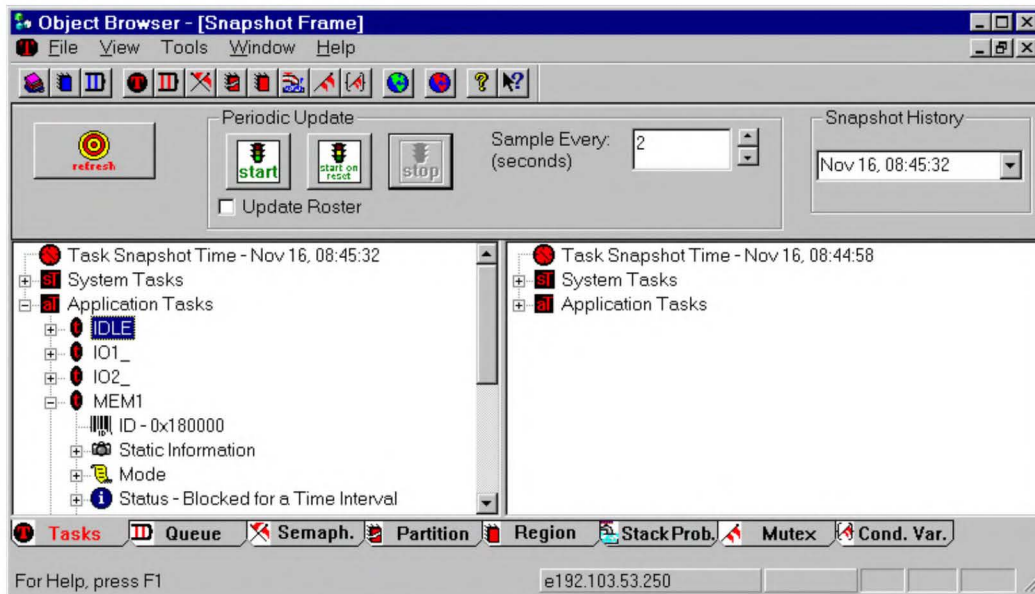


FIGURE 3-27 Snapshot Frame Window

Object Browser is used to take snapshots of your target system, and the data is displayed in one of the two graphical presentation modes.

1. Determine how often you want to sample by changing the value in the **Sample Every** box.
2. Click **Start** to start periodic sampling of your target system.

In [Figure 3-28](#), a snapshot of tasks is taken every two seconds. The current sample appears on the left and the historic samples appear on the right.

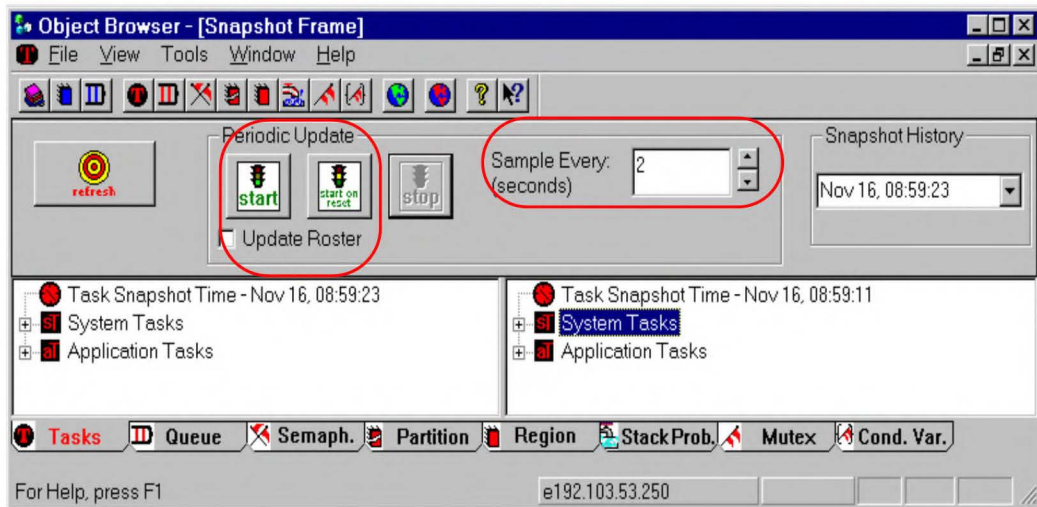


FIGURE 3-28 Periodic Sample

This concludes the Object Browser tutorial. For more information on Object Browser, refer to the Object Browser chapter.

3. Now stop any on-going sampling and quit Object Browser and proceed to the next section on ES_p.

3.13 Using ES_p

ES_p works like a logic analyzer for software. It can provide users with an event-by-event view of your target run-time behavior between a user defined trigger point and de-trigger point.

The host-based ES_p uses the pMONT target agent to obtain target information. Since pMONT runs as a set of tasks on the target, your target application must include pMONT and must be running for Object Browser to work.

NOTE: Make certain that your target is running prior to invoking ES_p.

To Launch ES_p, click on the **ES_p** button in pRISM+ Manager.

3.13.1 Configuring an Experiment

A session in which ES_p collects data from the target is called an *experiment*. Before you can start an experiment, configure the experiment by specifying the following:

- **Trigger** — This tells pMONT when to start a data collection. This can be any pSOSystem system calls or user events.
- **Log** — This tells pMONT what to log and what to ignore while a data collection runs. This can be any pSOSystem system calls or user events.
- **Detrigger** — This tells pMONT when to stop the data collection. This can be any pSOSystem system calls, user events or end-of-target-buffer condition.

1. To start a new experiment, select **File** → **New Experiment** from ES_p main menu.

The Configuration window appears (see [Figure 3-29 on page 3-37](#)).

For this tutorial section you will perform the following steps:

1. Enter **singbuf1** in the **Experiment Name** box. This is the name of your experiment.
2. Use the pSOS+ call **rn_getseg** as the trigger. To do this, right click on the **rn_getseg** call in the **SVC** list.
3. Select **Trigger** to tell pMONT to begin logging data on the first **rn_getseg** call after you start the data collection.
4. Leave the default setting for **Log**. The default instructs pMONT to ignore the **i_enter**, **i_return**, and **tm_tick** calls.

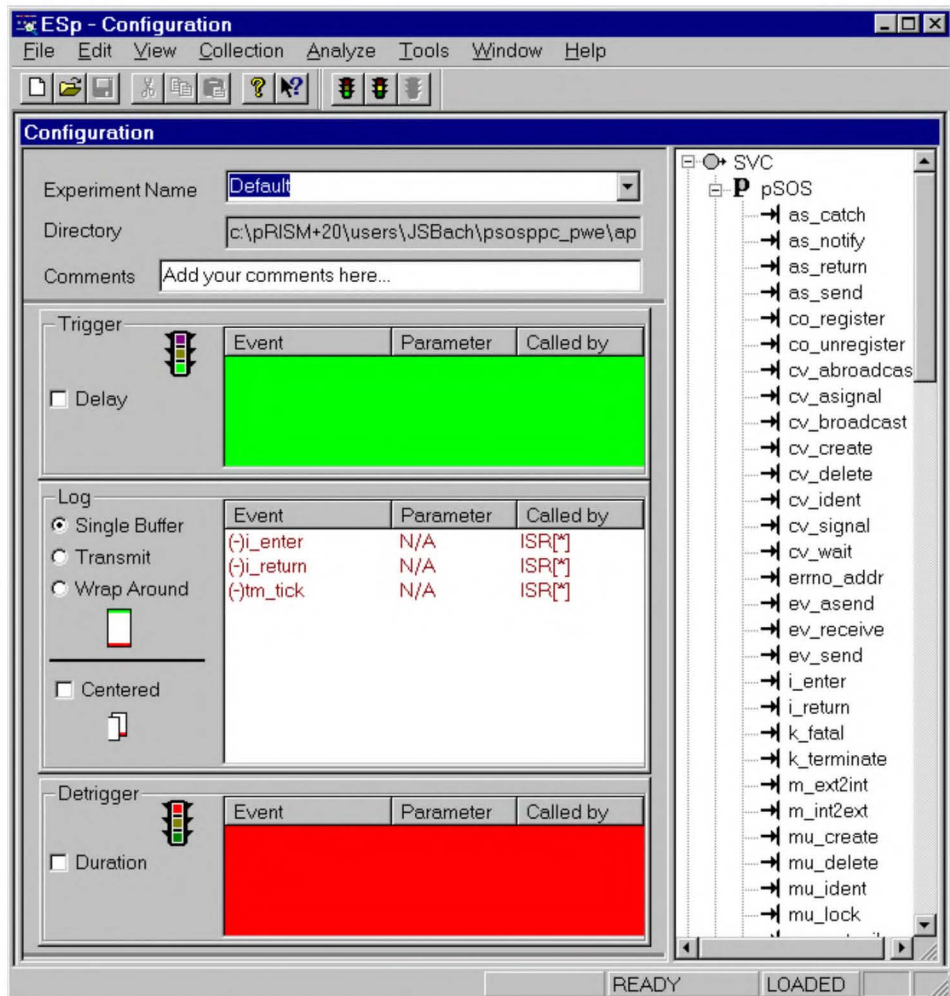


FIGURE 3-29 ES — Configuration Window

If these calls were not made or did not function correctly, you would not have been able to do everything you have done so far in this tutorial. By ignoring these events, you are saving more space in the target memory buffer for events you do want to log and analyze.

5. Choose **Single Buffer** in the **Log** area as the target buffer management scheme.

This tells pMONT to start gathering data on **Trigger**, and stop gathering data on a buffer full condition if it happens before the **Detrigger** point is reached. This **Single Buffer** of data will then be sent to host-based ES_p for analysis.

6. Make sure that your settings match those shown in [Figure 3-30](#).

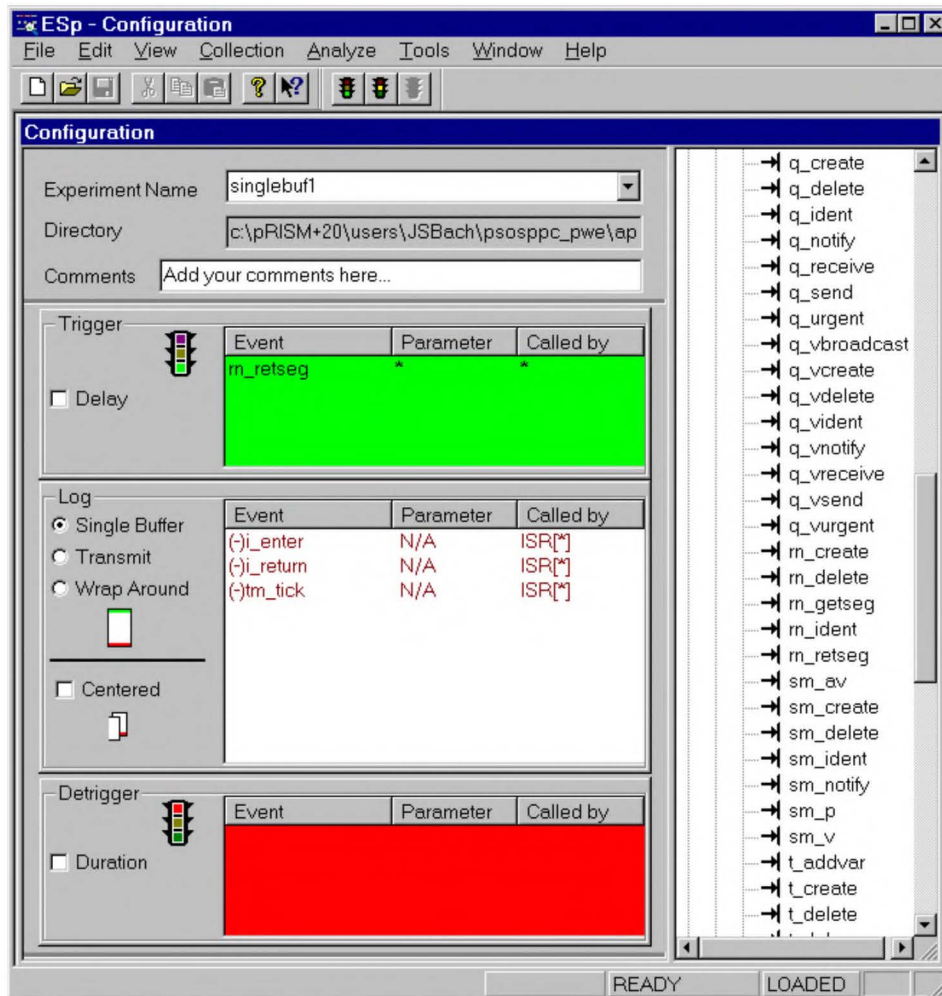


FIGURE 3-30 Experiment Configuration

3.13.2 Starting a Data Collection

Use this procedure to initiate a data collection.

1. To start a data collection, click on the **Green Traffic Signal** button.

The Experiment Monitor appears to show you the progress of ES_p. A few seconds later, you are notified that the experiment was ended by a buffer full condition (see [Figure 3-31](#)).

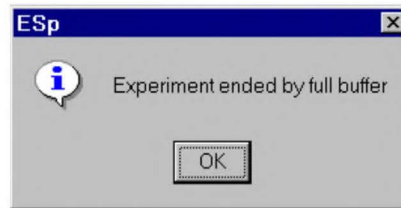


FIGURE 3-31 Experiment Notification

2. Click **OK** to see the display of target events.

3.13.3 Analyzing the Data

You can look at what happened on the target from the time you started the data collection to the time the target buffer is filled (see [Figure 3-32 on page 3-40](#)).

To analyze the data, use these steps.

1. Click on the very first event.

You get a time-stamped report on the event in the lower window (you might need to zoom in). This is a `rn_getseg` call, which is expected since this was set as the trigger point.

2. To turn on the legend, select **View** → **Legend**.
3. Click **Task State** to get a full display of task states. The solid green line represents the CPU execution path.
4. Follow each event on the execution path to see exactly what happened on the target.
5. Right click on any event. This allows you to make that event a reference point to calculate delta time between that event and any other event in both directions (see [Figure 3-33](#)). Click on another event.

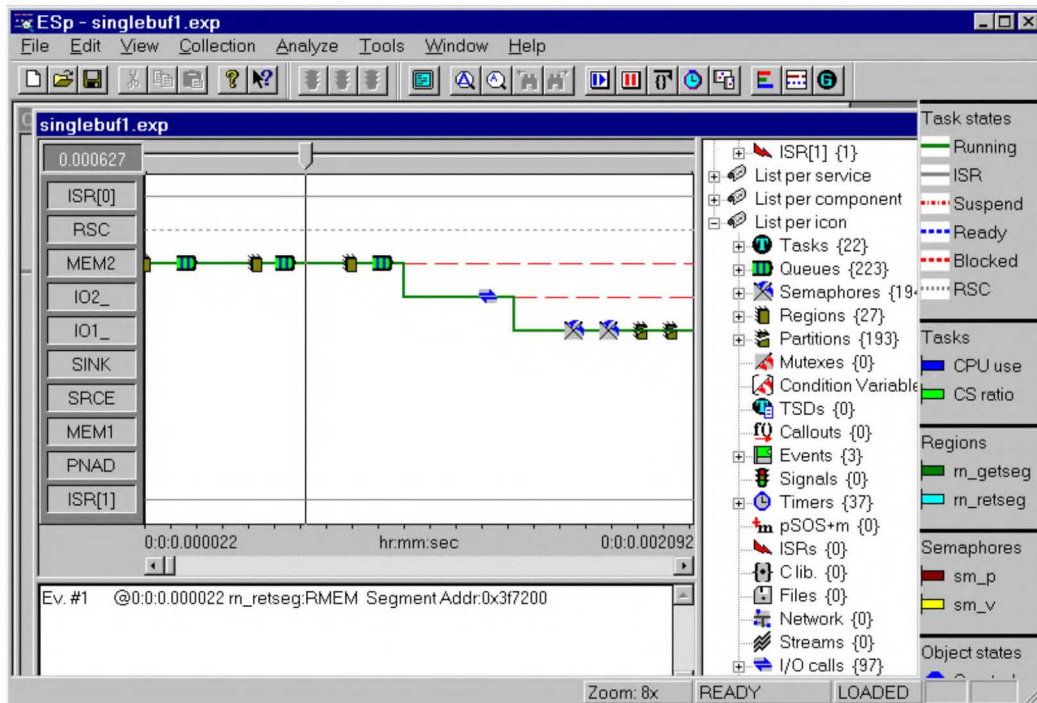


FIGURE 3-32 Data Analysis

6. Choose to see only the CPU execution trace without any events; right click anywhere on in the events window and choose **Execution only**.

This gives you a way to look for patterns in CPU scheduling behavior (see [Figure 3-34 on page 3-42](#)).

7. To get CPU use by task, click on the name of a task in the Events window.
8. Quit ESsp.

This concludes the ESsp tutorial. For more information about using ESsp, see [Chapter 10](#).

This also concludes the pRISM+ tutorial.

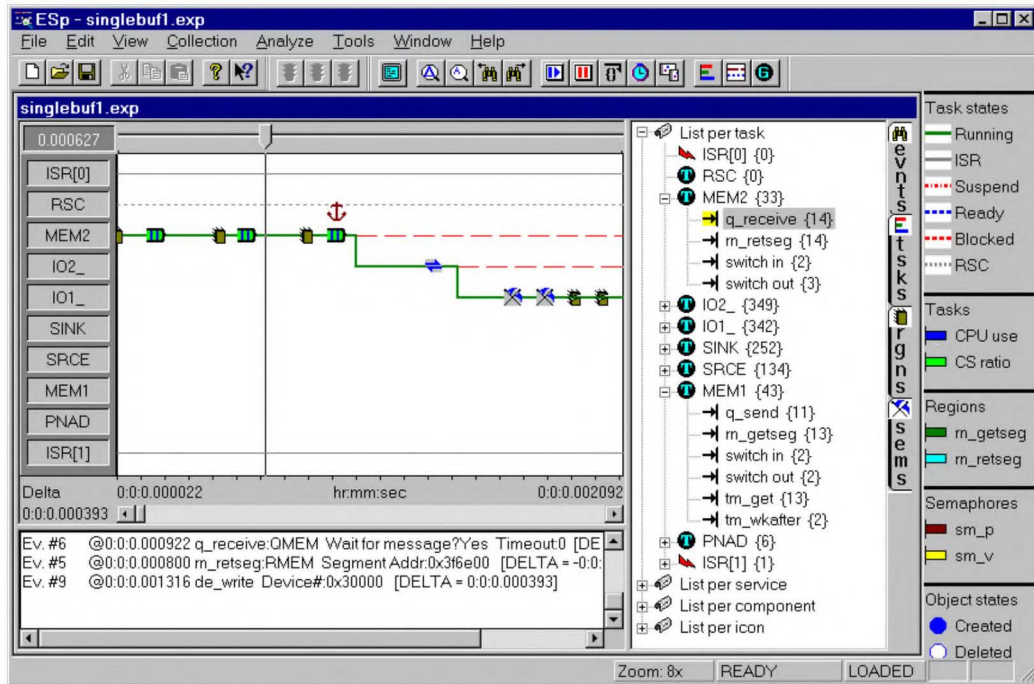


FIGURE 3-33 ESp Experiment Example

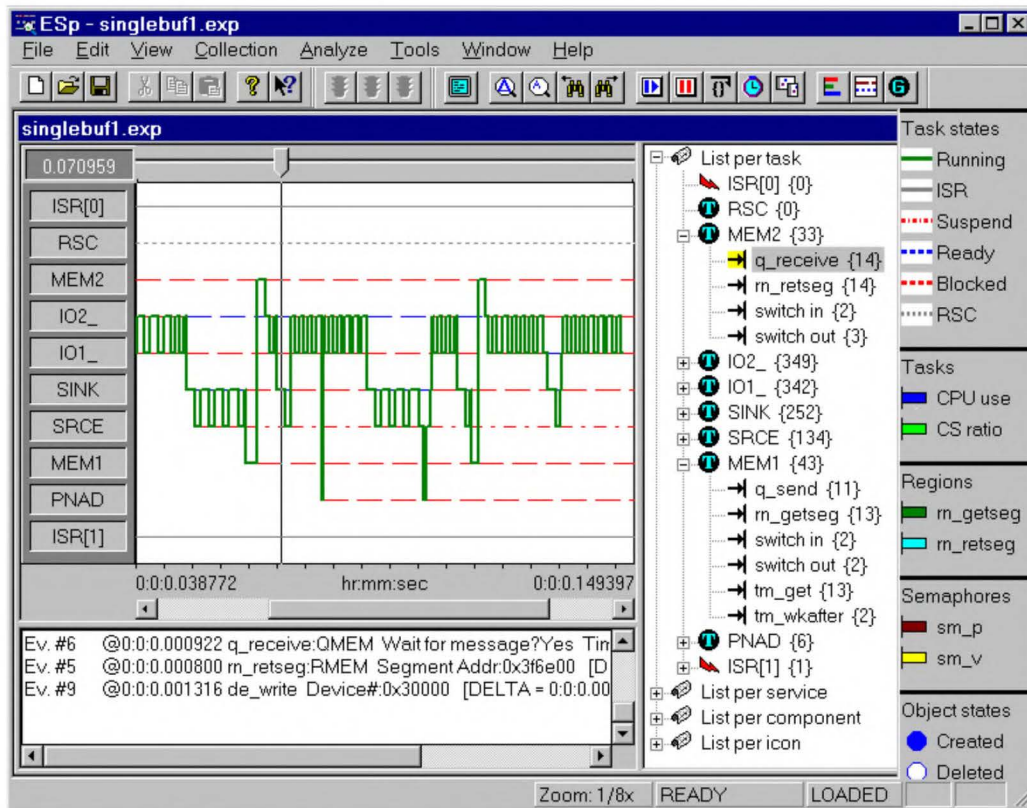


FIGURE 3-34 Patterns in CPU Scheduling Behavior

4

Understanding the pRISM+ Manager

This chapter explains more about the pRISM+ Manager, how to use some of the pRISM+ Manager's features, and how these features affect the other pRISM+ Tools.

4.1 The pRISM+ Development Environment

The pRISM+ Development Environment provides a comprehensive set of tools for constructing embedded applications. The central application you will use is the pRISM+ Manager. It provides a context for your project, called a pRISMSpace. All other tools work within this pRISMSpace context. Each of these tools will be explained in the remaining chapters of this manual.

4.1.1 Overview

Using pRISM+ Manager, you will create a pRISMSpace for your project. The pRISMSpace is the pRISM+ project definition. It contains information that enables pRISM+ Manager to invoke additional tools in your project's context. This information includes the project directory, the currently selected BSP, the current project editor, and target-related information.

After creating a pRISMSpace, you will use pRISM+ Configuration Wizard to select which operating system components you want to include in your application. Then you will use the project editor's or pRISM+ Manager's build command to create your application executable.

Once you have a downloadable image, you can use pRISM+ Manager to define and select a physical target board. You can now use pRISM+ Manager or the debugger to download your application to the board.

When your executable code is running on the target board you can use run-time-analysis tools such as the SearchLight Debugger, ESp, or Object Browser to determine the state of your embedded application.

A typical development cycle involves these processes:

- Writing source code in a project editor.
- Compiling and linking the executable image.
- Downloading and debugging the embedded application.

pRISM+ provides alternate paths to accomplish this edit-compile-debug cycle.

While setting up a pRISMSpace, you are asked which project editor you want to use. pRISM+ currently supports two project editors: pRISM+ Editor and SNiFF+. In addition, pRISM+ supports a variety of debuggers, including SearchLight for PPC, MIPS, and 68K, from Integrated Systems; and SingleStep debugger for PPC and 68K, from SDS. The default project editor and debugger are pRISM+ Editor and the SearchLight debugger.

Which project editor you choose depends on what type of development you will be doing.

- pRISM+ Editor is a fast-start environment targeted specifically at firmware developers who are bringing up a custom board.

Its makefile orientation and simplicity are ideal for working with multiple makefiles, including switching between multiple BSPs. pRISM+ Editor focuses on working with existing makefile, and presenting the optimal Compile-Edit cycle in a familiar user interface.

- The optional project editor, SNiFF+, is targeted at larger groups of developers and/or larger code bases. SNiFF+ is a code comprehension tool, also known as a Source Code Engineering tool.

It is a collection of static analysis tools for source code analysis, browsing and comprehension. The benefits are automating and simplifying manual and error prone programming tasks, resulting in dramatic improvements in developer productivity.

Choosing which debugger to use can be done later. You can use any Integrated Systems-supported debugger for your target.

4.1.2 pRISM+ Manager and the pRISMSpace

pRISM+ Manager is your central control panel for pRISM+. (See [Figure 4-1 on page 4-3](#).) It provides three major services:

- Project management through the pRISMSpace.
- Target services for defining and selecting target boards.
- Tool services for integrating custom tools into the pRISM+ environment.

Each of these services is managed independently so that any new project can access any previously defined tool or target.



FIGURE 4-1 pRISM+ Manager Toolbar

Creating a pRISMSpace

Select **File** → **New** from the pRISM+ Manager menu to initiate the pRISMSpace Wizard. The pRISMSpace Wizard is a series of dialogs that lead you through the construction of a pRISMSpace. The pRISMSpace Wizard presents options that pertain to the project editor you are using, what code base to start with, and where you want the pRISMSpace to be created.

1. If you have purchased and installed the SNIFF+ product, the first dialog shown is the **Tools Options** dialog. Select the project editor that most meets your needs.

NOTE: If you did not purchase or install SNIFF+, the pRISMSpace Wizard skips this dialog.

When you select either **SNIFF+** or **pRISM+ Editor** in the **Tools Options** dialog and then click the **Next** button, the **Choose a Starting Point** dialog box displays.

2. The **Choose a Starting Point** dialog is where you choose between using sample code or existing code. You can pick pRISM+ sample applications or your existing code or makefile to start a pRISM+ project.

- For pRISM+ Editor, you can choose between a pRISM+ sample application or your own makefile-based project.
- For SNIFF+, you can choose between a sample application and an existing code base.

The subtle difference here is that pRISM+ Editor requires that you have a makefile, while SNIFF+ does not. In addition, SNIFF+ requires you to adjust the User Shared Source Working Environment (SSWE) to point to your source tree (refer to [Chapter 8](#)).

NOTE: If you want to use pRISM+ Editor but you do not have a makefile, you can copy one of the makefiles from the sample application directory under the pSOSystem directory. These makefiles contain the appropriate references and structure for building a pSOS+ application. For example, you can copy the `$PSS_ROOT/apps/pdemo/makefile` to the directory that contains your source code, and then modify the makefile to add your own source code files.

When you choose **Start with a pSOSystem sample application**, then click **Next**, the **Choose a pSOSystem example** dialog appears.

3. The **Choose a pSOSystem example** dialog shows various sample applications you can select.

The sample applications are useful for providing a starting point for new projects. Select the sample that most closely matches your target application requirements. Then you can modify the sample application to fit your needs.

NOTE: For additional information about the sample applications, refer to the *pSOSystem Application Examples* manual or the sample application README files in each of the sample application directories.

After you select a code base to work with and click **Next**, the **Finish this new project** dialog appears.

4. The **Finish this new project** dialog asks where the pRISMSpace should be located and what it should be called.

By default, sample applications are set up under your home directory. You can change this to point anywhere you want; for example, `c:\MyEproject\pdemo`.

After you specify the location of your new pRISMSpace, click the **Finish** button to begin creating the new pRISMSpace.

How pRISM+ Manager Sets Up Projects

pRISM+ Manager sets up projects slightly differently for pRISM+ Editor and SNiFF+.

- For pRISM+ Editor projects, pRISM+ Manager copies all necessary files to the pRISMSpace directory.
- For SNiFF+, pRISM+ Manager copies only a subset of the files to the pRISM-Space directory. SNiFF+ uses an advanced feature called a *virtual path* to access source files that are not in your pRISMSpace directory.

pRISMSpace Project Settings

In the pRISMSpace **Project Settings** dialog (see [Figure 4-2 on page 4-6](#)), you tell pRISM+ Manager the following:

- the name of the pSOSystem configuration file (usually `sys_conf.h`)
- the Board Support Package (BSP) to use with this project
- the default makefile target for the project

For more information about changing your BSP, see [Section 15.1.2, *Incorporating a Custom BSP for pSOSystem* on page 15-3](#).

The pRISMSpace **Project Settings** dialog also displays which project editor has been selected for this project (SNiFF+ or pRISM+ Editor).

Normally, the pSOS configuration file is named `sys_conf.h`. However, you can enter a new name into the **pSOSystem Configuration File** field. Note that changing this file requires your application to be completely recompiled and re-linked.

The **Board Support Package** field allows you to switch between BSPs. By default, the drop-down list shows all the BSPs provided with pRISM+. These BSPs are in the pSOSystem directory under the `bsps` subdirectory. Any additional BSPs you add to this directory will show up in the list.

To add additional BSPs that do not reside in the `bsps` directory, you can enter the path to your BSP directory, or use the **Browse** button to navigate to your BSP directory.

The last field in the pRISMSpace **Project Settings** dialog is **Build Make Target**. This is used as the Current Target by pRISM+ Manager and pRISM+ Editor when building the project.

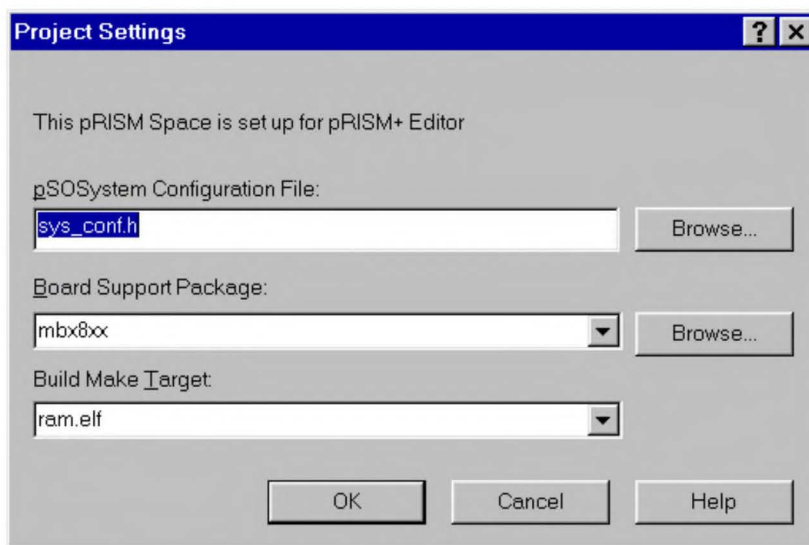


FIGURE 4-2 Project Settings Dialog Box

Build Command

pRISM+ Manager passes the Current Target name to make when you press the **Build** button on the toolbar (or select the **PrismSpace** → **Build** menu command). Normally, you will use the project editor to build your applications. The pRISM+ Manager Build command is provided as a shortcut.

The pRISM+ Manager Build command is configured by the Tools Manager. Under the **Standard** tab of the **pRISM+ Tools** dialog is an entry for the Build command. You can edit this command to customize your build process. See [Section 4.1.3, The Tool Manager on page 4-7](#) for information about the **pRISM+ Tools** dialog.

When the Build command is run, the results display in the Log window. You can access the Log window by selecting **View** → **Log Window**.

Switching to a Different pRISMSpace

To switch to a different pRISMSpace, use the **File** → **Open** command to find the new pRISMSpace, or select one from the “recently used” list at the bottom of the **File** menu. Switching to a different pRISMSpace loads a new project context and closes any open tools that were launched in the previous pRISMSpace context.

4.1.3 The Tool Manager

Tools are accessed from the **Tools** menu or from the buttons on the toolbar. Tools have multiple levels of integration into pRISM+. The simplest integration is running a program passing in pRISMSpace context information. Some tools integrate further by implementing special interfaces that allow pRISM+ Manager to dynamically update their pRISMSpace context.

pRISM+ Manager allows you to customize your standard pRISM+ Tools and add new custom tools through the **pRISM+ Tools** dialog box. Choose **Tools** → **Customize** to open this dialog box (see [Figure 4-3](#)).

Selecting a tool from the **Tool List** displays the properties of that tool. You can add new tools and order the menu using the buttons on the dialog.

- The **Title** field is the name that appears on the **Tools** menu and in the tool tip for the button.
- The **Command** field defines the name of the program to run.
- The **Arguments** field defines a list of items to pass when the command is invoked. Use pRISM+ macros to pass current context information onto the tools. These macros are available from the list displayed when you click the arrow button.
- You can control the current directory for the program you are running by setting the **Initial Directory** field.
- The check boxes allow you to place the custom command onto the **Tools** menu or the toolbar. If you select **Add To Toolbar**, you can specify bitmaps that will display on the large and small toolbars.

The **Advanced** button brings up the **Advanced Tool Properties** dialog.

Advanced Tool Properties

The **Advanced Tool Properties** dialog is where you control when tools are launched. Each tool can be started or stopped when certain events occur. These events occur when a project or target is opened or closed and when the current application is started. Check the box to enable the selection drop-down.

- If the tool needs exclusive access to the target, enable **Used for controlling the Target**. When this control is enabled, pRISM+ Manager warns when conflicts occur.
- You can use the tool manager to start a CORBA service that may be needed by other tools. Check the **CORBA Server** check box and specify the CORBA service name.

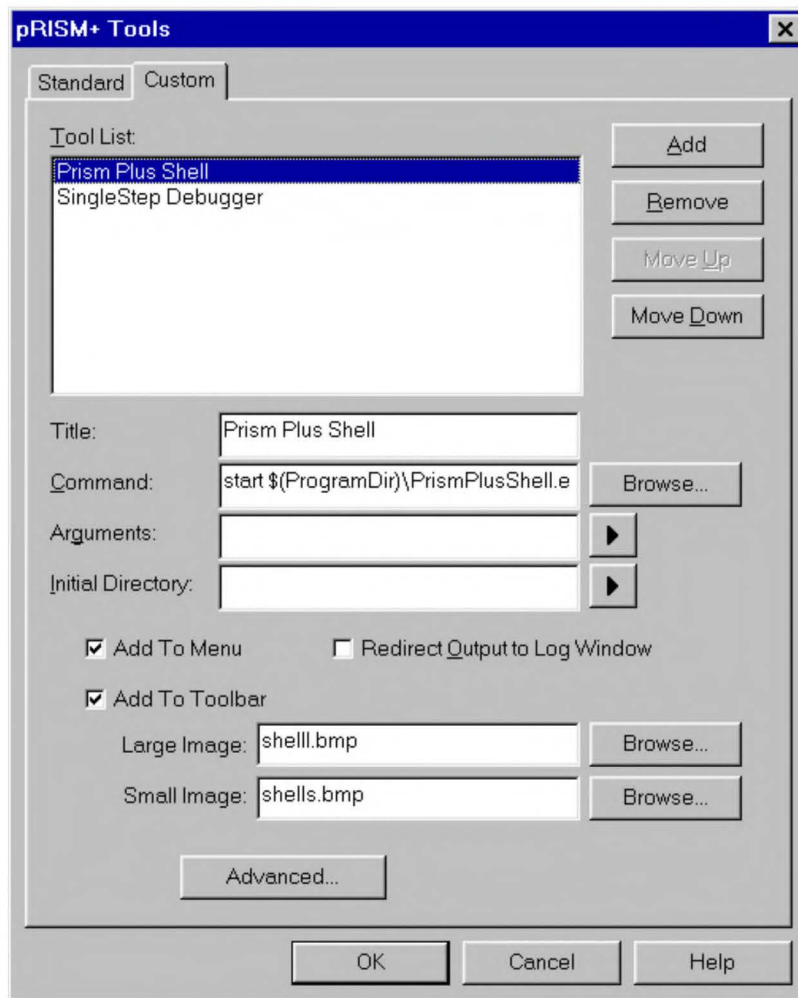


FIGURE 4-3 pRISM+ Tools Dialog Box — Custom Page

- The **Implements pRISM+ Tool Interface** check box is used for tools that want to communicate with pRISM+ Manager. This allows the tool to receive dynamic changes to the pRISMSpace context.

For more information about integrating tools into pRISM+, see the *Third Party Integration Guide*.

4.1.4 The Target Manager

You can set up different target board definitions in pRISM+ Manager using the Target Manager's **Target List** dialog. (See [Figure 4-4 on page 4-9](#).)

The **Target List** dialog is where you specify the Target Configuration Directory and define targets.

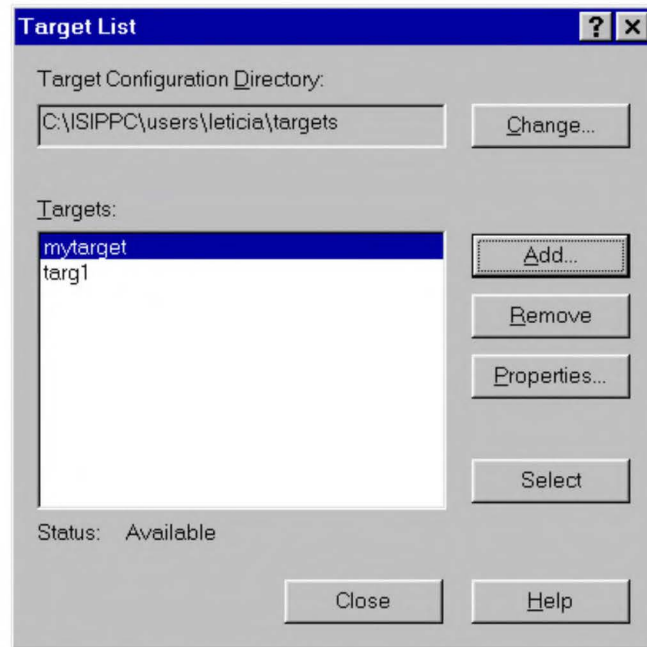


FIGURE 4-4 Target List

In a multi-user environment, you can share a **Target Configuration Directory** by setting the directory to a shared network resource. pRISM+ Manager will then help manage access to the targets, giving status and warning if the target is already in use. Click on the **Change** button to locate and set the shared network directory.

Use the **Add**, **Remove**, and **Properties** buttons to edit target definitions. Select the **Add** button to see a prompt that asks for a symbolic name for the target. This name is displayed in the Targets list box, and in the Target Selection control in the toolbar. The **Properties for Target** dialog is displayed when you define a new target or when you click the **Properties** button. See [Figure 4-5 on page 4-10](#).

To activate a target, click the **Select** button — or use the Target Selection drop-down on the pRISM+ Manager toolbar.

Properties for Target

The **Properties for Target** dialog is where you specify the attributes of the Target Communication Server and the type of connection to use for both pROBE and pMONT target agents.

Properties for Target mytarget

Server

Server Selection

☒ Use local communications server

☐ Use local BDM communications server

☐ Use remote communications server

Remote Registration

Remote Server Host Name: localhost

Remote Communications Server Executable:

pROBE Target Connection

☒ Network Network Name: mytarget

☐ Serial Port Number: 2

pMONT Target Connection

☒ Network Network Name: mytarget

☐ Serial Port Number: 2

OK Cancel Help

FIGURE 4-5 Target List — Properties Page

The **Server Selection** area of the dialog allows you to specify a local server, a remote server, or a BDM connection.

- Normally, you **Use a Local Communication Server** for both serial and network connections.

- A **Remote communications server** is used primarily when your target hardware has a serial connection to a machine other than your workstation. To use a remote Communication Server, you enter the host name of the remote machine and the path to the CommServ.exe on the remote machine.

pROBE and pMONT target agents can be configured independently. This allows you to do debugging with a network or serial connection while dynamic analysis can be done with the opposite.

- When selecting a network connection, you enter the IP address of the target. You can use either the number form (xxx.xxx.xxx.xxx) or the symbolic name form, if the name is resolvable by an available DNS server or appears in your local hosts file. You can also set the network port number.
- For serial targets, you specify the serial device name (for example, COM1) and the Baud Rate.

Setup Target

pRISM+ Manager provides a **Setup Target** dialog that downloads your executable code to the target and starts it running. You can independently specify any of three sequential operations.

1. Optionally download a file that you specify.
2. Boot the machine at the default or specified address.
3. You can optionally run the initialization of pSOS.

4.1.5 After Downloading the Application

After successfully downloading, you can use the **Halt** and **Go** buttons on the pRISM+ Manager toolbar. Once a program is downloaded and running, you can invoke debugging and analysis tools from the toolbar. SearchLight, ES_p, and Object Browser are available. You can also download your executable code through your debugger.

Target communications can be reset with the **Target** → **Reset** menu command. This causes the Communications server to disconnect and reset itself for a future session. This command must be used each time the target board is physically reset. The **Target** → **Reset** command does not affect the state of the target board.

To reconnect to a running target board, select **Target** → **Connect** to re-establish communications.

The pRISM+ Editor is a fast-start programming environment targeted specifically at firmware developers who are bringing up custom boards. Its makefile orientation and simplicity are ideal for working with multiple makefiles and switching between multiple BSPs. pRISM+ Editor focuses on working with existing makefiles and presenting the optimal Compile-Edit cycle in a familiar user interface.

pRISM+ Editor is composed of three major systems: Makefile Browser, Program Editor, and Message View. These three systems work together with pRISM+ Manager to form a comprehensive suite of embedded development tools. See [Figure 5-1 on page 5-2](#).

5.1 Makefile Browser

When a pRISMSpace is selected in pRISM+ Manager, pRISM+ Editor will load the associated makefile and restore any state from the previous working session. Restoring state loads additional makefiles, opens previously loaded files, and restores window locations.

The Makefile Browser reads the makefile, parses it and displays the file names found from the makefile. There are two views in the Makefile Browser: Makefile view and Source view. The Makefile View displays a dependency graph. The Source View displays a list of files referenced by the makefile.

The Makefile Browser's knowledge of dependencies comes only from the makefile. This means that only files referenced by the makefile are displayed in either view. When the makefile is modified and saved, the Makefile Browser re-parses the makefile and updates the Makefile Browser's views.

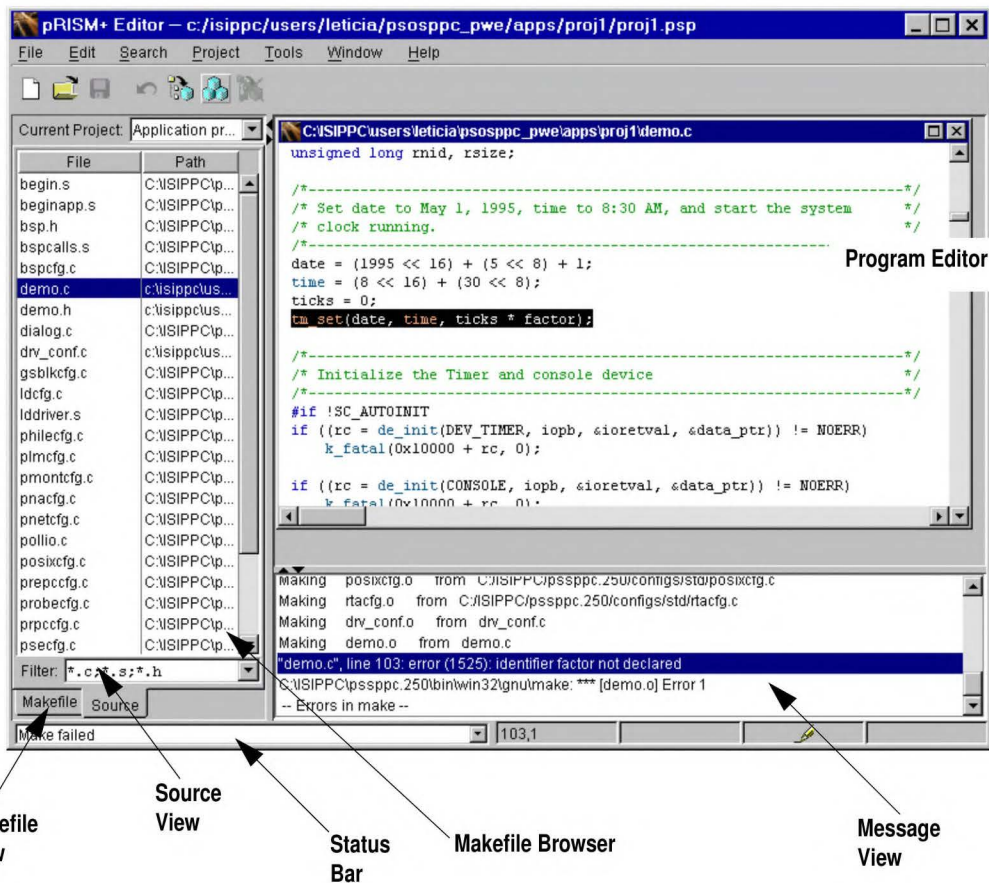


FIGURE 5-1 pRISM+ Editor Main Window

To add a new file to the Makefile Browser's view, you edit the makefile and then save it. This assumes that you have a knowledge of makefile rules and syntax.

The Makefile View's dependency graph is controlled by the currently selected Makefile Target. Changing the Current Target (such as `ram.elf`) setting in the **Project** → **Settings...** dialog will change the Makefile View's display.

The Source View's File List is controlled using the view's local (right-click) menu. You can display files from all makefiles, from the current project (makefile), or from just the current target (for example, `ram.elf`).

5.1.1 Makefile View

The Makefile View displays targets, sources and makefiles in a hierarchy. The Makefile tab at the bottom of the Makefile Browser's view selects the dependency hierarchy. Each of the target's dependencies will be displayed hierarchically in this view. The dependency hierarchy is displayed with include files underneath source files, which are in turn underneath object files.

The makefile is represented by the top node of each hierarchy. This node has a descriptive text field that is used for displaying the makefile node. You can edit the description using the **Settings...** dialog on the Makefile View's popup menu. You can load the makefile into a Program Editor by right-clicking in the Makefile View and selecting **Edit Makefile**. The popup menu also provides a context sensitive Make command; for example, performing a right-click on the `root.o` or `root.c` will cause the popup menu to have **Make root.o** as the first command on the popup menu.

Each makefile in the Makefile Browser has a **Current Target** setting. To access this setting right-click in the Makefile View and select **Settings...** You can change the **Current Target** to any of the top-level targets in the makefile. Top-level targets filter the file list. Top-level targets are defined to be targets that do not appear as dependent files of another target.

5.1.2 Source View

The Source tab, at the bottom of the Makefile Browser's view, displays a flat list of files derived from the makefile. The Source View can show sources from All Makefiles, from the Current Project, or the Current Target. Use the Source View's popup menu to select the set of files you want displayed.

Dependent files that match the **Filter** mask pattern are displayed. The **Filter** can be re-defined by editing the field at the bottom of the Source View. It can be set to a list of wild cards of the form: `*.c`, `*.s`, `*.h`. The Source View displays both the name and the path to the files.

5.1.3 Additional Makefiles

pRISM+ Editor has the ability to support multiple makefiles. Select **Project → Add Makefile...** to add makefile that you want to work within your project. pRISM+ Editor has special support for Board Support Packages (BSPs). pRISM+ Editor works in conjunction with pRISM+ Manager to determine which BSP is used during the build.

pRISM+ Manager has a BSP setting in the **pRISMSpace → Settings...** dialog. When this BSP value is changed, pRISM+ Manager notifies pRISM+ Editor of the change. Then,

pRISM+ Editor will use the new value for `PSS_BSP`, which causes any new builds to use the new BSP. In this way, you can switch between two different BSPs.

NOTE: You must completely rebuild your application whenever you change between BSPs. To rebuild the application, use **Project** → **Rebuild All** or Alt-F9.

Select **Project** → **Add BSP Makefile...** to add a BSP project. pRISM+ Editor displays a list of BSPs found in the `PSS_ROOT\bsp` directory. You can add your custom BSPs to that directory to easily switch between sample and custom versions. Alternatively, you can use pRISM+ Manager to add a BSP that resides in another directory. Use pRISM+ Manager's **pRISMspace** → **Settings...** dialog to browse to your BSP directory. BSPs added this way show up in pRISM+ Editor's **Add BSP Makefile...** dialog.

5.1.4 Current Project and Current Target

pRISM+ Editor starts up with a pRISMspace passed in by pRISM+ Manager. Each pRISMspace can contain multiple projects, where each project is defined to be a makefile. The **Current Project** is set using the drop-down list at the top of the Makefile Browser. This allows you to switch between different makefile projects in your pRISMspace. The **Project** → **Make Target** → **all** and **Project** → **Rebuild All** commands operate on the Current Project.

Each project can have unique Project Settings. Setting the **Current Project** and then selecting **Project** → **Settings** allows you to customize the **Project Description**, change the Build Command, and select the default Current Target. The **Project** → **Make current target** command operates on the Current Target. The Current Target specifies the default makefile target for makes and builds.

5.2 Program Editor

The Program Editor provides text editing capabilities commonly found in programmer's editors. It supports on-the-fly syntax highlighting, brace matching, regular expression searching and keystroke macros. For additional information, see Editor Commands in the on-line help. The editor supports opening multiple files into Program Editors in the Program Editor panel.

The Program Editor supports the notion of buffers. This allows you to work within one Program Editor (maximized perhaps) and switch between any files open in other windows. Select the **Edit** → **Buffer list...**, or Alt-B to open the buffer list. In the **Buffer List** window you can quickly access, save, and close files displayed in the Program Editor window.

You can arrange and manage the Program Editor windows by using the commands on the **Window** menu.

5.3 Message View

The Message View collects output from the builds. The text is filtered into the message view. The message view displays the errors can make or compile. Double-clicking on the error message opens the source file and displays the line of code where error occurred.

The Message View displays messages from any tools that are run during the course of a build. Messages which conform to a particular format are Trackable. This means that the compiler or other tool has emitted file and line number information and the Message View can display the source file and line that is referenced in the message.

5.4 Using the pRISM+ Editor

In the Getting Started chapter you used the pRISM+ Editor to create a project and compile your project application. In this section will discover other pRISM+ Editor features that will assist you in your project development.

You will learn how to:

- Create a new source file
- Save a new source file
- Rename a source file
- Add a new source file to your project
- Error check your project
- Include custom libraries

5.4.1 Creating New Source Files

1. From the pRISM+ Editor, select **File** → **New**.

An empty text window is displayed and available for you to use.

2. Start editing your file.

3. To explore other procedures in this section lets create a new file called `greeting`. Type the following in your new source file:

EXAMPLE 5-1: `greeting.c`

```
void greeting (void)
{
    printf("Howdy.\n");
}
```

4. Save your new source file such as `greeting.c`. See [Section 5.4.2, *Saving New Source Files*](#), for directions.

5.4.2 Saving New Source Files

1. From the pRISM+ Editor, select **File** → **Save**. For *Untitled* files the **Save file as** dialog is displayed.
2. In the **Save file as** dialog fill in the following fields:
 - a. Enter the name of the file in the **File name** field.
 - b. Select the location where you want the file to be saved. The default location is where your current opened project is stored.
3. In the **Save file as** dialog, click **Save**. This saves the new source file.

5.4.3 Copying an Existing Source Files

1. In pRISM+ Editor, click on the Program Editor you want to save in a new directory.
2. From the pRISM+ Editor, select **File** → **Save As**. The **Save file as** dialog is displayed.
3. In the **Save file as** dialog fill in the following fields:
 - a. Enter the name of the file in the **File name** field.
 - b. Select the location where you want the file to be saved. The default location is where your current opened project is stored.
4. In the **Save file as** dialog, click **Save**. This saves the new source file. You are now ready to modify the newly copied source file.

5.4.4 Adding Source Files to Your Project

Accessing the Makefile

1. From the pRISM+ Editor, select the Makefile View
2. In the Makefile view, right-click on any of the node of the makefile you want to edit. A popup menu is displayed. Select **Edit Makefile**.

Editing Makefile to include new source file

1. Scroll down to the end of the makefile. (See [Figure 5-2 on page 5-7](#).)
2. As the last entry point of the file, add the new source file name to the makefile. In this example you are going to add the file `greeting.c`. This file was created in [Section 5.4.1, Creating New Source Files on page 5-5](#).

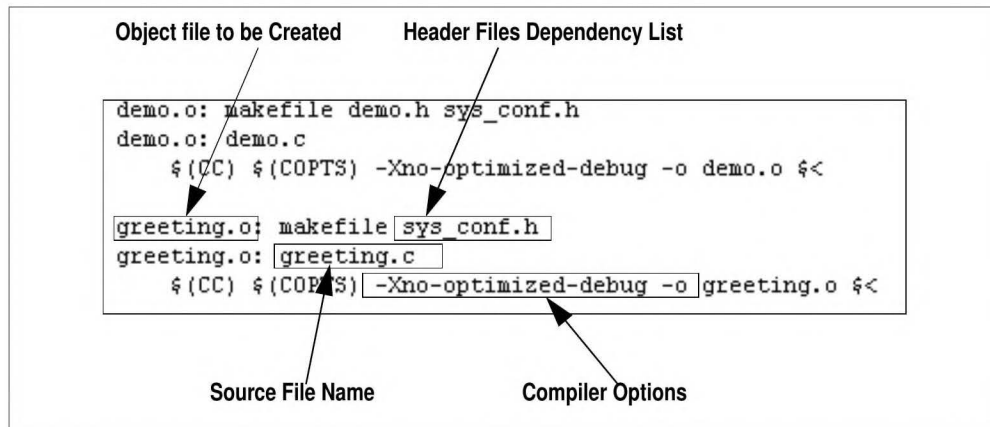


FIGURE 5-2 Example of Adding a Source File

3. For the new entry, include any Header Files Dependencies that the new file depends on.
4. For the new entry, include any compiler options.
5. For the new entry, include the name of the object file to be created.
6. Invoke the **Search → Find** dialog and type into the **Text to find** field
PSS_APOBJJS
7. In the **Find** dialog, select **Search backward**.

8. Click on the **Find** button.
9. In the `PSS_APOBJJS` line, add the name of the object you want created. This is the same name you defined in previous step.
10. From the pRISM+ Editor, select **File** → **Save**. The makefile is now saved and re-parsed.
11. Click on the Makefile tab. In the Makefile Browser the new source file will appear. See [Figure 5-3](#).

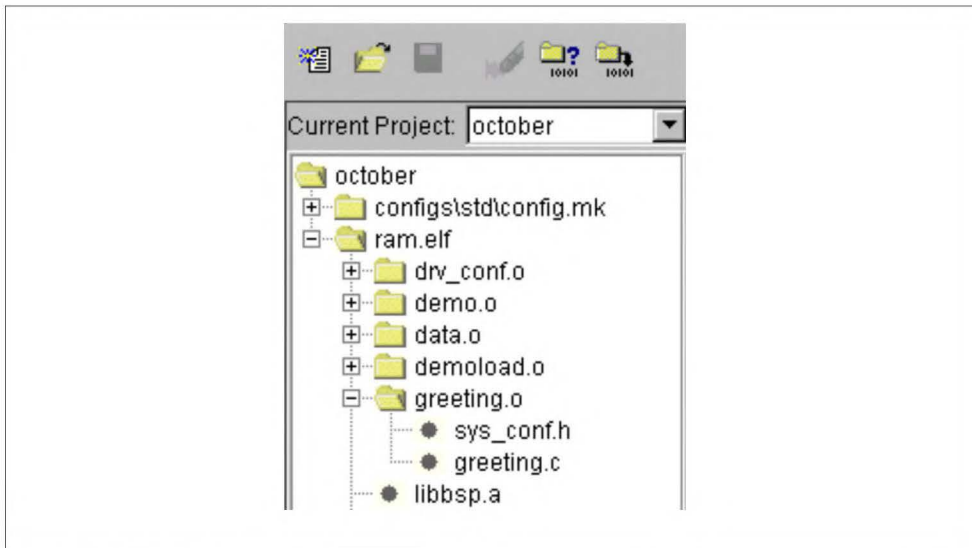


FIGURE 5-3 Makefile Browser

5.4.5 Error Checking Your Files

When you execute the make command, it reports any syntax errors in the Message View. In this section you will learn how to locate your errors using the Message View.

NOTE: In this example we are going to use `greeting.c` file. This file was created in [Section 5.4.1, *Creating New Source Files on page 5-5*](#).

1. In the Makefile Browser, double-click on the `greeting.c` file.

The `greeting.c` file will display in the Program Editor view.

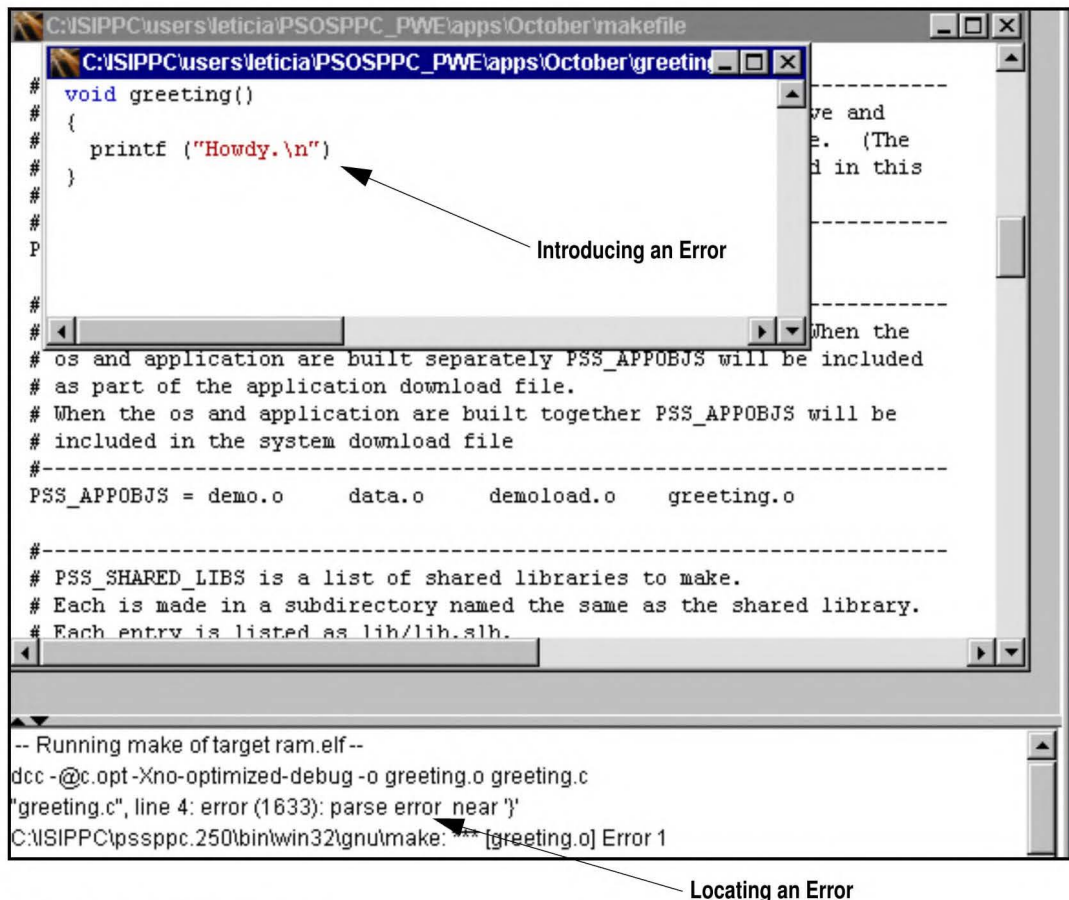


FIGURE 5-4 Locating an Error

5.4.6 Introducing an Error

1. In the `greeting.c` file, remove the semicolon (;) from the `printf` line. See [Figure 5-4](#).
2. From the pRISM+ Editor, select **Project** → **Make greeting.o**.
3. Observe the Message View for any errors. See [Figure 5-4](#).
4. The Message View will automatically track to the first error after the build is completed.

You can double-click on any error message to display the source at the reported error's line number. This will open the file with the error. It will also highlight the error.

In this instance you can add the semicolon (;) to the `printf` line to correct the error.

5.4.7 Profiling Your Project

Selecting **Tools** → **Profiler** invokes the optional Run-Time Analysis Tools (RTA). The profiler is invoked from pRISM+ Editor either from the **Tools** menu or by right-clicking on the target in the Makefile View. Before you can profile your project you must edit your makefile and the `sys_conf.h` file to include profiling compiler switches. For more information, see the sample application, RTADEMO. To learn more about RTA, refer to the *Visual Run-Time Analysis Tools User Guide*.

5.4.8 Accessing the Link Map Analyzer Tool

Selecting **Tools** → **Link Map Analyzer** invokes the Run-Time Analysis Tools (RTA). The Link option can be invoked from within the pRISM+ Editor. Before you can Analyze your project with the Link Map Analyzer Tool you must edit your makefile and the `sys_conf.h` file. To learn more about RTA, refer to the *Visual Run-Time Analysis Tools User Guide*.

5.4.9 Including Custom Libraries

Additional makefiles are generally used to add libraries to the project. For example if you have a sub-system that is built into a `.lib` file and then linked into your application, you can include the makefile that builds the `.lib` into the pRISMspace. This allows you to access the source files from the `.lib` subproject. In addition, BSPs are generally built by a separate makefile and pRISM+ Editor provide special support for this operation (see [Section 5.4.11, Adding a BSP Makefile on page 5-11](#)).

5.4.10 Adding a Makefile

1. From the pRISM+ Editor, select **Project** → **Add Makefile**. A browser is displayed.
2. Using the Browser, locate and select the makefile.
3. Click the **OK** button to include this file to your current project.
4. Click on the Makefile tab. In the Makefile Browser the new makefile will appear.

The **Add BSP makefile** menu item is a shortcut that allows you to select from the BSPs in the `PSS_ROOT/bsps` directory. Placing your custom BSP under this directory will cause it to be included in this list.

5.4.11 Adding a BSP Makefile

1. From the pRISM+ Editor, select **Project** → **Add BSP Makefile**. A BSP file list is displayed. The list shows all the bsp's in the bsp's directory.
2. Select the bsp makefile.
3. Click the **OK** button to include this file to your current project.
4. Click on the Makefile tab. In the Makefile Browser the new BSP makefile appears.

5.4.12 Removing a Makefile

1. Click on the Makefile tab.
2. In the Makefile Browser, select the makefile you want to remove and right-mouse click. A popup menu is displayed.
3. In the popup menu, select **Remove Makefile**.

NOTE: The **Remove Makefile** command is enabled only if the project (such as `pdemo`) already has an associated makefile.

5.4.13 Using the Buffer List

The buffer List allows you to manage open files during your pRISM+ Editor session. To access the buffer list complete the following steps:

1. From the pRISM+ Editor, select **Edit** → **Buffer List**, or Alt-B. The **Buffer List** window is displayed.

Accessing a file

1. In the **Buffer List** window, select the file you want to access.
2. Click the **Edit** button. This displays the file in the Program Editor view.

NOTE: Double-clicking on a file loads that file into the Program Editor.

Saving All Opened Files

1. In the **Buffer List** window, hold the Shift key down and select the all files in the list.
2. Click the **Save** button. This saves all the opened files.

Another way to save all opened files is to select **File** → **Save all**.

6

Using SNIFF+ in the pRISM+ Environment

6

This chapter explains more about SNIFF+, the optional pRISM+ project editor. This chapter consists of two parts.

- The first part ([Section 6.1](#) through [Section 6.6](#)) offers concepts and reference information on an application development framework section which is the result of integrating SNIFF+ with pSOSystem.
- The second part offers step-by-step instructions detailing how to use this application development framework from various common starting points.

For a complete description of the SNIFF+ functionality, refer to the SNIFF+ documentation located on the pRISM+ Documentation CD-ROM.

6.1 Overview

pRISM+ offers a range of powerful source code engineering tools collectively known as SNIFF+. The integration of SNIFF+ with pSOSystem provides users of pRISM+ with a powerful and versatile application development framework to develop pSOSystem-based applications. Some highlights of what this application development framework offers users of pRISM+ are as follows:

- pSOSystem code comprehension.
- Powerful source code browsers for the user's application code.
- Integrated Make support.
- Interface to configuration management tools.
- Support for team development.

- Support for mixed-platform development.
- Flexible application development framework.

6.2 Key Features of pRISM+ Application Development Framework

The pRISM+ application development framework is designed for today's team-based software development environment. It's application-centric and aimed at helping developers to enhance productivity by providing a wide range of powerful source code engineering tools that are seamlessly integrated with the pSOSystem code base. This framework can easily be extended and adapted to specific development environments and source code bases to optimize individual needs.

This sections summarizes the major features of this development framework. More details will be offered in subsequent sections.

6.2.1 Source Code Comprehension

Rapid source code comprehension is essential to software development productivity. Today's software developers need to understand legacy code bases, purchased source code software, and software developed by other team members. pRISM+ offers an extensive set of source code browsers for code comprehension. Since pRISM+ browsers can work on code that is not necessarily syntactically correct, users can begin with pRISM+ browsers from the very beginning, before the code is compiled.

In fact, Integrated Systems has applied the pRISM+ source browsers to the very pSOSystem code base you are using. Every pRISM+ is shipped with pre-parsed source projects so you can browse pSOSystem from the first day to understand exactly how it works and its interface to application code.

6.2.2 Team Development

pRISM+ offers real team development support for today's development environment without compromising the ease of use for single users. pRISM+ offers sophisticated support for code sharing amongst team members. The default pRISM+ configuration allows a team to share a common pSOSystem code base, which resides on a server machine while individual developers can build against the common code base from their individual workstations. This code sharing framework can be easily extended to a customer's code base as well. pRISM+ offers precise instructions on how to extend this framework and how to achieve a seamless level of integration of a customer's code with pSOSystem.

Furthermore, pRISM+ offers integration with most popular configuration management tools, such as ClearCase, PVCS and RCS, making pRISM+ a complete team-development solution.

6.2.3 Mixed-Platform Development

pRISM+ is designed to support mixed-platform development. In pRISM+ you can compile and debug on different host platforms. Many of today's development teams share common code repository on a server machine while team members compile remotely from individual workstations of a different type. pRISM+ is designed to support this development configuration.

6.2.4 Integrated Make Support

pRISM+ offers a powerful integrated make support system that consists of three parts:

- Support for code-sharing team-development.
- Support for makefile generation.
- Support for pSOSystem-specific make-requirements.

Functionally, pRISM+ make support is an integral part of the pRISM+ team development support and mixed-platform development support. pRISM+ make support allows multiple users to compile against a common code base across multiple platforms with ease, leaving the tools to handle the complexity of team-based builds.

With pRISM+ you can leave the complex task of managing makefiles for a team-based project to the tools. pRISM+ can automatically generate makefiles to support team development and mixed-platform development. These pRISM+ generated makefiles are flexible enough to be used from the GUI framework or at the command line. Of course, pRISM+ can also be easily configured for you to use existing makefiles.

In order to produce target executables, pRISM+ also provides easy-to-use utilities and concise instructions to help you integrate your applications with pSOSystem code. The hybrid-make model implemented in pRISM+ provides the best of both worlds — controlled and seamless integration with pSOSystem; flexibility and choice with application make.

6.2.5 Flexible Application Development Framework

pRISM+ Application Development Framework is designed to be an application-centric development environment that allows a maximum level of flexibility to adapt our tools to your environment and your application. It has a configurable and scalable design, making it equally relevant for a single user developing on one local machine and team members developing across multiple platforms.

pRISM+ also provides you with ample flexibility without losing the level of specific pSOSystem support. Specific attention is given to the pSOSystem-to-application interface to ensure that you can easily incorporate your work into the development framework. pRISM+ offers many utility programs and concise documentation to help you to adapt the development framework to your environment and your application code base.

The following sections describe the pRISM+ Application Development Framework, as well as how to adapt it to your environment.

6.3 Key SNIFF+ Concepts

To understand the integration of SNIFF+ with pSOSystem, you need to become familiar with some basic SNIFF+ concepts. This section offers a list of relevant concepts for the pSOSystem integration, together with brief descriptions of how these concepts are used by the integration.

Refer to the SNIFF+ documentation set for complete reference information on the SNIFF+ concepts discussed in this section.

6.3.1 Code Comprehension and Browsing

SNIFF+ provides the most advanced browsing and cross referencing capabilities to help you understand more code, more efficiently. Powerful filtering and visualization techniques work even with the biggest projects with many thousands of files, tens of thousands of symbols and millions of lines of code. No compilation is necessary to extract the symbolic information. With SNIFF+, you can browse code that has not yet been compiled.

6.3.2 Source Code Parsing

SNIFF+ uses an efficient C/C++ parser which analyzes C++, ANSI C, or Kernighan & Ritchie C source code. No compilation is necessary in order to extract symbolic information. The parser is highly configurable and can optionally preprocess the

source code. The symbolic information is kept continually on disk, so that parsing is done only once for each file and then parsed again only after a change.

If the source code of a project is edited, the symbolic information is updated immediately. Saved files are re-parsed and all browsing tools are updated. Therefore, you are always working with the newest symbol information that correctly mirrors the source code. Also, cross reference information is instantly updated.

6.3.3 Projects

In this section you will learn about a very important concept in the SNIFF+ project. A *project* is the main structuring element in SNIFF+ for grouping together files and directories that logically belong together in your file system. Once projects are created, you can then use SNIFF+ browsers to browse and understand the source code.

Project Directories and SNIFF+ Generated Files

Generally, you create a project from existing source files. When you create the project, you must specify the directory that will contain these source files. The directory you specify is the *project directory*. Each project in SNIFF+ corresponds to a project directory in your file system.

During project creation, SNIFF+ generates the following files and directories in a project directory:

- **Makefile:** This is the project makefile, generated when you choose to build your targets executables using SNIFF+ Make Support.
- **Project Description File (PDF):** Each SNIFF+ project is described by a Project Description File (PDF) that stores the structure, the list of files, and the attributes of the project. SNIFF+ maintains a project's PDF for you.
- **Project Generate Directory:** This directory contains a number of files generated for the project and maintained by SNIFF+. Its default name is `.sniffdir`.

Contents of a Project

Each SNIFF+ project contains the following:

- **Your source files:** You can include any type and number of source files in a project. For example, a typical SNIFF+ project might have C++ implementation and header files, yacc sources, documentation files, and files of a third-party documentation tools like FrameMaker.

- **A Makefile:** This is either your own makefile or SNIFF+'s makefile, depending on whether or not you use SNIFF+'s Make Support.
- **The Project Description File (PDF):** When you open a project, you are really telling SNIFF+ to load the project's PDF. When you modify a project's structure in any way (for example, by adding or removing files to the project), its PDF will be changed accordingly.

Subproject Structures

You can include other projects to create a hierarchical project structure. The process of including one project in another project is referred to in SNIFF+ as *adding a subproject*. The included project is known as a subproject.

Project Attributes

Each SNIFF+ project is described by a project description file (PDF). The PDF stores information such as the structure, the list of files, and the attributes of the project. A project's attributes would include file types added in the project, make parameters, parser options, and your choice of version control tools. These attributes are user-modifiable. Refer to the *SNIFF+ User Guide* for a complete list of project attributes.

Project Types

SNIFF+ distinguishes between two different project types: *shared* and *absolute*. The following table outlines the differences between these two project types:

Project Type	Default	Can project files be shared among Developers?	Project attributes refer to Extension files and subprojects using
Shared	*.shared	yes	path relative to a root directory
Absolute	*.proj	no	absolute path names

Shared Projects

Shared projects are for team development. Each team member has access to a shared project and can make changes to its files and structure. Shared projects are always used in conjunction with a configuration management and version control (CMVC) tool of your choice.

Shared projects offer a great deal of flexibility. Since all references to files and sub-projects are relative to a root directory, you can easily move a shared project to another location on a file system.

It is recommended that you work with shared projects even if you do not initially work in a team development environment, since most single-user development work is eventually incorporated into a team development environment sometime during a project's life. With shared projects, the transition from a single-user to a team environment is much smoother than with absolute projects.

Absolute Projects

Absolute projects are most suitable for browsing code. Setting up an absolute project is easy. If you need to get your source code into SNIFF+ for browsing only, it makes sense to use SNIFF+'s absolute project type. For development, however, it is recommend that you use shared projects.

Organizing Project Structures

Project structures in SNIFF+ do not need to map directly to file system structures. [Figure 6-1 on page 6-8](#) illustrates this idea, using a pSOSystem example.

In [Figure 6-1](#) and [Figure 6-2](#), you can see that although these directories are not subdirectories of `$PSS_ROOT/bsps/mbx8xx/src`:

- `$PSS_ROOT/bsps/devices/lan`
- `$PSS_ROOT/bsps/devices/mfp`
- `$PSS_ROOT/drivers`

these projects are subprojects of `bsps_src.shared`:

- `mfp_mbx8xx.shared`
- `lan_mbx8xx.shared`
- `drivers_mbx8xx.shared`

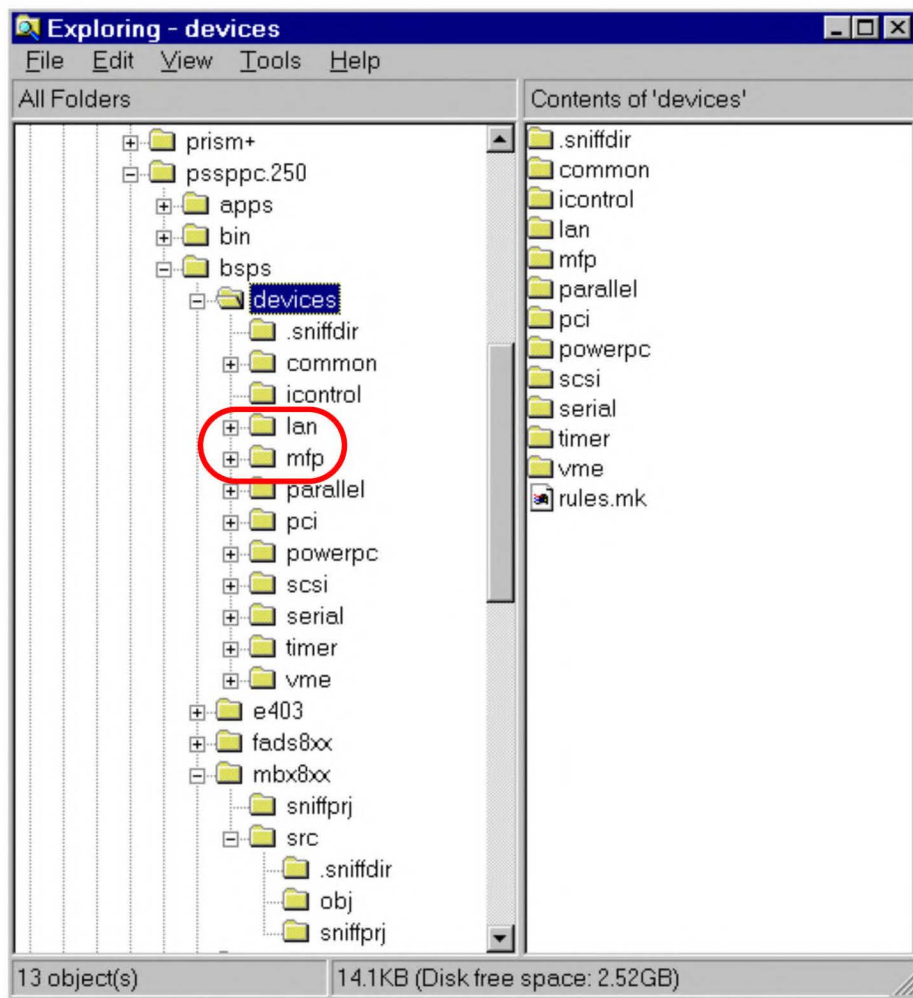


FIGURE 6-1 File System Structure (Partial View) of pSOSystem Code Base

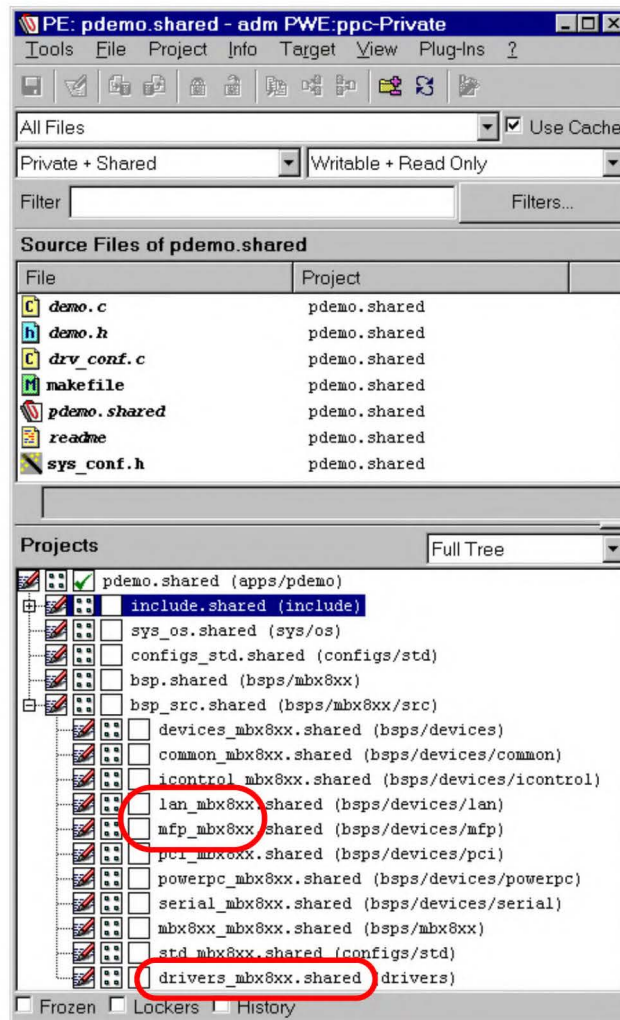


FIGURE 6-2 Project Structure

Tracking Dependencies in a Project

If you use SNIFF+ Make Support, SNIFF+ tracks dependencies among source files. As a result, you can add files to a project or remove files from a project without having to worry about which files need to be recompiled. Only source files that need recompiling are recompiled. Before each build, just tell SNIFF+ to update a project's dependency information to reflect your changes.

If you do not use SNIFF+ Make Support, you must update your own makefiles to reflect any changes in dependencies.

How to Create a SNIFF+ Project

[Section 6.4, Using the pRISM+ Application Development Framework on page 6-18](#) shows how pRISM+ can help you to create source projects and integrate your source project with pSOSystem.

You can also refer to the *SNIFF+ User's Guide* for a detailed description of how to use the SNIFF+ Project Setup Wizard to create source projects.

Choosing Which Project to Open

To work on a source project, you first must open it. Suppose you have a project structure similar to that shown in [Figure 6-2 on page 6-9](#). You have the following options:

- If you plan to modify and rebuild a single project — for example, any subproject of `pdemo.shared` — you can open only that project.
- If you plan to modify and then rebuild the entire application `pdemo.shared`, SNIFF+ will automatically open all of its subprojects. You can then work on `pdemo.shared` and all of its subprojects.
- If you plan to modify and then rebuild the Board Support Package (`bsp_src.shared`) SNIFF+ will automatically open all its subprojects. You can then work on `bsp_src.shared` and all the subprojects it includes.

6.3.4 Workspaces

Workspaces are the means by which SNIFF+ implements the solution for two important requirements:

- De-coupling the changes of a single developer from those of other team members.
- Sharing as much information as possible.

A *workspace* is a directory tree where complete projects or parts of complete projects reside. Workspaces can override each other; SNIFF+ provides a merged view of these workspaces.

SNIFF+ distinguishes between private workspaces and shared workspaces. A *private workspace* is the directory that belongs to only one user and is modified only by that

user. Every user has a private workspace and all the modifications to a project are done in the private workspace. A *shared workspace* is a directory that is accessible to any number of team members. There can be any number of shared workspace(s).

All private workspaces must have the same directory structure as the shared workspaces. Thus a private user makes a copy of a shared file or checks out a version of a shared file from the shared workspace. This private copy is stored in the private workspace which mimics the shared space in structure. During the rest file SNIFF+ will use the private copy to override the shared version of the file to reflect any changes made to the file.

For repository-based version tools, SNIFF+ also treats the repository as a workspace. Extensive discussion on workspaces is provided in the *SNIFF+ User's Guide*.

6.3.5 Working Environments

6

Working environments are physical directories on your file system in which SNIFF+ shared projects reside. In SNIFF+, you open shared projects by first specifying in which working environment you work in.

When workspaces are associated with a default version control configuration, they are referred to as Working Environments (WE). In this document, however, the terms workspace and working environment are used interchangeably.

You **must** use Working Environments if:

- You are a member of a development team that works on the same set of files, and you do not use a third-party configuration management tool that furnishes a workspace model of its own, such as ClearCase.
- You develop software for multiple platforms (as a member of a development team or alone).
- You work alone on projects and plan to share them in the future.

NOTE: pRISM+ uses the Working Environments concept to enable team-development out of the box. The following concepts are relevant only if you are using SNIFF+ outside of the pRISM+ Application Development Framework.

You **need not** use Working Environments if:

- You work alone on a project and do not need to share your project with others now or in the future.

- You already use a third-party configuration management tool such as ClearCase.
- You use SNIFF+ to browse source code only.

For team-based development, Working Environments enable:

- shared access to your team data Repository.
- shared and transparent access to team source code.
- shared access to platform-specific object code.
- individual team members to work in isolation from the rest of the team.
- individual team members to work on selected configurations of a team project.

Single users can also benefit from using the Working Environments for the following reasons:

- Working Environments are easily movable.
- Working Environments enable you to always know which projects you are working on.
- A Repository Working Environment allows you to maintain one directory for your data Repository and another for your workspace.
- A Working Environment can be used by single users for single-platform or multi-platform development.

Types of Working Environment

There are four types of working environments:

- Repository Working Environment (RWE).
- Shared Source Working Environment (SSWE).
- Shared Object Working Environment. (Not supported by pRISM+ Development Environment)
- Private Working Environment (PWE).

Make Support and Working Environments

SNIFF+ Make Support maintains information about dependencies and `include` directives across working environment boundaries, by supplying this information to

your make utility and compiler. Although this information can be maintained in your own makefiles, it is recommended that you use SNIFF+ Make Support when you are doing team-based development within SNIFF+.

Working Environment and Teams

Working Environments are designed to be used by teams. This section explains how Working Environments support team development. It also summarizes each working environment type and how the four types interact with each other.

Shared Access to Your Team Repository

Team members access and modify shared files using commands provided by your configuration management and version control (CMVC) tool. SNIFF+ provides an interface to your CMVC tools. This interface needs to know the location of your Repository.

You provide this information by defining a Repository Working Environment (RWE), which specifies the root directory of your Repository.

Shared and Transparent Access to Team Source Code

SNIFF+ requires you to specify the root directory of your team's shared source code. Once you have such a root directory, you must tell SNIFF+ where it is located. This is done by defining a Shared Source Working Environment (SSWE).

All team members can view or share the latest version of your software system as reflected by the source files in the SSWE. When browsing the source files, this view is read-only. When editing source files, team members work on private copies of the shared source files they want to modify. Team members never directly modify the shared source files in the SSWE. The view to all other source files (those not being modified) remains read-only.

Directories for Platform-Specific Object Code

The SNIFF+ Shared Object Working Environment is not used by pRISM+. Refer to the *SNIFF+ User Guide* for a complete description of this type of working environment.

Isolating Individual Work from the Team

Developers must be able to work in isolation from other team members. They need their own workspaces to edit, compile and debug projects without interfering with

the work of their team members. They also continually need to have access to their software system's most current source code and object code base.

SNIFF+ supports this type of work environment by allowing each team member to work in a private workspace. In SNIFF+, a Private Working Environment (PWE) is defined in order to specify the root directory of each team member's private workspace.

When working in your PWE, you have a read-only view of the shared source files located in your team's SSWE. When you need to modify shared source files, you check out the necessary files from your team's Repository. When you are satisfied that your changes are error free, you can check the modified files back into your team's Repository.

The next time your team's SSWE is updated, these changes are incorporated, and the shared source files in the SSWE once again reflect the most current state of your software system.

6.3.6 How File Sharing Works

SNIFF+ supports file sharing among Working Environments by requiring that all affected Working Environments have the same project directory structure. This is the easiest way for file sharing to work.

A SNIFF+ project's PDF stores structural information about the project such as the names of project files, their location relative to the project directory, and the names and locations of any subprojects. When all Working Environments that share files have the same project directory structure, SNIFF+ can easily find any project files or subprojects.

The project directory structure of the Shared Source Working Environment (SSWE) is the basis for all other working environment project directory structures. SNIFF+ automatically copies the SSWE's project directory structure into your private working environments when you open any shared projects from your private Working Environment. SNIFF+ copies only the SSWE directory structure, not the directory contents. [Figure 6-3 on page 6-15](#) illustrates the idea of equivalent project directory structures.

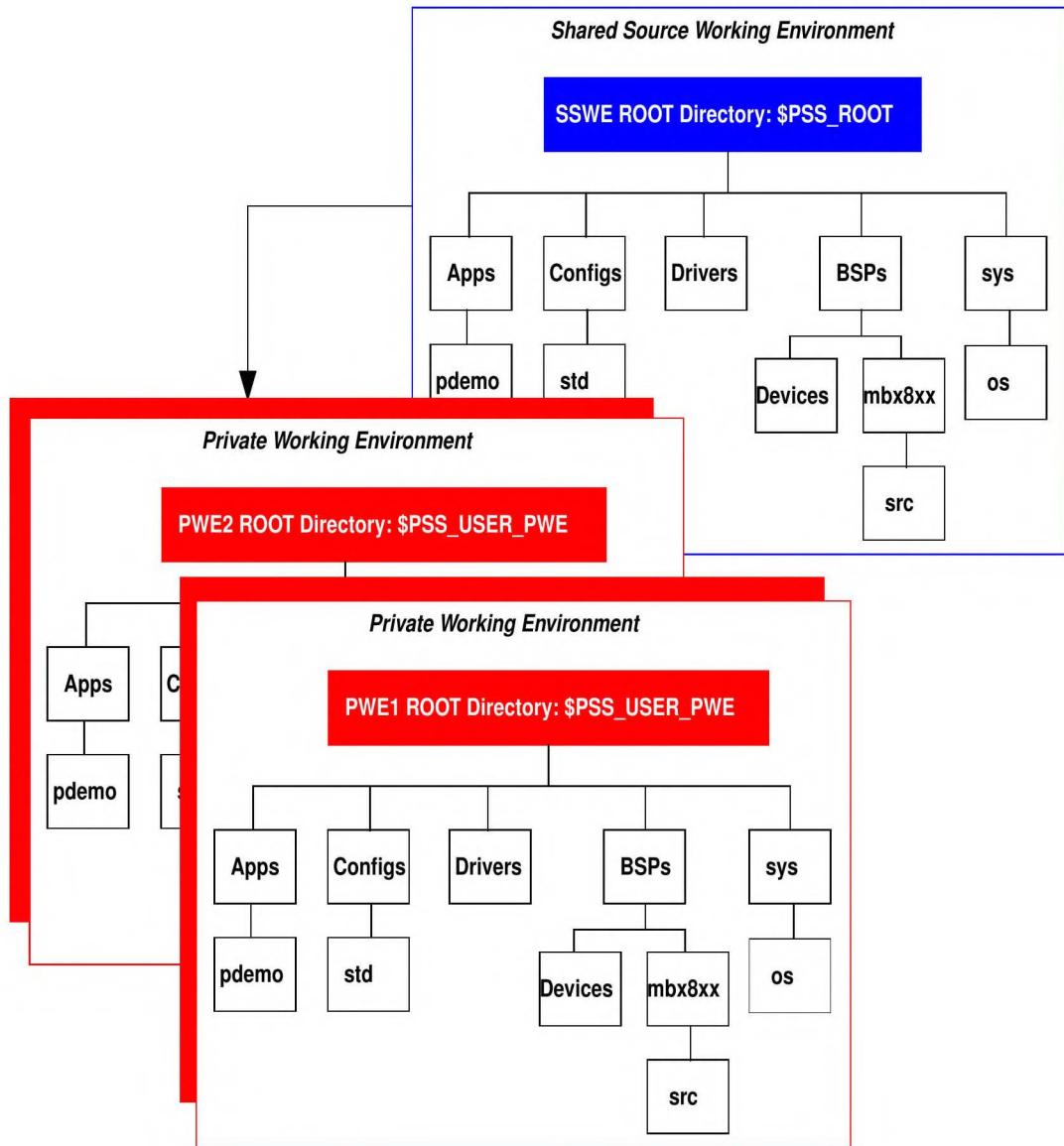


FIGURE 6-3 How File Sharing Works

The PWEs have the same project directory structure as the SSWE. The two team members working in PWE1 and PWE2, respectively, share the source files in the SSWE. When browsing source files, their view to the files is read-only. When editing source files, they work on local, writable copies of the source files they have checked out from the Repository. When compiling in their PWEs, object code is created locally from both shared (read-only) source files and local (writable) sources files.

A Closer Look at File Sharing

Let's look more closely at the SSWE, PWE1 and PWE2. For example, the `foo` project directory in the SSWE contains the following:

- The Project Description File `foo.shared`.
- The Project Makefile.
- The following source files: `foo.c`, `foo.h`, `bar.c`, and `bar.h`.

Figure 6-4 shows the contents of the `foo` project directory in the SSWE, PWE1 and PWE2. In this example, two developers (Joe Developer and Jane Developer) own the PWEs. Joe Developer owns and works in PWE1; Jane Developer owns and works in PWE2. Both Joe Developer and Jane Developer share common source files located in the SSWE.

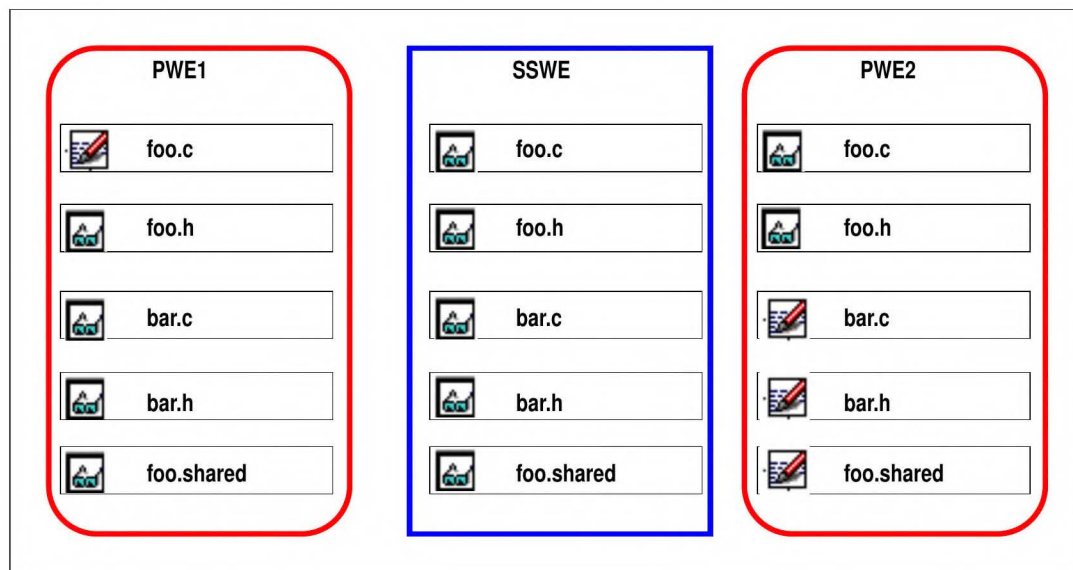


FIGURE 6-4 File Sharing

As [Figure 6-4](#) shows, Joe Developer has checked out only one file from the `foo` project directory in his PWE: `foo.c`. He has a read-only view to all other files. Jane Developer has checked out three files from the `foo` project directory into her PWE: `bar.c`, `bar.h`, and the Project Description File, `foo.shared`.

NOTE: To make structural changes to a SNIFF+ project, you must check out the project's Project Description File. Examples of structural changes include adding and removing project files and subprojects, and changing project attributes, such as the name of project targets.

[Figure 6-4](#) shows that Joe Developer has a read-only view to files checked out by Jane Developer, and Jane Developer has a read-only view to files checked out by Joe Developer. While Joe Developer is making changes to his local copy of `foo.c` in his PWE, Jane Developer can only browse the original copy of the file located in the SSWE.

This is an example of the exclusive file locking; when one team member has checked out a file in his PWE, all other team members can only browse this file. SNIFF+ configuration management and version control (CMVC) interface can provide other file-locking mechanisms as well. Your CMVC tool determines which mechanisms are available for use.

When Joe Developer builds `foo.o` from his private area, SNIFF+ ensures that his build will use the local copy of modified `foo.c`. SNIFF+ does this by having the local copy override the same file in the shared area for Joe Developer. SNIFF+ allows Joe Developer to use all other files in the shared area in order to complete his build.

Changes made to `foo.c` are local to Joe Developer and are not visible to Jane Developer. Similarly, Jane Developer can derive from the shared area her own copy of any of the files and make her modifications, eventually overriding the shared versions of the same files.

6.3.7 SNIFF+ Build and Make Support

SNIFF+ Make Support offers the following features:

- It comes with its own makefiles.
- It is based on standard UNIX Make Tools.
- It is fully integrated with Working Environments to build targets across multiple shared Working Environments.
- It automatically generates make support files that contain data about include paths, dependencies lists, object files lists, and `VPATH` information for shared projects.

- It automatically provides make rules for recursively building a project's target.
- It provides automatic support for multi-platform development and works with compilers, linkers, archivers, and other build tools of your choice.
- It maintains your build system by automatically updating make support files.

6.3.8 Building Targets When Using Team Working Environments

If you use SNIFF+ Working Environments for your team software development projects, you must use SNIFF+ Make Support in its entirety (including makefiles and make support files) for building your object files and targets.

SNIFF+ Make Support allows you to take full advantage of Working Environments by providing a mechanism for automatically sharing source and object files between members of a team. As a result, it is not possible to use any makefiles with shared Working Environment.

One major exception is the pSOSystem makefiles which have been extended to support team environments. This allows you to use the hybrid make model in a team development environment. The hybrid make model is described in [Section 6.6.8, Hybrid Make Model on page 6-46](#). For details about pSOSystem makefile extensions for team support, refer to [Appendix E](#).

6.4 Using the pRISM+ Application Development Framework

This section provides a detailed description of the pRISM+ Application Development framework. In this section you will see how the SNIFF+ concepts discussed in the previous section are applied in pRISM+

6.4.1 Team Development Support

The pRISM+ Application Development Framework is designed to address the needs of team-based embedded development projects based on pSOSystem. While the default configuration supports team development, single users can also reap the benefits of this set-up. This section describes the Team Development Support aspects of the pRISM+ Application Development Framework.

You are encouraged to refer to the SNIFF+ manuals for related concepts on team development. This section does not replace the SNIFF+ reference material on this subject. While all the SNIFF+ features are available to pRISM+ users, this document is produced to describe the use of these features within pRISM+.

6.4.2 pRISM+ Default Working Environments Settings

The pRISM+ Application Development Framework provides the following default Working Environments:

- *RWE:pSOSystem-Repository*
- *SSWE:pSOSystem-target*
- *SSWE:pSOSystem-target-User*
- *PWE:target-Private*

RWE:pSOSystem-Repository

This is the Repository Working Environment (RWE) for pSOSystem code base. A repository contains version-controlled files of a project. Close examination of the **Working Environment Root** field shows that this Working Environment is pointing to the location `$PSS_ROOT/repository`.

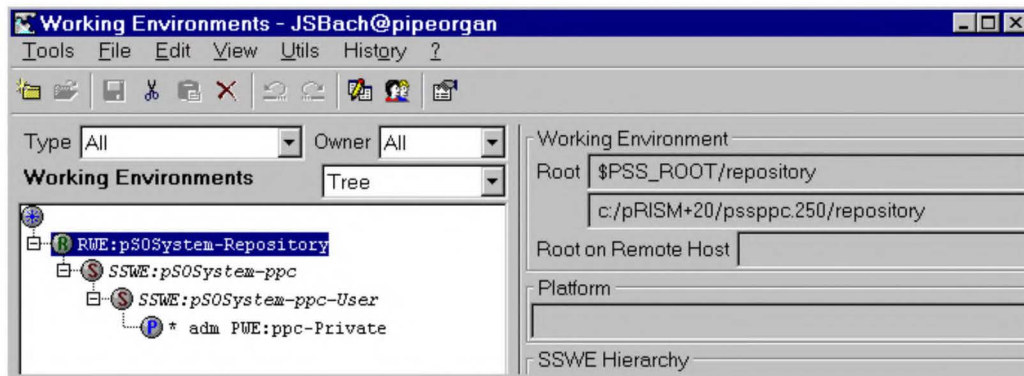


FIGURE 6-5 Repository Working Environment (RWE)

This **Working Environment Root** can be modified to point to any other directory where you keep source control information for your code base. If your project has an existing repository and you would like pSOSystem to be checked into your existing repository, then you should point the Repository **Working Environment Root** to the location of your repository.

Once you have set up your RWE root, you should check in all of your pSOSystem source files. For instructions on how to do this with various CMVC Tools, refer to the *SNIFF+ User's Guide* located on the pRISM+ Documentation CD-ROM.

SSWE:pSOSystem-target

This is the Shared Source Working Environment (SSWE), which contains the actual pSOSystem source-code base and pre-parsed pSOSystem source projects. Close examination of the **Working Environment Root** field shows that this Working Environment is pointing to `$PSS_ROOT`, the location of pSOSystem on your machine.

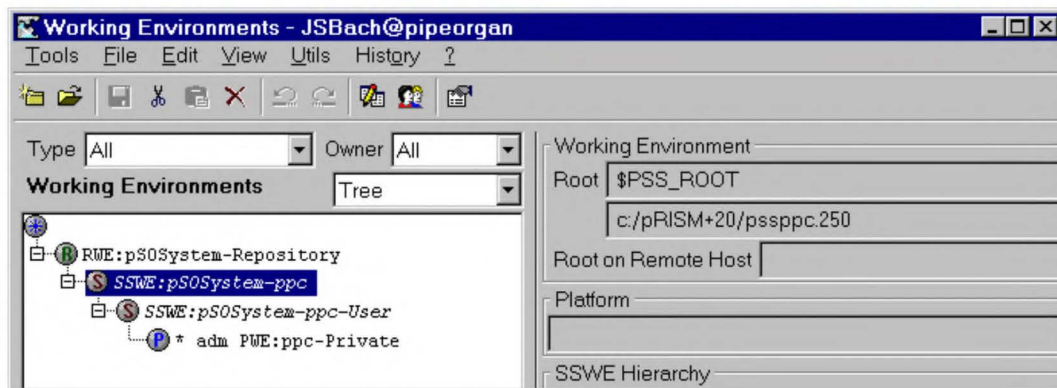


FIGURE 6-6 Shared Source Working Environment (SSWE)

This **Working Environment Root** can be modified to point to the actual location of the pSOSystem code base your development team will share.

You can easily modify this WE root by redefining the `$PSS_ROOT` environment variable in the start-up script in your pRISM+ installation directory.

- **On Windows hosts:** Modify the `envtarget.ksh` file.
- **On UNIX hosts:** Modify the `envvtarget.sh` or `envvtarget.csh` file.

Once you have pointed this SSWE to your team's shared version of pSOSystem, you are on your way to doing team development with a common pSOSystem with the team members.

NOTE: Figure 6-6 shows a PowerPC-specific version of the SSWE. In general, the SSWE is identified as `SSWE:pSOSystem-target`, where *target* can be any one of *ppc*, *68k*, *mips*, and so on, as appropriate for your particular target processor.

SSWE:pSOSystem-target-User

This Shared Source Working Environment (SSWE) is pointed to by a user-defined environment variable `$PSS_USER_SSWE`. This environment variable points to the root directory of any existing code base which you will integrate with pSOSystem.

You can redefine the `$PSS_USER_SSWE` environment variable in the start-up script in your pRISM+ installation directory.

- **On Windows hosts:** Modify the `envtarget.ksh` file.
- **On UNIX hosts:** Modify the `envvtarget.sh` or `envvtarget.csh` file.

You can also choose to copy your existing code base into the default location provided by `$PSS_USER_SSWE`.

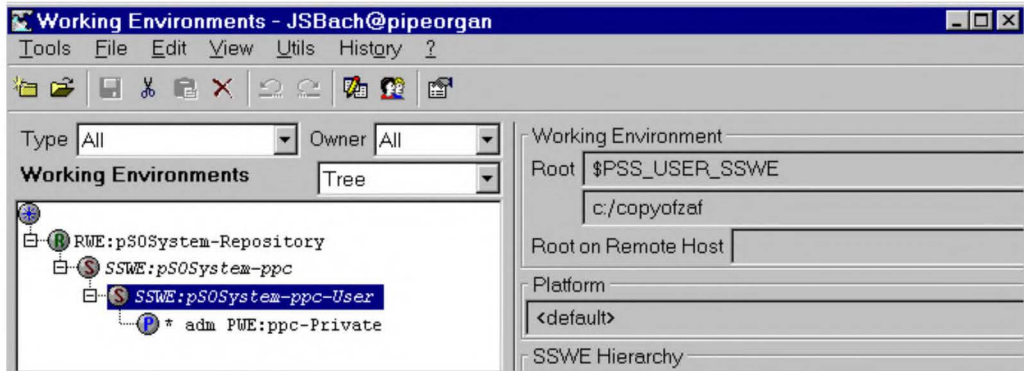


FIGURE 6-7 Shared Source Working Environment for Customer's Code

Once you define this SSWE to contain your existing code base, you can then create your source projects in this SSWE. This results in a source projects that a team members can share. This SSWE is derived out of the first SSWE which points to `$PSS_ROOT`.

This use of the SNIFF+ Working Environments allows you to easily integrate your code with pSOSystem code so you can browse them together. Code you do not plan to use with pSOSystem does not have to be located in a SSWE derived from the SSWE which points to the shared pSOSystem.

You can extend this concept further to more than one additional SSWE if your existing code base resides under more than one root directory.

NOTE: Figure 6-7 shows a PowerPC-specific version of the SSWE. In general, this SSWE is identified as `SSWE:pSOSystem-target-User`, where *target* can be any one of *ppc*, *68k*, *mips*, and so on, as appropriate for your particular target processor.

PWE:target-Private

This is the Private Working Environment (PWE) for a private user who is on the team sharing the common pSOSystem. By default the PWE points to your `$PSS_USER_PWE`, a subdirectory of your home directory.

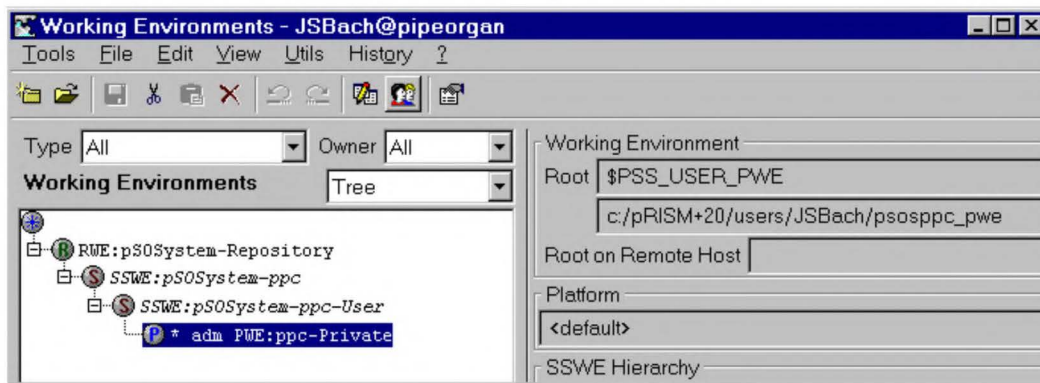


FIGURE 6-8 Private Working Environment (PWE)

NOTE: Figure 6-8 shows a PowerPC-specific version of the PWE. In general, this PWE is identified as `PWE:target-Private`, where *target* can be any one of *ppc*, *68k*, *mips*, and so on, as appropriate for your particular target processor.

You can redefine the `$PSS_USER_PWE` environment variable in the start-up script in your pRISM+ installation directory.

- On Windows hosts: Modify the `envtarget.ksh` file.
- On UNIX hosts: Modify the `envvtarget.sh` or `envvtarget.csh` file.

6.4.3 Restoring the Default Working Environment Settings

When you start SNIFF+, it sources a set of preference files to get its initial settings. To view the default preferences set by the pRISM+ installation program, use **Tools** → **Preferences** to display the SNIFF+ Preferences Window, as shown in Figure 6-9.

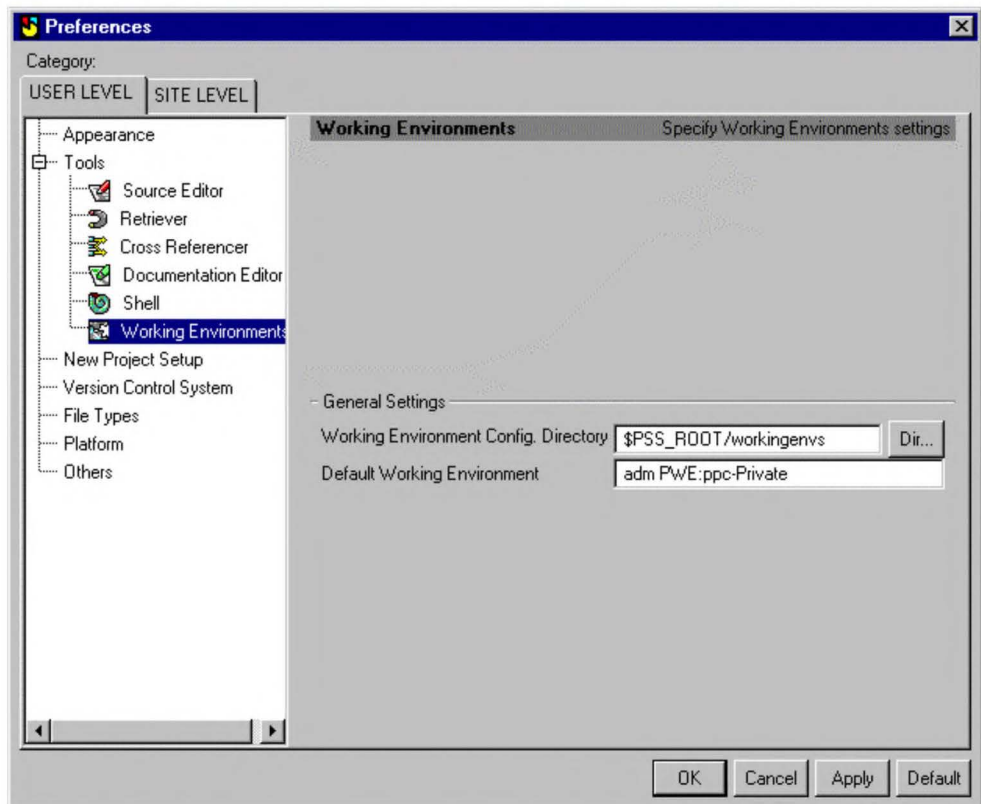


FIGURE 6-9 Default Working Environment Settings Location

Figure 6-9 shows that when you start SNIFF+, it looks to directory `$PSS_ROOT/workingenvs` for Working Environment settings, and that it uses **PWE:target-Private** as the default working environment to open projects.

NOTE: Figure 6-9 shows PowerPC-specific **General Settings**.

6.4.4 What Can You Do with pRISM+ Team Support?

Here let's use a typical set-up of a team to take a closer look at what pRISM+ team support can do for you. The example set-up is as follows:

- The team shares one common pSOSystem, located on a remote UNIX host named *muse*.
- The *muse* host contains the version control repository in directory *team_repository*.
- The team is comprised of two developers, Joe Developer and Jane Developer, who use PCs as their development stations.
- Joe Developer and Jane Developer both have third-party PC NFS software which allows them to access the UNIX file system.
- Joe Developer and Jane Developer have installed pRISM+ on each of their PCs and they are ready to start development of their project based on a pSOSystem sample application, *pdemo*.

Before they started, their SSWE administrator performed the following tasks:

- Created a copy of pSOSystem code on *muse* and checked all of the pSOSystem code into RCS, this team's version control tool of choice.
- Made sure that Joe Developer and Jane Developer have been able to mount *muse*'s file system as a local drive, as *e:\muse*, on their respective PCs.

For both Joe Developer and Jane Developer, pSOSystem is now located at *e:\muse\pSOSystem_share*.

- Edited *envtarget.ksh* in Joe Developer and Jane Developer's individual pRISM+ installation directory to make *\$PSS_ROOT* point to *e:\muse\pSOSystem_share*.

Now Joe Developer and Jane Developer are able to open *pdemo.shared* and see the read-only version of the shared files. They can both work on *pdemo.shared* by checking out files from the repository. When they make changes to a local copy of the shared files, this local file will override the shared file during a make.

When no more changes are needed, Joe Developer and Jane Developer can check their changes back into the repository. Their changes will be made visible to the team when their system administrator performs an update of their SSWE.

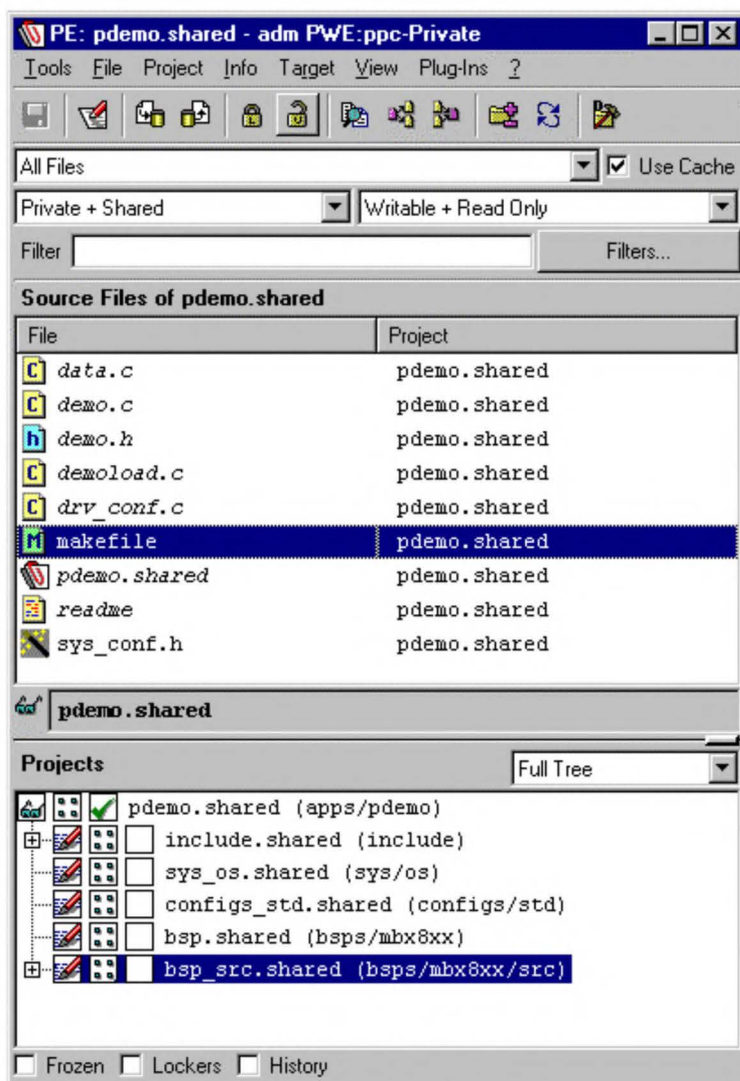


FIGURE 6-10 Private View of a Shared Project

6.5 pSOSystem Source Projects

The pRISM+ Application Development Framework is application-centric, designed specifically for pSOSystem-based application development. To develop with pSOSystem, you must first understand pSOSystem and its structure, its contents, and its interface to the application code. This critical understanding of pSOSystem-based code can be accomplished with the aid of the powerful SNIFF+ source code browsers.

This section describes the pre-parsed pSOSystem source projects that are shipped with pRISM+, which you can browse to gain the key understanding of pSOSystem.

This section focuses on one particular kind of source projects; pSOSystem sample applications. These sample applications are perfect for studying the pSOSystem-to-application interface since they are also designed as starting points for users to begin development with pSOSystem.

Refer to the SNIFF+ manuals for related concepts on source projects, how to create them and share them with team members. This section does not replace the SNIFF+ reference material on these subjects. While all the SNIFF+ features are available to pRISM+ users, this document describes the use of these features *within pRISM+*.

6.5.1 File and Directory View of a pSOSystem Sample Application

pSOSystem sample applications are designed to serve as perfect starting points for developing a pSOSystem application. Some very simple applications, such as `hello` and `pdemo`, can also be used by Board Support Package (BSP) developers to test the basic working condition of newly developed BSPs.

Each sample application is designed to illustrate one aspect of pSOSystem, but all of them have some things in common. Each application is made up of the following:

- sample application code
- `sys_conf.h`
- `drv_conf.c`
- `makefile`

Sample Application Code

Sample application code is the actual sample code that shows how to use certain pSOSystem features.

sys_conf.h

The `sys_conf.h` file is used to configure pSOSystem for your application. pSOSystem is a scalable operating system. In the `sys_conf.h` file, you can simply say YES or NO for each operating-system component to either include or exclude it from the application.

For operating-system components to be included in an application, you can also use the `sys_conf.h` file to configure them. This file includes many other configurable settings, such as boot mode and I/O devices to include. This file is key to configuring pSOSystem for your application.

drv_conf.c

The `drv_conf.c` file is used to configure and initialize pSOSystem drivers based on information entered in the `sys_conf.h` file. For each I/O device included by `sys_conf.h` file, a set-up routine is called in this file for the device.

makefile

The makefile associated with each sample application is set up for building the sample application. Each makefile is a precise definition of files from pSOSystem needed to make a target executable for this application.

By invoking the make command using the pSOSystem makefile, you can build a target execution image from the following:

- files of sample application code
- operating system configuration code and start-up code from the directory `$PSS_ROOT/config/STD`
- an object library known as the Board Support Package, `libbsp.a`, located in the `$PSS_BSP` directory.

BSP `libbsp.a` also contains high-level driver code located in `$PSS_ROOT/drivers` and device code located in directory `$PSS_ROOT/bsps/devices`
- an object library, `libsys.a`, which contains all the operating-system components in the `$PSS_ROOT/sys/os` directory
- other object libraries required by sample applications in directory `$PSS_ROOT/sys/libc`
- any other libraries an application might need

NOTE: The `$PSS_ROOT` points to the location of pSOSystem, and `$PSS_BSP` identifies one of the Board Support Packages in the `$PSS_ROOT/BSPS` directory.

6.5.2 pSOSystem Projects

In pRISM+, pSOSystem comes as a set of pre-parsed shared source projects. These source projects are provided so you can get a quick start without having to learn all about SNIFF+ right away. They are pre-parsed so they can be browsed immediately for code comprehension. Most importantly, the sample application projects can serve as starting points for development.

Other projects such as BSP projects and driver projects can be integrated with your code as subprojects in much the same way as they are used as subprojects for pSOSystem sample application projects.

This section looks closely at these pSOSystem source projects and how they are used.

Types of pSOSystem Projects

Pre-parsed pSOSystem projects can be categorized into the following groups:

General pSOSystem Projects

<code>include.shared</code>	Project for pSOSystem include files which are in <code>\$PSS_ROOT/include</code> directory and subdirectories.
<code>sys_os.shared</code>	Project for pSOSystem OS components which are in <code>\$PSS_ROOT/sys/os</code> directory.
<code>configs_std.shared</code>	Project for pSOSystem configuration files which are in <code>\$PSS_ROOT/configs/std</code> directory.

pSOSystem Libraries Source Projects

<code>sysclass.shared</code>	Project for C++ pSOS class library source files in <code>\$PSS_ROOT/sys/libc/src/sysclass</code> directory.
------------------------------	---

pSOSystem Drivers Project

NOTE: This list may vary depending on version of pRISM+.

`driver_name_drv.shared` Project for pSOSystem drivers in `$PSS_ROOT/drivers/` directory.

Board Support Package (BSP)

`bsp_src.shared` Project for individual pSOSystem BSP source in `$PSS_ROOT/bsps/<bsp_name>/src` directory.
Each `bsp_src.shared` project also includes all the devices projects and drivers project that are relevant for this BSP.

pSOSystem device code is located in `$PSS_ROOT/bsps/devices` directory.

`bsp.shared` Project for individual pSOSystem BSP in `$PSS_ROOT/bsps<bsp_name>` directory.

Sample Application Projects

`<app_name>.shared` Project for pSOSystem sample application in `$PSS_ROOT/apps/<app_name>` directory.

For a complete list of all the source projects that are available in pSOSystem, refer to [Appendix E](#).

Sample Application Projects

We have established that in order to build a target executable for a sample application, we also need many other parts of pSOSystem. In SNIFF+ terminology, the project from which the executable target is defined is the *super-project*. Other projects that are needed for building the executable target in the super-project are its *subprojects*.

In the case of a pSOSystem sample application project, it is the super-project. It includes things such as a Board Support Package and operating system components as subprojects.

A typical pSOSystem application is made up of the following:

- a sample application (super-project).
- `include.shared` (added as a subproject to the sample application super-project).
- `sys_os.shared` (added as a subproject to the sample application super-project).
- `configs_std.shared` (as a subproject to the sample application super-project).
- `bsp_src.shared` (subproject to the sample application super-project). Each `bsp_src.shared` also includes as its subprojects all the devices projects that are relevant for this BSP.
- `bsp.shared` (subproject to the sample application super-project).
- any other projects from the `$PSS_ROOT/drivers` directory, added as subproject(s), if referred by the application
- any other projects from the `$PSS_ROOT/sys/libc/src` directory, added as subproject(s), if referred by the application

Figure 6-11 on page 6-31 shows the `pdemo.shared` example used throughout this chapter. Note the project and subproject relationship that exists between `pdemo.shared` and its subprojects.

pSOSystem as Source Project

For your development, the pSOSystem sample application is analogous to the software you are developing. The Board Support Package is analogous to the drivers you are developing for your custom hardware. All the other pieces in pSOSystem such as the operating systems components and configuration code are additional supporting software for your application. They can be thought of as pre-made, ready-to-use supporting subprojects for your application project.

Converting Your Application to a pSOSystem Application Project

As we will show you in the tutorial, pRISMSpace Wizard can help you turn your existing code base into a shared source project. Once this source project is made, you can use the **Convert to pSOSystem Application** option to append pSOSystem subprojects to your project. Depending on the type of application you have, you may need to adjust the subproject list slightly, but the **Convert to pSOSystem app proj** option provides a quick way of adding most of the common code you need out of pSOSystem.

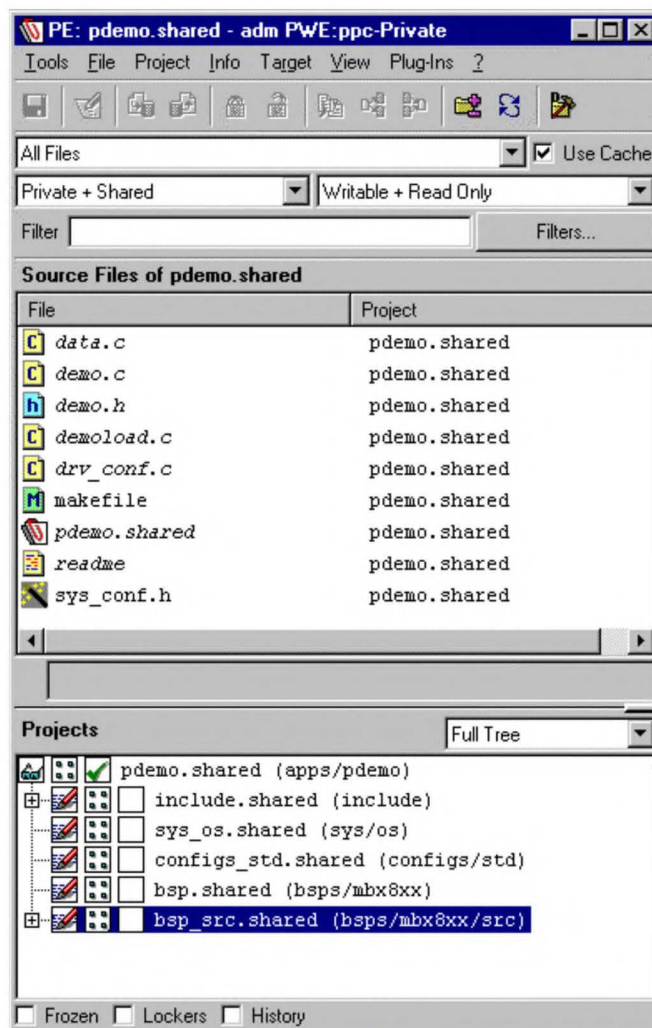


FIGURE 6-11 pSOSystem Sample Application Source Project Hierarchy

Figure 6-12 on page 6-32 shows an example of a source project made by pRISM+ out of an existing code base prior to using the option **Convert to pSOSystem App Proj**. The option **Convert to pSOSystem App Proj** is located on the SNIFF+ **Plug-Ins** menu.

In Figure 6-13 on page 6-33, you can see the results of choosing the option **Convert to pSOSystem App Proj** for your project.

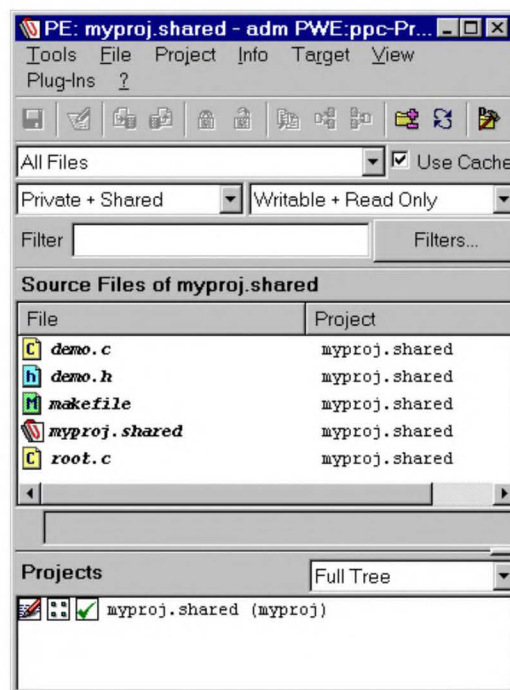


FIGURE 6-12 Source Project Before **Convert to pSOSystem app proj** is Performed

As you can see in [Figure 6-13](#), the conversion made a pSOSystem superproject `pss_main.shared` and added your code as a subproject. It also added other pSOSystem subprojects to the `pss_main.shared` superproject.

Depending on what kind of application you are developing, the default pSOSystem projects added by the conversion might not be sufficient. Refer to [Appendix E](#) for other source projects your application might also need.

pss_main.shared Project

`pss_main.shared` is the top-most pSOSystem super project which integrates your code base with pSOSystem code. `pss_main.shared` contains the set of three files that are essential to every pSOSystem application: `sys_conf.h`, `drv_conf.c` and a pSOSystem makefile.

The `sys_conf.h` and `drv_conf.c` files used in `pss_main.shared` are generic template files. They are sufficient for a simple application such as the `pdemo` sample application but they might not entirely fulfill your application requirements.

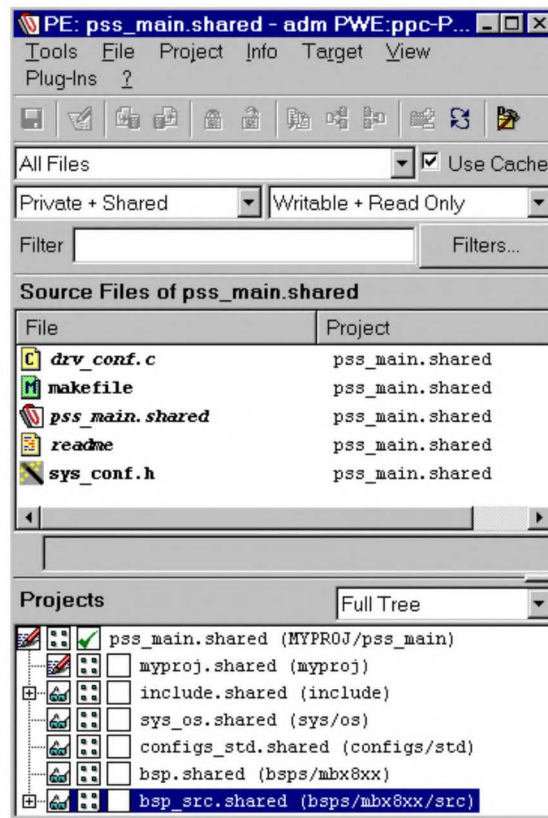


FIGURE 6-13 Source Project After Performing **Convert to pSOSystem app proj**

Compare the `sys_conf.h` and `drv_conf.c` files in a pSOSystem sample application that closely resembles the type of application you are developing with the template files. If there are differences, you can either import the changes needed by yourself or simply copy `sys_conf.h` and `drv_conf.c` from the pSOSystem sample application that most closely resembles the type of application you are developing.

If your application code already contains `sys_conf.h` and `drv_conf.c`, your working version should replace the template version.

To see the role of `pss_main.shared` plays in the build stage, refer to [Section 6.6.8 Hybrid Make Model](#) on page 6-46.

6.5.3 Browse View Versus Build View of pSOSystem Source Projects

pSOSystem is make-centric. Each pSOSystem sample application is defined by the makefile used to build that application. Each pSOSystem application is defined by a set of makefiles, each requiring a different set of files from the same pSOSystem directory structure. Depending on the kind of application, the makefile explicitly includes other makefiles from other parts of pSOSystem to pull in all the other files necessary to build the application.

In order to present an accurate browse view for pRISM+ users, each pSOSystem sample application project is specially constructed based on a unique file list as defined by each sample application's makefile. There are, however, several exceptions where the browse view contains more files than what's actually used to make a target.

The following table shows the level of accuracy of the "browse view" of pSOSystem sample applications projects compared to the "build view" of the same projects as defined by pSOSystem makefiles.

Project Name	Browse View Accuracy with Build View
Sample application super project	100% reflection of build
include.shared	Contains all the include files in \$PSS_ROOT/include
sys_os.shared	100% reflection of build
configs_std.shared	Contains all the start modules; only one is needed per target
bsp_src.shared	100% reflection of build
bsp.shared	100% reflection of build
Drivers projects	100% reflection of build
Library projects	100% reflection of build

The slight deviation in the file list does not affect the building of an executable because the pSOSystem makefiles ultimately decide what files are included in the build.

The slight deviation in file lists does affect accuracy in browsing. You can make adjustments to project file list simply by adding or removing files, or subprojects, from the projects. For example, the `beginapp.s` can be removed from the

`configs_std.shared` if you are not going to build the `app.elf` target, and consequently will not need to browse `beginapp.s` with the application.

Refer to the pSOSystem makefiles for a complete file list for each target. Use this as your guide to adjusting the file list for browsing.

6.5.4 Browsing pSOSystem

Browsing pSOSystem with SNIFF+

Refer to the SNIFF+ User's Guide for instruction on how to use SNIFF+ browsers.

Browsing pSOSystem with Preprocessing Enabled

pSOSystem code makes heavy use of preprocessing macros. Refer to the SNIFF+ User's Guide on how to enable preprocessing for browsing.

6.5.5 Utilities Programs

pSOSystem source projects were created using some utility programs in the form of Bourne shell scripts. These scripts are included in pRISM+ so you can use them to create source projects for your existing code base. These scripts are located in `$PSS_ROOT/bin/source/plugins/scripts` directory. Functional descriptions of these scripts are included in the script source.

It is recommended that you follow the steps in [Using the pRISM+ Application Development Framework with SNIFF+ on page 6-50](#) to create and work with project until you are familiar with SNIFF+.

6.6 pRISM+ Make Support

The pRISM+ Application Development Framework offers comprehensive make support which is pSOSystem-centric yet flexible enough to be extended to your environment. You can use the supplied pSOSystem makefiles or use SNIFF+ makefile generation feature to automatically generate makefiles for your code base.

pRISM+ make support is also scalable, designed to address the need of single developers as well as team developers. This section describes the make support provided by the pRISM+ Application Development Framework.

6.6.1 pRISM+ Make Options at a Glance

pRISM+ offers many make options ranging from simple to very advanced. These make options are summarized below. Extensive details will be offered in subsequent sections.

- Build your application using pSOSystem makefiles
- Build your application using SNIFF+ Make Support
- Build your application using a combination of pSOSystem makefiles and SNIFF+ Make Support – the Hybrid Make Model
- Using your own make and makefile
- Building from the command line

6.6.2 pSOSystem Application Make Structure

This section describes pSOSystem makefiles structure. pSOSystem is supplied with makefiles for building sample applications, BSP libraries, OS libraries and other libraries that come in source form with pSOSystem. These makefiles can be used with or without SNIFF+. When used with SNIFF+, pSOSystem makefiles provide overriding of SNIFF+ Workspaces.

NOTE: This document briefly explains SNIFF+ workspaces and general concepts.

For detailed description refer to the *SNIFF+ User's Guide and Reference*

This section is a reference for anybody modifying, using, and writing makefiles for pSOSystem. pSOSystem makefiles can be divided into three categories:

- Sample application makefiles.
- BSP makefiles.
- Makefiles to build system libraries and other libraries.

Sample Application Makefiles

Every sample application comes with a makefile to build the application targets. This makefile ties the application to the rest of pSOSystem. It serves as a definition of files that are needed for pSOSystem to build a target executable. If you want to expand pSOSystem makefiles for your project, you should begin with this makefile.

Each sample application makefile or application makefile imports common definitions and rules from the `config.mk` file in `$PSS_ROOT/configs/std` directory.

This makefile is included by every application makefile. Each sample application makefile also includes `bsp.mk` file from the BSP directory. Each sample application makefile might also include one or more `drivers/<drv_name>/rules.mk` file if the application uses driver `drv_name`.

If the application is built in the SNIFF+ environment, a sample application makefile also includes the file `configs/std/$(SNIFF_MAKE_CMD).mk`. This file implements workspace overriding for pSOSystem applications in case of SNIFF+. In a non-SNIFF+ environment, inclusion of this file has no effect.

NOTE: `SNIFF_MAKE_CMD` is defined to `pss_gnu`. By default, the `include` statement to add this file is commented out in the makefile.

The following sections are brief summaries of the makefiles included by the sample application makefile.

\$PSS_ROOT/configs/std/config.mk

This makefile contains common compiler defines and options, rules for making configuration file objects (`psoscfg.o`, `pnacfg.o` etc.). It also includes rules for all the common application targets such as `ram.elf`, `ram.hex` etc. This makefile is included by every sample application makefile.

\$PSS_ROOT/bSPs/<bsp_name>/bsp.mk

This makefile contains board specific defines and targets (for example, `DFP=H`). This is included by every application makefile. It is also included by the BSP makefile.

\$PSS_ROOT/drivers/<drv_name>/rules.mk

This makefile contains rule for making `<drv_name>` driver (for example, `PPP`). It is included by an application makefile if the application needs the `<drv_name>` driver.

BSP Makefiles

Every Board Support Package comes with a makefile to build an object library. This makefile normally resides in `$PSS_ROOT/bSPs/<bsp_name>/src` directory. Each BSP makefile provides a definition of all other files that are needed out of pSOSystem in order to build a BSP library. To expand the pSOSystem BSP makefile for your custom board support package, you should begin with this makefile.

Each BSP makefile includes `$PSS_ROOT/bSPs/<bsp_name>/bsp.mk` file to get the BSP specific defines.

It also includes `$PSS_ROOT/drivers/rules.mk`, `$PSS_ROOT/bsps/devices/rules.mk` and `$PSS_ROOT/bsps/devices/target/rules.mk`. These `rules.mk` files contain rules for making objects from the source files in the respective directories.

In a SNIFF+ environment this makefile also includes the `$(SNIFF_MAKE_CMD).mk` file.

The following are brief summaries of the makefiles included by BSP makefiles.

`$PSS_ROOT/drivers/rules.mk`

Contains rules to make the high level drivers from the `drivers` directory. It is included by BSP makefiles using drivers from this directory.

`$PSS_ROOT/bsps/devices/rules.mk`

Contains rules for making low level device drivers which come from `$PSS_ROOT/bsps/devices/<device_name>` directory. It is included by every BSP makefile.

`$PSS_ROOT/bsps/devices/target/rules.mk`

Contains rules for making target-specific files from the `$PSS_ROOT/bsps/devices/target` directory. It is included in every BSP directory.

Makefiles to Build System Libraries and Other Libraries

`$PSS_ROOT/sys/os` directory contains makefiles to build the system libraries `libsys.a`.

Putting It All Together

To generate a target executable, execute the `make` command on the project makefile in the sample application directory. This makefile calls `config.mk`, `bsp.mk` and `rules.mk` to compile the operating systems configuration code, BSP configuration code, and any high-level driver code this application needs.

The object files generated are then linked with a BSP library (determined by the environment variable `$PSS_BSP`) and the OS library to generate a target executable, such as `ram.elf`.

The BSP libraries and OS libraries are built during installation. By default, they are not recompiled with each application build. These libraries need to be recompiled only if you have made modifications to files in any of the `$PSS_ROOT/bsps` directories or the `$PSS_ROOT/sys` directory.

6.6.3 Make Attributes of pSOSystem Source Projects

In the previous section the pSOSystem make structure was described. This section examines how pSOSystem make is integrated with SNIFF+. This will be done by examining the make attributes of pSOSystem projects. Using the `pdemo` sample application, you will examine the various aspects of Make Attributes. Double-clicking on a project name in the PE window displays the project's attribute sheets. Use the *SNIFF+ Reference Guide* for descriptions of all make attributes. This section only explains parameters relevant to pSOSystem integration.

Build Options

Figure 6-14 on page 6-40 shows the **Build Options** category in the **Attributes** dialog box for `pdemo.shared`.

- **Use SNIFF+ Make Support:** This box is checked because you will be using the SNIFF+ Make Support system to generate the macros to support team development. This is true even when you are using pSOSystem makefiles.
- **Make Command: psosmake SNIFF_MAKE**
 - **psosmake:** The actual make command used on the command line.
 - **SNIFF_MAKE:** This macro is used to turn on the options in the pSOSystem makefile to enable the file overriding feature. You should always use it when compiling from within the SNIFF+ environment using pSOSystem makefiles.

If you follow the procedures in [Section 6.7, Using the pRISM+ Application Development Framework with SNIFF+ on page 6-50](#), the pRISMspace Wizard ensures that you use the correct make command based on your starting mode. You do not need to modify project make attributes when you follow the procedures given in [Section 6.7](#).

[Table 6-1](#) contains a list of make commands used by various pRISM+ make models on Windows and UNIX hosts.

TABLE 6-1 Make Command

UNIX Command	Windows Command	Descriptions
<code>psosmake</code>	<code>psosmake</code>	Used by command-line make.
<code>psosmake SNIFF_MAKE</code>	<code>psosmake SNIFF_MAKE</code>	Used with SNIFF+ when compiling with pSOSystem makefiles
<code>psosmake SNIFF_MAKE</code>	<code>psosmake SNIFF_MAKE</code>	Used with SNIFF+ when compiling with SNIFF+ generated makefiles

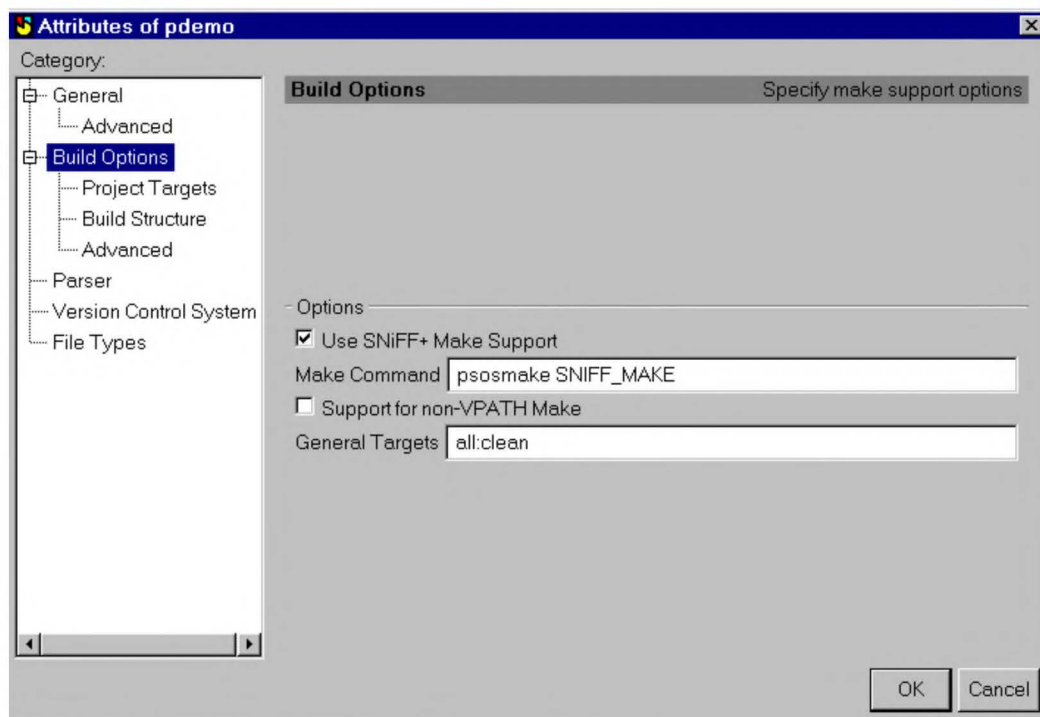


FIGURE 6-14 Build Options Category in Attributes Dialog Box

Project Targets

Figure 6-15 on page 6-41 shows the **Build Options** → **Project Targets** category in the **Attributes** dialog box for `pdemo.shared`.

- **Executable:** Here you can see that, for the project `pdemo.shared`, the default target is `ram.elf`.
- **Other:** This field shows all the other targets that can be made from this project.

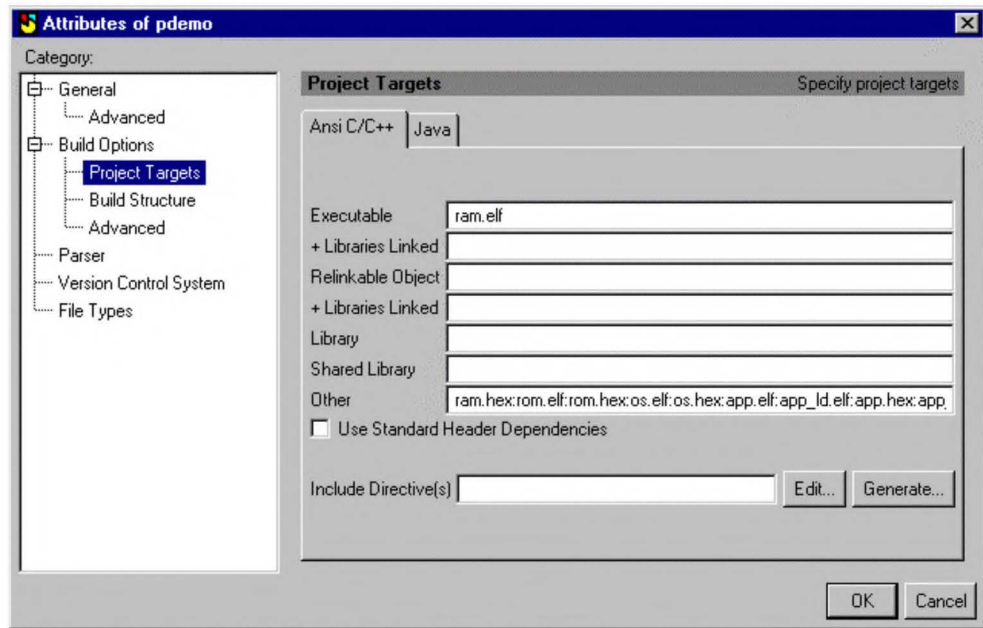


FIGURE 6-15 Build Options → Project Targets Category in the Attributes Dialog Box

Advanced Options

Figure 6-16 on page 6-42 shows the Build Options → Advanced category in the Attributes dialog box for `pdemo.shared`.

- **Use Generated Files Directory:** Location of the generated make support files. By default, the generated make support files are located in the directory specified in the **General File Directory** field of the Advanced Options view.
- ***.incl:** These files contain SNIFF+ generated macros for this project. pSOSystem makefiles use `vpath.incl` to support team development. These files normally reside in the location indicated by the **Use Generated Files Directory**.

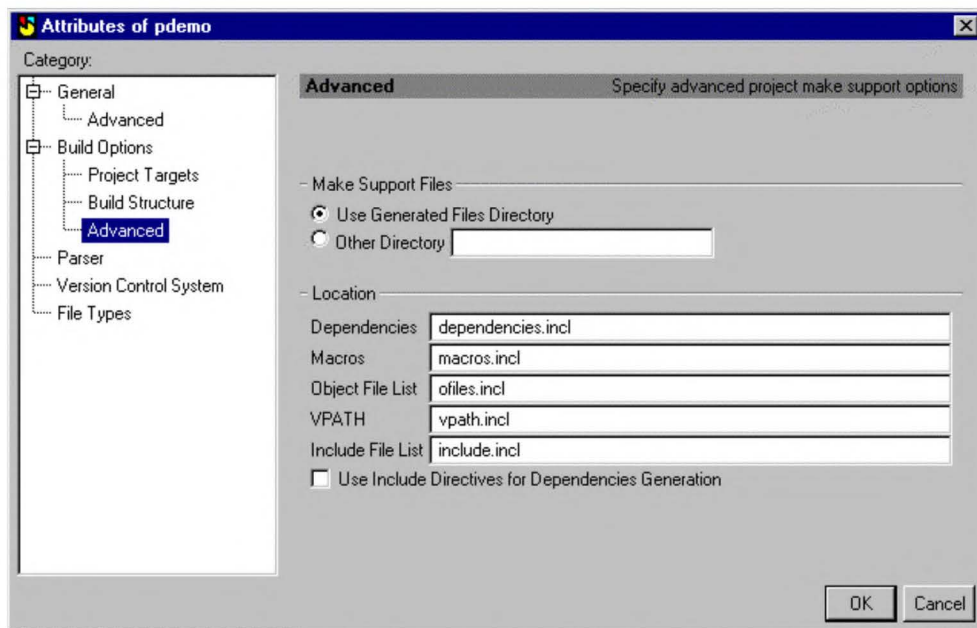


FIGURE 6-16 Build Options → Advanced Category in the Attributes Dialog Box

6.6.4 Making a pSOSystem Target Executable

Using the `pdemo.shared` example, to make the `ram.elf` target, select **Target → Make → ram.elf** in the PE window.

6.6.5 Using pSOSystem Makefiles

pSOSystem makefiles are the default makefiles used by pRISM+ and integrated into the SNIFF+ Make Support structure.

pSOSystem makefiles represent the way pSOSystem is built and tested. All the tests done on pSOSystem are based on builds done with these makefiles. For these reason you should not fundamentally alter the structure of these makefiles or attempt to regenerate these makefiles with SNIFF+.

pSOSystem makefiles have been extended for the integration with SNIFF+. Although pSOSystem makefiles implement workspace overriding when used with SNIFF+ Working Environments, these makefiles themselves do not have SNIFF+ awareness. For example, when you start your development based on a pSOSystem sample

application, you will start by default start with a pSOSystem makefile. When a new file is added to this project, the pSOSystem makefiles are not automatically updated with the new file information. The makefiles should be updated for the change to take effect.

pSOSystem was not compiled using SNIFF+ generated makefiles for many reasons. Each pSOSystem application defines multiple targets, for execution in RAM, in ROM, in `.elf` format or `.hex` format, etc. Each of these targets is built using a different set of files out of the same pSOSystem source tree. pSOSystem makefiles provide the mapping for what is needed for each target. These targets require specific ordering of object files at link time. Many of the pSOSystem files require specific compiler flags on a per-file basis. These special make requirements makes it impractical to use SNIFF+ generated makefiles to compile pSOSystem code because too many projects would have to be made, specifically, one separate for every target. This is also the reason why Integrated Systems discourages you from regenerating makefiles with SNIFF+ to compile pSOSystem sample applications.

6.6.6 Using the SNIFF+ Makefile-Generation Feature

SNIFF+ provides automatic make support for multi-platform development that can be configured to work with any compilers, linkers, archivers, and other build tools of your choice. Once a project source tree is constructed using SNIFF+ Project Editor, makefiles can be automatically generated for this project. When additional files are added to the project source tree, the generated makefiles are automatically updated to reflect the changes. The SNIFF+ automatic makefile generation feature is tightly integrated with the project management aspects of SNIFF+, namely the Workspace and Working Environment concepts. Together they allow a team of engineers to share and compile against a common code base between them.

pRISM+ supports and extends this SNIFF+ feature with some additional pSOSystem specific make support files as well as a mechanism to allow SNIFF+ “made” modules to be incorporated back into a pSOSystem build in order to produce a target executable. Together with modification to pSOSystem makefiles, pRISM+ offers a powerful solution for team-based development project based on pSOSystem.

Refer to the *SNIFF+ User Guide* for detailed information about the SNIFF+ Make Support system. This section documents only the integration of SNIFF+ with pSOSystem.

SNIFF+'s Makefiles and Make Support Files

Refer to the *SNIFF+ User Guide* for information about SNIFF+'s makefiles and Make Support files.

pRISM+ Specific Makefiles

In addition to the standard SNIFF+ Makefiles and Make Support Files, there are several additional files to support the use of the automatic makefile generation feature for pSOSystem-based applications. They are as follows:

- `diab_target_${HOST}.mk`: (Located in `$SNIFF_DIR/make_support` directory) This is an additional pRISM+ platform makefile which integrates SNIFF+'s make system with pRISM+ embedded platform.
- `general.mib.mk`: (Located in `$SNIFF_DIR/make_support` directory) This is an additional language makefile to support the mib file type.

pRISM+ Platform Makefile

In addition to SNIFF+ Makefile and Make Support files, also located in `$SNIFF_DIR/make_support` is a platform makefile that supports the use of the SNIFF+ automatic makefile generation feature for pSOSystem-based applications. Each platform makefile is unique for a pRISM+ for a specific processor family.

This platform makefile is included by the SNIFF+ general makefile `general.mk`. All the pRISM+ specific make options are specified in this file. These options include compiler, assembler, linker, and archiver invocation commands and options.

Per-File Compile Options

SNIFF+ Make Support uses all compile options on a per-platform basis. This means that the compiler options in the pRISM+ platform makefiles are used for every file for this platform. However, in embedded development it's common to compile files with per-file options. To support this, an additional macro, `COPT_PER_FILE` is defined in `general.c.mk` to allow you to specify compile options on a per-file basis.

If a file with the `.cop` extension exists, the content of it is passed to the compiler when compiling the corresponding `.c`, `.cc`, `.cxx`, and `.s` files. For example, if you want to instrument the source file `demo.c` with the Diab compiler option `-Xrtc` in order to use RTA Suite to perform run-time error checking on this file, you need to make a file named `demo.cop` to include `-Xrtc`. This `.cop` file should be kept in your private workspace.

6.6.7 Generating Makefiles for Your Project

When to Use This Feature

pRISM+ provides integration to support use of SNIFF+ makefile generation feature because it is a very powerful paradigm for building large applications and for managing a team build environment. It is not recommended that you remove pSOSystem makefiles and regenerate them using this makefile generation feature for the following reasons:

- Possible exposure of complexity of pSOSystem make structure to users
- Certain functionality limitations in SNIFF+ Make Support system.
- Possible difficulties for Integrated Systems support staff to recreate your environment in order to track down a problem.

Use SNIFF+ to generate makefiles for your code base only. pRISM+ provides mechanisms for you to integrate your modules that are compiled with SNIFF+ generated makefiles in to a pSOSystem based build. This is the base of the Hybrid Make Model which is recommended by Integrated Systems to users who want to use the automatic makefile generation feature.

How to Use This Feature

In order to use the makefile generation feature, you must create a source project for your code base. The `SSWE:pSOSystem-target-User` is specifically designed to hold your code.

To make this SSWE contain your code, edit the `$PSS_USER_SSWE` environment variable in the start-up script in your pRISM+ installation directory; define `$PSS_USER_SSWE` to the root of your code.

- **On Windows hosts:** Modify the `envtarget.ksh` file.
- **On UNIX hosts:** Modify the `envvtarget.sh` or `envvtarget.csh` file.

After editing the start-up script file, you need to restart SNIFF+ for the new setting to take effect.

Now you are ready to make a source project under the `SSWE:pSOSystem-target-User` which will enable sharing of the new project. Since your new project is derived from a SSWE that is derived from `SSWE:pSOSystem-target`, you can later make your project a pSOSystem subproject.

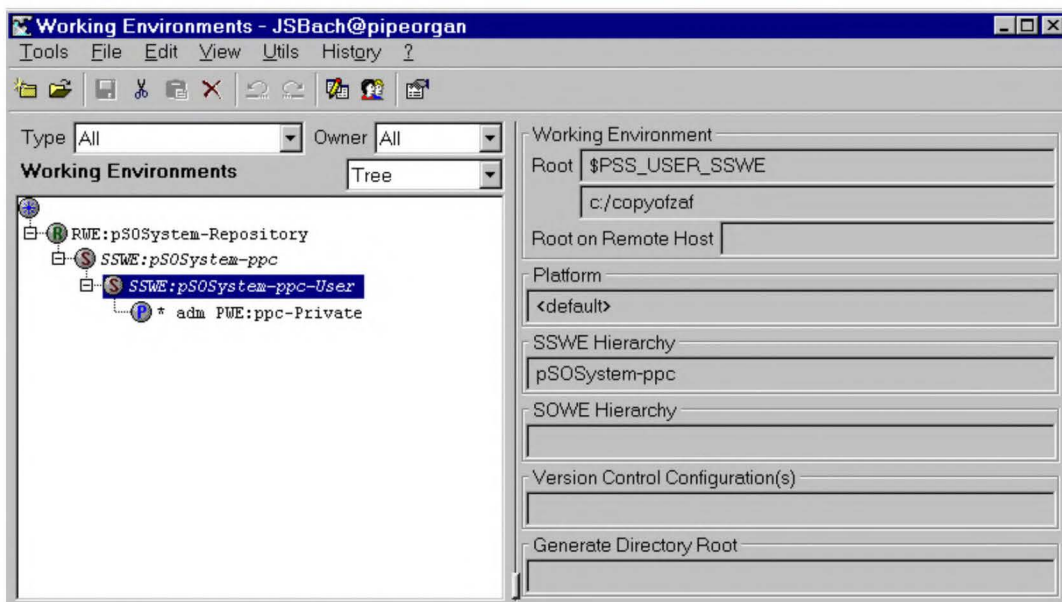


FIGURE 6-17 Working Environment Window

Once a project is made with this target-sharing enabled, you can compile by selecting **Target** → **Make** → **Update Makefile** followed by **Target** → **Make** → **your target** in the PE window. Refer to [Section 6.7, Using the pRISM+ Application Development Framework with SNIFF+ on page 6-50](#), for additional step-by-step instructions.

6.6.8 Hybrid Make Model

The Hybrid Make Model is the method you use to combine the SNIFF+ automatic makefile generation with the pSOSystem make system in order to produce a pSOSystem-based target executable. This make model offers the best of both worlds:

- Use of Integrated Systems-supplied pSOSystem makefiles for OS specific compilation requirements.
- Use of the SNIFF+ powerful automatic makefile generation feature for your code base.

The Hybrid Make Model combines control with flexibility. Integrated Systems strongly recommends that pRISM+ users avoid regenerating pSOSystem makefiles

with SNIFF+. The Hybrid Make Model is the recommended method if you want to use the SNIFF+ Make Support system with pRISM+.

This section explains the benefits of this make model and integration that exists in the pRISM+ Application Development Framework to support this model.

Who Should Use the Hybrid Make Model?

The Hybrid Make Model is designed for the following users:

- Users with an existing code base and who would like to use SNIFF+ to generate and manage makefiles.
- Users who are starting a new project and would like to automate the makefile generation and management process.
- Users who are using SNIFF+ support for team development.

How Does Hybrid Make Model Work?

The Hybrid Make Model works as follows:

1. You begin by pointing pRISM+ to an existing code base and create a shared source project for the existing code base. In the simplest case, this “existing code base” can be an empty directory to be populated by a new project. This step is performed with the help of the pRISMSpace Wizard.
2. Once you have started in this mode from the pRISMSpace Wizard, pRISM+ will automatically enable the makefile generation option.
3. You can then compile your code with the SNIFF+ generated makefiles and a relinkable object is generated by default.

To integrate your code with pSOSystem, you run the **Convert to pSOSystem app proj** script. This script does the following:

- It adds a pSOSystem superproject name `pss_main.shared` to your source project. Your source project then becomes a subproject to `pss_main.shared`.

`pss_main.shared` contains a template pSOSystem makefile with rules for pSOSystem-based target executables such as `ram.elf`. This template makefile also contains a macro which is to hold the name of your relinkable object in order to include it in the final build.

- It enters the name of your relinkable object into the template pSOSystem makefile in `pss_main.shared` so when you invoke the pSOSystem make, your object module is linked into the target executable.
- It also appends most of the common pSOSystem subprojects to `pss_main.shared`, which can be browsed with your code.
- You complete the final build by invoking make on the top-level pSOSystem makefile to generate a pSOSystem-based target executable.

pss_main.shared Project

`pss_main.shared` is the top-most pSOSystem super project which integrates your code base with pSOSystem code. `pss_main.shared` contains a set of three files that are essential to every pSOSystem application: `sys_conf.h`, `drv_conf.c` and a pSOSystem makefile.

The `sys_conf.h` and `drv_conf.c` files used in `pss_main.shared` are generic template files. They are sufficient for a simple application such as the `pdemo` sample application but they might not reflect the needs of your application entirely.

Compare the `sys_conf.h` and `drv_conf.c` files in a pSOSystem sample application (one that closely resembles the type of application you are developing) with the template files. If there are differences, you can either import the needed changes, or simply copy the `sys_conf.h` and `drv_conf.c` from the pSOSystem sample application that closely resembles the type of application you are developing.

If your application code already contains `sys_conf.h` and `drv_conf.c`, your working version should replace the template version.

The `pss_main.shared` makefile is a slightly modified pSOSystem application makefile. The structure and function is similar to all the pSOSystem makefiles that can be found in any pSOSystem sample application in `$PSS_ROOT/apps` directory. This makefile, however, differs slightly from other pSOSystem sample application makefiles in the following aspects:

- The `PSS_APOBJS` macro contains the name of your custom module. The option **Convert to pSOSystem Application** inserts the project name. This module is then linked when any targets are made.
- Unlike sample application makefiles that also contain all the specific libraries those applications need, the template makefile of the `pss_main.shared` does not contain the name of any other libraries. You need to enter into the makefile any other libraries that your application requires.

6.6.9 Doing Team-Based Builds

SNIFF+ make system is integrated tightly with the concepts of workspace and overriding of workspaces. This is reflected by the fact that SNIFF+ generated makefiles use the concept of `VPATH` to allow team-based builds and sharing of files.

To support this team-build concept consistently, for files that are compiled with SNIFF+ generated makefiles as well as those made by pSOSystem makefiles, pSOSystem makefiles have been written to support the concept of workspace as well. These extensions in pSOSystem makefiles assist the team of developers in sharing a common pSOSystem.

This section describes the sharing of pSOSystem code in a team build environment.

SNIFF+ and Overriding of Workspaces

A workspace is a directory tree in the file system in which complete SNIFF+ projects or parts of projects reside. SNIFF+ distinguishes between workspaces that are owned by only one developer (Private Workspaces or PWSs) and workspaces that are shared by a team (Shared Workspaces or SWSs).

A workspace can share files that it does not have, but contained in another workspace. For example, private workspace can share files with shared workspace. A practical application of Private Workspace and Shared Workspace file sharing allows individual team members to build against a common code base without having to maintain local copies of the common files.

A workspace can override another workspace. Files in one workspace can hide files that have the same name and relative position in another workspace. For example, suppose both PWS and SWS contain files `apps/hello/root.c`. The `apps/hello/root.c` file in PWS hides the same file in SWS.

A practical application of file overriding between workspaces allows a team member to check out a copy of a shared file into his Private Workspace. After some modifications, this modified version of the file will be used in his next build, overriding the original shared version of the file by the same name in the Shared Workspace.

Both sharing and overriding of files in workspaces are collectively referred as *overriding* in this document.

Sharing of pSOSystem Code

In the pSOSystem context, `$PSS_ROOT` serves as the root of the SWS. A PWS is created for every developer in the team in his own `$HOME/psosppc_pwe` directory. When a shared project is opened in a developer's PWE, only a makefile (and `.mk` files

included by the makefiles) is created in the PWE. No other source files are copied to the PWS.

When modification to a shared file is needed, you can either make a local copy of a file or check it out from the version control tool. This local copy of the file will then hide the same file in the SWE for the developer. pSOSystem makefiles understand SNIFF+ workspaces and they implement sharing and overriding. Because of this feature, users on the same development team can effectively share on common pSOSystem tree.

Refer to [Appendix E](#) for specific modifications made to pSOSystem makefile to enable the support for overriding workspace.

6.6.10 Building from the Command Line

Use the command `psosmake SNIFF_MAKE target_name` to build from the command line.

6.7 Using the pRISM+ Application Development Framework with SNIFF+

This section provides step-by-step instructions on how to use pRISM+ Application Development Framework with SNIFF+, how to start, how to configure the tools for your environment, and how to proceed with developing your application.

The material in the section is organized in terms of several typical usage scenarios, each with a distinct starting point. These starting points are as follows:

1. [Starting a New Project with pRISM+ on page 6-51](#)
2. [Starting a Project from Your Existing Code Base on page 6-63](#)
3. [Integrating a Custom Board Support Package into pRISM+ on page 6-82](#)
4. [Converting a Project Made with pRISM+ Editor on page 6-87](#)
5. [Starting with an Existing Application for a Previous Version of pRISM+/pSOSystem on page 6-87](#)

If you are new to pRISM+, we strongly recommend that you start from [Section 6.7.1, Starting a New Project with pRISM+ on page 6-51](#) and go through all the material in that section to familiarize yourself with the tools. After that, pick a starting point that is the closest to your real development needs and proceed from there.

6.7.1 Starting a New Project with pRISM+

Who Should Use This Procedure?

This usage scenario is intended for the following users:

- First time users of pRISM+ who would like an in-depth tutorial on the pRISM+ Application Development Framework.
- Platform developers who build Board Support Packages (BSPs) and application developers who work closely with pSOSystem.
- Users who are starting a brand new application with no legacy code base, and therefore want to begin development based on a pSOSystem sample application.

NOTE: For this usage scenario, automatic makefile generation is not enabled by default. SNIFF+ will use pSOSystem makefiles when you build your application. If you are starting with no existing base, but would like to use the SNIFF+ makefile generation feature for your project, go to [Section 6.7.2, Starting a Project from Your Existing Code Base on page 6-63](#).

Step-by-Step Instructions

In this usage scenario, you are starting your development with a pSOSystem sample application. There is a variety of sample applications in `$PSS_ROOT/apps` directory, each illustrating one aspect of pSOSystem; for example, SNMP, NFS, etc. Choose one that is the closest to your application type. Once you have selected a sample application to begin with, pRISM+ will attach a Board Support Package (BSP) to this sample application. This way, when you generate a target executable, the executable will be able to run on the board supported by the BSP.

The default Board Support Package is selected at installation time by the installer. To see or to change the default setting of BSP, from pRISM+ Manager, select **pRISMSpace** → **Settings**. When you are starting development in this mode, you will begin by opening a shared pSOSystem sample application project in your private workspace. After you begin, your private workspace will contain the following set of files:

- `sys_conf.h` — pSOSystem configuration file that belongs to the sample application you have selected.
- `makefile` — pSOSystem makefile that belongs to the sample application you have selected.

- `.sniffdir` — a directory used by SNIFF+.
- `sniffprj` — a directory used by SNIFF+.

Other files, such as the actual source files and header files that belong to the sample application and other pSOSystem files necessary to build a target executables, are residing in the shared source workspace. You do not have private copies of shared files in your private workspace by default. Private, or local copies are made when you either manually make copies of files or when you check files out from your source control tool through SNIFF+.

To this base line pSOSystem sample application you will:

- Add your own files into your private workspace directory.
- Modify the pSOSystem makefile in your private workspace to include new files you have added in your build.
- If necessary, switch between different BSPs in order to run your application on a variety of target hardware.
- Make target executable and proceed to debugging and testing.

Version Control

We strongly recommend that you put all your source files under source control before starting development. SNIFF+ supports a number of CMVC tools. If you are not currently using a CMVC tool, we advise using RCS, which is shipped with SNIFF+.

We recommend that you check the entire pSOSystem directory structure into your CMVC tool prior to using SNIFF+. For details on how to check source files into CMVC tools, contact your Systems Administrator and reference the *SNIFF+ User's Guide*.

For the purpose of this tutorial, RCS is used as the version control tool and the entire pSOSystem source tree is checked in. All the examples used in this section assume this.

Start New pRISMSpace

Now you are ready to make a new pRISMSpace for your application. A pRISMSpace holds all the information regarding each pRISM+ session such as your host tools settings, your choice of targets, the location of your source project etc. This session

information is stored in a pRISMSpace file *name.psp*, where *name* is a name you can give your pRISMSpace.

1. To start a new pRISMSpace from the pRISM+ Manager, select **File** → **New** to start the pRISMSpace Wizard. This Wizard will guide you through the pRISMSpace configuration process.
2. In the **Tools Options** dialog, select **SNIFF+** as your project editor and click the **Next** button.
3. In the **Choose a starting point** dialog, choose **Start with a pSOSystem Sample Application** and click the **Next** button.
4. In the **Choose a pSOSystem example** dialog, you will see a list of sample applications. Choose an application that is the closest to the type of application you are developing. This tutorial uses *pdemo*, which is a simple sample application that demonstrates some basic use of pSOSystem.

Choose *pdemo* from the list, then click the **Next** button. This displays the last dialog of the Wizard, **Finish this new project**.

5. In the **Finish this new project** dialog:
 - **pRISMSpace Name** is the name you use to identify your new pRISMSpace. It is always the same as the name of the sample application you opened.
 - **pRISMSpace directory** is the directory that contains your pRISMSpace file, *name.psp*. This directory is your private workspace directory.

See the section [What Really Happened? on page 6-75](#) for a detailed discussion of working with shared projects as a private user.

Click the **Finish** button.

Congratulations, you have completed the steps to start a new pRISMSpace!

pRISM Manager will now call SNIFF+ with your project settings and start SNIFF+ for you. A log window now appears and it shows all the communication between pRISM Manager and SNIFF+. A little later, a SNIFF+ Project Editor Window appears with the pSOSystem sample application you chose and the BSP you chose (see [Figure 6-18](#)).

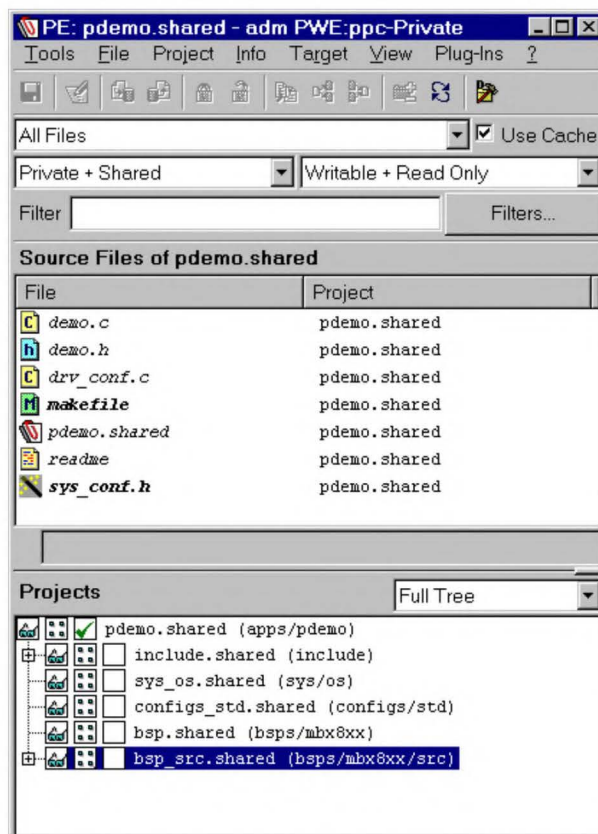


FIGURE 6-18 First PE Window

By completing the steps in previous sections, you have opened a pSOSystem sample application, `pdemo.shared`, as a private user. From the PE window, you see the source files in this project `pdemo.share` as well as the project hierarchy.

Working with the Sample Application

By completing the steps in previous sections, you have opened a pSOSystem sample application, `pdemo.shared` as a private user. Now you can browse all the files in the sample application, build the target of the project, beginning development with this example. Now let us look at how to perform some basic tasks within this development framework.

Building a Target Executable

To build a target executable, do the following from the Project Editor window:

1. Highlight the project for which you want to build an executable. In our case, highlight `pdemo.shared`.
2. From the Project Editor menu, choose **Target** → **Make** → **ram.elf** or another target.

In this usage scenario, you are using a pSOSystem makefile, not SNIFF+ generated makefiles. But because pSOSystem makefiles have been enhanced to support file overriding, they use the `VPATH` macro generated by SNIFF+ to locate the shared files.

Because file overriding is supported, any local copy of a source file will override the shared version of the same file. For example, if you checked out `demo.c` from your version control tool and modified it, the next time you compile, your modified version of `demo.c` is used instead of the `demo.c` in the shared workspace, such as `$PSS_ROOT`.

Start a New File and Add It to the Project

To start a new file and add it to the project:

1. Check out `pdemo.shared`, the PDF file, from version control so you can modify the project structure. You will also be prompted to reload the project. Perform the reload. The PE window will refresh its display.
2. Start a new file and add it to your project by selecting **Project** → **Add New File to pdemo.shared**.
3. Enter the name of the new file you are about to compose and add to `pdemo.shared`, then click **OK**.

The name of your file appears in the file list in the Project Editor window (see [Figure 6-20 on page 6-57](#)). Double-click the file name to open an editor window. Refer to the *SNIFF+ User's Guide* for information about how to change the default new file template used by the SNIFF+ Source Editor when you start a new file.

4. Save your changes to the project structure by selecting **Project** → **Save Project** in the PE Window.
5. Preserve the project structure change by checking in the modified PDF file for the project, `pdemo.shared` (choose **File** → **Check In**).

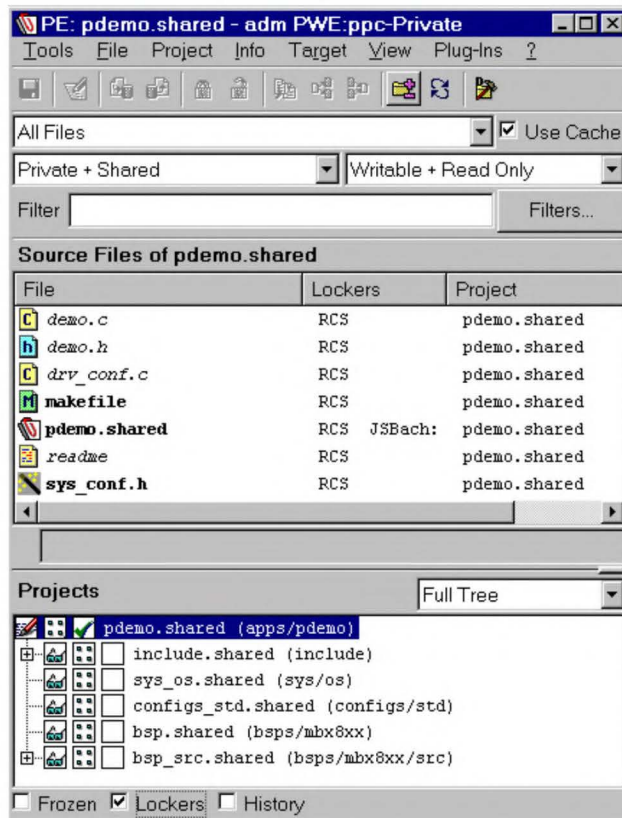


FIGURE 6-19 Project Reloaded

- Update SSWE if you want others to see and share the changes. For complete instructions on how to update SSWE, refer to the *SNIFF+ User's Guide*.

NOTE: You need to edit the makefile in order to add this file to your next build.

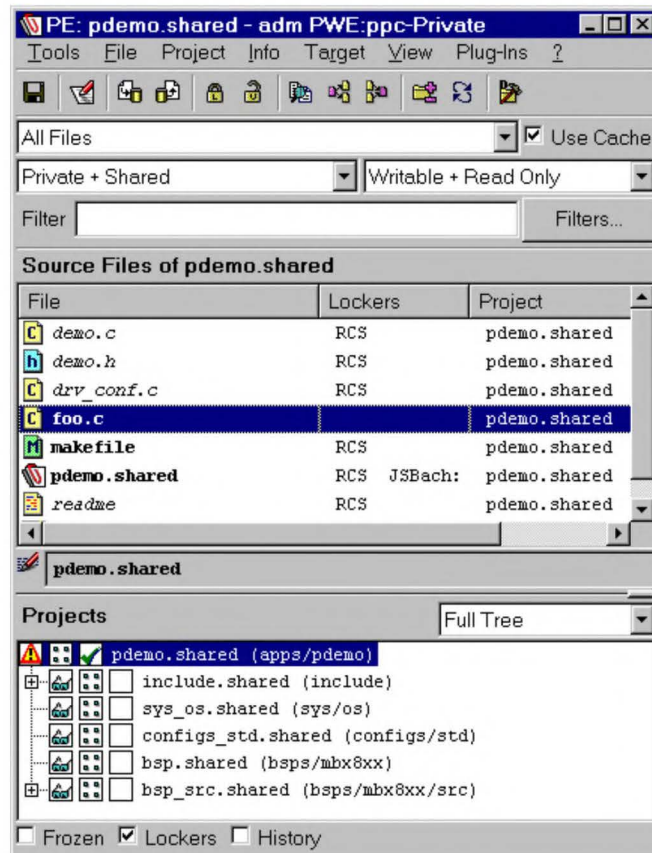


FIGURE 6-20 Foo File Added to Project

Adding Files To and Removing Files From The Project

To add files to the project, or remove files from the project:

1. Check out `pdemo.shared`, the PDF file, from version control so you can modify the project structure. You will also be prompted to reload the project. Perform the reload.
2. From the Project Editor window, choose **Project** → **Add/Remove files to/from pdemo.shared**. This will add or remove files to or from the `pdemo.shared` project from your private workspace directory.

3. Save your changes by selecting **Project** → **Save** in the PE Window. Preserve the project structure change by checking in the modified PDF file for the project, `pdemo.shared`.
4. Update SSWE if you want others to see and share the changes. For precise instructions on how to update SSWE, refer to the *SNIFF+ User's Guide*.

NOTE: You need to edit the makefile in order to reflect the changes you have made to your project in your next build.

Adding/Removing a Whole Directory of Code to/from `pdemo.shared`

To add a whole directory of code to `pdemo.shared` project:

1. For the directory of code you want to add to `pdemo.shared`, recursively make shared source projects out of it and all of its subdirectories.
2. Check out `pdemo.shared`, the PDF file, from version control so you can modify project structure. You will also be prompted to reload the project. Perform the reload.
3. Add the new source project as a subproject to `pdemo.shared` by choosing **Project** → **Add Subproject to `pdemo.shared`**
4. Save your changes by selecting **Project** → **Save in PE Window**. Preserve the project structure change by checking in the modified PDF file for the project, `pdemo.shared`.
5. Update SSWE if you want others to see and share the changes. For complete instructions on how to update SSWE, refer to *SNIFF+ User's Guide*.

Several methods can be used to perform [step 1](#):

- Using instructions given in [Section 6.7.6, *Starting with an Existing Application for a Previous Version of pRISM+/pSOSystem* on page 6-87](#). This is the recommended method.
- Using SNIFF+ Wizard — Refer to the SNIFF+ manuals for instructions on how to use this. Use this method only if you are a proficient user of SNIFF+ already.
- Manually make the project with SNIFF+. Use this method only if you are a proficient user of SNIFF+ already.

After you add a subproject or subprojects to `pdemo.shared`, note that you need to edit the makefile in order to reflect the changes you've made to your project in your next build.

Modifying a Shared File

In the top part of the PE window's file list, the files whose names are in italic are local copies of the shared files. Other files are shared and should NOT be modified until you do the following:

1. Make a local copy of any shared file you want to modify.
2. Check your local copy into a version control tool and then check it out again. This effectively gives you a private copy of this file.

NOTE: A right click on any file name will give you a local menu to perform copy, check in, check out, and edit functions.

The file `sys_conf.h` file uses the pRISM+ Configuration Wizard as the default editor. Since you have a copy of this file in your private directory, you can simply double click on it to modify it with the pRISM+ Configuration Wizard. If you choose to use the SNIFF+ Source Editor to edit `sys_conf.h`, you can access the simple edit function by performing a right mouse click.

Switching to Another BSP

pSOSystem comes with many Board Support Packages for off-the-shelf single board computers. One BSP is chosen as the default BSP at installation time by the installer. This BSP is then attached to all pSOSystem sample applications you open.

Figure 6-21 on page 6-60 shows an example of a **Project Settings** dialog. This figure shows the default BSP as `mbx8xx`, a PowerPC-specific BSP. If your target is a different processor, the default will be something different.

To attach the sample application you are working with to another BSP, you must modify your pRISMspace settings. To change your pRISMspace settings, select **PrismSpace** → **Settings** from pRISM+ Manager, and then change the default Board Support Package.

For your changes to take effect, you must quit out of SNIFF+ from its Launch Pad and then restart it again from pRISM Manager by clicking on the Development Tool button. This will let you reopen your sample application with another BSP.

If you have a custom BSP that you would like to integrate into the pRISM+ Application Development Framework, refer to the section [Integrating a Custom Board Support Package into pRISM+ on page 6-82](#).

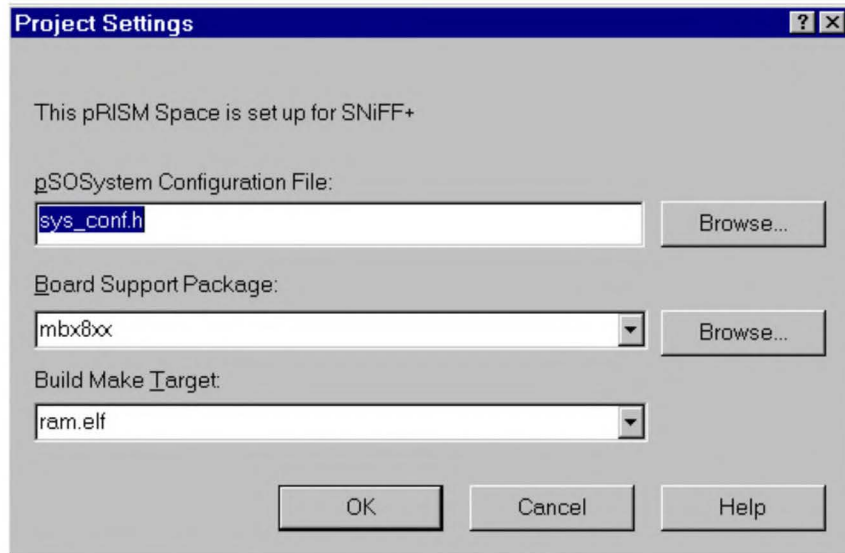


FIGURE 6-21 Project Settings Dialog Box

What Really Happened?

What really happened in your file system when you started a new pRISMSpace with a sample pSOSystem application?

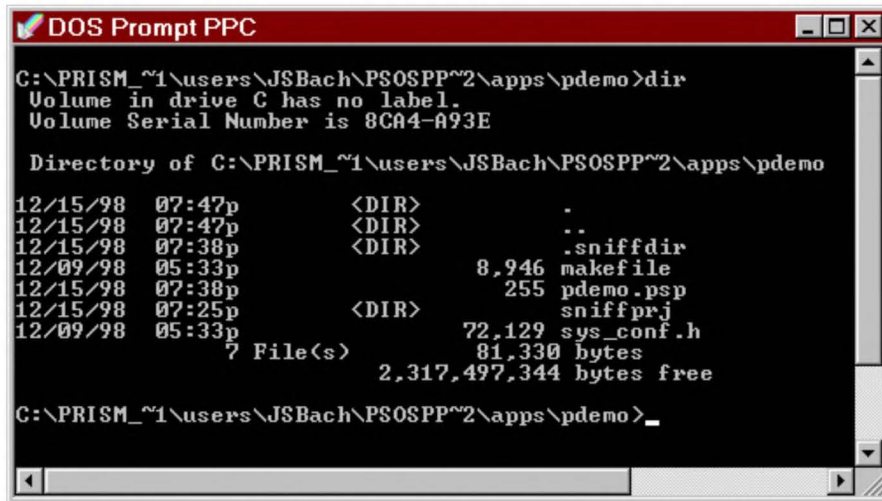
On UNIX hosts, you can do the exploring from the command line.

On Windows hosts, take a look with one of the pRISM+ utility programs. From the Start button, select **Programs** → **pRISM+ 2.0***target_CPU* → **Utilities** → **DOS Prompt** *target_CPU* (where *target_CPU* can be PPC, MIPS, or 68K). This opens a DOS window within the pRISM+ environment settings.

Your Private Workspace

Now, change directory to `$PSS_USER_PWE`, your Private Working Environment root directory. This is the root of your private workspace. You can confirm this by using the SNIFF+ Working Environment Tool.

In the `$PSS_USER_PWE` directory you will immediately see a mirroring directory structure that resembles the pSOSystem top-level directory structure. This is because SNIFF+ created the directories when you opened a shared pSOSystem sample application project, namely `pdemo.shared`.



```

DOS Prompt PPC
C:\PRISM_~1\users\JSBach\PSOSPP~2\apps\pdemo>dir
Volume in drive C has no label.
Volume Serial Number is 8CA4-A93E

Directory of C:\PRISM_~1\users\JSBach\PSOSPP~2\apps\pdemo

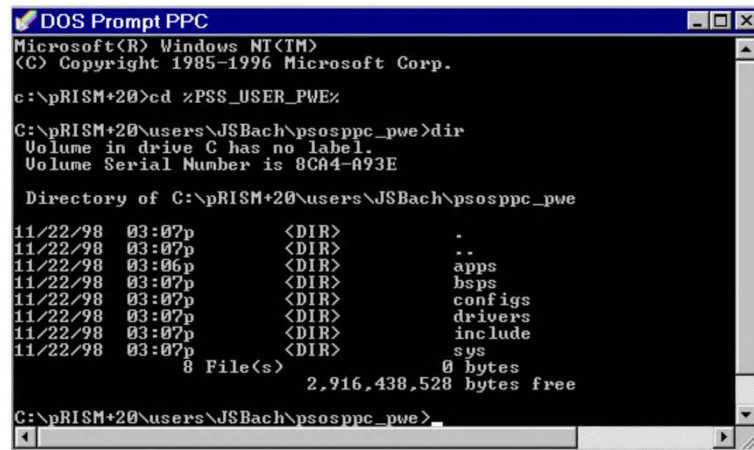
12/15/98  07:47p      <DIR>          .
12/15/98  07:47p      <DIR>          ..
12/15/98  07:38p      <DIR>          .sniffdir
12/09/98  05:33p                8,946 makefile
12/15/98  07:38p                255 pdemo.psp
12/15/98  07:25p      <DIR>          sniffprj
12/09/98  05:33p       72,129 sys_conf.h
? File(s)                81,330 bytes
                        2,317,497,344 bytes free

C:\PRISM_~1\users\JSBach\PSOSPP~2\apps\pdemo>_

```

FIGURE 6-22 Contents of pdemo Directory

When a shared project is opened from a PWE, SNIFF+ always creates a mirroring directory structure in the private working space to mimic the directory structure of the shared source workspace. This mirroring directory structure in your private workspace is later used to hold the files you copy or check out from the shared source workspace.



```

DOS Prompt PPC
Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.
c:\pRISM+20>cd %PSS_USER_PWE%
C:\pRISM+20\users\JSBach\psosppc_pwe>dir
Volume in drive C has no label.
Volume Serial Number is 8CA4-A93E

Directory of C:\pRISM+20\users\JSBach\psosppc_pwe

11/22/98  03:07p      <DIR>          .
11/22/98  03:07p      <DIR>          ..
11/22/98  03:06p      <DIR>          apps
11/22/98  03:07p      <DIR>          bsp
11/22/98  03:07p      <DIR>          configs
11/22/98  03:07p      <DIR>          drivers
11/22/98  03:07p      <DIR>          include
11/22/98  03:07p      <DIR>          sys
8 File(s)                0 bytes
                        2,916,438,528 bytes free

C:\pRISM+20\users\JSBach\psosppc_pwe>_

```

FIGURE 6-23 Contents of Your Private Workspace

Because `pdemo.shared` resides in `$PSS_ROOT/apps/pdemo` in the shared source workspace, your private workspace for it is `$PSS_USER_PWE/apps/pdemo`.

Change directory to `$PSS_USER_PWE/apps/pdemo` to examine its contents. Depending on the type of sample application you have opened, the number of files might vary from what is shown in [Figure 6-23](#). However, what is shown is typical of what happens after the opening of any shared pSOSystem sample application `sample_app.shared`.

The `pdemo` directory shown in [Figure 6-22](#) contains the following files and subdirectories:

<code>makefile</code>	A local copy of the pSOSystem makefile for your project. You always have a local copy of the makefile by default in pRISM+.
<code>sys_conf.h</code>	A local copy of the pSOSystem configuration file. You always have a local copy of <code>sys_conf.h</code> files by default in pRISM+ to allow the use of pRISM+ Configuration Wizard within SNIFF+.
<code>sniffprj</code>	This is an empty directory used by SNIFF+ internally.
<code>.sniffdir</code>	Contains all the intermediate files SNIFF+ generates. For an explanation of the files found in this directory, refer to the <i>SNIFF+ Reference Manual</i> .
<code>pdemp.psp</code>	Your pRISMSpace file.

Any new source files you add to `pdemo.shared` will be kept in this `pdemo` directory, as will any private copies of shared files.

As a small experiment, you can right-click on `demo.c` and choose **Make local copy**. After that, you will see `demo.c` in your private workspace.

Source Files and File Overriding

Notice that in your private workspace for `pdemo.shared`, there are no other source files associated with the `pdemo` sample application because those files are shared. You get a local copy only when you:

- Make a local copy.
- Check a copy out of the version control system (if one is in use).

When you do make a local copy of a file or check out a version for local use from your version control tool, this local version of the file will override the file by the

same name in the shared code base when you perform a build. pSOSystem makefile automatically handles this file overriding feature. For more details about file sharing and overriding, refer to the *SNIFF+ User's Guide*.

As a small experiment, you can right click on `demo.c` and choose **Make local copy**. After that, you will see `demo.c` in your private workspace. This `demo.c` will override the shared `demo.c` in the shared source workspace (namely, under `$PSS_ROOT/apps\pdemo`), the next time you compile `pdemo.shared`.

This concludes the tutorial on how to use pRISM+ to begin development with a pSOSystem sample application using SNIFF+.

6.7.2 Starting a Project from Your Existing Code Base

Who Should Use This Procedure?

This usage scenario is intended for the following users:

- Users who have gone through a pRISM+ tutorial and are now ready to begin development starting with their existing code.

NOTE: If the code base you refer to is a custom Board Support Package you have developed, go to [Section 6.7.4, *Integrating a Custom Board Support Package into pRISM+* on page 6-82](#). If you have not gone through a pRISM+ tutorial, begin with Usage Scenario 1 in the previous section.

- Application developers who have a medium- or large-sized existing code base which they would like to browse, build and eventually integrate with pSOSystem code to produce an embedded application.
- Users who are starting a brand new application with small- or no existing code base, who would like pRISM+ to automatically generate and manage makefiles for the project.

Step-by-Step Instructions

First pRISM+ will make a shared source project out of your code base. By default, the project is created recursively to include all directories and subdirectories in a source tree. Makefiles are generated when your project is created. Once your code is turned into a source project, you can then browse this code, add files to your project, automatically update makefiles, and continue your development.

Automatic makefile generation is by default enabled for this usage case. As pRISM+ is making source projects out of your code, when a makefile is not detected in a directory, pRISM+ will place a generated makefile there. pRISM+ will then update the makefile when new files are added and dependencies change as you add files to your project.

If you already have working makefiles for your code that resides within your code base, don't worry, pRISM+ will not over-write your makefile. You can go on using your own makefile instead of generating new makefiles.

If you are starting with no code but anticipate your project code base to grow and eventually have a substantial amount of code, you can start by generating an empty source project. pRISM+ will then update the makefile when new files are added and dependencies change.

Once you have made source projects out of your own code base, pRISM+ can automatically integrate your code with the rest of pSOSystem code in order to produce a target executable.

Version Control

We strongly recommend that you put all your source files under source control before starting development. SNIFF+ supports a number of CMVC tools. If you are not currently using a CMVC tool, you are advised to use RCS, which is shipped with SNIFF+.

We also recommend that you check in the entire pSOSystem directory structure into your CMVC tool prior to using SNIFF+. For details on how to check in source files into CMVC tools, contact your Systems Administrator and reference the SNIFF+ User's Guide.

For purpose of this tutorial, RCS is used as the version control tool and the entire pSOSystem source tree is checked in. All the examples used in this section assumes this.

Locate Your Existing Code Base

Before you start pRISM+, it is necessary that you set an environment variable which pRISM+ will use to locate your code base. For simplicity, we are now assuming that your code base has a single root. If your code base has more than one root, refer to the section [Working with Multiple Source Trees on page 6-80](#).

While your code can reside anywhere in your file system, the need to integrate with pSOSystem code requires that your existing code base is located in a known loca-

tion to pRISM+. pRISM+ uses the environment variable `$PSS_USER_SSWE` to point to your code.

To point this environment variable to your code, edit `envtarget_CPU.ksh` (on Windows hosts) or `envvtarget_CPU` (on UNIX hosts) in the pRISM+ installation directory.

Modify this line:

```
PSS_USER_SSWE="$HOME/psosTarget_CPU_workspace"
```

to this:

```
PSS_USER_SSWE="location_of_your_code"
```

where `location_of_your_code` is the root directory of your code.

Upon a new pRISM+ installation, `$PSS_USER_SSWE` points to the default location `$HOME/psosTarget_CPU_workspace`. For a large existing code base, it is easier for you to redefine the environment variable than to copy your entire code base into the `$HOME/psosTarget_CPU_workspace` location.

Once you have redefined `$PSS_USER_SSWE`, SNIFF+ will treat the directory that `$PSS_USER_SSWE` points to as the Shared Source Working Environment root directory. SNIFF+ can then make a shared project for you out of the source code in `$PSS_USER_SSWE` directory.

You can then work with this shared project in your PWE, `$PSS_USER_PWE`, just as you can with any shared pSOSystem sample application, as illustrated in [Section 6.7.1, *Starting a New Project with pRISM+ on page 6-51*](#).

Later on, you can run the **Convert to pSOSystem App Proj** command to integrate it with pSOSystem.

NOTE: If you and your team are sharing this code, this step should be performed by a team shared code administrator. For more information on how to set up team-based development, refer to the *SNIFF+ User's Guide*.

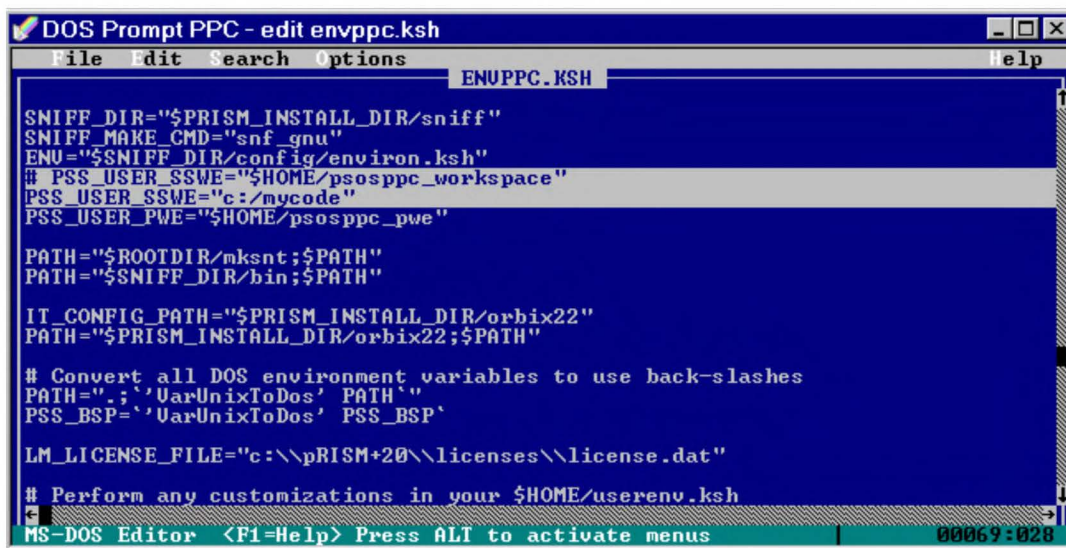
NOTE: You must change `$PSS_USER_SSWE` prior to invoking any pRISM+ tools for the change to take effect.

NOTE: Makefile generation is automatically enabled when you use pRISM+ in this usage scenario. If you want to take advantage of this feature, remove or rename any existing makefiles in your source tree. If SNIFF+ detects a makefile in a directory as a source project is being made, it will not overwrite the existing makefile even when Make Support is enabled.

Locating Existing Code Base

This tutorial uses a small example to simulate an existing code base. This example code is in `c:\mycode\myproj`. You will tell pRISM+ the location of this code base by modifying the pRISM+ environment file `envtarget_CPU.ksh` and by setting `$PSS_USER_SSWE` to `c:\mycode`, as shown in [Figure 6-24](#).

If your code base resides in more than one location, refer to [Section 6.7.3, Working with Multiple Source Trees](#) on page 6-80.



```

DOS Prompt PPC - edit envppc.ksh
File Edit Search Options ENUPPC.KSH Help
SNIFF_DIR="$PRISM_INSTALL_DIR/sniff"
SNIFF_MAKE_CMD="snf_gnu"
ENV="$SNIFF_DIR/config/enviren.ksh"
# PSS_USER_SSWE="$HOME/psosppc_workspace"
PSS_USER_SSWE="c:\mycode"
PSS_USER_PWE="$HOME/psosppc_pwe"

PATH="$ROOTDIR/mksnt;$PATH"
PATH="$SNIFF_DIR/bin;$PATH"

IT_CONFIG_PATH="$PRISM_INSTALL_DIR/orbix22"
PATH="$PRISM_INSTALL_DIR/orbix22;$PATH"

# Convert all DOS environment variables to use back-slashes
PATH=". ; 'VarUnixToDos' PATH"
PSS_BSP='VarUnixToDos' PSS_BSP'

LM_LICENSE_FILE="c:\\pRISM+20\\licenses\\license.dat"

# Perform any customizations in your $HOME/userenv.ksh
MS-DOS Editor <F1=Help> Press ALT to activate menus 00069:028

```

FIGURE 6-24 Change `PSS_USER_SSWE`

Now you are ready to start pRISM+.

Start New pRISMSpace

Now you are ready to make a new pRISMSpace for your application. A pRISMSpace holds all the information regarding each pRISM+ session such as your host tools settings, your choice of targets, the location of your source project etc. This session information is stored in a pRISMSpace file `[name].psp`, name is a name you can give your pRISMSpace.

3. To start a new pRISMSpace from the pRISM+ Manager, select **File** → **New** to display the pRISMSpace Wizard. This Wizard will guide you through the pRISMSpace configuration process.

4. In the **Tools Options** dialog, select SNIFF+ as your project editor choice and click on the **Next** button.
5. In the **Choose a starting point** dialog, choose **Start with an existing codebase** and click on the **Next** button.
6. The **Locate the code starting point** dialog prompts for the location of your existing code base.

If you have successfully redefined the `$PSS_USER_SSWE` environment to point to the root directory of your code base, you will see your new definition of `$PSS_USER_SSWE` expanded and displayed in this dialog. Browse to the location of your code.

In this tutorial example, `$PSS_USER_SSWE` points to `C:\mycode`, and the source files are in the subdirectory `\myproj`.

7. The **Set relinkable object name** dialog prompts for the name to be used by pRISM+ to refer to the relinkable object made from your code.

NOTE: When you start pRISM+ in this mode, with an existing code base, automatic makefile generation is enabled by default. If your code base does not have working makefiles, pRISM+ can generate makefiles automatically and build a relinkable object out of your code base. This relinkable object is then linked with the rest of pSOSystem code when you perform the **Convert to pSOSystem App Proj** operation followed by building of a target executable such as `ram.elf`. For more information about the pRISM+ Hybrid Make Model, refer to [Section 6.6.8, *Hybrid Make Model* on page 6-46](#).

For this tutorial example, you will name the executable `myproj.o`. This name will be entered into pSOSystem makefiles by pRISM+ when you perform the **Convert to pSOSystem App Proj** operation later in this tutorial.

Note that even if you are not using SNIFF+ to generate a makefile but want to use your existing makefiles, you can also enter a name for a relinkable object for the purpose of integration with pSOSystem code. After you do so, make sure you modify your makefile to make this relinkable object.

8. The **Finish this new project** dialog prompts for the name of your pRISMSpace and shows the default location of your pRISMSpace file.
 - **pRISMSpace Name** is the name you use to identify your new pRISMSpace. It is always the same as the name of the shared project you open as a private user.

- **pRISMSpace directory** is the directory which contains your pRISMSpace file, [name].psp. This directory is your private workspace directory. See [What Really Happened? on page 6-75](#) for a detailed discussion of working with shared projects as a private user.

9. Click on the **Finish** button.

Congratulations, you have completed the steps to start a new pRISMSpace!

pRISM Manager will now call SNIFF+ with your project settings and start SNIFF+ for you. A log window appears and shows the communication between pRISM Manager and SNIFF+.

A little later, a SNIFF+ Project Editor Window appears showing a shared source project made out of your code opened in your private working environment.



FIGURE 6-25 myproj.shared in Project

Working with Your Source Project

By completing the steps in the previous section, you have accomplished the following:

- Made a shared source project that a team can share and compile against.
- From your screen, you can see that you have also opened this shared project as a private user.
- Generated makefiles that were placed in your source directories when makefiles were not detected in the directories as the source projects were being made.

Now you are ready to beginning development with pRISM+ Application Development Framework. Now let us look at how to perform some basic tasks within this development framework.

Compiling Your Code

Automatic makefile generation is enabled by default. To compile your code:

1. Update the makefile by selecting **Target** → **Update Makefile** in the PE window.
2. Make the relinkable object needed for later integration with pSOSystem by selecting **Target** → **Make myproj.o** in the PE window.

Now you are ready to add files to browse your source projects, add files to your projects, and compile them.

Start a New File and Add It to the Project

To start a new file and add it to the project:

1. Check in the newly created source project `myproj.shared`, then check out the PDF file `myproj.shared` from version control so you can modify the project structure. You will also be prompted to reload the project. Perform the reload.
2. Start a new file and add it to your project by selecting **Project** → **Add New File to pdemo.shared** in the PE window.
3. In the **New File** dialog box, enter the name of the new file you are about to compose and add to `pdemo.shared`, then click **OK**.

The name of your file now appears in the file list in the Project Editor window. Double-clicking the file name to open an editor window.

Refer to the *SNIFF+ User's Guide* for information about how to change the default new file template used by the SNIFF+ Source Editor when you start a new file.

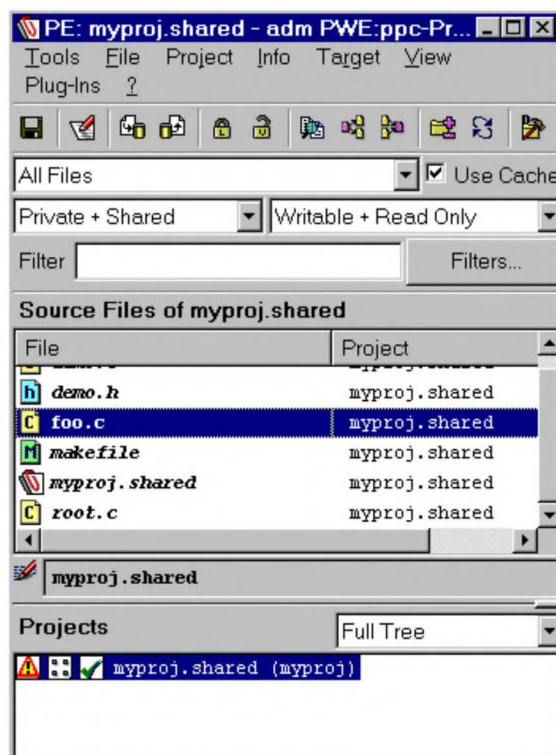


FIGURE 6-26 New File Added

4. Save your changes to project structure by select **Project** → **Save Project** in the SNIFF+ window. Preserve the project structure change by checking in the modified PDF file for the project, `myproj.shared`.
5. Update SSWE if you want others to see and share the changes. For complete instructions on how to update SSWE, refer to the *SNIFF+ User's Guide*.

After adding files to your project, you must update your makefile to reflect the changes. To update your makefile, select **Target** → **Update Makefile** in the PE window. SNIFF+ will then update your makefile automatically.

Adding Files To and Removing Files From the Project

To add files to or remove files from the project:

1. Check out `pdemo.shared`, the PDF file, from version control so you can modify the project structure. You will also be prompted to reload the project. Perform the reload.
2. From the Project Editor window, choose **Project → Add/Remove files to/from myproj.shared**. This will add files to or remove files from the `myproj.shared` project from your private workspace directory.
3. Save your changes by selecting **Project → Save** in PE Window. Preserve the project structure change by checking in the modified PDF file for the project, `myproj.shared`.
4. Update SSWE if you want others to see and share the changes. For complete instructions on how to update SSWE, refer to the *SNIFF+ User's Guide*.
5. You need to update the makefile in order to reflect the changes you've made to your project in your next build. To update your makefile, from PE window, select **Target → Update Makefile**. SNIFF+ will then update your makefile automatically.

Adding/Removing a Whole Directory of Code to/from a Project

To add a whole directory of code to the `myproj.shared` project:

1. Make a source project out of the directory (and all of its subdirectory) of code you want to add to `myproj.shared`, and then save the project.
2. Check out `myproj.shared`, the PDF file, for the `myproj.shared` project, so you can modify the project structure.
3. Add this new source project as a subproject to `myproj.shared` by choosing **Project → Add Subproject to myproj.shared**.
4. Save your changes by selecting **Project → Save** in the PE Window. Preserve the project structure change by checking in the modified PDF file for the project, `pdemo.shared`.
5. Update SSWE if you want others to see and share the changes. For precise instructions on how to update SSWE, refer to *SNIFF+ User's Guide*.

NOTE: After you add a subproject or subprojects to `myapp.shared`, note that you need to update makefile in order to reflect the changes you have made to

your project in your next build. To update your makefile, from PE window, select **Target** → **Update Makefile**. SNIFF+ will then update your makefile automatically.

Several methods can be used to perform [step 1 on page 6-71](#). They are as follows:

- Using instructions given in [Section 6.7.2, Starting a Project from Your Existing Code Base on page 6-63](#). This is the recommended method. Once you've source project out of the directory or directories of code you want added to your project, add them as subprojects to your project.
- If the root of this directory isn't under your current \$PSS_USER_SSWE, refer to the section titled *Working with Multiple Source Trees*.
- Using SNIFF+ Wizard – Refer to the SNIFF+ manuals for instructions on how to use this. Use this method only if you are a proficient user of SNIFF+ already.
- Manually make the project with SNIFF+. Use this method only if you are a proficient user of SNIFF+ already.

Modifying a Shared File

In the top part of the PE window's file list, the files whose names are in italic are local copies of the shared files. Other files are shared and should NOT be modified until a user:

- Make a local copy of it.
- Check it into a version control tool and then check it out again. This effectively gives you a private copy of this file.

A right-click on any file name will pop up a local menu to perform copy, check in, check out and edit functions.

In the case of `sys_conf.h` file, it uses the pRISM+ Configuration Wizard as the default editor. Since you by default have a copy of this file in your private directory, you can simply double-click on it to modify it with the pRISM+ Configuration Wizard. If instead you choose to use the SNIFF+ Source Editor to edit `sys_conf.h` file, you can access the simple edit function by performing a right-mouse click.

Integrate Your Code with pSOSystem

In order to browse your code together with pSOSystem code and to produce a pSOSystem-based target executable, you must integrate your code with pSOSystem.

1. Run the **Convert to pSOSystem app proj** option on your source project to integrate it with pSOSystem code. [Figure 6-27](#) shows the results of this operation.
2. Build a target executable (for example `ram.elf`) from the pSOSystem super-project, `pss_main.shared`.

Convert to pSOSystem Application Project

1. To integrate your code with pSOSystem, from PE window, highlight `myproj.shared` and then select **Plug-ins** → **Convert to pSOSystem app proj**. [Figure 6-27](#) shows your project, `myproj.shared`, after the conversion.

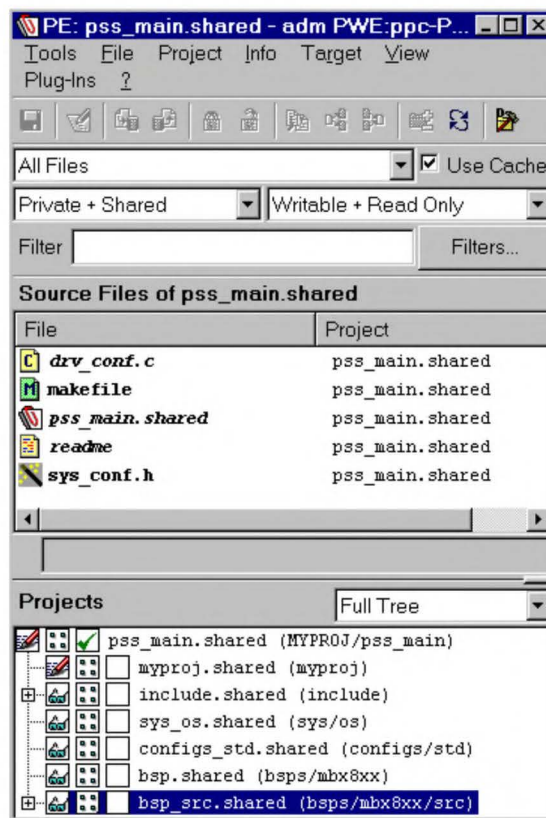


FIGURE 6-27 Converting Your Project

As you can see from the figure, the convert process performed the following:

- Added a pSOSystem superproject, `pss_main.shared` as the top-most project.
- Made your project a subproject to `pss_main.shared`.
- Added a collection of typical pSOSystem subprojects to `pss_main.shared`.

Refer to the section [Using `pss_main.shared` Project on page 6-79](#) for some important information regarding this pSOSystem superproject.

Building a Target Executable

After you run the **Convert to pSOSystem App Proj** on your project, you are ready to build a pSOSystem-based target executable.

1. To complete the target build, highlight `pss_main.shared` in the PE window and select **Target** → **Make** → **ram.elf** (or another kind of target executable).

Now you can proceed to downloading and debugging your module on the target.

Building a Target Executable Using Your Existing Makefile

When a source project is created and a makefile is detected by SNIFF+ in a directory, then no generated makefile will be placed in that directory even if automatic makefile generation feature is enabled. In other words, SNIFF+ will NOT overwrite any existing makefile you have in your source code base.

If you have existing and working makefiles, you may need to modify the default project attributes to have SNIFF+ to invoke your make command and using your makefiles. For complete instruction on how to configure SNIFF+ to use your make and makefiles, refer to the SNIFF+ User's Guide.

If you do choose to use your own make or makefiles to build your own module, you can follow the steps below to integrate your code with pSOSystem and build a target executable:

1. Modify your makefile to generate a relinkable object.
2. Integrate your code with pSOSystem code for browsing and build by selecting **Plug-ins** → **Convert to pSOSystem app proj**.

Switching to Another BSP

pSOSystem comes with many Board Support Packages for off-the-shelf single board computers. One BSP is chosen as the default BSP at installation time by the

installer. This BSP is then attached to all pSOSystem sample applications you open. In our example, by examining the PE window, we can see that mbx8xx is set as the default BSP.

To attach the sample application you are working with to another BSP, you must modify your pRISMSpace settings. To change your pRISMSpace setting, from pRISM+ Manager, select **PrismSpace** → **Settings** and change the default Board Support Package.

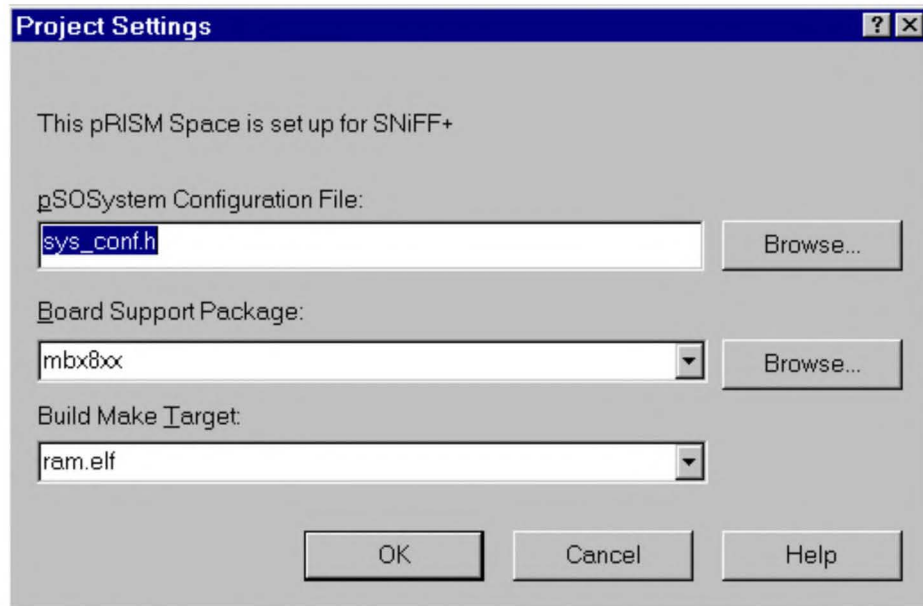


FIGURE 6-28 Project Settings Dialog Box

For your changes to take effect, you must quit out of SNIFF+ from its Launch Pad and then restart it again from pRISM Manager by clicking on the Development Tool button. This will let you reopen your sample application with another BSP.

If you have a custom BSP that you would like to integrate into the pRISM+ Application Development Framework, refer to [Section 6.7.4, Integrating a Custom Board Support Package into pRISM+ on page 6-82](#).

What Really Happened?

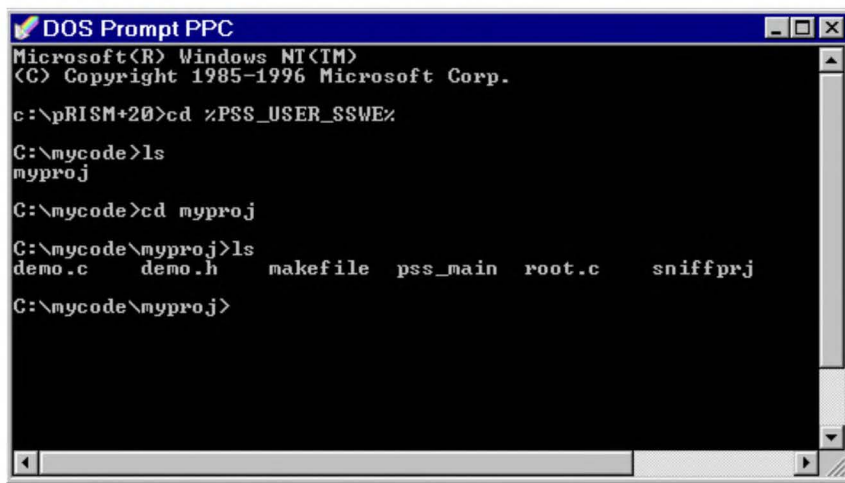
On UNIX hosts, you can explore from the command line.

On Windows hosts, using one of the pRISM+ utility programs, take a look at what actually happened.

From Start, select **Start** → **Programs** → **pRISM+ 2.0 target_CPU** → **Utilities** → **DOS Prompt target_CPU**. This opens a DOS window with the pRISM+ environment settings.

Your Shared Source Workspace

First look at what happened in your shared code base. Change directory to \$PSS_USER_SSWE; for example, to the location of your shared source workspace (in this example, `c:\mycode`) and then to `myproj`.



```

DOS Prompt PPC
Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

c:\pRISM+20>cd %PSS_USER_SSWE%

C:\mycode>ls
myproj

C:\mycode>cd myproj

C:\mycode\myproj>ls
demo.c  demo.h  makefile  pss_main  root.c  sniffprj

C:\mycode\myproj>

```

FIGURE 6-29 Shared Source Workspace

In addition to the files you have in your code base, there are now some new items in your directory:

makefile	This is a project makefile generated by SNIFF+.
sniffprj directory	This is the directory that holds the PDF (Project Definition File) for your shared project <code>myproj.shared</code> .
	For precise definition of Project Definition File, refer to the section SNIFF+ Basics and SNIFF+ <i>User's Guide</i> .

`pss_main` directory Contains files from the `pss_main.shared` project. This is added by the **Convert to pSOSystem App Proj** script you used to integrate your module with pSOSystem code.

For more information on the `pss_main.shared` and its use, refer to the section [Using `pss_main.shared` Project on page 6-79](#).

NOTE: This directory is not present before you run the **Convert to pSOSystem App Proj** command.

Your Private Workspace

Next, let's turn our attention to the PE windows which shows your private view of the shared project `myproj.shared`. Upon completing the steps above, you have in fact accomplished the following:

- Created a shared source project for your team.
- Opened `myproj.shared` in your private workspace.
- Generated makefile for your project

Now let us look at what's taken place in your private workspace. Close examination of `$PSS_USER_PWE` directory will reveal that a new directory with the name of your source project has been created. For our example, in `$PSS_USER_PWE` there is a new directory `myproj` which is the new private workspace for the shared project `myproj.shared`.

When you check out files from SSWE, a copy is placed here for you to modify. Same is true when you copy files from SSWE.

```

DOS Prompt PPC
11/22/98 04:02p      <DIR>          pss_main
11/22/98 03:50p      <DIR>          sniffprj
      8 File(s)          18,513 bytes
      2,825,288,192 bytes free

C:\pRISM+20\users\JSBach\psosppc_pwe\myproj>dir
Volume in drive C has no label.
Volume Serial Number is 8CA4-A93E

Directory of C:\pRISM+20\users\JSBach\psosppc_pwe\myproj

11/22/98 04:02p      <DIR>          .
11/22/98 04:02p      <DIR>          ..
11/22/98 03:50p      <DIR>          .sniffdir
11/22/98 03:50p      0 .Sniff_LastUpdateOfProject_myproj.shared
11/22/98 03:58p      230 foo.c
11/22/98 03:50p      18,283 makefile
11/22/98 04:02p      <DIR>          pss_main
11/22/98 03:50p      <DIR>          sniffprj
      8 File(s)          18,513 bytes
      2,825,288,192 bytes free

C:\pRISM+20\users\JSBach\psosppc_pwe\myproj>

```

FIGURE 6-30 Private Workspace

For now, upon creation, it contains the following:

makefile	SNiFF+ generated project makefile.
sniffprj	A directory used by SNiFF+.
.sniffdir	A directory used by SNiFF+.
pss_main.shared	This is your private workspace directory for pss_main.shared project, the pSOSystem superproject used to integrate your code with pSOSystem.

For more information on pss_main.shared, refer to the section [Using pss_main.shared Project on page 6-79](#).

NOTE: This directory is not present before you run the **Convert to pSOSystem App Proj** command.

Source Files and File Overriding

Notice that in your local workspace, there are no other source files present from your shared source project because those files are shared. You get a local copy when you:

- Make a local copy
- Check a copy out of version control system if one is in use

When you do make a local copy of a file or check out a version for local use from your version control tool, this local version of the file will override the file by the same name in the shared code base when you perform a build. pSOSystem makefile automatically handles this file overriding feature. For more details on file sharing and overriding, refer to SNIFF+ User's Guide.

The SNIFF+ Project Editor tool shows that you've opened a shared project as a private user, for example, in your private workspace. You are looking at shared source files and project files as if they were in your local directory, although earlier we verified that there are not local copies of files yet.

Using pss_main.shared Project

What Is It For?

pss_main.shared is a pSOSystem superproject designed specifically for integration of your code with pSOSystem code. It is a generic pSOSystem superproject to be used as the parent of the source project you want to integrated with pSOSystem. It contains a set of essential pSOSystem files needed by every pSOSystem application, including a pSOSystem makefile which integrates your build into a pSOSystem build in order to generate a pSOSystem-based executable.

Where Is It Stored?

pss_main.shared is stored in a subdirectory in your shared source workspace; that is, in the directory pointed to by \$PSS_USER_SSWE. It was put there by the **Convert to pSOSystem App Proj** command when you converted your project.

What Does It Contain?

pss_main.shared contains the following:

Makefile	This is a template pSOSystem makefile which contains rules to build pSOSystem targets.
drv_conf.c	This file is essential to every pSOSystem application.
sys_conf.h	This file is essential to every pSOSystem application.
readme	Readme file for pss_main.shared
sniffprj	A directory used by SNIFF+ which contains the PDF for pss_main.shared.

Using pSOSystem Application Signature Files (makefile, drv_conf.c, sys_conf.h)

Makefile in pss_main.shared

The makefile contained in `pss_main.shared` is a generic template pSOSystem makefile used to integrate a custom module in a pSOSystem build. This makefile is generic and might **NOT** include all the parts of pSOSystem code you would need for your application. For example, if you are using SNMP, you need to modify the makefile to include the pSOSystem SNMP library. You are responsible for making sure that this makefile is complete. Reference pSOSystem sample application makefiles for what's needed from pSOSystem for each type of application.

This makefile contains a macro `PSS_APPOBS` which should contain the name of the relinkable object made of your custom module. This module is placed in the makefile by pRISMSpace Wizard when you configure this pRISMSpace. This macro can be modified by users. If there are other libraries you want to be linked into the final build, you can also add them here. For information on the make system, refer to [Section 6.6.8, *Hybrid Make Model* on page 6-46](#).

This makefile assumes that it resides in the same directory as the `sys_conf.h` and `drv_conf.c` files which comes in the `pss_main.shared` project.

sys_conf.h in pss_main.shared

The `sys_conf.h` file contained in `pss_main.shared` is a generic template `sys_conf.h` file used to integrate a custom module in a pSOSystem build. This `sys_conf.h` file is generic and might NOT reflect the needs of your application. For example, you may be using more OS components than what the default is set for. You are responsible for making sure that this makefile is complete. Refer to the `sys_conf.h` file in pSOSystem sample applications for what is needed from pSOSystem for each type of application.

6.7.3 Working with Multiple Source Trees

In previous sections we have shown you how to incorporate your existing code base into the pRISM+ Application Development Framework if your code base existed under a single root directory. This section explains how to incorporate multiple source trees into pRISM+.

Suppose your legacy code consists of three source trees under directories `/root1` and `/root2`, and you would like to incorporate all the code into pRISM+. The recommended method is an extension of the method we used earlier to incorporate one source tree.

To incorporate all the code into pRISM+:

1. Edit `envtarget_CPU.ksh` in pRISM+ installation directory and point `$PSS_USER_SSWE` to `/root1`.
2. Edit `envtarget_CPU.ksh` in pRISM+ installation directory and add `$PSS_USER_SSWE2` environment variable to point to `/root2`.
3. Proceed with steps given in [Section 6.7.2, Starting a Project from Your Existing Code Base on page 6-63](#) to create source project for code in `/root1`.
4. Using SNIFF+ Working Environment Tool, create another SSWE derived from `$PSS_USER_SSWE` as shown in [Figure 6-31](#).

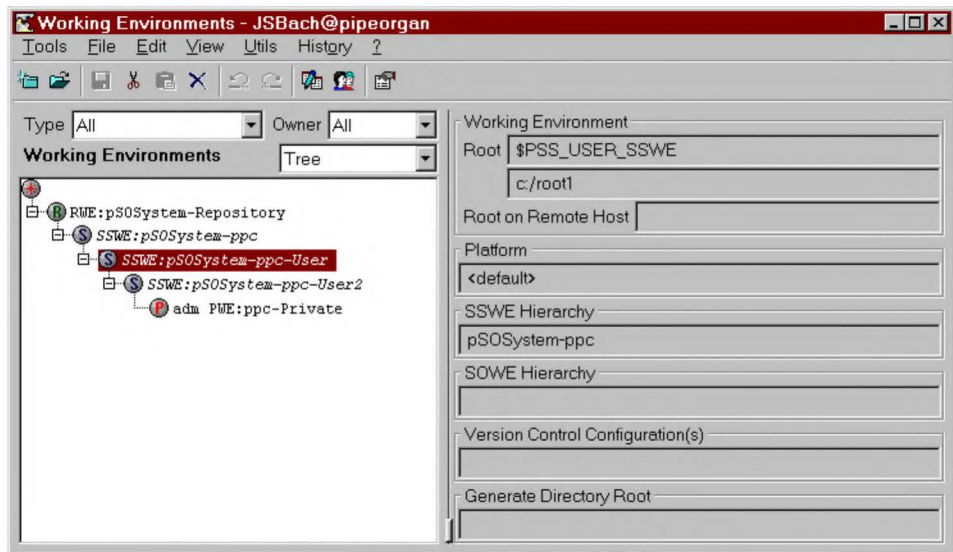


FIGURE 6-31 Incorporating Multiple Source Trees into pRISM+

The SSWE root should be set to `$PSS_USER_SSWE2` which points to `/root2`.

5. Move the PWE as shown in [Figure 6-32](#), to below the SSWE for code in `/root2`. Make sure that when you move the PWE, the **Owner** field is blank. Some pRISM+ scripts will not work correctly if your user is in the owner field.
6. Save the new Working Environment settings.
7. Using the SNIFF+ Working Environment Tool or the SNIFF+ Project Setup Wizard, you can make a source project for code in `/root2`.

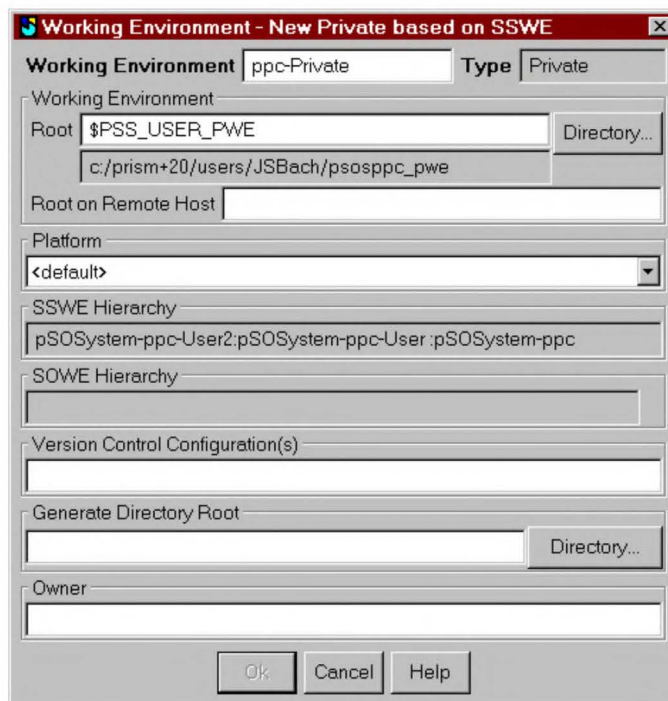


FIGURE 6-32 Make Sure Owner Field is Left Blank

8. Once you have set up a source project for code in `/root2` and code in `/root1`, you will be able to browse files from both source trees together by adding `root1.shared` as a subproject of `root2.shared`.

NOTE: Convert to pSOSystem App Proj only works for the child SSWE derived directly from the `$PSS_ROOT` SSWE.

6.7.4 Integrating a Custom Board Support Package into pRISM+

Who Should Use This Procedure?

This usage scenario is intended for the following users:

- Users who want to integrate custom BSPs into the pRISM+ Application Development Framework and use them in the same integrated fashion as other BSPs shipped with pRISM+.

- Users who want to derive a custom BSP out of a pRISM+ BSP and be able to toggle between the derived BSP and the pRISM+ BSP frequently for development and testing purposes.

Step-by-Step Instructions

For a custom BSP to be integrated into pRISM+ you must first perform the following tasks so your custom BSP conforms to the form needed by pRISM+ for integration. These steps are as follows:

- Copy your custom BSP under `$PSS_ROOT/bsps` so it can be visible to the rest of pSOSystem code which references BSPs with an environment variable relative to `$PSS_ROOT`.
- Organize your BSP directory structure so that it follows standard pSOSystem BSP format.
- Create a file list which contains all the files (including path information) for your BSP. SNIFF+ will use this file list to create a source project for your BSP.
- Run the pRISM+ supplied script `plugins_create_bsp` to create a SNIFF+ project for your custom BSP.

Upon completion of these steps, your BSP will be browser-ready and you are ready to continue with the development and testing of your BSP within the pRISM+ Application Development Framework.

Note that steps illustrated in this section show you how to integrate your BSP into the pRISM+ environment. If you need information on how to port a custom BSP to support this release of pSOSystem, refer to the pRISM+ Upgrade manual.

In this section we will use an example to illustrate the steps required to integrate a custom BSP into the pRISM+ environment for browsing, further development and integration with pSOSystem-based applications.

Copy Custom BSP into `$PSS_ROOT/bsps`

The pRISM+ development environment and tools are set up to allow users to build the same applications to run on many different target boards. pRISM+ accomplishes this by providing an easy toggling mechanism to allow users to work with many different BSPs. This is done with an environment variable `$PSS_BSP` which is defined relative to `$PSS_ROOT`, the root directory of pSOSystem. This is why your custom BSP must reside inside the pSOSystem tree in order for the rest of pSOSystem source projects to have visibility of your custom BSP.

Reorganize Your BSP Directory Structure

All pSOSystem BSPs have a certain directory structure as follows:

- Each BSP resides under the directory `$PSS_ROOT/bsps/<custom_bsp>`, where `custom_bsp` is the name of a BSP.
- Each BSP directory has a subdirectory `/src` which contains the source code for the BSP that's specific to this board.
- Each `/src` directory contains a makefile, in the form of a pSOSystem makefile, NOT SNIFF+ generated makefile, which:
 - Provides rules for compiling files in the `/src` directory.
 - Includes other makefiles, such as `rules.mk`, to include other source files needed by this BSP. Most commonly, these other files are drivers code and devices code that are board-independent. pSOSystem driver code resides in `$PSS_ROOT/drivers` directory and devices code reside in `$PSS_ROOT/bsps/devices` directory.
 - When used in a make, produces a object library, called `libbsp.a`, which is placed in the parent directory of `/src`, which is `$PSS_ROOT/bsps/<custom_bsp>`.

NOTE: Make sure that you organize your custom BSP to conform to this basic structure.

NOTE: Your current BSP makefile might not contain all the drivers and devices you need out of the current pSOSystem now. Do not worry, they can be added later.

Create a File List for Your BSP

An important part of integration of your BSP into the pRISM+ environment is to make it browsing-enabled for SNIFF+. For SNIFF+ to be able to browse your code, you must first turn your source tree into a source project. pRISM+ provides you with a script which automatically performs this for a custom BSP which resides in `$PSS_ROOT/bsps/<custom_bsp>` and which conforms to the basic BSP directory structure described in the last section. This script requires a file list which contains all the files that makes up your BSP, including the path for each file.

Generate a file list that meets the following requirements:

- It contains a list of ALL files which make up your BSP, including drivers code and devices code

- Name this file list `.sniffpl.lst`.
- Place your `.sniffpl.lst` in `$PSS_ROOT/bsps/custom_bsp/src` directory, where `custom_bsp` is the name of your custom BSP.
- First file name in this file should be from `$PSS_ROOT/bsps/custom_bsp/src` directory, where `custom_bsp` is the name of your custom BSP. You can place list for files in `$PSS_ROOT/drivers` and `/devices` after that.
- Your `.sniffpl.lst` should include the file: `configs/std/snf_gnu.mk` at the end.

All standard pSOSystem BSP source projects are created by Integrated Systems using file lists such as the one you are creating for your BSP. Go to any pSOSystem BSP directory to see an example file list used by Integrated Systems to generate the standard BSP projects.

Armed with the file list, you are ready to run the script to perform the final integration of your BSP into pRISM+.

Run `plugins_create_bsp` to Create a SNIFF+ Project

To perform the final step of integration of your custom BSP into the pRISM+ environment, you need to run the `plugins_create_bsp` to create a SNIFF+ project for it.

About `plugins_create_bsp`

`plugins_create_bsp` is:

- A sh script
- Located in `$PSS_ROOT/bin` directory.

`plugins_create_bsp` creates:

- `bsp_src.shared` under `$PSS_ROOT/bsps/custom_bsp/src/sniffprj`
- `bsp.shared` under `$PSS_ROOT/bsps/custom_bsp/sniffprj`

`plugins_create_bsp` assumes:

- You have pRISM+ environment set-up.
- You are using a makefile derived from a pSOSystem BSP makefile and not SNIFF+ generated makefiles.

`plugins_create_bsp` uses:

- `plugins_create_proj` from `$PSS_ROOT/bin/source/plugins/scripts`
- `plugins_add_target` from `$PSS_ROOT/bin`

`plugins_create_bsp` Usage Syntax:

```
plugins_create_bsp <bsp_dir> [ -f <file_list_file> ]
```

where `bsp_dir` is `$PSS_ROOT/bmps/custom_bsp`

Using `plugins_create_bsp`

On Windows hosts, execute this shell script by following the steps below:

1. Start the pRISM+ ksh by selecting **Programs** → **pRISM+ 2.0 target_CPU** → **Utilities** → **Korn Shell target_CPU** from the Windows **Start** menu. This starts the Korn Shell window.
2. From the command line, execute

```
plugins_create_bsp <bsp_dir> [ -f <file_list_file> ]
```

where

- `bsp_dir` is `$PSS_ROOT/bmps/custom_bsp`
- `file_list_file` is the file list you generated for your custom BSP

On UNIX hosts, this script can be executed from a `sh` command line.

You will see SNIFF+ invoked by the shell script through `sniffaccess` program to create your BSP project. On completion of this final step, you have integrated your custom BSP into the pRISM+ environment.

Verifying Your Integration

To verify that you have succeeded in integrating your BSP into pRISM+:

1. Open any pRISMSpace you have made according to the steps illustrated in [Starting a New Project with pRISM+ on page 6-51](#).
2. Change the BSP settings of your previously made project by selecting **PrismSpace** → **Settings** → **Board Support Packages**. By now you should see your BSP added to the list.
3. Select your BSP from the list

4. Shut down SNIFF+ from the SNIFF+ Launch Pad. There is no need to close the current pRISMSpace.
5. Restart SNIFF+ again from the pRISM Manager by clicking on the "Development Tool" button (second from the left). This should bring up the pSOSystem sample application again with your BSP.

6.7.5 Converting a Project Made with pRISM+ Editor

Who Should Use This Procedure?

This usage scenario is intended for the following users:

- Those who use pRISM+ Editor to do their development but want to parse and browse their source files with SNIFF+.
- Those who have previously used pRISM+ Editor but want to transition to using SNIFF+ to continue their projects.

Users of pRISM+ Editor who simply want to use SNIFF+ to browse their source files can treat the body of code they want to browse as an existing code base referred to in [Starting a Project from Your Existing Code Base on page 6-63](#).

Users of pRISM+ Editor who want to transition to using SNIFF+ to continue their projects should first evaluate their team development needs and then proceed with the instructions given in [Starting a Project from Your Existing Code Base on page 6-63](#).

6.7.6 Starting with an Existing Application for a Previous Version of pRISM+/pSOSystem

Refer to the *pRISM+ Upgrade Guide* for complete directions.

7

pRISM+ Configuration Wizard

pRISM+ Wizard helps you configure your pSOSystem application by providing easy editing of the configuration parameters that control pSOSystem and its components. These parameters are briefly described in the pRISM+ Wizard on-line help and fully described in the *Programmer's Reference* manual. These parameters include, but are not limited to, the following:

- Which operating system components are built into the system.
- Serial channel characteristics of the target.
- LAN driver inclusion; if so, the IP address.
- Shared memory network interface (SMNI) inclusion; if so, the IP address.
- Optional device drivers in the system, including SCSI and RAM disk drivers, the TFTP pseudo driver, and any application-specific drivers you may have added.
- Values for most component configuration table entries. For example, the maximum of currently active tasks and message queues in the system.

pRISM+ Wizard allows you to edit these parameters in a window environment with helpful editing features. For instance, pRISM+ Wizard provides intelligent default values for target-specific parameters and automatically checks for inconsistencies between parameter settings.

The output of pRISM+ Wizard is the file `sys_conf.h`, which is read at system start-up to initialize the pSOSystem configuration tables. pRISM+ Wizard enables you to quickly set the appropriate values in the pSOSystem file `sys_conf.h`, which controls pSOSystem configuration.

7.1 pRISM+ Wizard Features

The basic building blocks of pSOSystem are software components such as pSOS+, pROBE+, and pREPC+. Associated with each component is a configuration table, which is used to set configuration parameters.

During system startup, pSOSystem initializes all required component configuration tables. The code that initializes configuration tables is the shared, read-only file `$PSS_ROOT/configs/std/sysinit.c`.

The source code in `sysinit.c` contains many lines whose compilation depends on values defined in the application's pSOSystem configuration file, `sys_conf.h`.

pRISM+ Wizard includes the following features:

- A graphic view of configuration parameters, organized into folders, sub-folders, and pages
- Two parameter indexes, sortable by title or symbolic name
- A Find feature that lets you search by title or phrases within a title
- Fly-by help: Descriptions of all parameters, pages, and folders
- pSOSystem reference information
- Intelligent validation of field values and consistency checking
- Logically stepped instructions for configuration, including the ability to skip forward or backward and to return to the next necessary step
- The ability to import values from `bsp.h` files

7.2 pRISM+ Wizard Interface and Modes

This section describes the pRISM+ Configuration Wizard's interface and the different modes you use to modify your configuration file.

7.2.1 pRISM+ Wizard Interface

The pRISM+ Wizard is composed of a Navigation panel, a toolbar, a Parameter Setting panel, and a Wizard Control panel. See [Figure 7-1 on page 7-4](#).

- The toolbar provides quick access to quick most commonly used commands, such as:
 - The **Save**, **Open**, and **Create** commands, which assist in the updating of your configuration file.
 - The **Find** command, which assists in locating parameters.
 - The **Check** command, which provides error checking for your updated configuration file.
 - The **Help** command, which displays the parameter help window.
 - The **Wizard** command, which launches the Configuration Wizard.
- The Parameter Setting panel displays the current parameters.
- The Wizard Control panel has navigational buttons that are enabled when the Wizard command is invoked.
- The Navigation panel allows you select the mode you want to use to modify your configuration file.

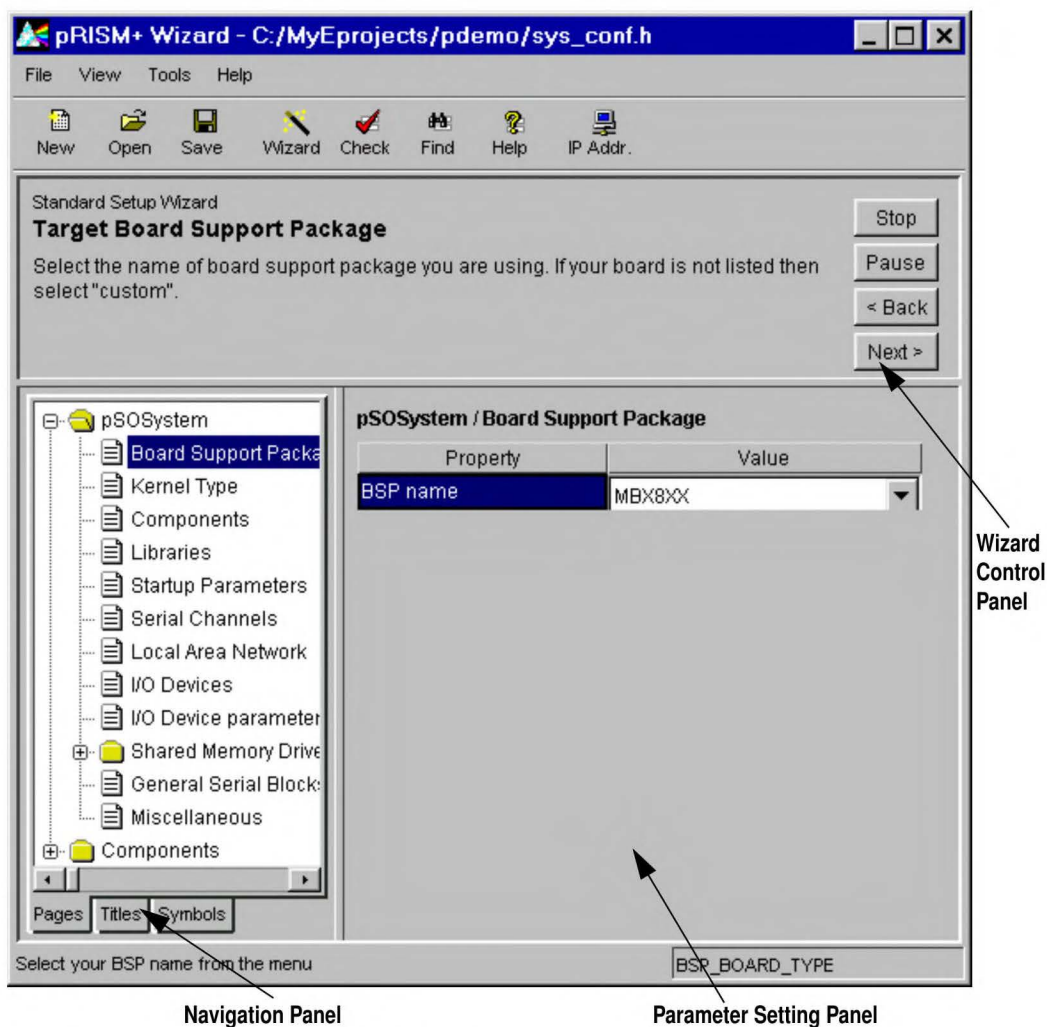


FIGURE 7-1 pRISM+ Configuration Wizard

7.2.2 pRISM+ Wizard Modes

pRISM+ Wizard provides three modes for editing configuration parameters. When you invoke pRISM+ Wizard, a selection dialog is displayed. In the dialog you can

select which configuration editing mode you want to use; these modes are described in [Table 7-1](#).

TABLE 7-1 pRISM+ Configuration Wizard Modes

Modes	Description
Run default configuration wizard	Runs the default wizard to set up a typical pSOSystem configuration.
Choose a configuration wizard	Allows you to choose a wizard to set up a special pSOSystem configuration (e.g., adding networking components).
Just edit configuration parameters	Puts pRISM+ Wizard in simple editing mode so you do not have to follow a wizard sequence.

- Each wizard provides an easy step-by-step process for making changes. Simply review the properties and values displayed in your configuration window, then click **Next** to go to the next step until completion.

If you wish to review what you have already done, the **Back** button takes you back sequentially.

And if you skip ahead to a different topic, pRISM+ Wizard brings you back to the last stage of configuration by graying out other options and highlighting **Resume**.

Verify your BSP settings before you save your file.

Once you have completed the configuration, select **File** → **Save** to save the `sys_conf.h` file. You can then exit the Wizard and go on to edit and compile your application as you would normally.

- To edit your configuration without the wizard, select the third option (Just edit configuration parameters). When the pRISM+ Wizard is displayed you can select the Symbols tab. Use the **Find** option to quickly locate your parameter.

Verify your BSP settings before you save your file.

7.2.3 Error Checking

Once you have completed the configuration, select the **Check** button from the toolbar to verify the settings you have made. If there are errors or incompatibilities between settings, you will be directed to the incorrect or incompatible settings.

You can modify the parameter settings and select the **Check** button again to verify the settings you have made.

Once you have completed your check, select **File** → **Save** to save the `sys_conf.h` file. You can then exit the Wizard and go on to edit and compile your application as you would normally.

7.2.4 Upgrading a Configuration File

To upgrade a `sys_conf.h` file which you have used in a previous version of pRISM+, select **File** → **Upgrade**, rather than **File** → **Save**.

This will upgrade your `sys_conf.h` file with the new fields for this version of pRISM+.

8

The SearchLight Debugger - A Tutorial

This chapter provides a brief introduction to the SearchLight debugger and a tutorial that shows how to use SearchLight to debug a pSOSystem application.

You will learn how to read and display memory variables, set breakpoints, and toggle between System Debug Mode and Task Debug Mode.

8

8.1 What is SearchLight Debugger?

The SearchLight Debugger is a source-level debugger that communicates to the Communication Server and Debug Server which in turn communicates to the pROBE+ target agent and pNA+ on your target. SearchLight has many features available to you to use to debug your pSOS+ application. The following list is the SearchLight product feature highlights:

- A graphical user interface.
- Tracking and control of target executable.
- Breakpoint services.
- Monitoring of language variables.
- C++ language support.
- System and Task Level Debug modes.
- OS breakpoints.
- Debugging Interrupt Service Routines (ISR).
- Query pSOS objects such as tasks, semaphores, and queues.

8.2 Starting SearchLight Debugger and Downloading an Application

This tutorial illustrates the features of SearchLight debugger using the pSOSystem sample application `pdemo`.

NOTE: For simplicity, the figures and samples in this tutorial are PowerPC examples. The SearchLight debugger supports PowerPC, 68K, and MIPS processors.

8.2.1 Accessing SearchLight Debugger

1. Complete the tutorial described in [Chapter 3, *Quick Start with a Tutorial*](#).
2. From the pRISM+ Manager, click the SearchLight debugger button: .



8.2.2 Downloading an Application

1. From the SearchLight main window, click on **File** → **Load**. The **Load** dialog box appears, as shown in [Figure 8-1](#).

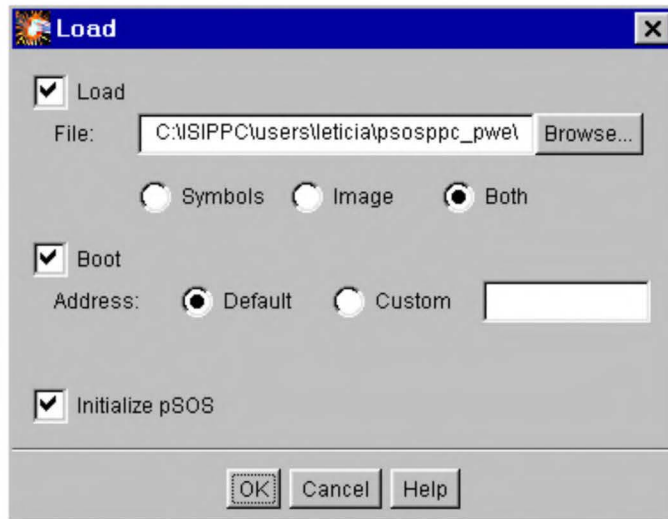


FIGURE 8-1 Load Dialog Box

2. If the application file to load already appears in the **File** text box, go to the next step, otherwise, click the **Browse** button in the **Load** dialog box to locate the file named `ram.elf`. If you already know the path and file name, you can type it

into the text entry field labeled **File**. Addresses and values may vary due to hardware differences.

3. Click on the **OK** button to start the load process.

The debugger proceeds to download the executable image and places a status box on the screen to indicate that the download has started. When the download is completed, the SearchLight Main window contains source code and context information as shown in [Figure 8-2](#).

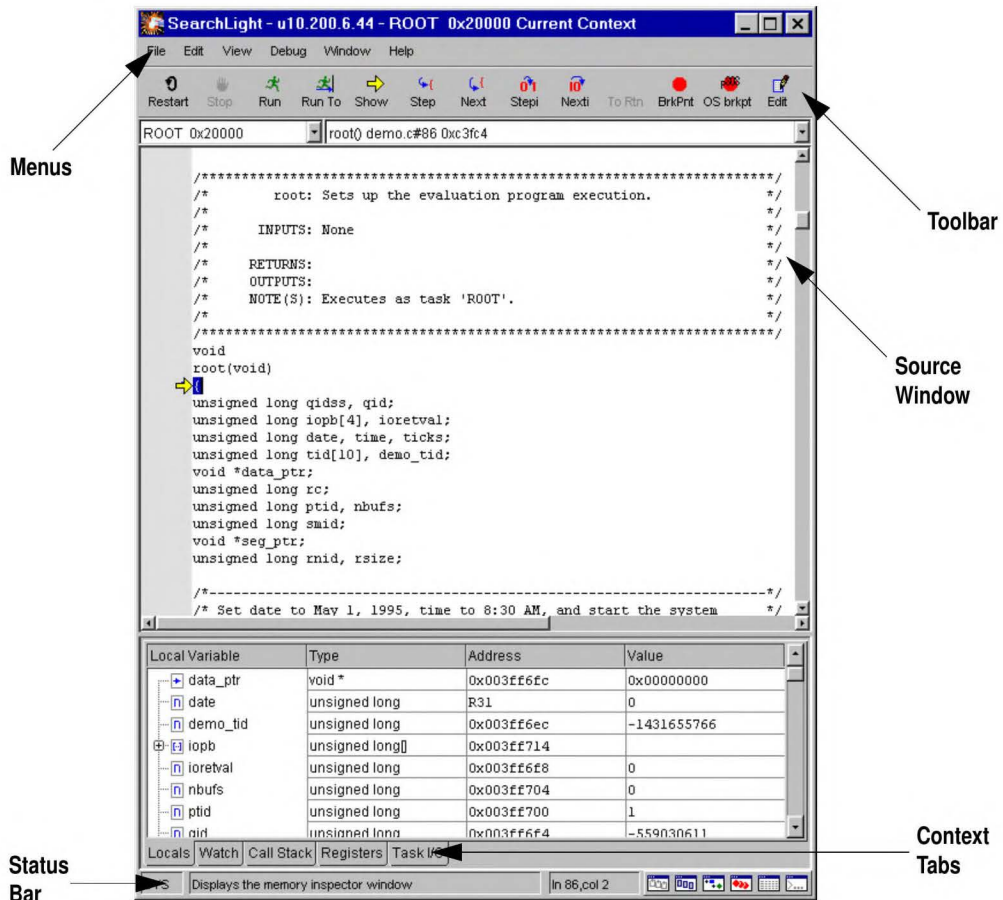


FIGURE 8-2 SearchLight Main Window





8.3 Debugging in System Debug Mode

This section describes how to use SearchLight in the System Debug mode. For information on how to use the Task Debug Mode, refer to [Debugging in Task Debug Mode on page 8-25](#).

8.3.1 Step, StepI, Next and NextI Commands and Code Views


The **Step**, **StepI**, **Next** and **NextI** commands are used to single step through the application code. A brief description of these commands is contained in the following table:

TABLE 8-1 Single Step Debugger Commands

Command	Icons	Description
Step		Executes one line of source code stepping into function calls.
StepI		Executes one assembly language instruction stepping into subroutine calls.
Next		Executes one line of source code stepping over function calls.
NextI		Executes one assembly code instruction and stepping over subroutine calls.

The following set of instructions demonstrate stepping commands and code viewing options.

1. Twice click on the **Step** command icon located in the **Tool** bar.

After the second step command has finished executing, the PC (Program Counter) pointer () is positioned at the source code line shown in [Figure 8-3](#).

```

unsigned long tid[10], demo_tid;
void *data_ptr;
unsigned long rc;
unsigned long ptid, nbufs;
unsigned long smid;
void *seg_ptr;
unsigned long rnid, rsize;

/*-----*/
/* Set date to May 1, 1995, time to 8:30 AM, and start the system */
/* clock running. */
/*-----*/
• date = (1995 << 16) + (5 << 8) + 1;
➡ time = (8 << 16) + (30 << 8);
• ticks = 0;
• tm_set(date, time, ticks);

/*-----*/
/* Initialize the Timer and console device */
/*-----*/
#if !SC_AUTOINIT
if ((rc = de_init(DEV_TIMER, iopb, &ioretval, &data_ptr)) != NOERR)
    k_fatal(0x10000 + rc, 0);

if ((rc = de_init(CONSOLE, iopb, &ioretval, &data_ptr)) != NOERR)
    k_fatal(0x10000 + rc, 0);
#endif

```

8

FIGURE 8-3 Show Pointer (PC) Position After Two **Step** Command Executions

In addition to a C/C++ source code viewing, the SearchLight debugger allows you to view the underlying assembly instructions or both at once.

2. Click on **View** → **Assembly** option.

The SearchLight debugger disassembles the machine code residing on the target board and displays it in assembly instructions as shown in [Figure 8-4](#).

b3fe0	3FE007CB	lis	r31,0x7CB
c3fe4	63FF0501	ori	r31,r31,0x501
→ c3fe8	3FC00008	lis	r30,0x8
c3fec	63DE1E00	ori	r30,r30,0x1E00
c3ff0	3BA00000	li	r29,0x0
c3ff4	7FE3FB78	mr	r3,r31
c3ff8	7FC4F378	mr	r4,r30
c3ffc	7FA5EB78	mr	r5,r29
c4000	48017535	bl	tm_set
c4004	39200002	li	r9,0x2
c4008	91210034	stw	r9,52(sp)
c400c	38810034	addi	r4,sp,52
c4010	38A10018	addi	r5,sp,24
c4014	38C1001C	addi	r6,sp,28
c4018	3C600003	lis	r3,0x3
c401c	480178C1	bl	de_init
c4020	7C7C1B78	mr	r28,r3
c4024	2C1C0000	cmpi	r28,0
c4028	41820010	beq-	demo#134
c402c	3C7C0001	addis	r3,r28,1
c4030	38800000	li	r4,0x0
c4034	480175B5	bl	k_fatal
c4038	3C60000C	lis	r3,0xC
c403c	38634E4C	addi	r3,r3,20044
c4040	38C10010	addi	r6,sp,16
c4044	38800008	li	r4,0x8
c4048	38A00006	li	r5,0x6

FIGURE 8-4 Assembly View

- From the SearchLight tool bar, click on the **Stepi** command icon several times to advance the pointer to the following assembly instructions. The **Stepi** command is used to step through assembly code. See [Figure 8-5](#).

PowerPC	mr r3, r31
68K	jsr _tm_set
MIPS	move a0, s8

43e04	3FE007CB	lis	r31,0x7CB
43e08	63FF0501	ori	r31,r31,0x501
43e0c	3FC00008	lis	r30,0x8
43e10	63DE1E00	ori	r30,r30,0x1E00
43e14	3BA00000	li	r29,0x0
43e18	7FE3FB78	mr	r3,r31
43e1c	7FC4F378	mr	r4,r30
43e20	7FA5EB78	mr	r5,r29
43e24	4800F91D	bl	tm_set
43e28	39200002	li	r9,0x2
43e2c	91210034	stw	r9,52(sp)
43e30	38810034	addi	r4,sp,52
43e34	38A10018	addi	r5,sp,24
43e38	38C1001C	addi	r6,sp,28
43e3c	3C600003	lis	r3,0x3
43e40	4800FCA9	bl	de_init
43e44	3C600004	lis	r3,0x4
43e48	38634A54	addi	r3,r3,19028
43e4c	39010044	addi	r8,sp,68
43e50	38800030	li	r4,0x30
43e54	38A01000	li	r5,0x1000
43e58	38C00200	li	r6,0x200
43e5c	38E00000	li	r7,0x0
43e60	4800F3FD	bl	t_create
43e64	3C600004	lis	r3,0x4

FIGURE 8-5 Stepi Example

- Click on **View** → **Source** option. The SearchLight debugger returns to source view mode. The PC pointer is positioned on the source code line corresponding to the assembly code instructions viewed in the previous step. See [Figure 8-6](#).

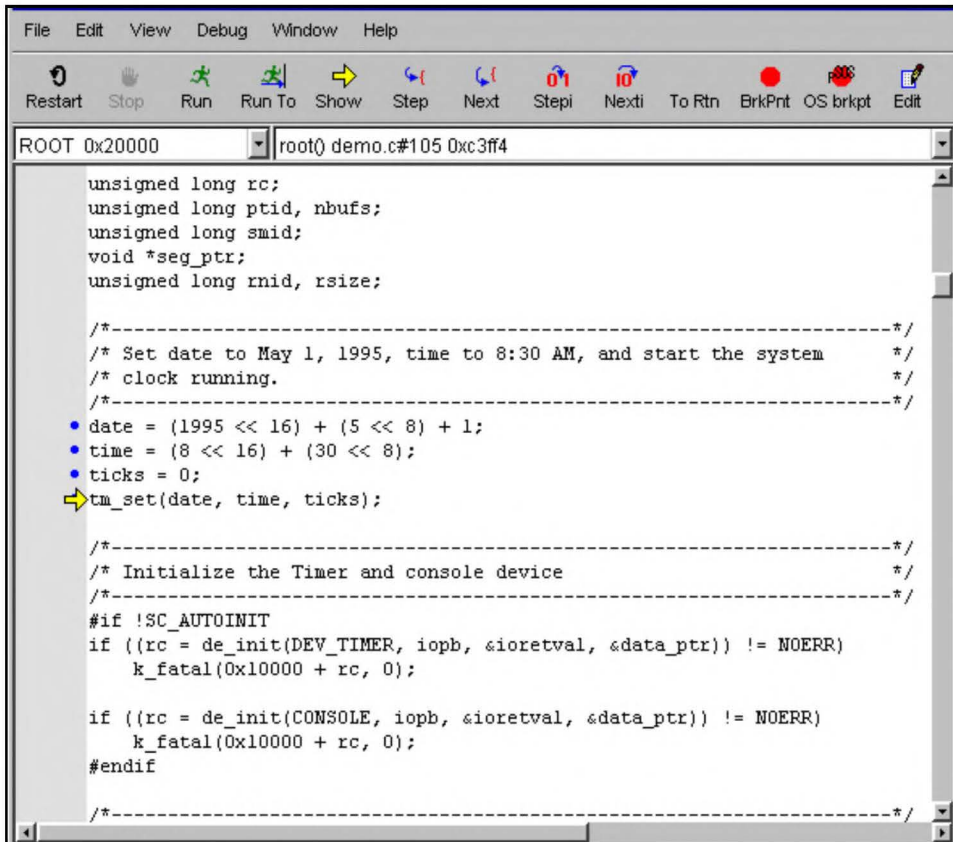


FIGURE 8-6 Source Code View

To step over function calls located in the C/C++ source code use the **Next** command.

- Click on the **Next** command icon located in the **Tool** bar. The SearchLight debugger steps over the `tm_set()` function call and positions the pointer on the next source code line.

8.3.2 Setting and Removing an OS Breakpoint

This section shows how to set and remove OS breakpoints. We are going to examine a complex breakpoint when a queue receives a message. This breakpoint will stop the program when a task or ISR make a queue receive call to any queue.

Setting an OS Breakpoint

1. Click on **Debug** → **OS Breakpoint** or click on the **OS brkpt** icon located in the **Tool** bar. The **OS Breakpoint** dialog box appears as shown in [Figure 8-7](#).

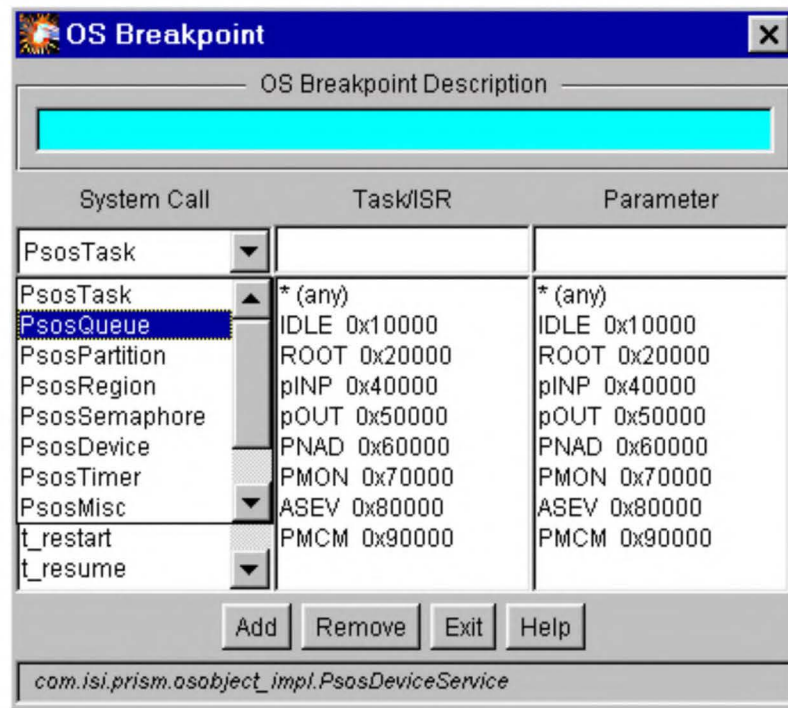


FIGURE 8-7 OS Breakpoint Dialog Box

2. Open the **System Call** drop-down selection list and select **PsoQueue** from the list. A list of system calls related to queues is placed into the **System Call** column.
3. Select `q_receive()` from the first (**System Call**) column then choose `*(any)` from the second (**Task/ISR**) and third (**Parameter**) columns.

After your selections are made the **OS Breakpoint Description** field contains a description of the OS breakpoint you selected as shown in [Figure 8-8](#).

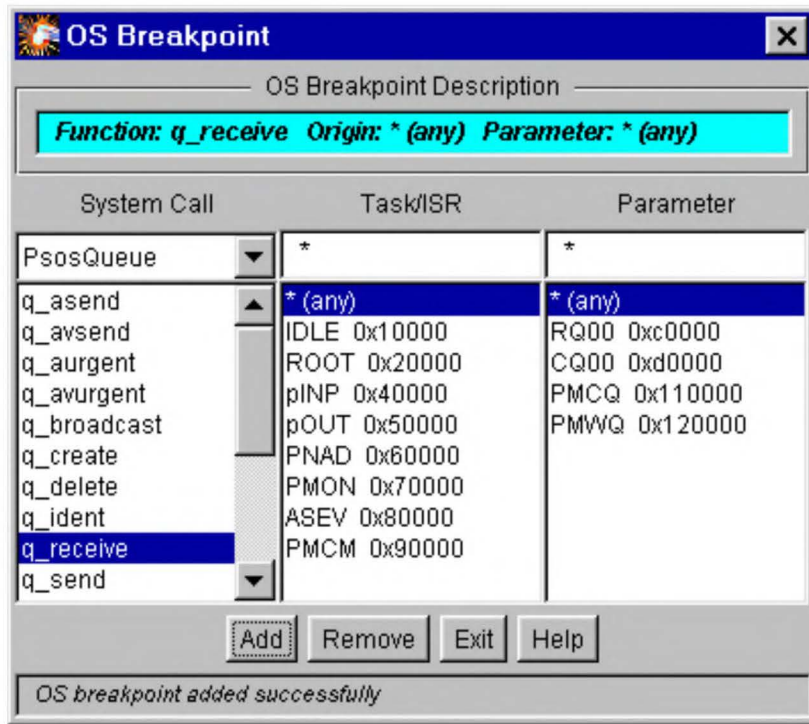



FIGURE 8-8 OS Breakpoint Description Field Filled In

4. Click on the **Add** button to establish the OS breakpoint.
5. Click on the **Exit** button to close the dialog box.
6. Click on the **Run** command icon  located in the **Tool** bar.

The debugger executes the program until it reaches the OS Breakpoint function call. The debugger will stop on the assembly instruction for the OS function call. To view the source code that made the call, click on the **Call Stack** tab and double-click on the second entry, `sink() demo.c` for this example. The **Call Stack** window will be explained in greater detail later in this tutorial.

7. To display the source code, click on the **Call Stack** tab.

8. In the **Call Stack** window, double-click on `sink()` demo.c The source code window brings into view the code containing the breakpoint. See [Figure 8-9](#).

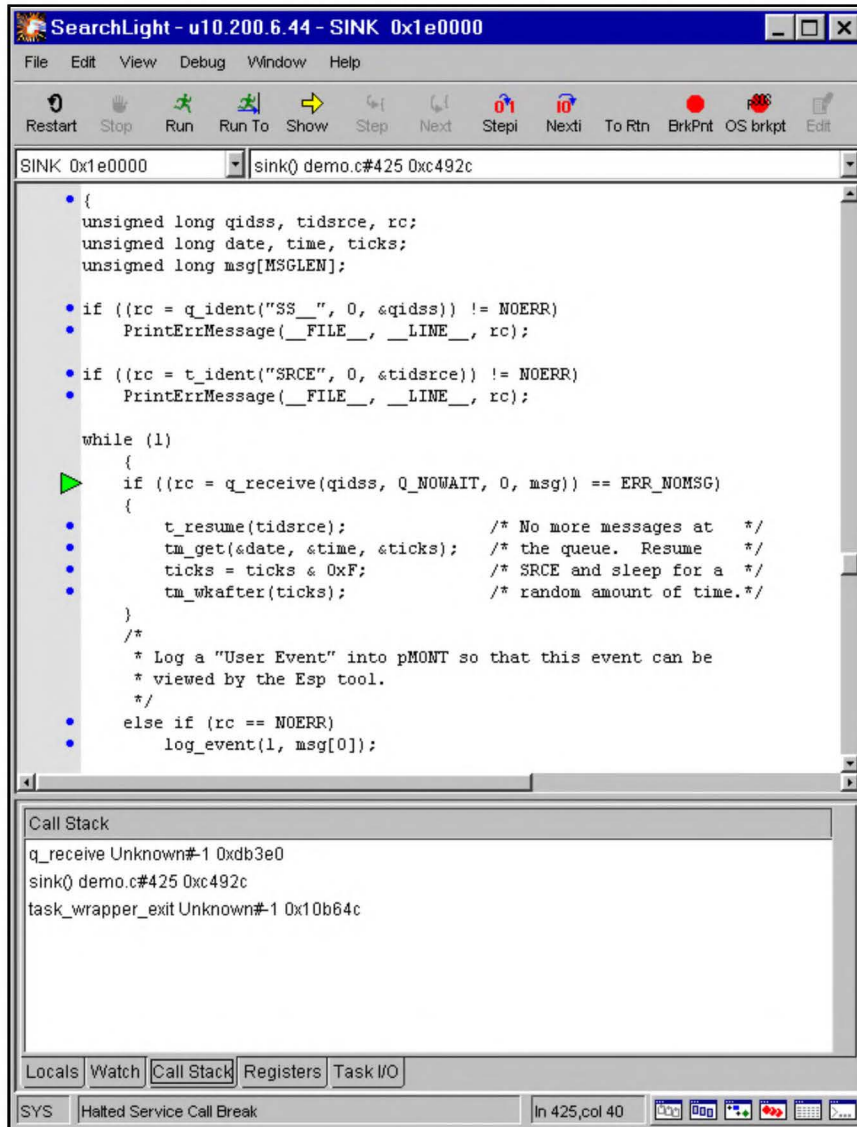


FIGURE 8-9 OS Breakpoint Encountered

Removing an OS Breakpoint

In this section you will remove the breakpoint you set in the previous section.

1. Click on **Debug** → **OS Breakpoint**.
2. From within the **OS Breakpoint** dialog box ensure that the `q_receive` system call with parameters is selected then click on the **Remove** button.

This action removes the previously set OS breakpoint.

3. Select the **Exit** button, to close the **OS Breakpoint** dialog box.

8.3.3 Viewing Memory Variables

The following instructions demonstrate how to change a variable value using the SearchLight debugger.

You can modify the `msg[0]` array element. The memory variables for the current context are viewed in the context view window area of the SearchLight main window.

1. In the SearchLight main window, select the **Locals** tab. The memory variables are displayed in the **Locals** tab as shown in [Figure 8-10](#).

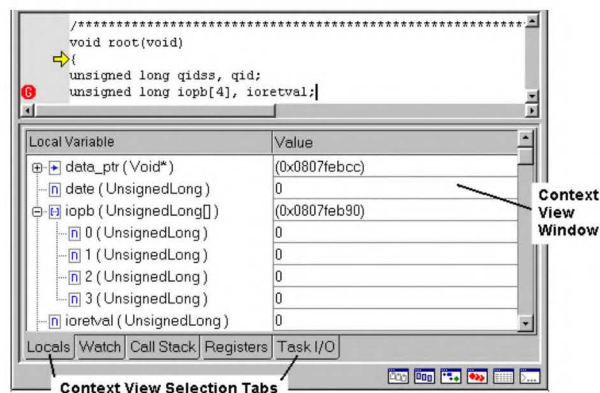


FIGURE 8-10 Current Context View Window

2. Ensure that the local variables display is in view by selecting the **Locals** context view tab.


- Expand the view of the `msg` array by clicking on the expand icon ().

Figure 8-11 shows the expanded view of the `msg[]` array.



Local Variable	Type	Address	Value
  <code>date</code>	UnsignedLong	0x3ed848	0
 <code>msg</code>	UnsignedLong[]	0x3ed854	0x3ed854
 <code>0</code>	UnsignedLong	0x3ed854	1
 <code>1</code>	UnsignedLong	0x3ed858	0
 <code>2</code>	UnsignedLong	0x3ed85c	406448
 <code>3</code>	UnsignedLong	0x3ed860	36914
 <code>qidss</code>	UnsignedLong	0x3ed840	1966080
 <code>rc</code>	UnsignedLong	in register	0

FIGURE 8-11 `msg[]` View Expanded

- Change the value of the `msg[0]` array element by selecting the value field and entering `0xff000000`. You must press `<Enter>` for the change to take effect.

The **Locals** window allows values to be entered in either hexadecimal or decimal format regardless of the current display format. The value will automatically be converted to conform to the current display format. The display format can be changed through the **Edit** → **Preferences** dialog. The available choices are decimal, hexadecimal, and both.

Local Variable	Type	Address	Value
  <code>date</code>	UnsignedLong	0x3ed848	0
 <code>msg</code>	UnsignedLong[]	0x3ed854	0x3ed854
 <code>0</code>	UnsignedLong	0x3ed854	4278190080
 <code>1</code>	UnsignedLong	0x3ed858	0
 <code>2</code>	UnsignedLong	0x3ed85c	406448
 <code>3</code>	UnsignedLong	0x3ed860	36914
 <code>qidss</code>	UnsignedLong	0x3ed840	1966080
 <code>rc</code>	UnsignedLong	in register	0

decimal value of
`0xff000000`

FIGURE 8-12 `msg[0]` Modified

- Click on **View** → **Memory** to access the **Memory** dialog box.
- Type the address of the `msg` array variable (for example: `3ed854`) into the address field and press `<Enter>`. Do not enter the leading numbers, `0x` of the address `0x3ed854`.

2. Click on the **General** expand button to view the individual general purpose registers. See [Figure 8-15](#).

Name	Value
[-] General	
R0	0x8000002a
R1	0x003f05b0
R2	0x00143aac
R3	0x00140000
R4	0x00000001
R5	0x00000000
R6	0x003f05cc


FIGURE 8-15 General Registers Type

Register values can be changed by selecting a value field and entering a new value. For now leave the register values unchanged.

8

8.3.5 Navigating Through the Files Window

This section shows how you can access a source code file through the Files Window. You will also learn how to set a breakpoint from the source code window.

1. Click on **View** → **Files** or click on the **Files** icon () that is located in the **Status** bar to bring up the **Files** window. See [Figure 8-16 on page 8-16](#).

The **Files** Windows displays the list of files that are part of your application. Through this window you are able to easily access and edit your selected file. Double-click on the file you want to view.

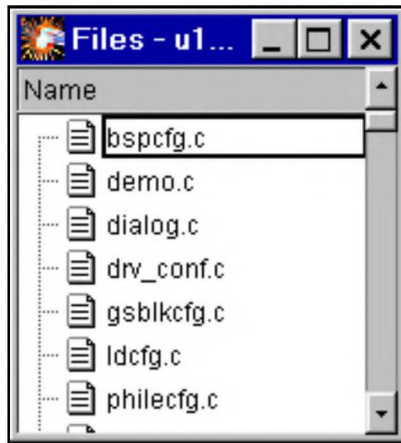


FIGURE 8-16 The Files Window

2. Double-click on the `drv_conf.c` filename, in the **Files** window. This opens the `drv_conf.c` file and displays it in a source code window.
3. From within the `drv_conf.c` source code view window you can use the vertical scroll bar to examine this source code file. See [Figure 8-17 on page 8-17](#).
4. To set a breakpoint in this window, click on the breakpoint icon located just below the **Menu** bar.

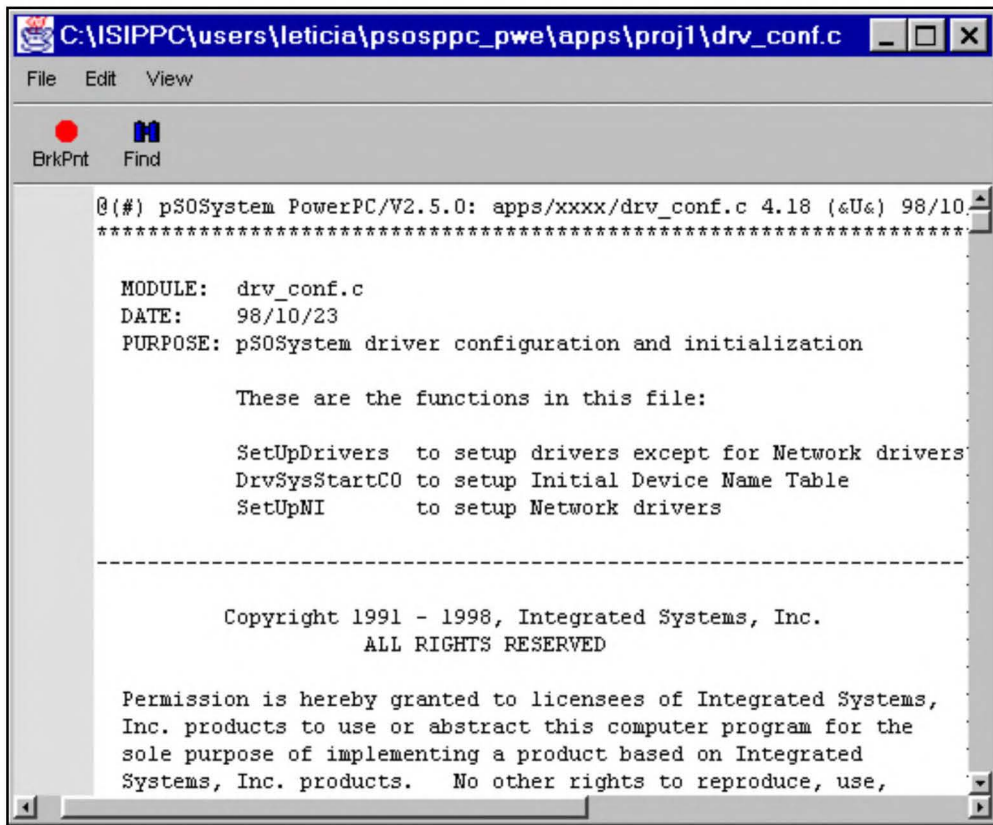


FIGURE 8-17 Function Source Code window

5. To close the Function Source Code window, click on **File** → **Close**.
6. Close the **Files** window.

8.3.6 Using Find to Locate a Text String and Set a Breakpoint

This section you will use the Find command to search for a text string. You will also set a breakpoint.

1. Select **Edit** → **Find**. The **Find** dialog box will display. See [Figure 8-18 on page 8-18](#).
2. In the **Find** dialog box, type:

```
process_data
```


3. Click on the **Find** button. Select the **Find** button again until your cursor stops at the source line:

```
process_data(char *Buf)
```

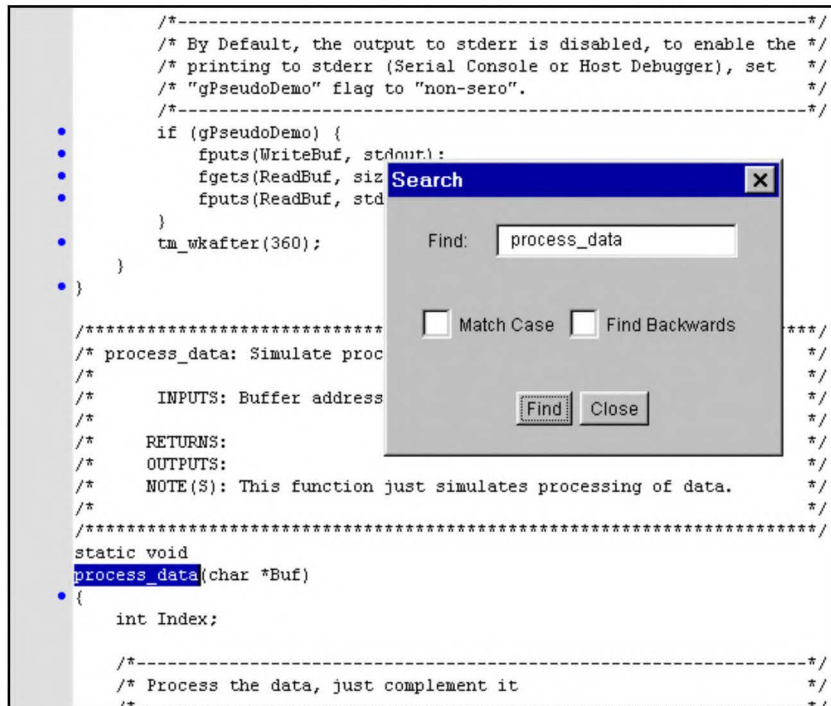


FIGURE 8-18 Locating Text String

- Place your cursor in the Source Window's left margin on the line:

```
for (Index = 0; Index < BLOCK_SIZE; Index++)
```

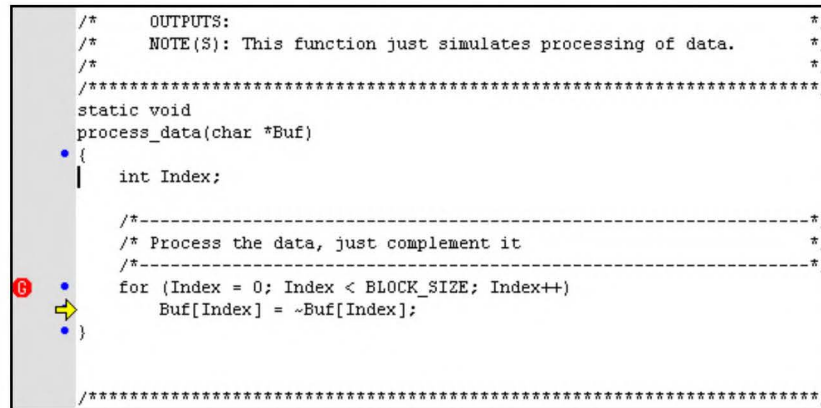



FIGURE 8-19 Setting a Breakpoint

- Perform a right-mouse click in the Source Window's left margin. This creates a breakpoint.
- Click on the **Run** command icon  located in the **Tool** bar.
The debugger executes then stops at the Breakpoint.
- Place your cursor in the Source Window's left margin on the breakpoint.
- Perform a right-mouse click in the Source Window's left margin. A dialog box is displayed asking if you want to remove this breakpoint.
- Click on the **Remove** button to remove the breakpoint.

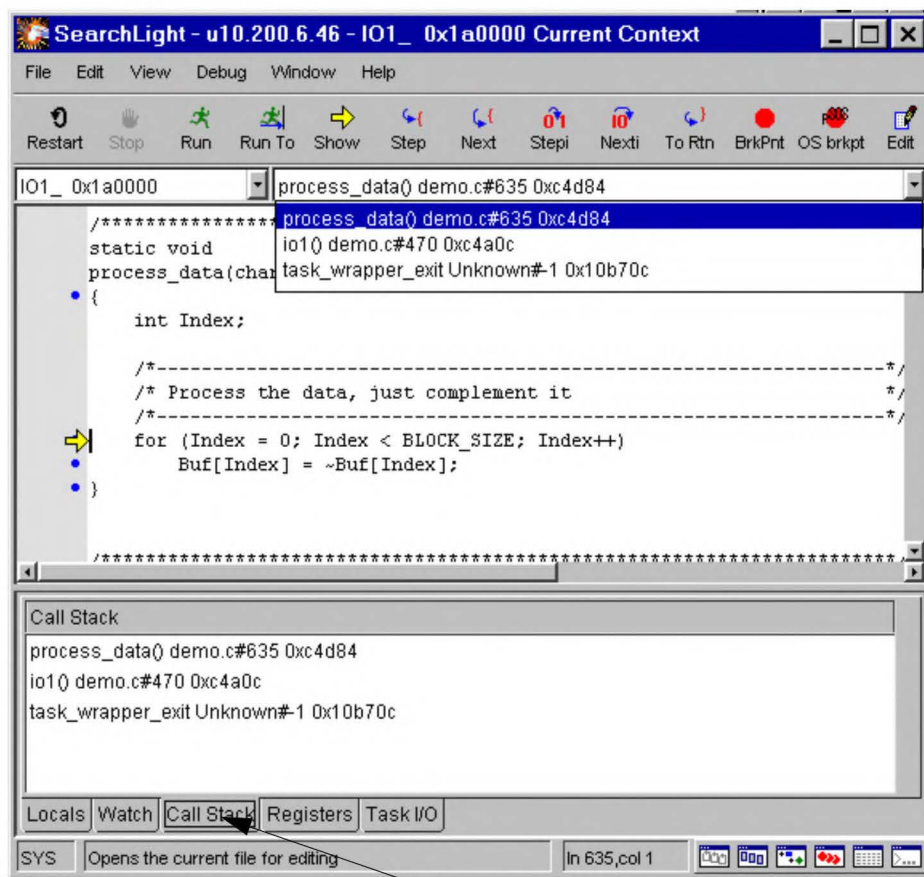
8.3.7 Examining the Call Stack

This section contains information on how to examine the call stack.

1. The **Call Stack** context tab to bring the call stack into view. See [Figure 8-20 on page 8-21](#). (See [Figure 8-10 on page 8-12](#) for a diagram showing the location of the context tabs.)

In this example, the **Call Stack** displays the function call trace of the displayed task. The first line of the display describes the next statement to be executed by the task. The remaining lines display the function call history from the most recent to the earliest. Each line lists the function name, source file name, line number, and the program address of the stack frame.

2. To view the corresponding function code of `io2()` or `io1()`, do one of the following steps:
 - a. Double-click on the `io2()` or `io1()` function call in the **Call Stack** tab. (See [Figure 8-20](#).)
 - b. Click on the Call Stack Context Control (see [Figure 8-20](#)). A drop-down list shows the call stack. Select the `io2()` or `io1()` function call from this list.

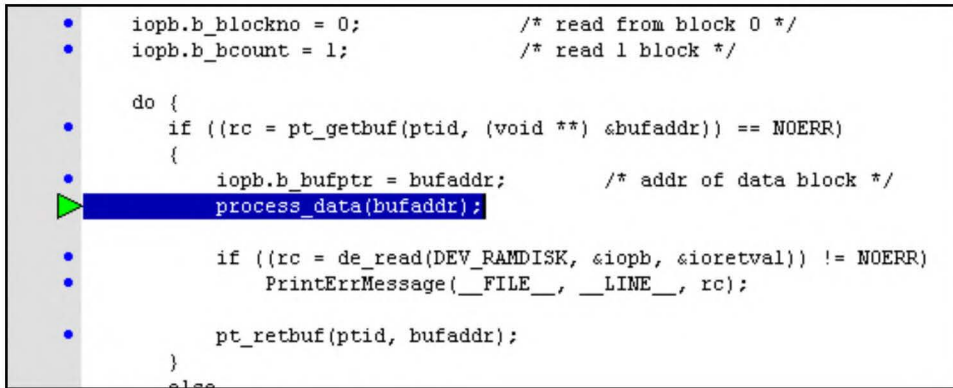


Call Stack Tab

FIGURE 8-20 Call Stack Context Tab

3. Select the `io2()` or `io1()` function from the call stack list.

The source window changes to show the source code of the `io2()` or `io1()` function. A call pointer indicates the call point. See [Figure 8-21](#).



```

•   iopb.b_blockno = 0;           /* read from block 0 */
•   iopb.b_bcount = 1;           /* read 1 block */

  do {
•   if ((rc = pt_getbuf(ptid, (void **) &bufaddr)) == NOERR)
    {
•       iopb.b_bufptr = bufaddr;   /* addr of data block */
•       process_data(bufaddr);
•
•       if ((rc = de_read(DEV_RAMDISK, &iopb, &ioretval)) != NOERR)
•         PrintErrorMessage(__FILE__, __LINE__, rc);
•
•       pt_retbuf(ptid, bufaddr);
    }
  }
  else

```

FIGURE 8-21 `io2()` or `io1()` Function

4. Click on the **Show** icon on the **Tool** bar to return to the source code of the current context.

8.3.8 Examining System Objects

The SearchLight debugger offers many features that help you perform “kernel aware” debugging. You can examine the state of kernel objects such as regions, queues, partitions, and semaphores. Or, you can examine threads of control (tasks). SearchLight provides a view into the internal kernel data structures and presents the relevant information to simplify your task of debugging a real time application.

1. Click on **View** → **pSOS Objects**.

The **pSOS Objects** window is displayed. A list of all of the tasks and their corresponding task IDs is displayed as shown in [Figure 8-22](#).

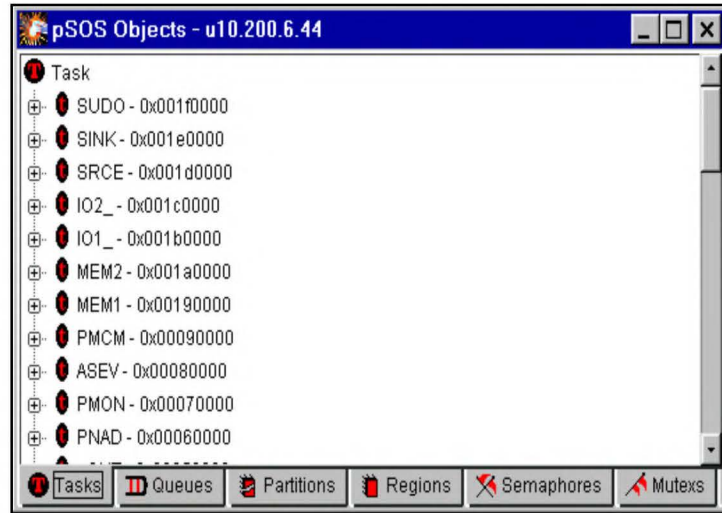



FIGURE 8-22 Tasks List View Tab

2. Click on the expand () control of the SRCE task to view more information about this task. A view of the SRCE task is displayed similar to [Figure 8-23](#).

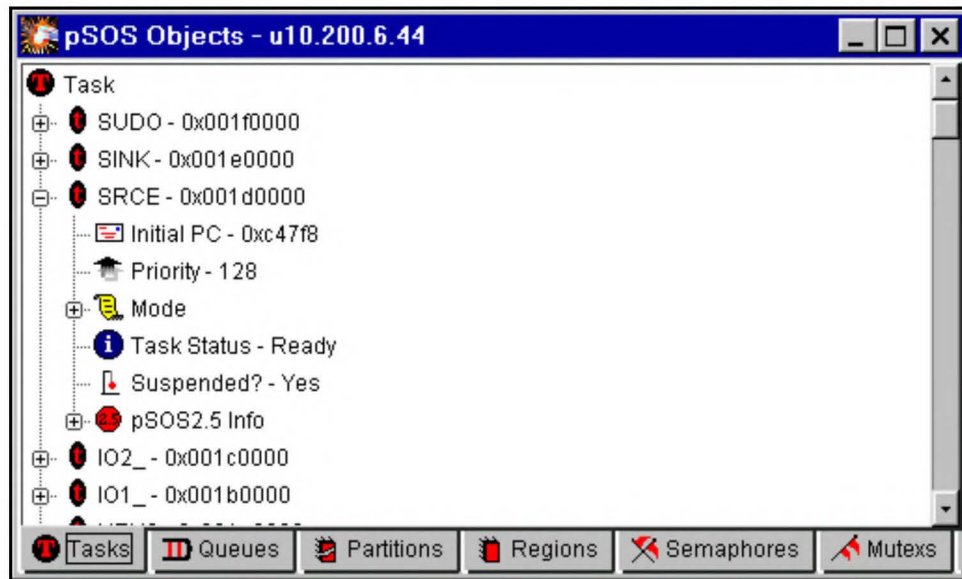


FIGURE 8-23 SRCE Tasks List Displayed

You can also view other system objects (queues, regions, partitions, and semaphores) by pressing the appropriate tab control located at the bottom of the pSOS Objects window.

3. Click the **Semaphores** tab.

The list of Semaphore system objects, similar to [Figure 8-24](#), is displayed.

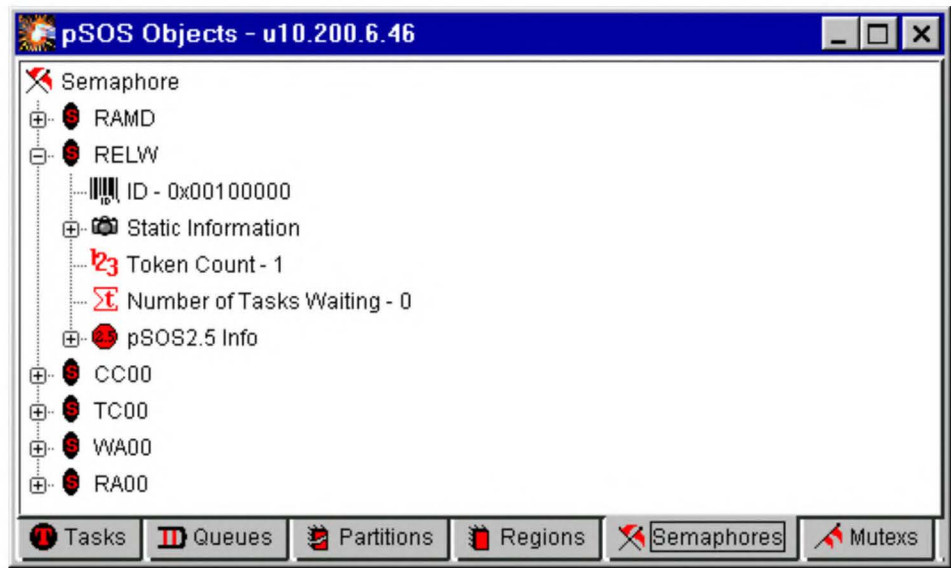


FIGURE 8-24 Semaphore Objects

4. Click on the other control tabs, **Queue**, **Region**, **Partition**, and **Mutexs** to examine their contents also.
5. Close the **pSOS Objects** window.

8.4 Debugging in Task Debug Mode

The following section describe how to use and access SearchLight's TDM (Task Debug Mode). For additional information on the usage of SearchLight access the online help within the SearchLight Debugger.

8.4.1 Accessing Task Debug Mode

1. From the SearchLight menu bar, click on **Debug → Mode**. The **Debug Mode** dialog box will appear. See [Figure 8-25 on page 8-26](#).
2. In the **Debug Mode** dialog box, click on the **Task** radio button.

3. In the task list window, select the task you want to debug. For example in this tutorial click on the Debug checkbox corresponding to the IO1 and IO2 tasks.
4. Click on the **OK** button.

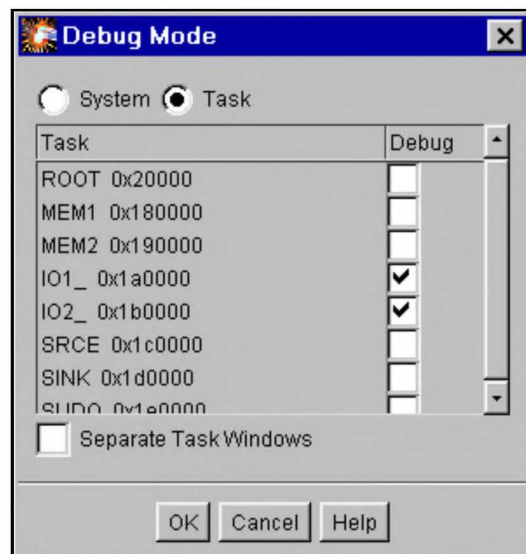
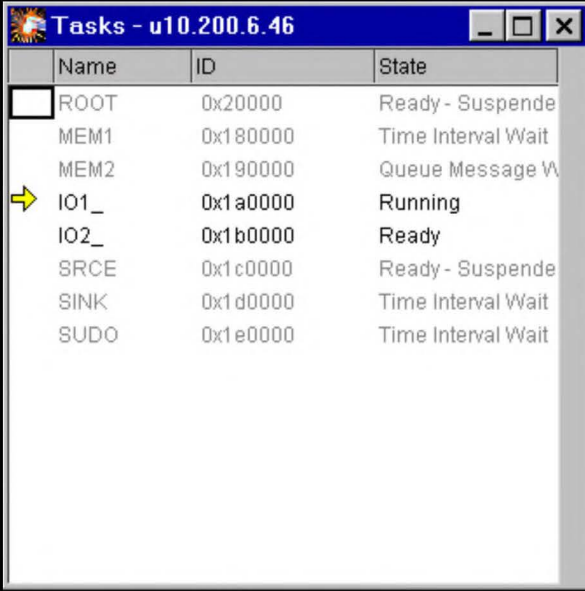


FIGURE 8-25 Debug Mode Dialog

5. From the SearchLight menu bar, click on **View → Tasks**. A task list window will appear.

All of the tasks which are not debug tasks are greyed in the list. The IO1 and IO2 tasks will not be greyed. The debugger is now in Task Debug Mode. See [Figure 8-26](#).



	Name	ID	State
<input type="checkbox"/>	ROOT	0x20000	Ready - Suspend
	MEM1	0x180000	Time Interval Wait
	MEM2	0x190000	Queue Message V
→	IO1_	0x1a0000	Running
	IO2_	0x1b0000	Ready
	SRCE	0x1c0000	Ready - Suspend
	SINK	0x1d0000	Time Interval Wait
	SUDO	0x1e0000	Time Interval Wait

FIGURE 8-26 Task View List in TDM

8.4.2 Setting Breakpoints in TDM

1. From the **Debug** menu, select **OS Breakpoint** or click **OS brkpt** button from the SearchLight main window. The **OS Breakpoint** dialog box appears.
2. Open the **System Call** drop down menu selection and select **pSOSPartition**. A list of related system calls is displayed in the **System Call** column. See [Figure 8-27](#).

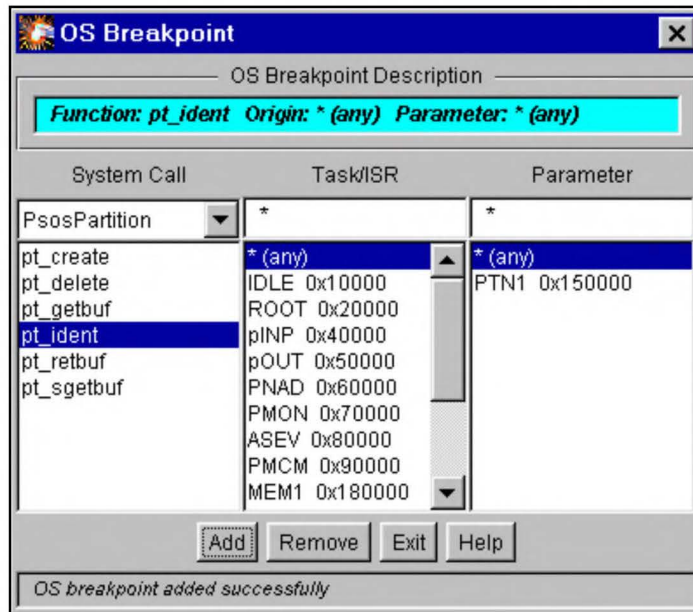


FIGURE 8-27 Setting OS Breakpoints in TDM

3. Select the **pt_ident** from the **System Call** column then choose *** (any)** from the **Task/ISR** and **Parameter** columns. Click the **Add** button to add the breakpoint.

- Repeat steps 2 and 3, selecting the functions shown in the following table. Click the **Add** button. A status bar will display. Check to see if all the breakpoints were added as specified.

TABLE 8-2 TDM Tutorial Settings

System Call	Function Calls	Task/ISR settings	Parameter settings
Partition	pt_ident	*	*
Partition	pt_getbuf	*	*
Partition	pt_retbuf	*	*
Device	de_read	*	-
Device	de_write	*	-

- Click on **Run** in the SearchLight main window.

SearchLight runs until the program execution finds IO1 or IO2 makes one of the above system calls. The source window is updated to show assembly code.

- In the SearchLight window, click on the **Call Stack** context selection tab. Double-click on the io1() or io2() task, whichever appears. The source window updates to show the line corresponding to one of the above function calls.
- Click on **Run** again until any five breakpoints have been found.

Observe the breakpoint source line using the Call Stack tab. By now you would have stopped in the IO1 or IO2 tasks. The debugger will only stop tasks on the TDM list. In this case the tasks are IO1 and IO2.

8.4.3 Removing Tasks from Task Debug Mode

- Choose **Debug** → **Mode** from the SearchLight main window, and deselect the IO1 task from the debug list and click on **OK**.

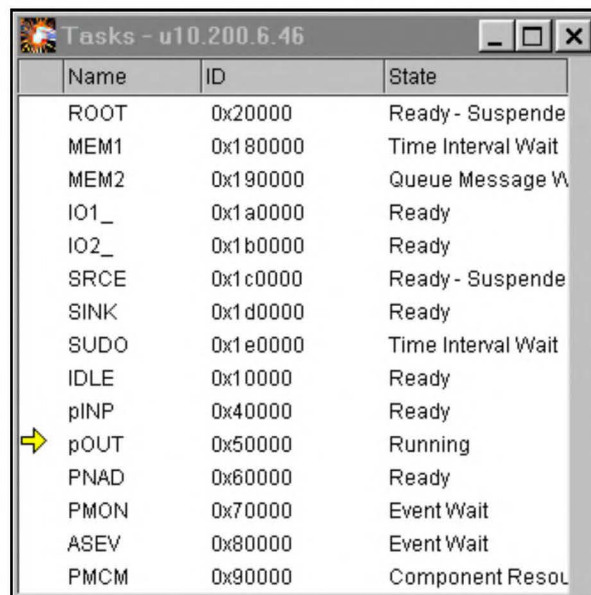
IO2 is the only debug task at this stage. IO1 should be grayed in the Tasks window.

- Click on **Run** and observe the breakpoint source line using the Call Stack tab. Check if the debugger stops execution only for a breakpoint for the IO2 task.

Continue execution this way, until the debugger finds any five breakpoints.

8.4.4 Exiting Task Debug Mode

1. From the SearchLight menu bar, click on **Debug** → **Mode**. The **Debug Mode** dialog box will appear.
2. In the **Debug Mode** dialog box, click on the **System** radio button.
3. Click on the **OK** button.
4. From the SearchLight menu bar, click on **View** → **Tasks**. A task list window will appear. See [Figure 8-28](#).



Name	ID	State
ROOT	0x20000	Ready - Suspende
MEM1	0x180000	Time Interval Wait
MEM2	0x190000	Queue Message W
IO1_	0x1a0000	Ready
IO2_	0x1b0000	Ready
SRCE	0x1c0000	Ready - Suspende
SINK	0x1d0000	Ready
SUDO	0x1e0000	Time Interval Wait
IDLE	0x10000	Ready
pINP	0x40000	Ready
pOUT	0x50000	Running
PNAD	0x60000	Ready
PMON	0x70000	Event Wait
ASEV	0x80000	Event Wait
PMCM	0x90000	Component Resol

FIGURE 8-28 Task View List in SDM

With the exception of Restart, all the commands available in the System Debug Mode are available in the Task Debug Mode.

5. From the SearchLight menu bar, click on **View** → **Breakpoints**. A list of all the breakpoints will appear.
6. Highlight all the breakpoints.
7. Select **Delete Breakpoint** button. This will remove all the breakpoints you set during this tutorial.

8.4.5 Conclusion

You have now concluded the SearchLight debugger tutorial. Additional information on the SearchLight Debugger is located in the on-line help in the SearchLight main window. To access SearchLight help from the SearchLight main window click on **Help** → **Contents**. In the Windows environment, you must have a default browser configured for your system in order to access the SearchLight html help files.

9

The SingleStep Debugger - A Tutorial

The SingleStep debugger from Software Development Systems, Inc. is included as an optional component in pRISM+ for 68K and PowerPC processors. This chapter introduces SingleStep and provides a tutorial that shows how to use SingleStep to debug a pSOSystem application.

9

9.1 What is SingleStep Debugger?

SingleStep debugger lets you control the execution of source-level or assembly language programs, so you can easily find the errors in your applications. You control program execution by setting breakpoints on specified memory address or source location. Execution is then suspended enabling you to examine the variables accessed. The SingleStep debugger also allows you to step line-by-line through a program, either in source-level or assembly language.

The SingleStep debugger operates with the pROBE+ target level debugger. pROBE+ provides a debug connection to the target using a Serial or Ethernet connection.

SingleStep debugger supports BDM (Motorola 68K) and JTAG (IBM and IBM/Motorola PowerPC) target control mechanisms, which are especially useful in situations where target resources are extremely constrained and communication must be simplified. For additional information on BDM or JTAG, refer to [Appendix C](#).

SingleStep Debugger product features include:

- A graphical user interface with multiple windows.
- Automatic tracking of program execution through source code.
- Traces and breaks on high-level language statements.

- Monitoring of language variables and system-level objects such as tasks, queues, and semaphores.
- Full-featured C++ language support.
- Ability to debug optimized code.

9.2 Using SingleStep Debugger

This section illustrates the features of SingleStep debugger using the pSOSystem pdemo sample application.

9.2.1 Before You Begin

Before you can complete this tutorial you must have completed defined in the [Chapter 3, *Quick Start with a Tutorial*](#).

9.2.2 Starting SingleStep Debugger for pSOSystem

To start SingleStep Debugger for pRISM+ and download the pdemo application to the target, complete the following steps:

1. From the pRISM+ Manager, click **Tools** → **SingleStep Debugger**. This launches the SingleStep Debugger.

The Debug window and the SingleStep main window are displayed. See [Figure 9-1](#) for an example of the Debug window.

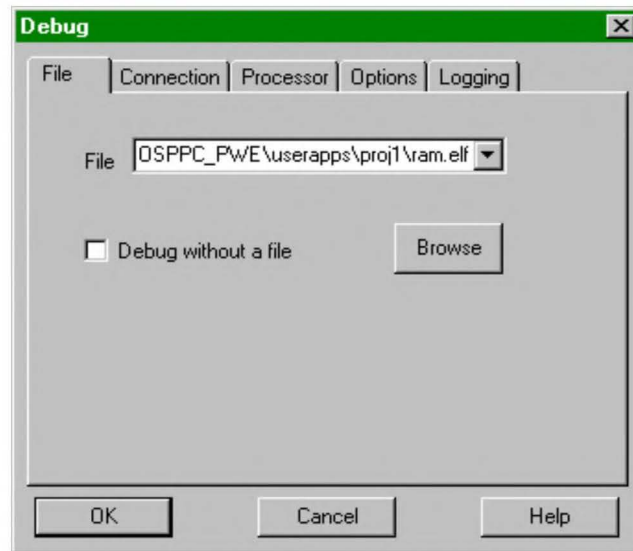


FIGURE 9-1 Debug Window

NOTE: The name of file you want to download should appear in the File field. If not, complete [step 2](#) through [7](#). If your filename does appear in the file field, go directly to [step 8](#).

2. Click the **Browse** button in the **Debug** window and locate the `ram.elf` file.

NOTE: If you already know the path and file name, you can simply type it in the space labeled **File**.

3. Highlight the `ram.elf` file by clicking on it and click the **OK** button.

4. Click on the **Connection** tab.

The **Connection** window is displayed (Figure 9-2).

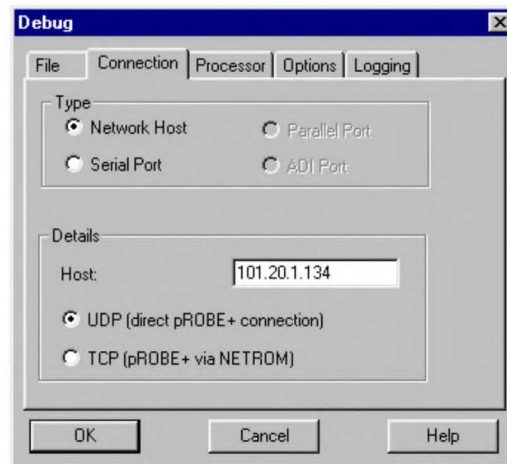


FIGURE 9-2 Debug Window with Connection Selected

5. Select **Network Host** in the **Type** section of the **Connection** window.
6. In the **Details** box, select **UDP** and enter the name or your target board (if DNS is available) or its IP address in the **Host** field.
7. Click on the **Logging** tab and select the **Log to screen (always)** option.
8. Click the **OK** button.

The system proceeds to make the network connection and download the executable image. The **Debug Status** window displays status messages as this takes place. When the download is complete, the **Image Downloading**, **Target Reset**, and **Execute until 'main'** fields should show **Completed**, and the **Debug Session** field should show **Started Successfully** (Figure 9-3 on page 9-5).

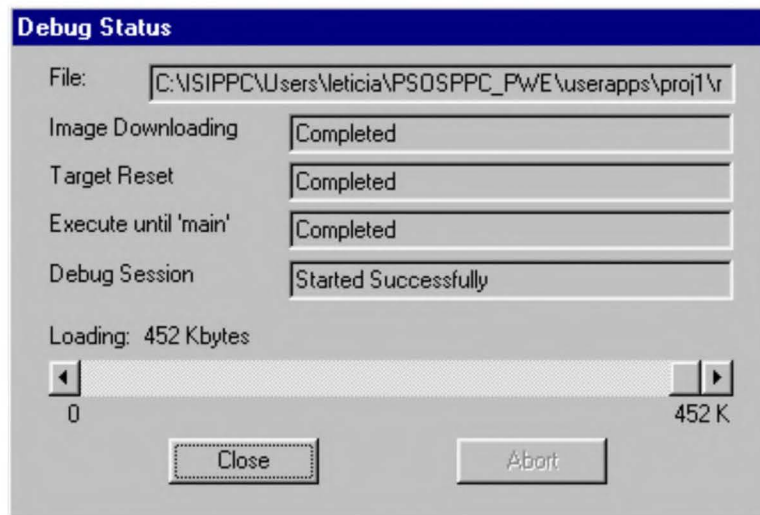


FIGURE 9-3 Debug Status Window

NOTE: The status of the download is displayed in the bottom of the Debug Status window.

9. Click the **Close** button to close the Debug Status window.

9.2.3 The Toolbar and Source Windows

After establishing the connection and successfully downloading the executable image, SingleStep opens the working windows (see [Figure 9-4](#)). These working windows are your main work area.

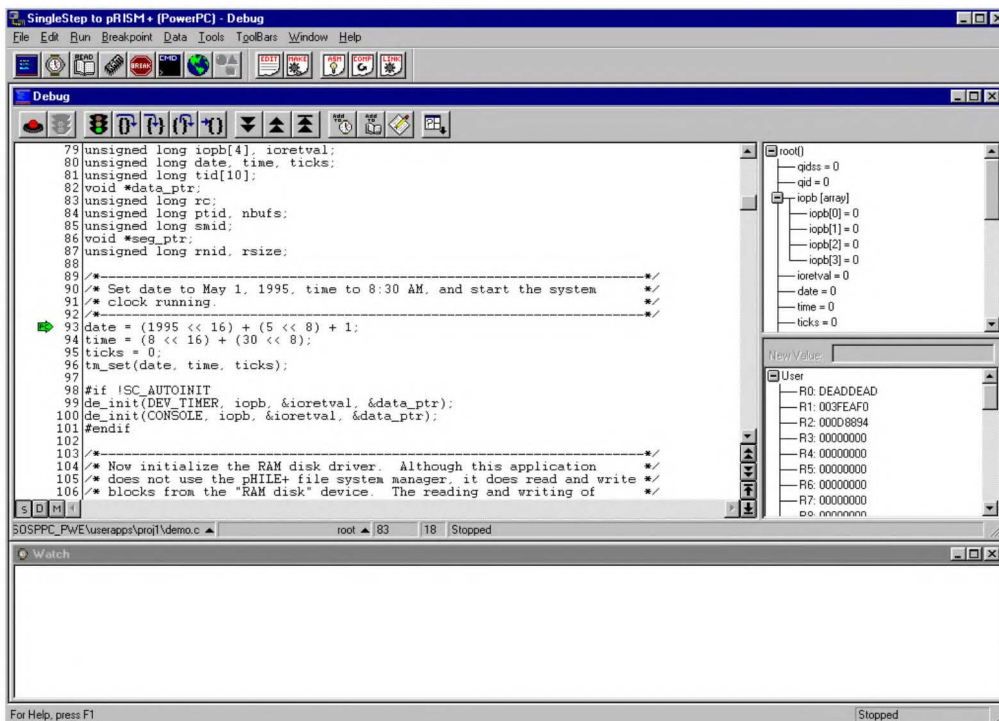


FIGURE 9-4 Toolbar and Source Windows

The first time SingleStep is invoked, three working windows are displayed:

- Debug window
 - Source panel
 - Stack panel (shows both function calls and local variables)
 - Register panel
- Toolbar window
- Watch window

NOTE: By default, these three windows are *detached* windows: The windows are not connected together. Additional windows that are not visible at this time are accessed from the SingleStep toolbar menu. Refer to the *SingleStep User Guide* for more complete information on these windows.

9.2.4 Invoking the Command Window

The Command window, as shown in [Figure 9-5](#), is the interactive shell for entering commands.



FIGURE 9-5 Command Window

To invoke the Command window, select **Command** from the **Window** menu selection on the Toolbar menu.

9.2.5 Running the System Debug Tutorial

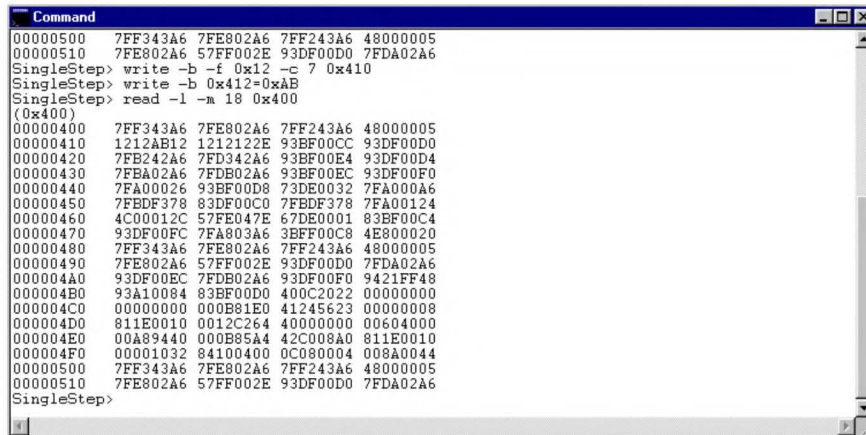
This section covers various basic SingleStep Debugger for pSOSystem tasks.

Memory Manipulation

To examine memory, complete the following steps:

1. Enter **read -l -m 18 0x400** from the Command window.

The **read** command requests a hex display of memory. The **-l** requests memory to be displayed in long words. The **18** requests (and displays) eighteen lines of memory as shown in [Figure 9-6](#).



```

Command
00000500  7FF343A6 7FE802A6 7FF243A6 48000005
00000510  7FE802A6 57FF002E 93DF00D0 7FDA02A6
SingleStep> write -b -f 0x12 -c 7 0x410
SingleStep> write -b 0x412=0xAB
SingleStep> read -l -m 18 0x400
(0x400)
00000400  7FF343A6 7FE802A6 7FF243A6 48000005
00000410  1212AB12 1212122E 93BF00CC 93DF00D0
00000420  7FB242A6 7FD342A6 93BF00E4 93DF00D4
00000430  7FBA02A6 7FDB02A6 93BF00EC 93DF00F0
00000440  7FA00026 93BF00D8 73DE0032 7FA000A6
00000450  7FBD378 83DF00C0 7FBD378 7FA00124
00000460  4C00012C 57FE047E 67DE0001 83BF00C4
00000470  93DF00FC 7FA803A6 3BFF00C8 4E800020
00000480  7FF343A6 7FE802A6 7FF243A6 48000005
00000490  7FE802A6 57FF002E 93DF00D0 7FDA02A6
000004A0  93DF00EC 7FDB02A6 93DF00F0 9421FF48
000004B0  93A10084 83BF00D0 400C2022 00000000
000004C0  00000000 000B81E0 41245623 00000008
000004D0  811E0010 0012C264 40000000 00604000
000004E0  00A89440 000B85A4 42C008A0 811E0010
000004F0  00001032 84100400 0C080004 008A0044
00000500  7FF343A6 7FE802A6 7FF243A6 48000005
00000510  7FE802A6 57FF002E 93DF00D0 7FDA02A6
SingleStep>

```

FIGURE 9-6 Output of the read Command

2. Fill an area of memory with 0x12 by entering:

```
write -b -f 0x12 -c 7 0x410.
```

The **-b** directs the **write** command to operate on byte (8-bit) elements. Each byte in the range of 410 through 417 is now set to 0x12.

NOTE: The address range may be unique to each board. Check for a valid address range.

3. Now set one byte in this range to a different value by entering:

```
write -b 0x412=0xAB
```

This sets the byte at location 0x412 to AB Hex.

4. View the results of the write commands by entering:

```
read -l -m 18 0x400
```

5. If your target is not responding, complete [step 6](#) through [8](#). If your target is responding, go to [step 9](#).
6. Press the reset button on your target board.
7. From the pRISM+ Manager toolbar, click the **Reset** button.
8. Download the application again by selecting **File → Debug** and clicking **OK**.
9. Click on the **Close** button in the Debug Status Window after the download is completed.

9.2.6 Source, Mixed, and Disassembly Display Modes

SingleStep Debugger supports three display modes while you are debugging:

- *Source*

In source-level mode, you debug code at the C/C++ language level, so the Code window shows the C/C++ language source code.

NOTE: When in source-level mode, a single **Step** command lets you execute one or more C/C++ language statements.

- *Mixed*

In mixed mode, you are shown assembly language with the corresponding high-level source statements interspersed.

- *Disassembly*

In disassembly mode, debugging is at the assembly-language level, so the Code window shows assembly-language code.

Executing C/C++ Statements One Line at a Time

The source window now displays source code for the `ROOT` task. The SingleStep Debugger for pRISM+ has highlighted the opening brace of the `ROOT` task, which is the current point of execution. (When control is entering a procedure, SingleStep highlights the opening brace.)

Two commands can be used to step through either source lines or machine instructions. These are:

- **Step**
- **Step In**

The difference between **Step** and **Step In** is that **Step In** steps into subroutines, and **Step** executes entire subroutine calls and halts when the called subroutine returns.

Step Command

1. You can execute C/C++ statements one at a time by pressing the **F10** button or selecting **Step** from the **Run** menu.
2. Repeat the **Step** command until the line containing the subroutine call `tm_set()` is highlighted (which may require more than one **Step**).

The highlighted line moves down because you are single-stepping lines of executable code. The complexity of the code determines whether the SingleStep Debugger requires more than one step to complete a single line.

NOTE: In some cases, SingleStep may appear to execute several lines of C or C++ code with a single **Step**. This is a result of compiler optimizations.

3. Single-step again and repeat until you are on the `de_init()` (this is another assembly routine).
4. Now switch to mixed mode by selecting **M** button on the Debug window.

The source window display changes, showing that the instructions making up the current source line consist of preparation for the call (argument passing), the actual subroutine call, and maybe some cleanup after the subroutine call, depending on the target processor architecture.

Step In and Go Until Command

The **Step In** command single-steps either source lines or machine instructions, according to the debugger mode. **Step In** can be invoked either by selecting it from the **Run** pull-down menu, clicking StepIn button or by pressing **F8**.

1. Press the **F8** key several times, until the actual assembly-language subroutine call (e.g. `jsr` on a 68k, `bl` on a PowerPC) is highlighted.
2. Press **F8** once more, and the first instruction of the subroutine should be highlighted.

3. Now return to high-level mode by selecting **Source** from the mode selection bar in the source window.

SingleStep either switches back to source code mode, or continues to display assembly-language, depending on the target processor you are using. This occurs because it is not always possible to trace back up the call chain from the first instruction of a subroutine. In this case you may get out of the called subroutine and back to C code by using the **Step Out** command from the **Run** menu. If necessary, try it now, and you should return to C source code.

The **Go Until** command allows you to set a temporary breakpoint and resume execution of the application.

1. Select **Go Until** from the **Run** menu.
2. In the dialog box that opens, specify `root#166` as the location for the temporary breakpoint, and click the **OK** button.

SingleStep should break at line #166.

Querying System Objects

SingleStep offers many features that help you perform “kernel aware” debugging. You can:

- Examine the state of other kernel objects and the pSOSystem configuration.
- Examine the state of the currently executing task.
- View into the internal kernel data structures.
- Debug your application using a command line interface.
- Set task-specific breakpoints.

1. Select **Kernel Objects** from the **Data** pull-down menu. See [Figure 9-7](#).

The pSOS+ Kernel Objects and Configuration window is displayed:

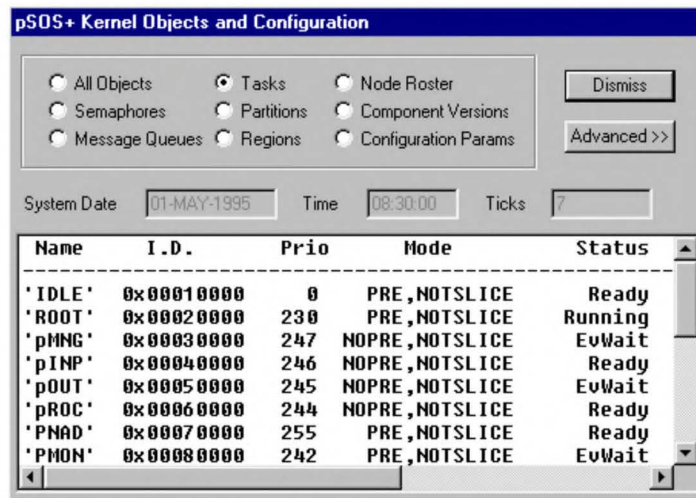


FIGURE 9-7 Tasks Displayed

A list of all of the tasks is shown, including information about each one. Notice that task 'ROOT' is currently running, and that all of the other tasks are either ready or blocked for some reason (for example, waiting for events).

You can view other system objects by pressing the appropriate radio button in the kernel objects window.

2. Click the **Semaphores** radio button.

The list of Semaphore system objects is displayed in [Figure 9-8](#).

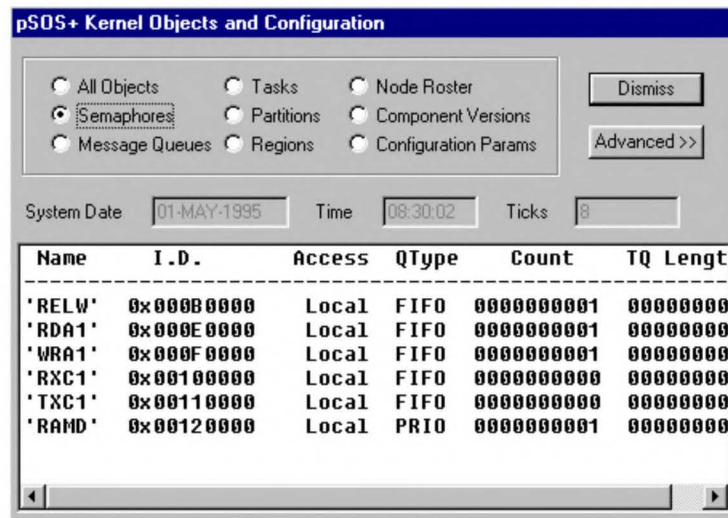


FIGURE 9-8 Semaphores Displayed

3. Now click the **Message Queues** radio button.

The list of Message Queue system objects is displayed in [Figure 9-9](#).

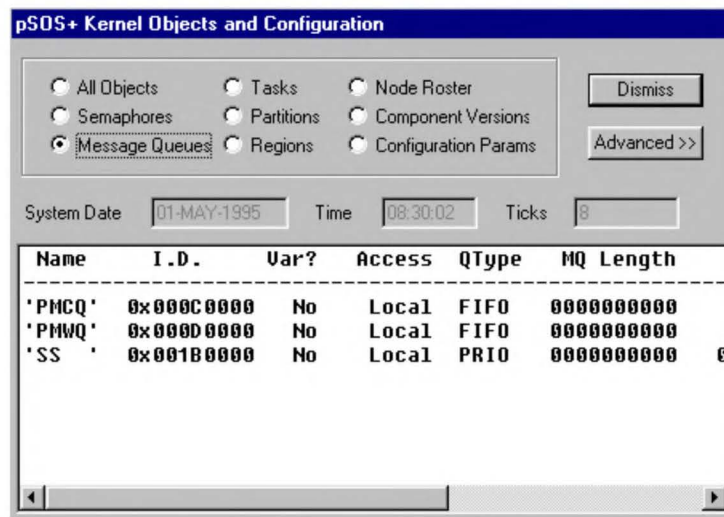


FIGURE 9-9 Message Queues Displayed

4. Now close the Kernel Objects window by clicking on the **Dismiss** button.
5. Execute the application until just past the point where another message queue has been created by selecting **Go Until** from the **Run** menu.
6. Enter `root#190` in the dialog box, and click **OK**.
7. When execution stops at line #190, look at the list of message queues again by selecting **Kernel Objects** from the **Data** pull-down menu (you may need to click the **Message Queues** radio button again).

Message queue "QMEM," created by the system call on line 182, is now displayed as well.

8. Click on the **Dismiss** button to close the **Kernel Objects and Configuration** window.

Reading a Variable

To read the value of program variables, complete the following steps:

1. In the Source window, double-click on the 'seg_ptr' variable on line 190.
2. Select **Read** from the **Data** pull-down menu so that the **Read** window appears.

This method can be used to display the value of a local variable in the function currently displayed in the **Read** window:

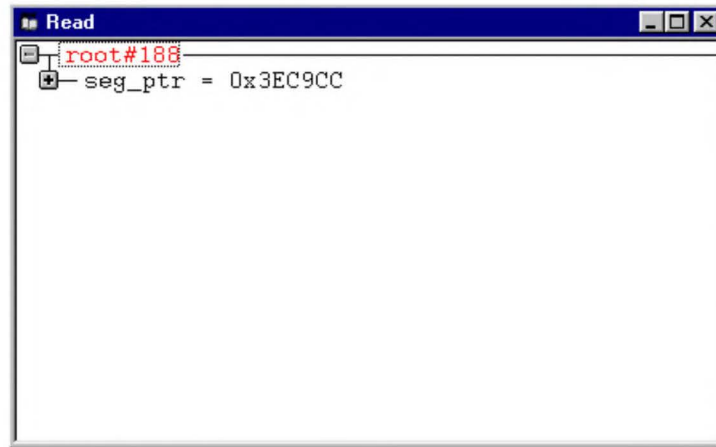


FIGURE 9-10 Read Window

3. Close the **Read** window.

Displaying a Variable

You can also control the information displayed for a variable.

1. In the **Stack** panel, select the `nbufs` variable and then click the right mouse button anywhere in the window.

A pop-up menu appears.

2. From the pop-up menu, select **Properties**.

The "Display Properties for `nbufs`" window is displayed.

3. Select the **Format** tab, turn on the **Address** checkbox, and click the **OK** button.

The **Locals** window should now show the address of variable `nbufs` as well as its value, as shown in [Figure 9-11 on page 9-16](#).

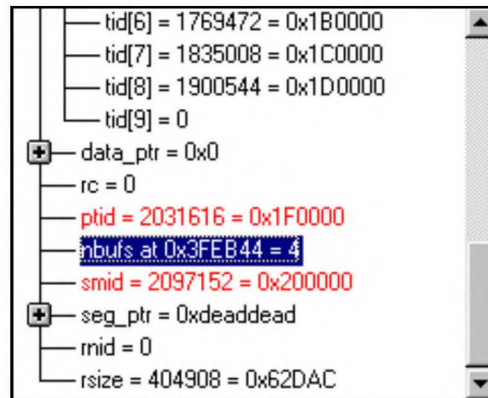


FIGURE 9-11 Locals Window

Modifying a Variable

To modify a variable, complete the following steps:

1. Select **rc** in the **Stack** panel, and then right click and select **Modify** from the popup menu that appears.

A dialog box opens where the value of **rc** can be changed.

2. Enter any new value and click **OK** when done.

Watching a Variable

A watched variable is displayed in the Watch window. SingleStep updates the value of a watched variable each time control is returned from the target to the debugger. In contrast, the Read window gives the value of the variable at a point in time but the value is not updated as the execution of the program continues.

To demonstrate the difference:

1. In the Source window, double-click on **rsize** (on line 192).
2. Select **Watch...** from the **Data** menu.
3. Select **Read...** from the **Data** menu.

Notice that both the Watch and Read windows show the same value for **rsize**.

4. Execute three lines of source by hitting **F10** three times.

Notice that the value of **rsize** changed in the Watch window but not in the Read window.

NOTE: You can control the information displayed for watched variables just as you did for read variables. Just click the right mouse button in the Watch window and select **Properties...** from the pop-up menu.

This concludes the SingleStep Debugger for pRISM+ tutorial.

5. To complete your tour of the pRISM+ tools, return to the [Quick Start with a Tutorial](#) chapter.

NOTE: For additional information on the SingleStep Debugger, refer to the *SingleStep User Guide* on the pRISM+ Documentation CD-ROM.

10

ESp

A pSOSystem application usually consists of several tasks. When the application is executed, these tasks can be blocked, waiting, or pre-empted by higher priority tasks. These tasks request resources by means of semaphores and queues. In this situation, untangling the interaction using a standard debugger can be difficult. ESp helps you visualize the tasks interaction.

ESp displays the interactive behavior of groups of tasks and events. Color coding shows whether a task is running, idle, blocked, or waiting for a semaphore. You can visually follow context switches, task-state transitions, interrupts, system calls, and other major application activities.

ESp also enables you to view real-time stack and CPU usage, context information, and user-defined events.

ESp views your application as a series of pSOSystem-specific events. ESp collects events as your application runs on the target in a session that you start and stop. It can collect any of the following events:

- Service calls — all pSOS+, pHILE+, pNA+, pRPC+, pSE+, pTLI and pSKT system calls.
- I/O calls — `de_init`, `de_open`, `de_cntrl`, `de_read`, `de_write`, and `de_close`.
- Context switches — transitions of tasks from and to the Running state.
- Interrupts — Entry and exit from interrupt handles.
- User events — Events defined by the user in their application.

The session in which ESp collects events is called an *experiment*. Before you begin an experiment, you must create an *experiment configuration* that tells ESp which events to collect and how to collect them. You create the experiment configuration in the Configuration Window.

After you run an experiment, you can study the events in ESp's main window, the Analyzer Window. The Analyzer window displays the execution thread of your application with icons that represent different events. In addition, it displays profiles of task, semaphore, and region use.

In a typical ESp session you will run an initial experiment to identify the general area of a problem. You will run more experiments and fine-tune your collection of events, enabling you to narrow down the area of analysis.

ESp stores all the information for each experiment — the configuration information and the collected events — under a single name that you provide in the Configuration window. As you run successive experiments, ESp asks you to provide a new name for each experiment. You may want to name the experiments sequentially; for example `exp1`, `exp2`, etc.

ESp expects you to create a new configuration each time you run an experiment, but you can use the same configuration repeatedly as long as you provide a new name for each new experiment.

10.1 ESp Prerequisites

Before you can use ESp, your application must meet the following prerequisites:

1. Your application must be built with the pMONT+ target agent enabled.
2. ESp uses Serial or Ethernet to communicate to the target. ESp needs its own serial channel for communication to the target.
3. Your application must be compiled, downloaded and running on your target board. Refer to [Chapter 3, *Quick Start with a Tutorial*](#) for instructions.

10.2 Placing User-Defined Event in the Application

The user event call `log_event()` is an option that lets you be more specific when you determine the events that begin and end data collection. If, for example, you are not sure about which area of an application is creating a problem, you can insert user event calls to begin and end a data collection run around the suspected

portions of code. During normal operation, the call has a negligible effect on system overhead.

10.3 Refining Data Collection Needs

The appropriate specification of events and buffer management is crucial to making data analysis concise and effective. For example, with an overly general event specification, the Analyzer window shows an unnecessarily large amount of information. Therefore, the sections that follow describe the effects of various buffer management schemes and event specifications.

10.3.1 Buffer Management

The allowable buffer management overhead and the frequency of application problems influence your buffer management choice. You may want to collect only a small amount of data, or you may need pMONT+ to monitor the application for days to capture the relevant events. In descending order of overhead, the most complex buffer management scheme is Transmit, then Wraparound, and finally, Halt on Buffer Full. The buffer management choice should be appropriate to your analysis goals.

Transmit

The Transmit buffer option has the most impact on application behavior because it is the only option with a periodic update mechanism. This causes pMONT+ to communicate with the host system during the data collection run (if it does not halt first because of other factors). Because it consumes a greater amount of system resources, you should select Transmit buffer only when necessary, such as when you need a large amount of collected data.

With the Transmit buffer management option, you should be aware of the influence of the communication medium and event parameters. Ethernet does not create a problem for pMONT+. For a serial connection, however, a lower baud rate (below 19.2 Kbps) can keep pMONT+ from transmitting to the host fast enough if pMONT+ rapidly logs events. As the following subsections explain, the choice of events affects the rate at which pMONT+ logs events. If pMONT+ cannot transmit events fast enough because of a slow serial connection, it stops collecting data. The ESp tool then posts the following message to the console:

```
Experiment ended because buffer full.
```

If you must use a serial connection at a low baud rate, try to specify events efficiently as described in [Section 10.3.2, *Event Specification*](#).

Wraparound

The Wraparound option is useful for analyzing program errors because it directs pMONT+ to capture the events leading up to an exception. For example, if an undesirable condition occurs once every few days, you can select Wraparound so that pMONT+ captures only the one buffer that surrounds the event.

The Wraparound option is much less intrusive than Transmit because it requires pMONT+ to transmit to the host once — when data gathering ceases. While the data gathering proceeds, pMONT+ continuously overwrites the buffer until the end trace event or other termination occurs. If you want the data collecting process to run without pMONT+ reporting to the host until the run is terminated, select Wraparound.

Halt on Buffer Full

The simplest buffer management option is Halt on Buffer Full. If you want to examine the program's execution within a specified window of events, use this option.

10.3.2 Event Specification

Whenever possible, you should configure the data collection so it provides the needed information with the least amount of overhead. Otherwise, the result may be an inaccurate picture of what the application is doing. With a more complex event specification scheme, pMONT+ intrudes more on the application, because it continuously checks an application against the criteria specified in the data collection configuration. Furthermore, for a finely tuned application, the degradation that pMONT+ overhead causes to the application's performance is more significant.

A group of parameters requires less overhead than a specification with the same number of parameters you specify individually. For example, if you do not want to log any level of ISR, it is more efficient to specify ISR (for all ISRs) in Events to Ignore than to specify each ISR level for pMONT+ to ignore.

Begin Trace Events

Before event collection actually begins, pMONT+ looks for only a begin trace event. In general, the best approach is to specify the least number of begin trace events.

End Trace Events

After event collecting actually begins, pMONT+ checks for end trace events, the events to log, and the events to ignore, so you should consider your choices for these events in relation to overhead.

Center Trace

The effect of Center Trace (a feature of End Trace Events) is to log the events surrounding an end trace event. Its purpose is to help reveal what happened around the end of a data collection run.

Events to Log

To help minimize the overhead created by Events to Log, try using the Events to Ignore specification as a complement to Events to Log. See [Events to Ignore on page 10-5](#).

Events to Ignore

Events to Ignore overrides duplicate specifications in Events to Log, so be sure not to cancel event logging you really want in Events to Log. On the other hand, you can complement the Events to Log by using Events to Ignore to make data collecting more efficient. For example, if an application has six interrupt levels but you want to log only five, specify all interrupt levels in Events to Log and one level in Events to Ignore. This is more efficient than specifying each interrupt level in Events to Log, because pMONT+ checks for two conditions in the former approach and five conditions in the latter.

10.4 Tailoring the Configuration Table

When you consider the size of the pMONT+ buffer (`traceBuffSize`), make sure it is large enough to accommodate the scope of information you want your application to supply to pMONT+. The `traceBuffSize` value is part of the `typedef struct`. You set this parameter by changing the value of `PM_TRACE_SIZE` in the `sys_conf.h` file.

10.5 Tailoring the Application's Stacks

If an application's stack size is such that it can be pushed very close to its size limit, pMONT+ may detect an error. If pMONT+ detects a write operation to either the highest or the lowest eight bytes, pMONT+ flags it as a corrupted boundary (but only if Enable Checking is selected in the ESp Stack menu; otherwise, corrupted boundaries are undetected). Therefore, you may want to consider setting a slightly larger stack size within your application if peak stack usage tends to be at or near the stack size limit.

10.6 Post-Mortem Analysis in ESp

The post-mortem analysis capability in ESp allows you to capture the events leading up to a fatal error or target crash. You can later analyze this data with ESp to pinpoint the problem. The following are the steps to collect post-mortem data for a target crash:

1. In ESp, select **File** → **New Experiment**.
2. In the **Configuration window**, click on **Wrap Around** so that the data is continuously collected in a wrap around buffer on the target.
3. Select **Collection** → **Start Now** or **Collection** → **Start at Reset**.
4. If you selected **Collection** → **Start at Reset**, re-initialize pSOS+ and run the target application.
5. When you suspect the target application has crashed, stop the experiment by selecting **Collection** → **Stop**.
6. ESp will try to communicate with the target to stop the experiment. Since the target application has crashed, this operation will fail. ESp will report that the experiment is aborted. Click the **OK** button.
7. DO NOT exit ESp.
8. Soft-reset the target. Press the target board's reset button.

CAUTION: DO NOT power off or on.

9. Setup the target (i.e., load the application, boot pSOS+, and initialize pSOS+) and run the application, as described in [Chapter 3](#).
10. Go back to ESp and select **File** → **New Experiment**.
11. ESp will detect that the previous experiment was aborted. It will ask you if you want to get any unrecovered experiment data from the target.
12. Click on the **Yes** button.

ESp will recover the post-mortem data and bring up the Analyzer window so that you can analyze the crash. If the target reset corrupted pMONT's experiment buffer, the post-mortem data is not available. In this case, ESp will display an error message that the experiment buffer was corrupted.

11

Object Browser

Object Browser is a run-time analysis tool. It monitors target behavior by taking periodic snapshots of the operating system objects on the target while the target system is running. Information on OS objects such as tasks, message queues, semaphores, and other critical information such as stack and memory usage can be displayed graphically. This gives a sampled view of the target run-time behavior over time.

Two intuitive graphical display modes exist:

- The Snapshot View is best suited for displaying run-time attributes of system objects, for example, run-time status and configuration parameters of a task.
- The alternative, Graph View, is best used to display the level of usage, for an example, each task's stack usage as a percentage of its own maximum allowed stack size.

From these intuitive graphical displays, users can easily spot problems such as stack overflow or memory leak over time.

Each collection of data obtained from the running target system can either be stored in Object Browser and compared with past or future samples or exported to standard desktop tools such as Microsoft Excel for documentation purposes.

You can use Object Browser to analyze the runtime behavior of your target system after you download and execute your application on the target. The following are examples of what you can use Object Browser to learn about in your application:

- Error conditions, such as stack overflows, stack underflows, memory leaks, and deadlocks (See [Monitoring for Stack Problems on page 11-4](#), [Finding Memory Leaks on page 11-4](#), and [Checking for Deadlocks and Priority Inversion on page 11-5](#).)

- Operating system object status such as information on: tasks, regions, partitions and semaphores

You can also use Object Browser to learn an unfamiliar application. Without viewing the source code, you can start immediately to look at the runtime behavior to understand how an application works.

When multiple programmers work on the same project, each can use Object Browser to view the rest of the application and determine whether all the parts are synchronized well.

pSOSystem objects you can monitor

Using the corresponding snapshot or graph page, you can monitor the following pSOSystem objects:

Tasks	The smallest unit of execution that can compete on its own for system resources. The pSOSystem application is made up of a series of tasks. The task can be viewed in snapshots only.
Stacks	The memory allocated to each pSOSystem task. The stack can be viewed in graphs only.
Semaphores	A mechanism for inter-task and task-ISR synchronization that is commonly applied to the producer-consumer problem, and the problem of controlling access to shared resources. It is defined to be a counter with an associated task-wait-queue.
Regions	A user-defined, physically contiguous block of memory. Tasks allocate memory segments from regions.
Partitions	A user-defined, physically contiguous block of memory divided into a set of equal-sized buffers. Tasks allocate buffers from partitions.
Queues	A flexible, general-purpose mechanism for tasks to synchronize and communicate with each other. Tasks send and receive messages from queues.
Mutex	A synchronization primitive used to provide mutual exclusion among tasks by serializing access to the critical regions of the code. It is similar to a binary semaphore. It also provides the ability to prevent unbounded priority inversion.

Cond. Var. A general purpose synchronization primitive that provides a sleep-wakeup or signal-wait mechanism. A condition variable has an associated user-defined condition.

The flexibility of binding any user defined predicate with a condition variable makes it a very powerful primitive for building complex synchronization mechanisms.

It operates in conjunction with a mutex so that the evaluation and alteration of the predicate, and signaling/waiting for the predicate can be performed as an atomic operation, thereby avoiding the races inherent in implementing such synchronization mechanisms on a pre-emptible multi-tasking system.

Object Browser target overhead

Object Browser communicates with the pMONT+ target agent to obtain the information from your target system. This operation uses CPU time on the target. The amount of CPU used depends on which objects are monitored and the update rate.

Object Browser Prerequisites

Before you can use Object Browser, your application must meet the following prerequisites:

- Your project application must have pMONT+ target agent as part of its components.
- Object Browser uses Serial or ethernet to communicate to the target.
- Your application must be compiled, downloaded and running on your target board. Refer to the [Quick Start with a Tutorial](#) chapter for instructions.

11.1 Monitoring for Stack Problems

11.1.1 Stack Problem Setup

Stack overflows are among the most difficult problems for the real-time developer. With the Object Browser, the stack utilization of tasks can be monitored. If a problem occurs, the Object Browser will show it. To monitor for stack problems complete the following steps:

1. Complete the Object Browser prerequisites, [Object Browser Prerequisites on page 11-3](#).
2. From the pRISM+ Manager, click on the **Object Browser** button.
3. From the Object Browser toolbar, click on **View** → **Snapshot** → **Stack Problems**.

The **Stack Problems** window will appear.

4. In the Snapshot window, right-mouse click to display the pop-up menu. Click on the **Update Pages** menu. Verify that the **Update Stack Problems** menu item has been selected. The **Update current page** and **Update Stack Problems** menu items are selected by default.
5. Start sampling.

In the **Stack Problems** window, you can monitor any stack overflow issues.

11.1.2 Understanding Your Stack Graphics Data

This section describes how to analyze the Stack Graphics data for the PowerPC and 68K processors.

If a task is created with Supervisor Stack = X and User Stack Size =Y, Object Browser returns the stack information as Supervisor Stack Size = X + Y and the User Stack Size = 0. For additional details on stack usage information, refer to the *pSOSystem System Calls* manual.

11.2 Finding Memory Leaks

The Object Browser can display the amount of free memory in various regions. Since all systems have a Region 0 (required), that is often where programs will go for temporary needs. You can monitor the free space in Region 0 and, if you notice its slow decline through the use of the **Update All Pages** option as well as select other

items of interest in your system to determine the cause and effect relationship. To locate memory leaks complete the following steps:

1. Complete the Object Browser prerequisites, [Object Browser Prerequisites on page 11-3](#).
2. From the pRISM+ Manager, click on the **Object Browser** button.
3. From the Object Browser toolbar, click on **View → Graphs → Stack**.

The **Stack Usage** Graphs window appears.

4. In the **Periodic Update** area of the **Stack Usage** window, set the parameters to begin polling the **Stacks Usage** page.
 - a. Select **Update All Pages** option.
 - b. In the **Sample Every (Seconds)** field, click on the arrow to increase the sample time to 8 seconds.
 - c. Click on the **Start** button to begin the sampling of your running application.

You can monitor the free space in Region 0.

11.3 Checking for Deadlocks and Priority Inversion

11

A deadlock is a situation in which two tasks are unknowingly waiting for resources that are held by each other. You can use Object Browser to examine the behavior of your tasks and queues. The following procedures provides a brief scenario that will assist you in understanding how you can possibly detect if your application has deadlocks or priority inversion situations.

1. Complete the Object Browser prerequisites, [Object Browser Prerequisites on page 11-3](#).
2. From the pRISM+ Manager, click on the **Object Browser** button.
3. From the Object Browser toolbar, click on **View → Snapshot → Queue**.

The **Queues** Snapshot window appears.

Examining Messages in the Queue

4. Click on the + icon on each queue to observe the number of messages in your application's queues.

5. In the Snapshot window, right-mouse click to display the pop-up menu. Select **Update Page** menu. Verify that the **Queues** menu item has been selected.
6. In the **Periodic Update** area of the **Queues** window, set the parameters to begin polling this page.
 - a. Select **Update Roster** option.
 - b. In the **Sample Every (Seconds)** field, click on the arrow to increase the sample time to 8 seconds.
 - c. Click on the **Start** button to begin the sampling of your running application.

Examining Tasks Waiting for Messages

7. Click on the + icon on each queue to observe the number of messages in your application's queues.
8. If the queue indicated it had more than 0 messages, click on the + icon on **Number of Tasks Waiting for Messages**. Observe which tasks are waiting for message from which queue.

NOTE: An increased number of messages in a queue sometimes signifies that deadlock situation might have occurred.

9. Click on the **Snapshot History** arrow to compare the object's (queue, task, and message) status in the various snapshots.
 - a. Are the objects behaving as expected?
 - b. Are the objects waiting too long for a resource?
 - c. If there is a problem with an object's behavior then deadlock or priority inversion has occurred.
 - d. Use a pRISM+ debugger or a pRISM+ Project Editor to examine and locate this problem in your application.
 - e. Correct and compile your application and complete these steps again.

11.4 Logging Data in the CSV Files

Object Browser logging is done in `.csv` (Comma Separated Value) format, which any editor capable of supporting this format can view. You can use Microsoft Excel to reformat this file to aid in analyzing or presenting your data. You can also use a typical text editor to view the log data.

1. Complete the Object Browser prerequisites, [Object Browser Prerequisites on page 11-3](#).
2. From the pRISM+ Manager, click on the **Object Browser** button.
3. From the Object Browser toolbar, click on **View → Graphs → Queue**.

The **Queues** Graph window appears.

4. Right-click anywhere in the graph window and select **Log in CSV** format from the popup menu. This action saves the graph samples as text in a CSV file.
5. In the **Periodic Update** area of the **Queues** window, set the parameters to begin polling the **Queues** page.
 - a. Select **Update All Pages** option.
 - b. In the **Sample Every (Seconds)** field, click on the arrow to increase the sample time to 8 seconds.
 - c. Click on the **Start** button to begin the sampling of your running application.
6. Using Microsoft Excel or another text editor, open the CSV file.
7. Repeat the same procedure for Snapshot Frame.

11.5 Selective Logging of Data in Graph Frame

1. Click on options button in toolbar.
2. Select the objects (stack/queue/region) and the condition for displaying and logging data.
3. Start sampling.

12

Run-Time Analysis (RTA) Suite

The Run-Time Analysis Suite draws on information from Diab Data's D-CC and D-C++ compiler suites and the target application to provide the critical insight needed by each developer to improve program performance, reliability, and memory usage in advanced 32-bit applications.

For additional information on this optional product, refer to the RTA Suite *Visual Run-Time Analysis Tools User Guide*.

12.1 Overview

12

12.1.1 Run-Time Error Checker

Compiler options generate code to catch invalid pointer references, out-of-bounds array references, stack overflow, memory leaks, and other memory-related errors. When code is run in the interactive RTA, double-clicking on an error message opens the source file at the error.

12.1.2 Visual Interactive Profiler

Analyzes profile data collected from instrumented code run on the target, and displays tables and charts showing function timing and call counts, line counts, and code coverage. This tool can be accessed from the pRISM+ Editor and SNIFF+.

12.1.3 Link Map Analyzer

Displays a linker command file in three ways: text, tree, and maps. Graphically displays memory setup to precisely locate code and data. This tool can be accessed from the pRISM+ Editor and SNIFF+.

12.1.4 Stack Use Analyzer

Reports maximum stack depth and the functions called to reach it by combining static analysis of the target executable with data from profiling runs.

The pRISM+ Shell provides multiple levels of services to you by the means of TCL, Tool Command Language. For many applications, you will probably only need the interactive pSOS-aware commands. For some applications, you might use the scripting capability of TCL. For a few applications, you could use the ability to talk to pRISM+ CORBA services directly to allow dynamic interpretation of CORBA requests. In some instances you can use the pRISM+ shell's modified version of TCL/CORBA commands to create and run TCL/CORBA based scripts.

The levels of service are:

- Interactive pSOS-aware commands

This includes commands that communicate to pROBE+ by the means of the communications server and for targets that support it, commands that communicate to pROBE+ through the communications and debug servers.

These commands supplement the GUI tools for debugging for example, Searchlight and SingleStep. Commands at this level are typically run one command at a time and in an interactive manner. See Appendix C for the complete list of commands.

An example of using commands to modify pROBE+ communication parameters is provided in [Section 13.1.2, *Modifying Communication Timeouts*](#) on page 13-3. Other typical uses of these commands are to display information about pSOS objects and to modify their values.

For more information, refer to [Section 13.1.4, *Using pRISM+ Shell with Searchlight Debugger*](#) on page 13-5 and [Appendix D, *pRISM+ Shell Commands*](#).

- **TCL scripts**

This pRISM+ shell level is used to write scripts of commands for example to perform menial tasks or to automate testing. In addition to the pSOS-aware commands provided by pRISM+, standard TCL built-in commands are used to handle program control-flow, to assign variables, to do input/output etc.

Example TCL scripts are provided. For more information, refer to *Using and Invoking a pRISM+ Shell Tcl Script* on page 13-8 and *pRISM+ Shell Commands*. Refer to one of the numerous TCL textbooks for more information about TCL scripting.

- **Low-level TCL/CORBA services**

The pRISM+ Shell also allows any CORBA service to be called from TCL. In this case, it is possible to write TCL scripts that directly call the communications or debug server IDL interfaces. This level is only intended for advanced tool customization and is not normally needed to develop pSOS applications.

The interactive pSOS-aware commands, provided in TCL source form in the pRISM+ installation directory, uses this TCL/CORBA mechanism and may be seen as examples for how to use the underlying services. For more information, refer to *pRISM+ Shell Commands* appendix.

- **Interactive host commands**

The pRISM+ Shell passes unknown commands through to the underlying default host shell. For example, if on UNIX platform the `ls` command is run in the pRISM+ Shell, it is passed to the underlying shell. Then the resulting output is passed back to the pRISM+ Shell for display. Using the interactive host commands are useful you need to run commands in the same execution context as pRISM+ for example, the environment variables like `PSS_ROOT` etc. See the numerous TCL textbooks for more information about the use of TCL as a general shell.

13.1 Using Interactive pSOS-Aware Commands

In this section you will see different examples of how to use the pRISM+ Shell's with interactive pSOS-aware commands. You will see three variations on how to use the pRISM+ shell at this level.

In this section you start at simple usage to a more complex usage. You will learn how to:

- Obtain information about pSOSystem objects.
- Use communication commands for troubleshooting purposes.
- Debug your application with SearchLight and pRISM+ Shell.

13.1.1 Obtaining Status of a pSOS Object

The simplest and most commonly used pRISM+ Shell command is `show` command. You can use this command to obtain information about your tasks, queues, semaphores, mutexes, and conditional variables. With this version of the pRISM+ Shell you do not always need the ID number of the object you want to see. You can now call the object by name. For example:

If you want the status of the active tasks, type the following. An example of the results is also shown:

task show

Name	ID	Priority	Susp	Status	Parameters	Ticks
IDLE	0x00010000	0x00000000	NO	Ready		
ROOT	0x00020000	0x000000E6	Yes	Ready		

If you want the status of the `PROBE` flag settings, type the following. An example of the results is also shown:

probe show

```

RBUG Flag is ON
No Dots Flag is ON
No Manual Break Flag is OFF
No Page Flag is OFF
Profile Flag is OFF
Silent Mode Flag is OFF
Current Interrupt Level = 1
Default Interrupt Level = 1

```

13.1.2 Modifying Communication Timeouts

A more complicated method of using the pRISM+ Shell is in troubleshooting. You can use the pRISM+ Shell to modify your timeout commands. You might use these commands when the pRISM+ Manager reports communication problems. Special


communication timeout commands are available for times when heavy network traffic causes errors such as `Target Not Responding`.

When the Communication Server sends a packet to pROBE+, it expects an acknowledgment from pROBE+ indicating that pROBE+ received the packet.

This acknowledgment must arrive within the time specified by the acknowledgment timeout parameter (`acktimeout`). If the packet does not arrive in time, the Communication Server assumes that pROBE+ never received the packet and the packet is resent. The number of times the communication resends the same packet is determined by the `retries` parameter.

After the Communication Server gets the acknowledgment from pROBE+, it expects pROBE+ to process the request and return the result within the `replytimeout` period. If the reply does not arrive in time, the Communication Server assumes the connection to the target is down.

The following steps show how you can redefine communication timeouts to try avoid error communication error messages for example, `Target Not Responding`:

1. Setup and invoke pRISM+. Refer to the pRISM+ Tutorial in the pRISM+ Getting Started chapter.
2. To access the pRISM+ shell, click on the Shell  button from the pRISM+ Manager. A DOS-like window will appear.
3. To display the current settings, in the pRISM+ shell use the following syntax:

```
session open
debugger show
```

The current settings for `retries`, `timeout`, and `acktimeout` are displayed.

4. Set the `retries` number:
5. Verify that the `retries` was modified.


```
debugger show
```

This will display the current settings for `retries`, `timeout`, and `acktimeout`. Other related debugger communication commands include:

```
debugger set timeout
debugger set acktimeout
```

13.1.3 Downloading a pSOS+ Executable

If you want to use the pRISM+ Shell to download, boot, and initialize your pSOS executables.

1. Setup and invoke pRISM+. Refer to the pRISM+ Tutorial in the pRISM+ Getting Started chapter.
2. To access the pRISM+ shell, click on the Shell  button from the pRISM+ Manager. A DOS-like window will appear.

3. In the pRISM+ Shell, type

```
dssession open
```

This opens the communication to the debug server. OK will display at a successful completion.

4. In the pRISM+ Shell, use the following syntax:

```
dssession load C:/File_Path/ram.elf all
```

This will begin the downloading process of your ram.elf executable. OK will display at a successful completion.

5. In the pRISM+ Shell, type

```
boot
```

This boots your executable that is now on the target. OK will display at a successful completion.

6. In the pRISM+ Shell, type

```
initialize
```

This initializes your executable that is now on the target. OK will display at a successful completion.


You are now ready to run and debug your psos application.

13.1.4 Using pRISM+ Shell with SearchLight Debugger

The pRISM+ shell allows you access to the pSOSystem calls within the pROBE+ context. You can use this special pROBE+ access with the SearchLight debugger. The following steps provide an example of how this can be done. You can use the pRISM+ Shell separately with all the supported processors.

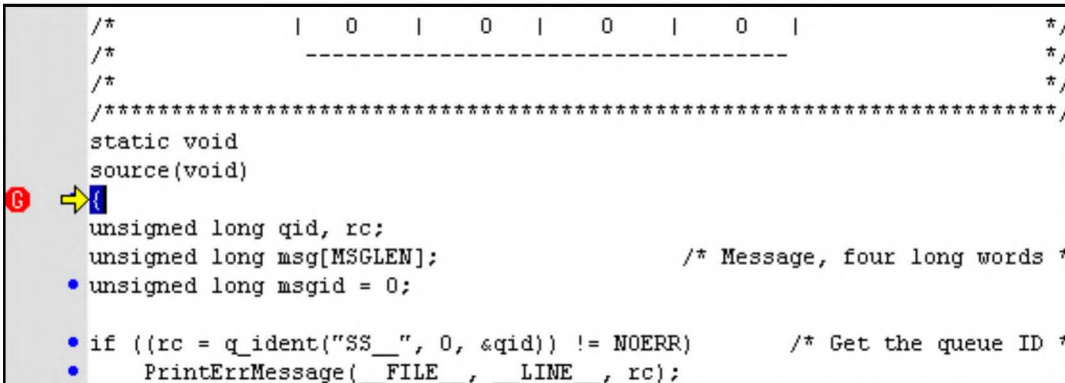
NOTE: Before you can begin the steps in this tutorial you must complete the pRISM+ Tutorial in the *pRISM+ Getting Started* chapter.

Accessing the pRISM+ Shell and Setting Up Your Project

1. To access the pRISM+ shell, click on the Shell  button from the pRISM+ Manager. A DOS-like window will appear.
2. From the pRISM+ Manager, select **File** → **proj1**. This project was created during the pRISM+ Tutorial.
3. From the pRISM+ Manager, select **target1** from the Target List pull-down menu. This target was defined during the pRISM+ Tutorial.

Accessing SearchLight and Setting Up Your Application

1. From the pRISM+ Manager, click on the **Debug** button to invoke the SearchLight debugger.
2. From the SearchLight window, click on **File** → **Load**. A status box will display. The download is complete when the status box displays: Download Complete and OS boot message.
3. From the SearchLight source window, scroll through the application and locate the line `void source (void)`. Place your cursor in this line, click on the **brkPnt** button. A red circle appears identifying the location of the breakpoint as shown in Figure 13-1:



```

/*          | 0 | 0 | 0 | 0 |
/*          -----
/*
/*****
static void
source(void)
6  → {
    unsigned long qid, rc;
    unsigned long msg[MSGLEN];           /* Message, four long words */
    • unsigned long msgid = 0;

    • if ((rc = q_ident("SS_", 0, &qid) != NOERR)      /* Get the queue ID */
    •   PrintErrMsg( FILE , LINE , rc);

```

FIGURE 13-1 void source Breakpoint

4. From the SearchLight tool bar, click on **View** → **pSOS Objects**. The **pSOS Objects** dialog box will display.
5. In the **pSOS Objects** dialog box, click on the **Queue** tab.
6. Click on the **Run** button. The application will run until it reaches the breakpoint set in step 4.

NOTE: By running the application you can see a group of tasks and queues were created. In the queue window notice an **SS__** queue has been created.

Sending a Message to a Queue

1. Click on the + button next to the **SS__** queue. The components of this queue appears.
2. In the pRISM+ shell, type:

```
session open
queue show
```

All the queues and their status will display. See [Figure 13-2](#)

```
% queue show
[4291233081: New Connection (leticiapc.isi.com,CommSrv_leticia_ffc3ef01,*,letici
a,pid=4291040293,optimised) ]
Name      ID      TQ  MQ  MQ      Limit  Mgb  Qtype  Vari- pSOS+m  Notify  Notify
=====  =====  =====  =====  =====  =====  =====  =====  =====  =====  =====  =====
RQ00  0x000C0000  0000  0000  00010  Sys-pool  FIFO  No  Local  00000000  00000000
CQ00  0x000D0000  0000  0000  00004  Sys-pool  FIFO  No  Local  00000000  00000000
PMCQ  0x00110000  0000  0000  none   Sys-pool  FIFO  No  Local  00000000  00000000
PMWQ  0x00120000  0000  0000  none   Sys-pool  FIFO  No  Local  00000000  00000000
SS__  0x00140000  0000  0000  00008  Sys-pool  Prio  No  Local  00000000  00000000
QMEM  0x00160000  0000  0000  none   Sys-pool  FIFO  No  Local  00000000  00000000
-----
Sys-pool total = 0x00000100
Sys-pool free = 0x00000100
ok
%
```

FIGURE 13-2 Queue Show Command Results

3. In the pRISM+ shell, type:

```
psos call q_send 0x00140000 123 456 789 0
```

Use the SS__ queue's ID number, as shown in the above example. A brief message will display in the pRISM+ shell indicating that the message has been sent and received by the queue.

Viewing your Message

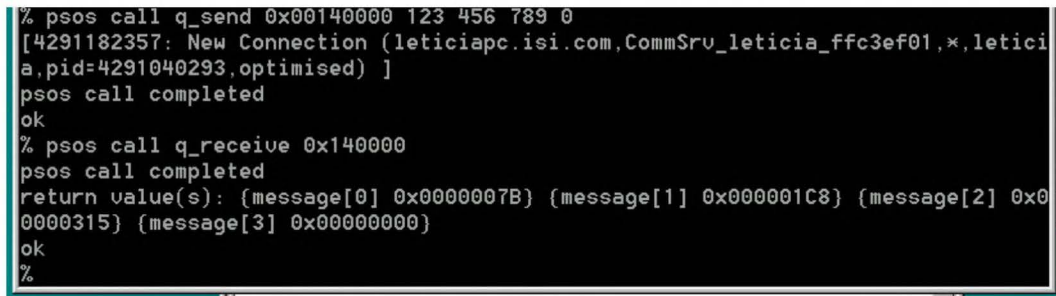
1. To refresh the **pSOS Objects** dialog box from the SearchLight tool bar, click on **View** → **pSOS Objects**. Click on the **+** button next to the SS queue.

Notice that the Number of Messages has changed status from 0 to 1.

2. To receive your message, in the pRISM+ shell type:

```
psos call q_receive 0x00140000
```

The message is displayed as show in [Figure 13-3](#).



```
% psos call q_send 0x00140000 123 456 789 0
[4291182357: New Connection (leticiapc.isi.com,CommSrv_leticia_ffc3ef01,*,letici
a.pid=4291040293,optimised) ]
psos call completed
ok
% psos call q_receive 0x140000
psos call completed
return value(s): {message[0] 0x0000007B} {message[1] 0x000001C8} {message[2] 0x0
0000315} {message[3] 0x00000000}
ok
%
```

FIGURE 13-3 psos call q_receive Command Results

Conclusion

This concludes this tutorial on how to use the pRISM+ Shell with the SearchLight debugger. For additional information on the pRISM+ Shell, refer to [pRISM+ Shell Commands](#) appendix.

13.2 Using and Invoking a pRISM+ Shell Tcl Script

You can use the pRISM+ shell commands within the pRISM+ Shell (see [Section 13.1.4, Using pRISM+ Shell with SearchLight Debugger](#)). You can also create a specialized tcl script that contains the pRISM+ Shell commands. With the pRISM+ Shell command you can create a test script to assist you in debugging your application. You can set a breakpoint at a certain address, send a message, or redefine the pROBE+ flags.

In this section you will learn how to attach a tcl script through the pRISM+ Shell. In order to do this procedure you must have already created a tcl script using the pRISM+ Shell commands. In this instance we will use a Tcl scrip provided in the PrismPlusShell directory.

Refer to [Appendix D, pRISM+ Shell Commands](#) for the list of the supported pRISM+ Shell commands.

13.2.1 Using an Existing Tcl Script for Testing

1. Locate the dsdemo_u.tcl and dsdemo_w.tcl files, which are located in the directory `/isiTarget_name/prism+/lib/PrismPlusShell`, where *Target_Name* is ppc, 68k, or mips.
2. Create a folder or directory labeled TestScripts.
3. Place a copy of the dsdemo_u.tcl and dsdemo_w.tcl files in the TestScripts directory.

NOTE: If you are developing in the UNIX environment, modify the dsdemo_u.tcl file. If you are developing in the Windows environment, modify the dsdemo_w.tcl file.

4. Use a text editor to modify the dsdemo_u.tcl or dsdemo_w.tcl file. Change all entries of the dummy project location with the location of the project you created when you completed the [Quick Start with a Tutorial](#). (See [Example 13-1 on page 13-10](#).)
 - a. Replace all *isiTarget* references with references to your processor (ppc, 68k, or mips).
 - b. Replace all the *user_name* references with your login name; for example, jsmith.
 - c. Save your modified script.
 - d. Copy the modified file dsdemo_u.tcl or dsdemo_w.tcl back to the original directory, `/isiTarget_name/prism+/lib/PrismPlusShell`.

EXAMPLE 13-1: Locating the Pathnames

```
# %dsession load

C:\isiTarget\users\user_name\PSOSTARGET_PWE\userapps\copy_of_pdemo\ram.elf
all

# %boot
# %...
# %csabout license
#####

proc demo_window {args} {

    tout << "%dsession open\n" ; # Print message on shell screen
    tout << "[eval dsession open]\n" ; # Execute the command "dsession open"

    tout << "%dsession load C:\isiTarget\users\user_name\PSOSTARGET_
PWE\userapps\copy_of_pdemo\ram.elf all\n"
    tout << "[eval dsession load C:\isiTarget\users\user_name\PSOSTARGET_
PWE\userapps\copy_of_pdemo\ram.elf all]\n"
```

5. From the pRISM+ Manager, select **File** → **proj1**. This project was created during the pRISM+ Tutorial.
6. From the pRISM+ Manager, select **targ1** from the **Target List** pull-down menu. This target was defined during the pRISM+ Tutorial.
7. From the pRISM+ Manager, click on the **pRISM+ Shell** button to invoke the pRISM+ Shell.
8. To execute your script, in the pRISM+ Shell use one of the following commands:

- a. In the Windows environment, type

```
dsdemo_window > output.txt
```

- b. In the UNIX environment, type

```
dsdemo_unix > output.txt
```

The results of the test script are located in the **output.txt** file.

13.2.2 pRISM+ Shell Script Example

In each pRISM+ for pSOSystem installation there are several Tcl script examples. The `demo_u.tcl` (UNIX script) and `demo_w.tcl` (Windows script) files are sample scripts you can use for this brief tutorial. You can also use these scripts as a starting point to create your own test scripts.

The `demo_w.tcl` script ([Example 13-2](#)) or `demo_u.tcl` script when invoked opens a debug session. It will load and boot your `ram.elf`, redefine your communication timeouts, then suspend and resume a task.

EXAMPLE 13-2: Windows Example Test Tcl Script

```

#*****
# Filename: demo_w.tcl
# Description: A demo program for Windows
# Details:
#   Running some shell commands.
# Date: Aug. 25, 1998.
#*****

#####
# Procedure to implement "demo_window" command. To execute the procedure,
# type "demo_window" on the shell.
# %demo_window
# To save the output result to a file named "output.txt", type
# %demo_window > output.txt
#-----
# Executing the above procedure is equal to typing the commands on the shell
# one by one:
# %dsession open
# %dsession load

C:\isiTarget\users\user_name\PSOSTARGET_PWE\userapps\copy_of_pdemo\ram.elf
all

# %boot
# %...
# %csabout license
#####

```


The next few commands will open a debug session and allow you to download, boot, and initialize your application.

```
proc demo_window {args} {

    tout << "%dsession open\n" ;
    # Print message on shell screen
    tout << "[eval dsession open]\n" ;
    # Execute the command "dsession open"

    tout << "%dsession load C:\isiTarget\users\user_
        name\PSOSTARGET_PWE\userapps\copy_of_pdemo\ram.elf all\n"
    tout << "[eval dsession load C:\isiTarget\users\user_
        name\PSOSTARGET_PWE\userapps\copy_of_pdemo\ram.elf all\n"
    tout << "%boot\n"
    tout << "[eval boot]\n"

    tout << "%initialize\n"
    tout << "[eval initialize]\n"

    tout << "%go\n"
    tout << "[eval go]\n"

    tout << "%halt\n"
    tout << "[eval halt]\n"

    tout << "%session open targ1\n"
    tout << "[eval session open targ1]\n"

    tout << "%debugger show\n"
    tout << "[eval debugger show]\n"
```

The next few commands will set and show the communication timeouts values.

```
tout << "%debugger set timeout 6000\n"
tout << "[eval debugger set timeout 6000]\n"

tout << "%debugger set acktimeout 300\n"
tout << "[eval debugger set acktimeout 300]\n"

tout << "%debugger set retries 6\n"
tout << "[eval debugger set retries 6]\n"

tout << "%debugger show\n"
tout << "[eval debugger show]\n"

tout << "%debugger set timeout 5000\n"
tout << "[eval debugger set timeout 5000]\n"
```

```
tout << "%debugger set acktimeout 200\n"
tout << "[eval debugger set acktimeout 200]\n"

tout << "%debugger set retries 5\n"
tout << "[eval debugger set retries 5]\n"
```

In the next few commands you will view various pSOS+ component tables.

```
tout << "%psos show table pna\n"
tout << "[eval psos show table pna]\n"

tout << "%psos show table pmont\n"
tout << "[eval psos show table pmont]\n"

tout << "%task show\n"
tout << "[eval task show]\n"
```

In the next few commands you will suspend and resume a task. You will also explore the `csabout` command.

```
tout << "%psos call t_suspend 0x00010000\n"
tout << "[eval psos call t_suspend 0x00010000]\n"

tout << "%session reopen\n"
tout << "[eval session reopen]\n"

tout << "%task show\n"
tout << "[eval task show]\n"

tout << "%psos call t_resume 0x00010000\n"
tout << "[eval psos call t_resume 0x00010000]\n"

tout << "%session reopen\n"
tout << "[eval session reopen]\n"

tout << "%task show\n"
tout << "[eval task show]\n"

tout << "%csabout version\n"
tout << "[eval csabout version]\n"

tout << "%csabout license\n"
tout << "[eval csabout license]\n"

set result [tout string]
tout clear
```

The following commands prints the results of this script to a file.

```

# Save output result to a file if user requests
set fileCheck [lindex $args 0]
if { $fileCheck == ">" } {
    set filename [lindex $args 1];
    # Obtain filename from user input
    set fileId [open $filename w]
    puts -nonewline $fileId $result
    close $fileId
}

return $result
}

```

For additional scripts to use, explore `/ISITarget_Name/prism+/lib/Prism-PlusShell` directory.

13.3 Using Low-Level TCL/CORBA Services

The pRISM+ Shell allows any CORBA service to be called from TCL. This allows you the capability to create TCL scripts that can communicate with the Communication or Debug Server IDL interfaces. Of course this type of pRISM+ Shell usage is specifically designed for the advanced usage.

13.4 Customizing the pRISM+ Shell

You can create a startup script, which will can be executed every time the pRISM+ Shell is invoked. The location of the startup script is:

- In the Windows Environment

```
%HOME%\ .tclshrc
```

- In the UNIX Environment

```
$HOME/ .tclshrc
```

Inside the startup script, you can specify the commands provided by pRISM+ Shell. For example:

```

%puts [session open $_targetName]
%puts [dsession open]

```

Every time the pRISM+ Shell starts, the target connection will be open automatically by the startup script. For example:

```
% puts [session open $_targetName]
% puts [dsession open]
% puts [breakpoint show]
% puts [task show]
% puts [semaphore show]
```

When the pRISM+ Shell starts, the pRISM+ Shell will connect to the target but it will also make these specific queries to the target to get information on tasks and semaphores.

pRISM+ has two target agents: pROBE+ and pMONT+. The target agents perform specific functions on the target as requested by the pRISM+ Tools. They assist in obtaining target status and communication. These target agent functions are described in this chapter. To use the pRISM+ Tools (such as ES_p, Object Browser, SearchLight, RTA Tool suite, or SingleStep for pRISM+) you must incorporate in your pSOS+ application one or more Target Agents. These target agents also make it possible for communication to occur between the target and the pRISM+ tools.

The pRISM+ Tools communicate to the pRISM+ Communication Server that resides on the host system. The pRISM+ Communication Server then communicates to the target agents through a Serial or Ethernet connection. In case of serial connection to the target, the pRISM+ Communication Server must be running on the host machine which is connected serially to the target.

14.1 pMONT+ Target Agent

The pMONT+ target agent performs the following functions on the target:

- Collects run-time events requested by you through ES_p.
- Establishes a connection with pRISM+ Communication Server.

This section describes the following pMONT+ target agent topics:

- Target requirements for monitoring an application
- Configuring pMONT+
- Target behavior

- `log_event()` call
- pMONT+ memory requirements
- Warnings about buffer support

14.1.1 Target Requirements for Monitoring an Application

For the ESsp and Object Browser tools to acquire information about an application, you must configure the target-resident pMONT+ component to be running when the application is running. The pMONT+ configuration and startup process is the same as for other pSOSystem components from Integrated Systems.

After start-up, the ESsp and Object Browser tools controls pMONT+ behavior according to your specifications. pMONT+ processes ESsp and Object Browser requests and interacts with the target's pSOSystem environment to supply information to ESsp and Object Browser.

14.1.2 Configuring pMONT+

The pMONT+ configuration table is defined in the `sys_conf.h` file. The `sys_conf.h` parameter settings become assignments in the `typedef` structure located in the `pmontcfg.h` file. For the definitions of pMONT+ Configuration Table entries, refer to the *Programmer's Reference* manual, Chapter 4.

```

:
typedef struct
{
    void (* code)();           /* Address of pMONT+ module */
    long data;                 /* start of pMONT data */
    long dataSize;             /* size of pMONT data */
    long cmode;                /* comm.mode:NETWORK_TYPE_CONN,PSOSDEV_..*/
    long dev;                  /* IO dev maj/minor# in form pSOS expects */
    char *traceBuff;           /* Buffer for logging trace events */
    long traceBuffSize;        /* trace events buffer size */
    unsigned long (* tmFreq)(); /* returns second timer frequency */
    void (*tmReset)();         /* resets second timer */
    unsigned long (* tmRead)(); /* reads counter value of second timer */
    long res1;
    long res2;
    long res3;
    long res4;

    pMONT_CT;

```

where the parameters are defined as follows:

<code>code</code>	Starting address of pMONT+ code.
<code>data</code>	Starting address of pMONT+ data area. If <code>data</code> is 0, the data area is allocated from Region 0.
<code>dataSize</code>	The size of the pMONT+ data area. If you specify the address with <code>data</code> , you must also specify <code>dataSize</code> .
<code>cmode</code>	Specifies the communication that pMONT+ uses: <ul style="list-style-type: none"> ■ <code>cmode=1</code> means Ethernet communication through the pNA+ network manager. ■ <code>cmode=2</code> means serial communication through a pSOS+ device.
<code>dev</code>	The pSOS+ I/O major:minor device number if <code>cmode</code> is 2. If <code>cmode</code> is 1, <code>dev</code> is not used.
<code>traceBuff</code>	Address of the buffer for logging trace data. If <code>traceBuff</code> is 0, <code>traceBuffSize</code> defines the size, and the pSOSsystem environment supplies the buffer. pMONT+ does not allocate <code>traceBuff</code> from Region 0 because the buffer should remain intact. If pMONT+ allocated <code>traceBuff</code> from Region 0, system initialization could result in unreliable buffer content.
<code>traceBuffSize</code>	The size of <code>traceBuff</code> in bytes, 1 kilobyte minimum.
<code>tmFreq</code>	Pointer to a user-supplied routine to return the frequency (counts per second) of an extra timer for finer timekeeping during resolution a data collection run.
<code>tmReset</code>	Pointer to a user-supplied routine to reset the extra timer and start counting.
<code>tmRead</code>	Pointer to a user-supplied routine to return the current count of the timer: the returned count must be between 0 and <code>tmFreq</code> and must indicate a sequence counted up from 0. The count must not exceed 24 bits within the span of 1 pSOS+ tick. If you do not use timers and are not running under pSOSsystem, then all three of the preceding timer entries must be 0.
<code>res[0-3]</code>	An array reserved for pMONT+ use. Each element of <code>res[]</code> should be initialized to zeroes (0000).

If you are configuring pMONT+ under the pSOSsystem environment, you can specify a macro in the `sys_conf.h` file to set or disable the extra timer automatically by setting `PM_TIMER` to YES or NO, respectively.

The node configuration table, defined through parameter settings made in the `sys_conf.h` file, includes a pointer to the pMONT+ configuration table and pointers to other pSOSystem components. The `struct NodeConfigTable` is as follows:

```
struct NodeConfigTable
{
    INT32      cputype;      /* CPU type */
    MPCT       *mp_ct;      /* pSOS+m configuration table pointer */
    pSOSCT     *psosct;     /* pSOS+ configuration table pointer */
    pROBECT    *probest;    /* pROBE+ configuration table pointer */
    pHILECT    *philect;    /* pHILE+ configuration table pointer */
    pREPCCT    *prepcct;    /* pREPC+ configuration table pointer */
    pICCT      *picct;      /* pIC+ configuration table pointer */
    pNACT      *pnact;      /* pNA+ configuration table pointer */
    pSECT      *psect;      /* pSE+ configuration table pointer */
    pMONTCT    *pmontct;    /* pMONT configuration table pointer */
    INT32      rsvd[6];     /* Unused entries */
}
NODE_CT;
```

To run pMONT+ with pSOS+, pROBE+, and any other components, for example pNA+ for networking, you need to have the necessary pointers set in the node configuration table as indicated above.

14.1.3 pMONT+ Driver Usage

pMONT+ does not initialize any drivers. It starts up as if the necessary driver initialization has already taken place. For pMONT+ to start successfully under this scheme, the driver must be configured to use the `autoinit` feature of a pSOS+ driver. Note that if you enable `autoinit` for a particular device, the kernel first calls the `de_init()` function of the driver with minor device number of 0. The kernel does this before any task starts running.

To use `autoinit`, you must set the eighth bit in the second reserved field of the pSOS+ I/O jump table of a particular device. The following example shows the pSOSystem convention for installing a driver in `drv_conf.c` (a file residing in each pSOSystem application directory):

```
InstallDriver(SC_DEV_SERIAL, CnslInit, NULL, NULL, CnslRead,\
CnslWrite, CnslCntrl, 0, 0, 1<<8);
```

where `1<<8` sets `autoinit`.

For pMONT+ to run, you must use the preceding method to initialize the timer. For serial communication, you must also initialize the serial driver, which then operates with the following characteristics:

- Blocking I/O
- ASCII mode
- Echoing off
- Carriage return to signal the end of a record
- No conversions for a new-line character

pMONT+ uses the serial driver through the pSOS calls `de_open()`, `de_read()`, and `de_write()`. It makes the `de_open()` call before proceeding to use the driver to read and write. The `de_open()` call should thus set the driver for pMONT+ usage if `autoinit` has not already done so.

You should use `autoinit` to initialize the driver and `de_open()` to change settings (if needed). In cases where the installed driver has specific functionality for each of the I/O calls, you can install a dummy driver for pMONT+ in which `de_open()` calls the actual serial driver to perform any necessary initialization that `autoinit` does not do.

14.1.4 pMONT+ Behavior on the Target

This section describes those aspects of pMONT+ behavior you should consider when planning the use of the system.

ESp and Object Browser communication with pMONT+ takes place across the medium that you define in the target definition. For its part, pMONT+ creates three tasks to communicate with the ESp and Object Browser tools and process their requests. These tasks are PMCM, PMON, and ASEV. They run at priorities 0xf1, 0xf2, and 0xf3, respectively. Any user task (including ROOT) must be at a priority below that of the pMONT+ tasks at the time of the user task's creation.

Using pROBE+ with pMONT+ requires caution. You should not set breakpoints in the application if the ESp and Object Browser tools and target frequently communicate with each other because this could break the connection. However, when no communication takes place between host and target, you can use the full functionality of the pROBE+ debugger. If you use pMONT+ and the pROBE+ debugger together, the pROBE+ interrupt level should be such that it prevents any interrupts in the system from occurring. Otherwise, if interrupts occur in the pROBE+ debugger, timing errors show up in the display of events.

pMONT+ does not require the presence of the pROBE+ debugger. However, if the debugger is present and configured correctly, you can use the pROBE+ `gs` command to start a data collection run from the beginning of an application. With the

pMONT+ and ES_p and Object Browser modules, `gs` can also cause an application *warm start*.

A warm start under pMONT+ means that the application restarts while an ES_p or Object Browser session is already in progress. With this feature, pMONT+ can collect trace data from the time an application starts up. Alternatively, you can reset the board to achieve the same result. You can specify a data collection run to start with the application by the following method:

- Through the ES_p and Object Browser interfaces, you can define a data collection run to begin upon the next restart of the connected target.
- On the target, you can break into pROBE+ by using a manual break or by pressing the RESET button on the board. You must then enter `gs` and, if pROBE+ is not set to silent startup mode, enter `go` to start the application. The data collection run begins automatically when the application starts running.

If you are not running the pROBE+ analyzer, you must restart your application manually before a data collection run can begin.

To perform its role in event logging and profiling, pMONT+ captures system activity through the kernel. By this method, pMONT+ minimizes the intrusion it causes to the application. The amount of intrusion depends on the level of requested services. For example, logging trace events from tasks only is less intrusive than logging all trace events. Setting up more items to filter or either to log or not log also adds to the load. Also, for dynamic profiling operations, metering fewer system activities is less intrusive than metering all.

14.1.5 `log_event()` System Call

pMONT+ supports one system call, `log_event()`. The `log_event()` call logs an event in the trace buffer. The `log_event()` call takes effect when the ES_p data collection run begins. Note that user-event logging happens only if your event specification has not made `log_event()` an event to ignore. The `log_event()` call always returns 0. The syntax of the `log_event()` call is as follows:

```
log_event(
    unsigned long user_event_id, /* User-defined event ID */
    unsigned long event_data    /* User-defined event data */
)
```

where the parameters are as follows:

<i>user_event_id</i>	A number for each user-event call. The maximum value the ID can have is 0xff. Providing an ID for each call can help you keep track of user events.
<i>event_data</i>	Optional 32-bit data you can log for test purposes.

14.1.6 Memory Usage

pMONT+ requires memory for two reasons:

- To keep track of information about creation and deletion of system objects.

This memory buffer is allocated from region 0 and the size is $96 * KC_NLOCOBJ$ bytes. `KC_NLOCOBJ` is the maximum number of pSOS+ kernel objects which is set in the `sys_conf.h` file. In case of a multi-processor system with pSOS+m, the equivalent number is $96 * (KC_NLOCOBJ + MC_NGLBOBJ)$. `MC_NGLBOBJ` is the maximum number of global objects.

- To Log the events during an ESsp experiment.

This memory buffer is known as the trace buffer. Its size and starting address are specified in the `sys_conf.h` file. If you want to allocate the memory for the trace buffer, the variable `PM_TRACE_BUFF` should be set to the starting address of such memory. The `PM_TRACE_SIZE` should be set to the size of this memory.

If `PM_TRACE_BUFF` is zero and `PM_TRACE_SIZE` is non-zero, then pMONT+ allocates this memory from `FreeMemPtr` during system startup.

`PM_TRACE_SIZE` should be at least 1000 bytes for an ESsp experiment to be configured.

14.2 pROBE+ Target Agent

pROBE+ is a target resident agent which functions as both a cross-development target agent and a stand-alone debugger. It provides the pSOS+ kernel-aware debugging functions, but is not dependent on pSOS+ kernel. This allows developers to obtain debug support during the BSP development process.

As a component, pROBE+ does not depend on certain types of peripheral hardware. It only requires the proper communication drivers and the simple exception wrappers. The interface of the communication drivers is common for all CPU families.

The interface of the exception wrappers is common for all CPU types within one CPU family.

14.2.1 pROBE+ Behavior on the Target

As a target agent pROBE+ enables advanced host-based source level debugging features. For example, System Debug Mode (SDM) and Task Debug Mode (TDM).

In SDM, if the target is stopped, due to exception or breakpoint, the whole application, pSOS+ kernel and all other non-pROBE+ components are stopped. Only the pROBE+ component and the communication driver which is used for the data transfer between the host and target are active. The developers will receive a snapshot of the target activities. The SDM is especially useful to debug the interrupt service routines or to examine the state of any pSOS+ kernel object and the values of any data structures when an exception occurs.

In TDM, the tasks are divided into two groups, the background and the foreground. Often the foreground group is called the debug task list or debug list. If the application hits a breakpoint or an unexpected exception occurs in a task context, all tasks in the debug task list will be stopped and the tasks in the background will keep running. You can add or remove a task to the debug task list through the host debugger. When a background tasks hits an unexpected exception, the task will automatically be added to the debug task list. If an unexpected exception occurs while in an interrupt service routine or if a pSOS+ fatal error occurs, pROBE+ will switch to the SDM mode if it was in TDM. The TDM is most useful when it becomes necessary to debug an individual task or an group of tasks while the reset of the system continues to run.

14.2.2 Configuring pROBE+

pROBE+ consists of five parts that provide degrees of the scalability. To allow pROBE+ to work with the host debugger, the Processor Service and remote Debugger Service have to be selected. If the pSOS+ kernel awareness is required, the Query Service has to be included. In the `sys_conf.h` provided by the pSOSystem sample application, you have to set `SC_PROBE` and `SC_PROBE_DEBUG` to `YES`. If the Query Service is needed, set `SC_PROBE_QUERY` to `Yes`.

The pSOSystem provides two communication drivers for the data transfer between the host debugger and target, one is for using the network and one is for using the serial port. You have to select the proper driver for your environment.

For more information on pROBE+, refer to the *pROBE+ User Guide* manual.

15

Customize the pRISM+ Tools/Environment

15.1 Customizing Your pRISM+ Tools

In this chapter you learn how to customize your pRISM+ tools. You will learn how to customize your toolbar and how to customize your project.

15.1.1 Customizing Your Toolbar

In this sample you are going to customize the pRISM+ Manager toolbar so you are able to list all the files in a pRISMSpace directory.

1. From the pRISM+ Manager menu bar, click **Tools** → **Customize**. The **pRISM+ Tools** dialog is displayed. See [Figure 15-1](#).
2. In the **pRISM+ Tools** dialog, click on the **Custom** tab.
3. In the **Custom** page, click the **Add** button. A default title of **New Tool Entry** appears. Delete this entry and enter **List pRISMSpace Files** as the tool name.

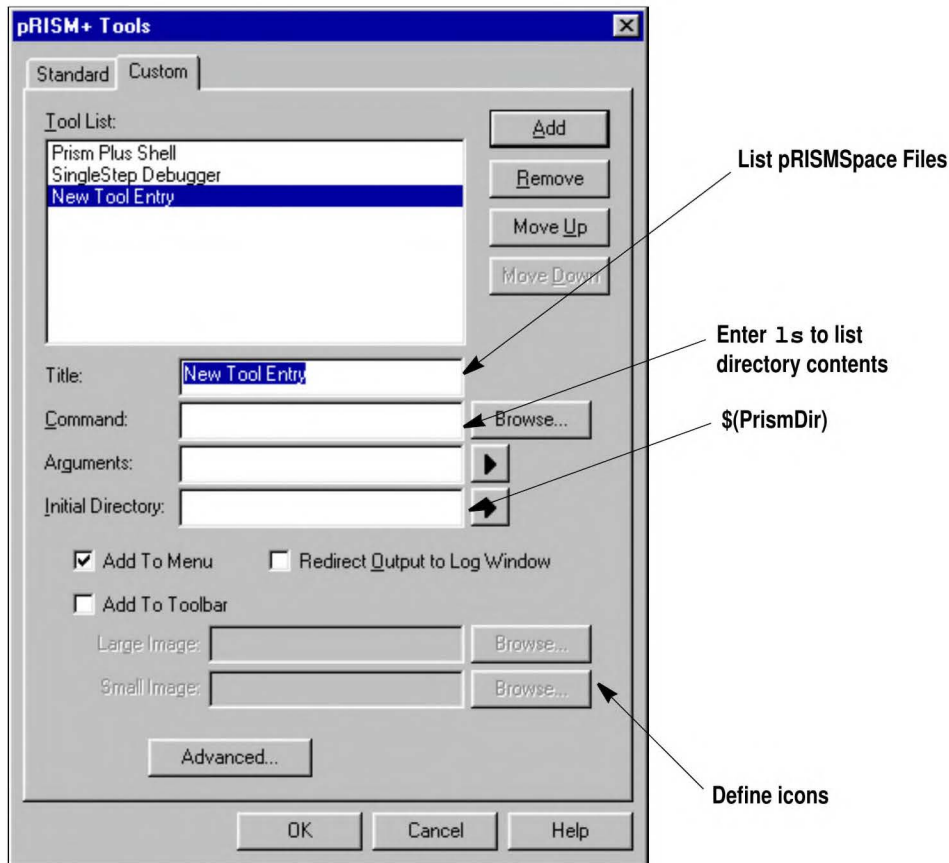


FIGURE 15-1 pRISM+ Tools Dialog

4. In the **Command** field, enter `ls` to list the contents of a directory.
5. Click the arrow next to the **Initial Directory** field.
6. Select **pRISMSpace Directory**. The proper environment variable, `$(PrismDir)`, is placed in the field.
7. Click the **Add to Menu**, **Add to Toolbar**, and **Redirect Output to Log Window** options.

By selecting the **Add to Toolbar** option your custom tool will display bitmaps icons on the pRISM+ Manager toolbar. Of course you need to specify which bitmaps the pRISM+ Manager will use for the toolbar.

8. Specify which bitmap files to display:
 - a. To specify which bitmap files to display in the pRISM+ toolbar, fill the file names in the **Large Image** and **Small Image** fields.
 - b. Click the **Browse** button next to the **Large Image** and/or **Small Image** fields and select any of the bitmaps included with pRISM+, or your own bitmaps. If you do not care which icons are used, leave both or either of the fields blank. The pRISM+ Manger, by default, will use a hammer icon.
 - c. If you wish to use your own bitmaps, note the following:
 - ◆ The format should be .bmp files, not X11 bitmaps.
 - ◆ Large bitmaps should be 32 x 32 pixels; small bitmaps should be 15 x 16 pixels.
 - ◆ Specify no more than 16 colors.
 - ◆ Make sure that the system has read access to your .bmp file(s). You can use the **Browse** button to direct the system to your bitmaps.
 - d. Click **OK** in the **Open** dialog after selecting the bitmaps.
9. Click **OK** on the **pRISM+ Tools** dialog.

Notice that a new icon appears for your tool on the right end of the toolbar, and that a Tool Tip string appears with the title of the tool when you mouse over the icon. Notice also that an entry for the tool appears when you click the **Tools** menu on the pRISM+ toolbar.
10. Open a project and click on the **List pRISMSpace Files** icon.
11. The pRISM+ Log Window appears and provides the listing.

15.1.2 Incorporating a Custom BSP for pSOSystem

In this section we will explore how to incorporate your custom BSP into an existing application. There are two methods you can use to incorporate your custom BSP into an existing project; you can copy your custom BSP into the `bsps` directory, or you can reference your custom BSP.

Copying the BSP

1. Copy the directory that contains your custom BSP into the following directory:

```
/ISI<TargetName>/pss<TargetName>.<version>/bsps
```

where *TargetName* is one of the following: 68k, ppc, or mips.

2. Launch **orbixd** and pRISM+. Refer to [Chapter 3, *Quick Start with a Tutorial*](#).
3. From the pRISM+ Manager, open your pRISMSpace project, where *Project_Name* is the name of your existing application you created with the pRISM+ Editor or SNIFF+.
4. From the pRISM+ Manager, click **pRISMSpace** → **Settings**. The **Project Settings** dialog is displayed.
5. In the **Project Settings** dialog, use the drop-down button next to the **Board Support Package** field to locate and select your BSP.
6. Click the **OK** button to accept the changes.
7. You must completely rebuild your application. Use the project editor to rebuild. For example perform a makeclean and a make all.

Referencing the BSP

1. Launch **orbixd** and pRISM+. Refer to [Chapter 3, *Quick Start with a Tutorial*](#).
2. From the pRISM+ Manager, open your pRISMSpace project.
3. Select an existing application you created with the pRISM+ Editor or SNIFF+.
4. From the pRISM+ Manager, click **pRISMSpace** → **Settings**. The **Project Settings** dialog is displayed.
5. In the **Project Settings** dialog, type in the full path and name of your custom BSP in the **Board Support Package** field. You can use the **Browse** button to locate the BSP directory.
6. Click the **OK** button to accept the changes.
7. You must completely rebuild your application. Use the project editor to rebuild. For example perform a makeclean and a make all.

15.2 Customizing Your pRISM+ Environment

In this section you will learn about some of the advanced features of pRISM+. You will learn how to:

- install multiple pRISM+ versions ([page 15-5](#))
- define your environment for a multiple-user configuration ([page 15-7](#))
- develop in a mixed-platform environment ([page 15-8](#))
- redefine your color settings ([page 15-13](#))
- configure your pRISM+ help print options ([page 15-13](#))

15.2.1 Multiple pRISM+ Installations

pRISM+ for pSOSystem allows you to have multiple pRISM+ installations on your PC and Workstation.

Multiple Installations In the Windows Environment

In this section you will learn how to install and use multiple pRISM+ installations in a Windows environment.

Installing Your Second pRISM+ Installation

During the installation, install your second pRISM+ installation in a different directory, for example:

```
C: \68K\isi68k\  
C: \powerpc\isippc\  
C: \PPC1_2_3\isippc\
```

The default installation directory is:

```
C: \isiTargetName\
```

where *TargetName* represents ppc, 68k, or mips.

During the installation, select **Browse directory** to create or choose a directory for your pRISM+ installation.

Multiple Installations In the UNIX Environment

In this section you will learn how to install and use multiple pRISM+ installations in a UNIX environment.

Installing Your Second pRISM+ Installation

During the installation, install your second pRISM+ installation in a different directory for example:

```
/User_Home_directory/68K/isi68k/

/User_Home_directory/Powerpc/isippc/

/User_Home_directory/PPC1_2_3/isippc/
```

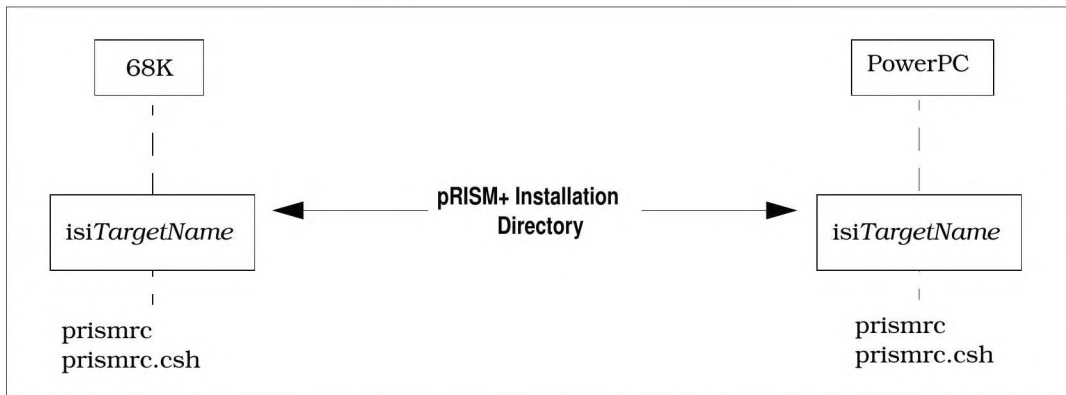


FIGURE 15-2 Multiple UNIX Environment pRISM+ Installation Sample Directories

The default installation directory is:

```
/User_Home_directory/isiTargetName/
```

where *TargetName* represents *ppc*, *68k*, or *mips*.

Running Your Second pRISM+ Installation

To run one of your pRISM+ installations, you need to reset your environment variables. The `prismrc` or `prismrc.csh` files identify your pRISM+ installation and the environment variables.

1. At the command line, type the following:

```
cd /User_Home_directory/isiTargetName
```

where *TargetName* represents ppc, 68k, or mips. You need to be in the directory of the pRISM+ installation you want to access. (See [Figure 15-2](#).) This command will take you to the pRISM+ installation directory.

2. At the command line, type one of the following:

```
source prismrc  
source prismrc.csh
```

You can now use this pRISM+ installation directory for pSOSystem development.

3. To switch to another pRISM+ installation, you must repeat steps 1 and 2.

15.2.2 Multiple-users Configuration (UNIX Only)

Multiple users can run pRISM+ on the same workstation. The default mode of operation for pRISM+ is for a single user to run it on a single workstation. This section describes the necessary steps you need to perform in order to have multiple users running on the same workstation.

Orbix Configuration for Multiuser Support

1. To enable multiple user support on a Solaris machine, run the following sample script, `$PRISM_DIR/bin/multi-user-support.sh`, after the installation is completed. Note that in order to run the script root privilege is required.
2. Issue this command to start the Orbix daemon after the `multi-user-support` script has been executed.

```
/etc/init.d/orbix start
```

3. Users on this workstation need to set an environment variable in their profile (for example, `.profile` or `.login`) to point to the directory containing `Orbix.cfg`. For example:

```
IT_CONFIG_PATH=/etc  
export IT_CONFIG_PATH
```

or

```
setenv IT_CONFIG_PATH /etc
```

A log file `/var/adm/orbix` will be created for logging Orbix daemon activities.

Memory Considerations (Solaris)

When multiple users are running pRISM+ from the same workstation, you can run into problems if the system is not adapted for multiple users. pRISM+ uses shared memory and when multiple users are using the workstation, the shared memory kernel parameters need to be tuned. You need to remember to allocate an equal amount of shared memory by using the swap space on the system.

See the document *SunOS 5.x Administering Security, Performance and Accounting*, Appendix A, *Tuning Kernel Parameters* for additional details. You can also use the answer book to get this information.

If the Target Setup window hangs when you are trying to download to your target, this can be one of the problems. The following error may appear to inform you that you need to tune your system:

```
No room for another process
```

15.2.3 Mixed-Platform Development for Solaris and Windows

This section describes how to develop a pSOSystem application in a mixed-platform environment. Specifically, it describes how to compile an application on the Solaris platform and debug the application using a Windows 95 or 98 or Windows NT based source level debugger.

System Environments and Configurations

- A UNIX workstation running Solaris 2.5.1 or 2.6 and pRISM+ version 2.0 or later.
- A PC running Windows 95, pRISM+ version 2.0 that includes SingleStep version 7.4, and NFS client software from Net Instrument.

Before You Begin

- Consult the UNIX `man` pages on the `share` command to find out how to export part of your UNIX file system so you can NFS-mount it from your PC.
- For this example, `/export/usrl` contains pRISM+ on your Solaris machine for the PowerPC processor type and pRISM+ has been installed into the directory `export/usrl/isiTargetName`.
- Install NFS client software on your Windows 95 or Windows NT machine. NFS-mount `/export/usrl` and map it to the local drive `F:\`.

Make sure you can browse to this directory before proceeding:

```
F:\isi<TargetName>\pss<TargetName>.<version>
```

Building Your Application in the UNIX Environment

Build your application according to the [Quick Start with a Tutorial](#) chapter. Ensure that the resulting `ram.elf` output file is place into a directory named `/export/usr1/myapp`. If you cannot copy your `ram.elf` to `/export/usr1/myapp` then you must ftp your file to your PC. Use a Windows ftp application to copy `ram.elf` from your UNIX workstation to your PC.

It is recommended to copy the `ram.elf` to this directory:

```
/ISI<TargetName>/users/<user_ID>/PSOS<TargetName>_PWE/apps
```

Debugging Your Application in the Windows Environment

To debug this application from the Windows based SingleStep Debugger, do the following:

1. Launch Orbixd and pRISM+ Manager. Refer to [Chapter 3](#).
2. From the pRISM+ Manager, select the **pRISM+ Shell** button.
3. In the pRISM+ Shell, type the following:
 - For PowerPC: **psmppc**
 - For 68K: **psm68k**

The Debug window and the SingleStep main window are displayed. See [Figure 15-3 on page 15-10](#) for an example of the Debug window.

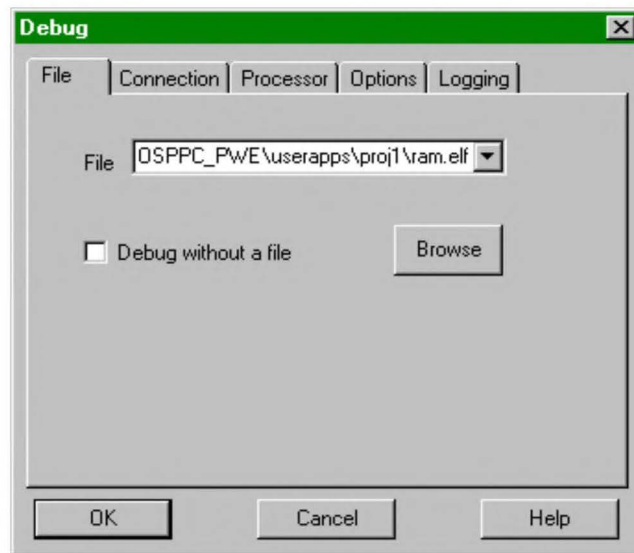


FIGURE 15-3 Debug Window

4. In the Debug window, enter the path and the name of the `ram.elf` file in the **File** field.
5. Click on the **Connection** tab.

The **Connection** window is displayed (Figure 15-4).

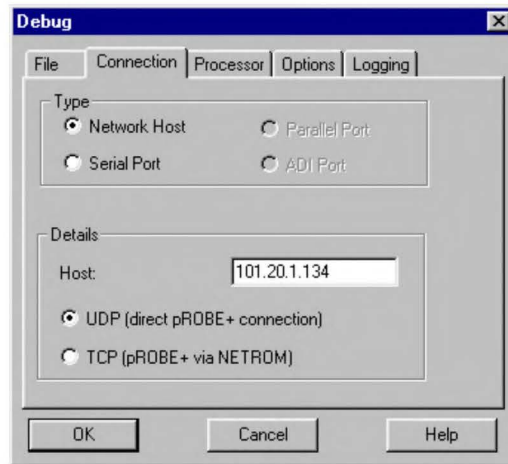


FIGURE 15-4 Debug Window With Connection Selected

6. Click **Network Host** in the **Type** section of the **Connection** window.
7. In the **Details** box, select **UDP** and enter the name of your target board (if DNS is available) or its IP address in the **Host** field.
8. Click on the **Logging** tab and select the **Log to screen (always)** option.
9. Click **OK**.

The system proceeds to make the network connection and download the executable image. The **Debug Status** window displays status messages as this takes place. When the download is complete, the **Image Downloading**, **Target Reset**, and **Execute until 'main'** fields should show **Completed**, and the **Debug Session** field should show **Started Successfully** (see Figure 15-5).

NOTE: The status of the download is displayed in the bottom of the Debug Status window.

10. Click **Close** to close the Debug Status window.

Your `ram.elf` file is now ready for you to debug.

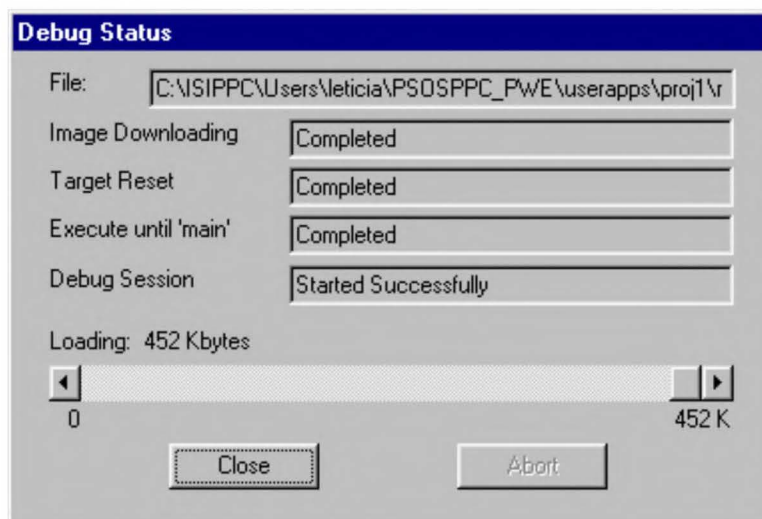


FIGURE 15-5 Debug Status Window

15.2.4 Redefining Your Environment Variables

The pRISM+ software installation includes a script file that sets up the pRISM+ environment. In the Windows environment the installation script automatically sets your environment variable to the default settings.

In the UNIX environment you must run a script in order to set your environment settings. These environment variables are set by the `prismrc` or `prismrc.csh` script (depending on your platform). To ensure that these settings are made every time you log in, add `prismrc` or `prismrc.csh` to your startup or profile file.

NOTE: The variable `PSS_BSP` setting is changed when you are using the pRISM+ Manager. When you change the `PSS_BSP` setting you must exit and restart pRISM+. The installation provides a default setting.

In the Windows Environment the environment file is called `envTarget_name.ksh`:

- `env68k.ksh` for the 68K environment
- `envppc.ksh` for the PowerPC environment
- `envmip.ksh` for the MIPS environment

This script file is created at installation and is run automatically each time you start pRISM+. You can edit the `envvTarget_name.ksh` file if you would like to change the pRISM+ environment settings. Each time you make a change to the script, you must first exit the Orbix daemon and pRISM+ Manager, make the change, then restart the Orbix daemon and pRISM+ Manager.

In the UNIX Environment, you can also edit these variables. You can modify your environment variables in the following files:

- `envvTarget_Name.csh`, where `Target_Name` can be one of the following: 68K, PPC, or MIPS
- `envvTarget_Name.sh`, where `Target_Name` can be one of the following: 68K, PPC, or MIPS.
- `prismrc`
- `prismrc.csh`

These environment files are text files that can be easily edited by using any text editor. For additional information on the environment variables, refer to [Appendix B, pRISM+ Environment Variables](#).

15.2.5 Redefining Your Color Settings (Solaris and HP-UX)

The color settings for pRISM+ Manager, ES_p, Object Browser, and pRISM+ on-line help can be set through your `.Xdefaults` file. A sample `.Xdefaults` file is provided in the `$PRISM_DIR/lib` directory. This sample `.Xdefaults` file can be appended to the end of your current `.Xdefaults` file.

Now, run the following command to replace the current property settings with the changes in your `.Xdefaults` file:

```
xrdb $HOME/.Xdefaults
```

The color settings for SNiFF+ are contained in the `.UserPrefs.sniff` file, which is copied to your `$HOME` directory the first time you run pRISM+. See the SNiFF+ documentation for information relating to the setting of the colors for SNiFF+.

NOTE: The colors cannot be set for pRISM+ Wizard or the SearchLight user interface at this time.

15.2.6 Setting a Printer for On-line Help (Solaris and HP-UX)

This section describes how to correctly define a printer so you can print the pRISM+ on-line help. These directions are for the UNIX environment only.

LPT1 and LPT2 are valid printer slots provided by pRISM+. By using pRISM+, you need to create a PostScript file and redirect the applicable file to a printer denoted by either LPT1 or LPT2.

To configure LPT1 or LPT2, do the following steps:

1. Edit the `win.ini` file. You can obtain this file from your `$HOME/windows/win.ini` directory, which resides in your home directory.
2. Change the print commands for LPT1 or LPT2 to redirect the output to a printer of your choice. For LPT1, you can choose between two commands.

For LPT1, type in the following default value:

```
"LPT1:=lp -c -s "%s" "
```

However, you can change the LPT1 command to the following

```
"LPT1:=lp -dprintrname -c -s "%s" "
```

where *`printrname`* is your specified printer.

For LPT2, type in the following default value:

```
'LPT2:=lp -c -s "%s" '
```

A

Board-Support Package Information

This appendix provides information about individual supported hardware products, including jumper settings, RAM configurations, and ROM locations. The sections are organized by manufacturer and product. [Table A-1](#) summarizes the specific boards described in this chapter.

TABLE A-1 Summary of Board-Specific Information

Board	Board Support Package (BSP)	Appendix	Page
IDT 79S465 Evaluation Board	\$PSS_ROOT/bsps/idt79465	A.2	A-3
IDT 79S440 Evaluation Daughtercard	\$PSS_ROOT/bsps/idt79465	A.3	A-10
IDT 79S500 Evaluation Daughtercard	\$PSS_ROOT/bsps/idt79465	A.4	A-14
LSI400X MiniRISC and LSI4101 TinyRISC Evaluation Boards	\$PSS_ROOT/bsps/lsi4101	A.5	A-16

A.1 pSOSystem/MIPS Operating Mode

This section describes the operating mode of pSOSystem/MIPS. It applies to all board-support packages and to all pSOSystem/MIPS components.

Operating Mode pSOSystem/MIPS runs exclusively in kernel mode. The processor must remain in kernel mode at all times.

All pSOSystem components operate in 32-bit mode, even on MIPS processors that support 64-bit mode.

Endian Mode pSOSystem/MIPS runs exclusively in big-endian mode.

All pSOSystem components are built as big-endian objects and will not work with little-endian code or libraries.

Memory Management pSOSystem/MIPS does not use the processor Memory Management Unit (MMU). Since MIPS pSOSystem does not support an MMU, the only valid address spaces are `kseg0` and `kseg1`. Access to any other virtual address space is undefined and not supported.

The processor converts `kseg0` virtual addresses to physical addresses by subtracting `0x80000000` from the virtual address. It converts `kseg1` virtual addresses to physical addresses by subtracting `0xA0000000` from the virtual address.

Table A-2 describes the virtual address space of the pSOS+ kernel.

TABLE A-2 Kernel Virtual Address Space

Segment	Virtual Address Range	Access
kuseg	0x00000000 - 0x7FFFFFFF	Not supported.
kseg0	0x80000000 - 0x9FFFFFFF	0.5 GB Unmapped Cached.
kseg1	0xA0000000 - 0xBFFFFFFF	0.5 GB Unmapped Uncached.
ksseg	0xC0000000 - 0xDFFFFFFF	Not supported.
kseg3	0xE0000000 - 0xFFFFFFFF	Not supported.

A.2 IDT 79S465 Evaluation Board

The `$PSS_ROOT/bsps/idt79465` directory contains a pSOSystem Board Support Package (BSP) for the IDT 79S465 Evaluation Board. The IDT 79S465 BSP supports the IDT R4640, R4650, R4700, and R5000 processors.

For R4640 and R4650 processors, see also IDT 79S5440 Daughtercard documentation.

For the R5000 processor, see also IDT 79S500 Daughtercard documentation.

A.2.1 Hardware Setup

[Table A-3](#) shows the IDT 79S465 Jumper/switch settings for the R4700 processor.

TABLE A-3 IDT 79S465 Jumper/switch Settings (R4700 Processor)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
J1	2-3	4 Mbyte SRAM	S1.1	On	R4k write-compatible mode
J2	2-3				
J3	2-3		S1.2	On	Clock divide by 2
J4	2-3		S1.3	On	
J5	2-3		S1.4	On	N/C
J6	2-3		S1.5	Off	DRAM Disabled
J7	2-3	X8-bit Flash	S1.6	Off	
J8	1-2		S1.7	On	SRAM Enabled
J9	1-2	4 Mbyte DRAM 64-bit mode	S1.8	Off	4 Mbyte SRAM
J10	1-2		S2.1	On	N/C
J11	1-2		S2.2	On	N/C
J12	1-2		S2.3	On	Clock divide by 2
J20	Close	Int. 5 routed to internal timer	S2.4	Off	Big Endian
J23	Close	Sync In Routed to CPU	S2.5	On	R4XXX Compatible mode
J24	1-2	Clock for R4700	S2.6	Off	64-bit bus mode

TABLE A-3 IDT 79S465 Jumper/switch Settings (R4700 Processor) (Continued)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
J25	Open	Clock for R4700	S2.7	On	100% CPU output drive strength
W1	1-2	4 Mbyte DRAM			
W2	1-2	4 Mbyte DRAM	S2.8	On	Int. 5 routed to internal timer
W3	1-2	5V System			

ROM Image Options

The IDT 79S465 Evaluation Board supports an 8-bit wide or a 32-bit wide Boot ROM interface. The jumper settings for the two options are in [Table A-4](#).

TABLE A-4 Flash Size for IDT 79S465 Evaluation Board

Jumper	X8 Bit	X32 Bit
J7	2-3	1-2
J8	1-2	2-3

The IDT79465 pSOSystem BSP supports both options.

RAM Options

The IDT 79S465 Evaluation Board supports three different RAM configuration options. These are the options:

- SRAM only
- DRAM only
- SRAM/DRAM

The switch settings for the three options are shown in [Table A-5](#).

TABLE A-5 RAM Options for IDT 79S465 Evaluation Board

Switch	SRAM Only	DRAM Only	SRAM/DRAM
S1.5	off	on	on
S1.6	off	on	on

TABLE A-5 RAM Options for IDT 79S465 Evaluation Board (Continued)

Switch	SRAM Only	DRAM Only	SRAM/DRAM
S1.7	on	off	on
S1.8	off	off	off

Ethernet Configuration

The pSOSystem BSP for the IDT79S465 board supports the Ethernet interface on the target board. The Ethernet interface is located on the AUI port marked J22. The Ethernet hardware address is a software configuration parameter for the IDT 79S465 Evaluation Board. This parameter should be set according to the Ethernet address assigned to the board. Consult the IDT documentation that comes with the board for the proper address. The Ethernet hardware address is configured in the pSOS startup dialog.

Serial Configuration

The pSOSystem BSP for the IDT 79S465 Evaluation Board supports two serial channels. The pSOSystem serial channel number 1 corresponds to the port marked J16 on the IDT 79S465 board, and channel number 2 corresponds to the port marked J17 on the board.

The pSOSystem Boot ROM shipped with this BSP uses serial channel 1 as the system console. The default serial protocol is 9600 baud, 8-bit data, 1 stop bit and no parity. You should connect a terminal (or terminal emulator) to the proper port.

SCSI Configuration

The pSOSystem BSP for the IDT 79S465 board supports the SCSI Bus interface on the target board. The SCSI interface is located on the port marked J13, and requires a special cable from IDT. Fuse F1 must be in place for the SCSI interface to work properly.

A.2.2 pSOSystem Boot Configuration

This section describes the various methods for configuring Boot ROMs for the IDT79S465 Evaluation Board.

Boot ROM images are configured in the `rom.dld` file in the `PSS_BSP` directory. The default boot configuration copies text and data to RAM and executes text from RAM. This configuration provides the fastest code execution, but requires more RAM

space. The default configuration requires one Megabyte of RAM space to operate properly. When building Boot ROMs with the default configuration, the `SC_RAM_SIZE` parameter in `sys_conf.h` must be set to `0xFF000`.

An alternative boot configuration is to copy data to RAM and leave text in ROM. This configuration will use less RAM space, but the code will execute slower. (Execution speed can be increased by using 32-bit wide Boot ROMs.) In this configuration, the `SC_RAM_SIZE` parameter in `sys_conf.h` can be left at its default value. To set up this alternative boot configuration, perform the following procedure:

1. Make a backup copy of `rom.dld`:

```
UNIX:  cp rom.dld rom.dld.org
PC:    xcopy rom.dld rom.bak
```

2. Edit the `rom.dld` file as follows:

- a. Search for an entry named:

```
.CpSrcBg (TEXT) : {}
```

- b. Move this entry so that it is just after the entry:

```
.textend (TEXT) : {}
```

- c. The very next entry should read:

```
} > mem8
```

- d. Change this entry so that it reads:

```
} > mem7
```

- e. Next, search for an entry named:

```
.CpDstBg (TEXT) : {}
```

- f. Move this entry so that it is just after the entry:

```
.data (DATA) : {}
```

- g. Save the new `rom.dld` file and follow the instructions for building pSOSystem Boot ROMs in the next section.

NOTE: In this configuration, the `mem6` memory definition in `ram.dld` can be made larger. For example, if `SC_RAM_SIZE = 0x31000`:

```
mem6:0x80031000 1 = 0x3CF000 /* SRAM/DRAM */
```

A.2.3 Building pSOSystem Boot ROMs

The boot ROM for the IDT79S465 Evaluation Board is built using the `tftp` sample application located in `$PSS_ROOT/apps/tftp`. Perform the following procedure to build new boot ROMs:

1. Copy `$PSS_ROOT/apps/tftp` to a working directory, and make the working directory the current directory:

```
UNIX:
% cp -r $PSS_ROOT/apps/tftp $PSS_ROOT/apps/idt79465boot
% cd $PSS_ROOT/apps/idt79465boot
```

```
PC:
> xcopy apps\tftp apps\idt79465boot /E
> cd apps\idt79465boot
```

2. Set the `PSS_BSP` environment variable to the absolute pathname of the IDT79465 BSP, as shown in the following example:

```
UNIX:
% setenv PSS_BSP ${PSS_ROOT}/bsps/idt79465
```

```
PC:
> set PSS_BSP = %PSS_ROOT%\bsps\idt79465
```

3. Depending on your boot configuration, edit `sys_conf.h` and change `SC_RAM_SIZE` to the appropriate value. The `SC_RAM_SIZE` value specifies the maximum amount of RAM available to the `tftp` Boot ROM.
4. Make the `tftp` application with the following command:

```
psosmake roms
```

The resulting image files are Motorola Srecord files. The 8-bit Boot ROM image file is named `rom.0` and the 32-bit Boot ROM image files are named `rom.u51 - rom.u54`. The 8-bit wide ROM must be placed in socket `u51` and the 32-bit wide ROMs must be placed in the sockets corresponding to the extension of the image file with which they were programmed.

A.2.4 Memory Layout and Usage

This section describes the memory layout for using the IDT79465 BSP.

Memory Layout

The IDT79S465 board comes default with 4 megabytes of SRAM and 4 megabytes of DRAM. The physical memory layout of the three RAM configuration options is described here: :

SRAM only	start	0x00000000
	end	0x003FFFFFFF
DRAM only	start	0x00000000
	end	0x003FFFFFFF
SRAM/DRAM	SRAM start	0x00000000
	SRAM end	0x003FFFFFFF
	DRAM start	0x00400000
	DRAM end	0x007FFFFFFF

The IDT79S465 peripherals are mapped as follows::

ROM	1FC00000 - 1FDFFFFFFF
Expansion CS	1F700000 - 1F7FFFFFFF
Ethernet	1F600000 - 1F6FFFFFFF
NVRAM	1F500000 - 1F5FFFFFFF
SCSI	1F400000 - 1F4FFFFFFF
SERIAL	1F300000 - 1F3FFFFFFF

Memory Usage

Table A-6 shows the ROM/RAM memory usage map for pSOSystem boot ROM.

TABLE A-6 ROM/RAM Usage Map for pSOSystem Boot ROM (IDT 79S465 Board)

ROM/RAM	Memory	Usage
ROM	0xBFC00000 – 0xBFC001FF	Reset vector
	0xBFC00200 – 0xBFC0027F	Bootstrap TLB vector
	0xBFC00280 – 0xBFC002FF	Bootstrap extended TLB vector
	0xBFC00300 – 0xBFC0037F	Bootstrap cache error vector
	0xBFC00380 – 0xBFC003FF	Bootstrap general vector
	0xBFC00400 – 0xBFC0047F	Bootstrap P3 vector
	0xBFC00480 – 0xBFC00FFF	Unused
	0xBFC01000 – 0xBFC7FFFF	pSOSystem Boot ROM text and initialized data
RAM	0x80000000 – 0x8000007F	TLB vector
	0x80000080 – 0x800000FF	Extended TLB vector
	0x80000100 – 0x8000017F	Cache error vector
	0x80000180 – 0x800001FF	General vector
	0x80000200 – 0x8000027F	P3 vector
	0x80000280 – 0x80000FFF	Reserved for pSOSystem
RAM	0x80001000 – 0x80001000 + SC_RAM_SIZE	Reserved for pSOSystem Boot ROM application
	0x80001000 + SC_RAM_SIZE – 0x803FFFFF	Free

A.2.5 Devices Supported for the IDT 79465 Evaluation Board

[Table A-7](#) provides a list of the devices supported by the IDT79465 BSP.

TABLE A-7 Supported Devices IDT 79S465 Evaluation Board

Device	Support	Description
National DP83932	<code>bsps/devices/lan/dp83932c</code>	Sonic Network Interface Controller
Zilog 85C30	<code>bsps/devices/serial/z85230.c</code>	Serial Communications Controller
NCR 53C90A	<code>bsps/devices/scsi/ncr53c90.c</code>	SCSI chip
R4XXXX	<code>bsps/devices/timer/r4000t.c</code>	R4000 internal timer

A.2.6 Miscellaneous

For SNIFF users, `src/.sniff1.lst` contains a list of all the pSOSystem files that make this BSP. The file is used by `bin/source/plugins/scripts/plugins_*` scripts to create a precise SNIFF+ project for the BSP. If you create a custom BSP using this BSP as a template and want to use the `plugins` script, update this file.

For further IDT79S465 board-specific information, see the *IDT 79S465 Evaluation Board Hardware User's Manual*.

A.3 IDT79S440 Board

The IDT79S440 Daughtercard interfaces an R4640 or R4650 processor to the IDT79S465 Evaluation Board. The R4640 processor operates in 32-bit bus mode only, and the R4650 processor can operate in 32-bit or 64-bit bus mode. The IDT79S465 board (described in [Section A.2 on page A-3](#)) provides setup information for the IDT79S465 board with an R4700 processor. This section describes additional information specific to the IDT79S440 Daughtercard.

A.3.1 Hardware Setup

Table A-8 shows the IDT 79S465 Jumper/switch settings for IDT79S440 in 32-bit bus mode.

TABLE A-8 IDT 79S465 Jumper/switch Settings (32-bit Bus Mode)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
J1	2-3	2 Mbyte SRAM	S1.1	On	R4k write-compatible mode
J2	2-3		S1.2	On	Clock divide by 3
J3	1-2				
J4	2-3		S1.3	Off	
J5	2-3		S1.4	On	N/C
J6	1-2		S1.5	Off	DRAM Disabled
J7	2-3	X8-bit Flash	S1.6	Off	
J8	1-2		S1.7	On	SRAM Enabled
J9	2-3	2 Mbyte DRAM 32-bit mode	S1.8	Off	2 Mbyte SRAM
J10	2-3		S2.1	On	N/C
J11	2-3		S2.2	On	N/C
J12	2-3		S2.3	On	Clock divide by 3
J20	Close	Int. 5 routed to internal timer	S2.4	Off	Big Endian
J23	Close	Sync In Routed to CPU	S2.5	On	R4XXX Compatible mode
J24	1-2	Clock for 4640/4650	S2.6	On	32-bit bus mode
J25	Open		S2.7	On	100% CPU output drive strength
W1	2-3	2 Mbyte DRAM	S2.8	On	Int. 5 routed to internal timer
W2	2-3				
W3	1-2	5V System			
NOTE: The 32-bit conversion kit (# IDT79S467) must be installed to run in 32-bit mode.					

Table A-9 shows the IDT 79S465 Jumper/switch settings for the IDT79S440 in 64-bit bus mode.

TABLE A-9 IDT 79S465 Jumper/switch Settings (64-bit Bus Mode)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
J1	2-3	4 Mbyte SRAM	S1.1	On	R4k write-compati- ble mode
J2	2-3				
J3	2-3		S1.2	On	Clock divide by 3
J4	2-3		S1.3	Off	
J5	2-3		S1.4	On	N/C
J6	2-3		S1.5	Off	DRAM Disabled
J7	2-3	X8-bit Flash	S1.6	Off	
J8	1-2		S1.7	On	SRAM Enabled
J9	1-2	4 Mbyte DRAM 64-bit mode (See NOTE:)	S1.8	Off	4 Mbyte SRAM
J10	1-2		S2.1	On	N/C
J11	1-2		S2.2	On	N/C
J12	1-2		S2.3	On	Clock divide by 3
J20	Close	Int. 5 routed to internal timer	S2.4	Off	Big Endian
J23	Close	Sync In Routed to CPU	S2.5	On	R4XXX Compatible mode
J24	1-2	Clock for 4640/4650	S2.6	Off	64-bit bus mode
J25	Open		S2.7	On	100% CPU output drive strength
W1	1-2	4 Mbyte DRAM	S2.8	On	Int. 5 routed to internal timer
W2	1-2				
W3	1-2	5V System			
NOTE: Only the R4650 processor can run in 64-bit bus mode. The R4640 must not be operated in this mode.					

RAM and ROM options described in [IDT 79S465 Evaluation Board](#) on page A-3 are also valid for the IDT79S440 Daughtercard.

NOTE: The IDT 79S466 Daughtercard will work with the same IDT79S465 Evaluation Board jumper/switch settings described in Tables A-8 and A-9.

Table A-10 gives the jumper settings for the IDT79S440 with R4650 enabled and for the IDT79S465 with R4640 enabled, respectively.

TABLE A-10 Jumper Settings for IDT 79S440 (R4650 Enabled) and IDT 79S465 (R4640 Enabled)

Jumper/ Switch	IDT 79S440 with R4650 Enabled	IDT 79S465 with R4640 Enabled
	Setting	Setting
W1	2-3	2-3
W2	2-3	2-3
W3	2-3	2-3
W4	2-3	2-3
W5	1-2	2-3
W6	1-2	1-2
W7	1-2	2-3
W8	2-3	1-2
W9	2-3	1-2
W10	1-2	2-3
W11	1-2	2-3
W12	1-2	2-3
J3	2-3	2-3
J4	open	open

The IDT79S440 Daughtercard operates at a different clock frequency than the default clock frequency in the IDT79S465 BSP. The CPU clock frequency setting must be changed for accurate timing. To change the CPU clock frequency, edit the file `bspcfg.c` in the `PSS_BSP` directory and change this variable:

```
const ULONG cpuClkFreq=50000000;
```

to the proper value and recompile the application.

For further IDT79S440 Daughtercard-specific information, see the *IDT 79S440 Evaluation Board Hardware User's Manual*.

A.4 IDT79S500 Board

The IDT79S500 Daughtercard interfaces the R5000 processor to the IDT79S465 Evaluation Board. The IDT79S465 board (described in [Section A.2 on page A-3](#)) provides setup information for the IDT79S465 board with an R4700 processor. This section describes additional information specific to the IDT79S500 Daughtercard.

A.4.1 Hardware Setup

CAUTION: Failure to connect a 5V power supply to connector J 12 may result in damage to both the 79S500 Daughtercard and 79S465 Evaluation Board.

[Table A-11](#) shows the IDT 79S465 Jumper/switch settings for IDT79S500.

TABLE A-11 IDT 79S465 Jumper/switch Settings (for IDT79S500)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
J1	2-3	4 Mbyte SRAM	S1.1	On	R4k write-compatible mode
J2	2-3		S1.2 *	Off	Clock divide by 4 (See NOTE:)
J3	2-3			On	
J4	2-3		S1.4	On	N/C
J5	2-3		S1.5	Off	DRAM Disabled
J6	2-3		S1.6	Off	DRAM Disabled
J7	2-3	X8-bit Flash	S1.7	On	SRAM Enabled
J8	1-2	X8-bit Flash	S1.8	Off	4 Mbyte SRAM
J9	1-2	4 Mbyte DRAM 64-bit mode	S2.1	On	N/C
J10	1-2		S2.2	On	N/C
J11	1-2		S2.3	On	Clock divide by 4
J12	1-2				

TABLE A-11 IDT 79S465 Jumper/switch Settings (for IDT79S500) (Continued)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description			
J20	Close	Int. 5 routed to internal timer	S2.4 **	Off	N/C (See NOTE:)			
J23	Close	Sync In Routed to CPU		On	R4XXX Compatible mode			
J24	2-3	Clock for 5000	S2.6	Off	64-bit bus mode			
J25	Close	Clock for 5000	S2.7	On	100% CPU output drive strength			
W1	1-2	4 Mbyte DRAM						
W2	1-2	4 Mbyte DRAM						
W3	1-2	5V System	S2.8	On	Int. 5 routed to internal timer			
NOTE: * For older boards with R5000 < 180 MHz: <div>S1.2 On S1.3 Off</div>								
NOTE: ** Switch S2.4 must be On for the 2 X clock option								

RAM and ROM options described in [IDT 79S465 Evaluation Board on page A-3](#) are also valid for the IDT79S500 Daughtercard.

[Table A-12](#) gives the IDT79S500 jumper settings for the 1 X Clock Option.

TABLE A-12 IDT 79S500 Jumper Settings (1 X Clock Option)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
W1	1-2	Normal Master Out to Socket	W10	2-3	512kb Secondary Cache
			W11	open	SRAM only (See NOTE:)
W2	1-2	Normal Master Clock to Socket	W12	1-2	1 X Clock
			W13	open	Big Endian
W3	1-2	Normal TClock1	W14	open	Sysclock from S500 oscillator
W4	1-2	Normal TClock0			
W5	1-2	Normal RClock1	W15	open	N/C

TABLE A-12 IDT 79S500 Jumper Settings (1 X Clock Option) (Continued)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
W6	1-2	Normal RClock0	W16	1-2	One buffer delayed clock for read operations
W7	1-2	512 KB Secondary Cache			
W8	1-2		W17	2-3	Refer to W16 for buffer delays
W9	2-3				
NOTE: For the DRAM only or DRAM/SRAM option, W11 must be shorted.					

The IDT79S500 Daughtercard operates at a different clock frequency than the default clock frequency in the IDT79S465 BSP. The CPU clock frequency setting must be changed for accurate timing. To change the CPU clock frequency, edit the file `bspcfg.c` in the `PSS_BSP` directory and change the variable:

```
const ULONG cpuClkFreq=50000000;
```

to the proper value and recompile the application.

For further information and 2 X Clock Jumper settings, see the *IDT 79S500 Evaluation Board Hardware User's Manual*.

A.5 LSI4101 Board

The `$PSS_ROOT/bsps/lsi4101` directory contains a pSOSystem BSP for the μ Meteor MiniRISC BDMR400X Evaluation Board, and the μ Meteor TinyRISC BDM4101 Evaluation Board. The MiniRISC Evaluation Board supports the LSI400X processors, and the TinyRISC Evaluation Board supports the LSI4101 processor. The LSI4101 processor is a MIPS 16 ISA-compatible processor.

A.5.1 Hardware Setup

Table A-13 shows the MiniRISC jumper settings for LSI400X. Table A-14 on page A-17 shows the TinyRISC jumper settings for LSI4101.

TABLE A-13 MiniRISC Jumper Settings (LSI400X)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
J1	Out	Big Endian	J11	Out	8 word I Cache Refill Size
J2	In	Write Burst Request	J12	In	Configuration Register

TABLE A-13 MiniRISC Jumper Settings (LSI400X) (Continued)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
J3	Out	D Cache Wraparound	J13	Out	4 word D Cache Refill Size
J4	In	No I Cache Wraparound	J14	In	Configuration Register
J5	In	0 SRAM Wait States	J15	In	4 word D Cache Refill Size
J6	In		J16	In	Configuration Register
J7	Out	Half speed BCLK	J17	In	On board 3.3V source
J8	In	MR400X as Arbiter	J18	In	
J9	Out	8 word I Cache Refill Size	J19	1-2	Sonic Enable
			J20	Out	Test Point
J10	In	Configuration Register	J21	In	ThinNet Ethernet

TABLE A-14 TinyRISC Jumper Settings (LSI4101)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
J1	Out	Big Endian	J13	Out	4 word D Cache Refill Size
J2	Out	No Write Burst Request	J14	In	
J3	Out	D Cache Wraparound	J15	In	4 word D Cache Refill Size
J4	In	No I Cache Wraparound	J16	In	Configuration Register
J5	Out	3 SRAM Wait States	J18	Out	Arbiter Enabled
J6	Out	3 SRAM Wait States	J19	Out	Enable Sonic Link
J7	Out	Half speed BCLK	J20	In	Connect 3.3V
J8	In	MR4101 as Arbiter	J21	In	

TABLE A-14 TinyRISC Jumper Settings (LSI4101) (Continued)

Jumper/ Switch	Setting	Description	Jumper/ Switch	Setting	Description
J9	Out	8 word I Cache Refill Size	J22	In	Power to CPU
J10	In	Configuration Register	J23	In	
J11	Out	8 word I Cache Refill Size	J25	1-2	Onboard oscillator for ICE port
J12	In	Configuration Register	J26	In	Serial Port B on J27

RAM Options

The MiniRISC and TinyRISC Evaluation Boards support both SRAM and DRAM. Both boards come default with one megabyte of onboard SRAM, and an optional four-megabyte or eight-megabyte DRAM module. The LSI4101 pSOSystem BSP can operate in either RAM space. Note that DRAM and SRAM are not contiguous.

Ethernet Configuration

The pSOSystem BSP for the LSI4101 board supports the Ethernet interface on the target board. The Ethernet interface for the MiniRISC board is located on the AUI port marked U72 or the BNC port marked U63 (see jumper J21 for port selection). The Ethernet interface for the TinyRISC board is located on the RJ45 port marked J29.

The Ethernet hardware address is a software configuration parameter for both the MiniRISC and TinyRISC Evaluation Boards. This parameter should be set according to the Ethernet hardware address assigned to the board. Consult the LSI documentation that comes with the board for the proper address. The Ethernet hardware address is configured in the pSOS startup dialog.

Serial Configuration

The pSOSystem BSP for the MiniRISC and TinyRISC Evaluation Boards supports two serial channels. The pSOSystem serial channel number 1 corresponds to the port marked U28 on the MiniRISC board and J28 on the TinyRISC board. Serial channel number 2 corresponds to the port marked U43 on the MiniRISC board and J27 on the TinyRISC board.

NOTE: The pSOSystem BSP for the LSI4101 TinyRISC Evaluation Board does not support the serial ICE port.

The pSOSystem Boot ROM shipped with this BSP uses serial channel 1 as the system console. The default serial protocol is 9600 baud, 8-bit data, 1 stop bit and no parity. You should connect a terminal (or terminal emulator) to the proper port.

The m68681 DUART serial chip implements two fixed baud rate sets for both channels. To select the baud rate set, set `BD_M68681_BRG_SET` in `board.h` to the desired set. Refer to the *M68681 User's Manual* for available baud rates in each set.

NVRAM

There is no Non-Volatile storage available on the MiniRISC and TinyRISC Evaluation Boards. All startup parameters are saved in SRAM and are not retained when power is removed from the board.

A.5.2 pSOSystem Boot Configuration

This section describes the various methods for configuring Boot ROMs for the MiniRISC and TintRISC Evaluation Boards.

Boot ROM images are configured in the `rom.dld` file in the `PSS_BSP` directory. The default boot configuration copies text and data to RAM and executes text from RAM. This configuration provides the fastest code execution but requires more RAM space. The default configuration requires one Megabyte of RAM space to operate properly. When building Boot ROMs with the default configuration, the `SC_RAM_SIZE` parameter in `sys_conf.h` must be set to `0xFF000`.

An alternative boot configuration is to copy data to RAM and leave text in ROM. This configuration will use less RAM space, but the code will execute slower. In this configuration, the `SC_RAM_SIZE` parameter in `sys_conf.h` can be left at its default value.

To set up this alternative boot configuration, perform the following procedure:

1. Make a backup copy of `rom.dld`:

```
UNIX:  cp rom.dld rom.dld.org
PC:    xcopy rom.dld rom.bak
```

2. Edit the `rom.dld` file as follows:

- a. Search for an entry named:

```
.CpSrcBg (TEXT) : {}
```

- b. Move this entry so that it is just after the entry:

```
.textend (TEXT) : {}
```

- c. The very next entry should read:

```
} > mem5
```

- d. Change this entry so that it reads:

```
} > mem4
```

- e. Next, search for an entry named:

```
.CpDstBg (TEXT) : {}
```

- f. Move this entry so that it is just after the entry:

```
.data (DATA) : {}
```

- g. Save the new `rom.dld` file and follow the instructions for building pSOSystem Boot ROMs in the next section.

NOTE: In this configuration, SRAM can be used for downloading applications, but first the `mem3` definition in `ram.dld` must be changed to reserve space for Boot ROMs. For example, if `SC_RAM_SIZE = 0x31000`:

```
mem3:0x80031000 1 = 0xCF000 /* SRAM */
```

A.5.3 Building pSOSystem Boot ROMs

The boot ROMs for the MiniRISC and TinyRISC Evaluation Boards are built using the `tftp` sample application located in `$PSS_ROOT/apps/tftp`. Perform the following procedure to build new boot ROMs:

1. Copy `$PSS_ROOT/apps/tftp` to a working directory, and make the working directory the current directory:

UNIX:

```
% cp -r $PSS_ROOT/apps/tftp $PSS_ROOT/apps/lsi4101boot
% cd $PSS_ROOT/apps/lsi4101boot
```

PC:

```
> xcopy apps\tftp apps\lsi4101boot
> cd apps\lsi4101boot
```

2. Set the PSS_BSP environment variable to the absolute pathname of the LSI4101 BSP, as shown in the following example:

```
UNIX:
% setenv PSS_BSP ${PSS_ROOT}/bsps/lsi4101
```

```
PC:
> set PSS_BSP = %PSS_ROOT%\bsps\lsi4101
```

3. Depending on your boot configuration, edit `sys_conf.h` and change `SC_RAM_SIZE` to the appropriate value. The `SC_RAM_SIZE` value specifies the maximum amount of RAM available to the `tftp` Boot ROM.
4. Make the `tftp` application with the following command:

```
psosmake roms
```

The resulting image files are Motorola Srecord files. The Boot ROM image files are named `rom.u5` - `rom.u8`. The ROMs must be placed in the sockets corresponding to the extension of the image file with which they were programmed for the TinyRISC board.

For the MiniRISC board,

- `rom.u5` goes in socket u8
- `rom.u6` goes in socket u19
- `rom.u7` goes in socket u29
- `rom.u8` goes in socket u40

A.5.4 Memory Layout and Usage

This section describes the memory layout for using the LSI4101 BSP.

Memory Layout

The MiniRISC and TinyRISC boards come default with one megabyte of SRAM and an optional four- or eight-megabyte DRAM module. The physical memory layout of the RAM is described here:

SRAM	start	0x00000000	
	end	0x0000FFFF	
DRAM	start	0x01000000	
	end	0x013FFFFFF	(4 megabyte option)
	end	0x017FFFFFF	(8 megabyte option)

The MiniRISC and TinyRISC peripherals are mapped as follows::

ROM	1FC00000 - 1FDFFFFF
Ethernet	1C000000 - 1C0000FF
Serial	1E000000 - 1E00003F

Memory Usage

Table A-15 shows the ROM/RAM memory usage map for pSOSystem boot ROM for the MiniRISC and TinyRISC boards.

TABLE A-15 ROM/RAM Usage Map for pSOSystem Boot ROM (LSI Boards)

ROM/ RAM	Memory	Usage
ROM	0xBFC00000 - 0xBFC000FF	Reset vector
	0xBFC00100 - 0xBFC0017F	Bootstrap TLB vector
	0xBFC00180 - 0xBFC001FF	Bootstrap general vector
	0xBFC00200 - 0xBFC00FFF	Unused
	0xBFC01000 - 0xBFC7FFFF	pSOSystem Boot ROM text and initialized data
SRAM	0x80000000 - 0x8000007F	TLB vector
	0x80000080 - 0x800000FF	General vector
	0x80000100 - 0x800003FF	Unused
SRAM	0x80000400 - 0x80000FFF	Reserved for pSOSystem
	0x80001000 - 0x80001000 + SC_RAM_SIZE	Reserved for pSOSystem Boot ROM application
	0x80001000 + SC_RAM_SIZE - 0x800FFFFF	Free

TABLE A-15 ROM/RAM Usage Map for pSOSystem Boot ROM (LSI Boards) (Continued)

ROM/ RAM	Memory	Usage
DRAM	0x81000000 – 0x813FFFFF	Free (4 megabyte option)
	0x81000000 – 0x817FFFFF	Free (8 megabyte option)

A.5.5 Devices Supported for the MiniRISC and TinyRISC Evaluation Boards

Table A-16 provides a list of the devices supported by the LSI4101 BSP.

TABLE A-16 Supported Devices for LSI4101 BSP

Device	Support	Description
National DP83932	bsps/devices/lan/dp83932.c	Sonic Network Interface Controller
Motorola M68681	bsps/devices/serial/m6861.c	DUART
AMD 29F0x0	bsps/devices/common/29f0x0.c	Flash memory

A.5.6 MIPS16 Support

The LSI4101 processor supports the MIPS16 ISA. Applications can be compiled for the MIPS16 ISA by two methods:

- To compile the entire application using the MIPS16 ISA, edit the application **makefile**. Find the definition for `PSS_APPCOPTS` and add the compiler switch for MIPS16 ISA. The new definition should be:

```
PSS_APPCOPTS = -tMIPS16EN:psos
```

- To compile individual C files using the MIPS16 ISA, edit the application **makefile**. Find the rule for the C file and add the compiler switch for MIPS16 ISA.

The following example shows what needs to be added (indicated by the bold text):

```
root.o:root.c \
    sysconf.h \
    makefile
$(CC) $(COPTS) -o root.o $<
```

will be changed to:

```
root.o:root.c \
    sysconf.h \
    makefile
$(CC) $(COPTS) -tMIPS16EN:psos -o root.o $<
```

A.5.7 Miscellaneous

For SNIFF users, `src/.sniff1.lst` contains a list of all the pSOSystem files that make this BSP. This file is used by `bin/source/plugins/scripts/plugins_*` scripts to create a precise SNIFF+ project for the BSP. If you create a custom BSP using this BSP as a template and wish to use the `plugins` script, update this file.

For further MiniRISC Evaluation Board-specific information, see the following:

- *MiniRISC BDMR400X Evaluation Board User's Guide*
- *MiniRISC CW400X Microprocessor Core Technical Manual*

For further TinyRISC Evaluation Board-specific information, see the following:

- *TinyRISC BDMR4101 Evaluation Board User's Guide*
- *TinyRISC TR4101 Microprocessor Core Technical Manual*

B

pRISM+ Environment Variables

This appendix describes how you can set up your pRISM+ environment. In this appendix you will learn what variables are available for modification.

The following sections contain an explanation of the environment variables set in the script. Each explanation is followed by the relevant code. All examples show the default values set by the pRISM+ installation.

NOTE: For simplicity, the remainder of this appendix makes use of the “ppc” target indicator. If you are using a 68k, x86, MIPS, or ARM targets, simply substitute “68k”, “x86”, “mip”, or “arm” wherever “ppc” is used.

B.1 pRISM+ Variables for the Windows Environment

The environment variable `PSS_ROOT` must be set to point to the directory that contains pSOSystem:

```
PSS_ROOT="C:/PRISM_INST_DIR/pssppc.<ver>"
```

`$PSS_ROOT\bin\${HOST}` must be added to your path so pSOSystem can find the binaries it needs for various utilities.

```
HOST=win32  
PATH="$PSS_ROOT/bin/${HOST};$PATH"
```

The environment variable `PSS_BSP` must point to the directory that contains your board support package. (Replace the path in the line below with the path to your target BSP).

```
PSS_BSP="$PSS_ROOT\\bsps\\mbx8xx"
```

NOTE: Use the pRISM+ Manager to set the BSP you want to use. See the pRISMSpace settings dialog in the on-line help.

The environment variable `BSP_TYPE` must be set when building application for ARM or THUMB processors. This variable is used to specify which execution model the application should be built for. The following table shows the `BSP_TYPE` values:

BSP_TYPE	Execution Model
321	ARM Mode 32-bit) Little Endian
32b	ARM mode (32-bit) Big Endian
161	THUMB mode (16-bit) Little Endian
16b	THUMB mode (16-bit) Big Endian

To build an ARM mode Little Endian application use the following syntax:

```
BSP_TYPE=32e1
```

ARM Compiler and Debugger Environment Variables

The ARM compiler and debugger require two environment variables called `ARMINC` and `ARMLIB`. These variables direct the compiler where to find include files and library files (for linking). In addition to these variables, the `BIN` directory must be added to the Windows `PATH`:

```
ARMINC="C:/PRISM_INST_DIR/ARM211.a/INCLUDE"
ARMLIB="C:/PRISM_INST_DIR/ARM211.a/LIB/EMBEDDED"
PATH="C:/PRISM_INST_DIR/ARM211.a/BIN;%PATH"
```

Diab Data Environment Variables

NOTE: The Diab Data environment variables are only for the 68K, MIPS, and PowerPC target processors.

The environment variable `DIABLIB` must be set to point to the directory where you installed the Diab Data compiler. This enables the compiler to find its libraries, headers and binaries. Also, the compiler's binary directory must be added to the `PATH`:

```
DIABLIB="C:/PRISM_INST_DIR/Diab/4.3p5"
PATH="%DIABLIB%/Bin;%PATH"
```

SingleStep Environment Variables

NOTE: The SingleStep environment variables are only for the 68K and PowerPC target processors.

For SingleStep, add the binary directory to the `PATH`. SingleStep can find all the other pieces it needs relative to the executable that is run:

```
PATH="C:/PRISM_INST_DIR/sds74/cmd;$PATH"
```

SNiFF+ Environment Variables

The environment variable `SNIFF_DIR` must point to the SNiFF+ installation directory. Also, the SNiFF+ binary directory must be added to the `path`:

```
SNIFF_DIR="C:/PRISM_INST_DIR/Sniff"
PATH="$SNIFF_DIR/Bin;$PATH"
```

The `IT_CONFIG_PATH` variable points to the Orbix configuration directory and is required by both the Orbix daemon and SNiFF+.

```
IT_CONFIG_PATH="$PRISM_INST_DIR/orbix"
PATH="$PRISM_INST_DIR/orbix;$PATH"
```

MKS Toolkit Environment Variables

The `ROOTDIR` and `SHELL` environment variables must be set using UNIX-style forward slashes instead of DOS-style back-slashes. `ROOTDIR` points to the base of the MKS executables. It also locates other files needed by the MKS tools:

```
ROOTDIR="c:/isi<target>/sniff/mks/mks-6.1"
```

`SHELL` is set to the full path and file name of the `sh` (ksh) executable for MKS:

```
SHELL="$ROOTDIR/mksnt/Sh.exe"
```

The environment variable `TMPDIR` must be set to an existing directory. The installation copies the value from either the `TEMP` or `TMP` variable in the MS-DOS environment, so really all three variables (`TMPDIR`, `TEMP`, and `TMP`) should point to the same directory. Also, add the binary directory for the MKS tools to the `PATH`:

```
TMPDIR="C:/TEMP"
PATH="$ROOTDIR/mksnt;$PATH"
```


CAD-UL Environment Variables

NOTE: The CAD-UL environment variables are only for the X86 target processor.

The environment variable CC386TMP must be set to an existing directory. The compiler and XDB debugger's binary directory must be added to the PATH:

```
CC386TMP="C:/PRISM_INST_DIR/CADUL/TMP"
PATH="C:/PRISM_INST_DIR/CADUL/BIN;$PATH"
PATH="C:/PRISM_INST_DIR/CADUL/XBD/X364b1XX;$PATH"
```

pRISM+ Variables

Set the environment variable PRISMDIR to the directory that contains the pRISM+ binaries:

```
PRISM_DIR="C:/PRISM_INST_DIR/pRISM+"
```

Set the environment variable CONFIG to the name of the configuration file to include in builds (config.mk, configxx.mk, etc.):

```
CONFIG="config"
```

Add the binary directory to the PATH:

```
TCL_LIBRARY="$PRISM_INST_DIR/pRISM+/Lib/PrismPlusShell/library"
PATH="$PRISM_INST_DIR/licenses/Bin/$HOST;PRISM_DIR/
bin;$PRISM_INST_DIR/JRE/1.1.7/Bin;$PATH"
```

Also, add the system directory to the path. This enables the pRISM+ executables to pick up the DLLs they need:

```
PATH=".;$PATH"
```

Additional PATH and Windows Settings

The pRISM+ installation path is added before the existing PATH to ensure that the pRISM+ executables come before any files you had in your PATH prior to the installation.

- USERNAME is set to the login name of the current user.

```
USERNAME="PRISM_INST_DIR/pRISM+/bin/MyName.exe"
```

```
LOGNAME = "$USERNAME"
```

- On both 95 and NT, we set the HOME variable to
"C:\\PRISM_INST_DIR\\Users\\\$USERNAME"

License File Environment Variable

All the pRISM+ tools use FLEXlm for licensing and will add the `LM_LICENSE_FILE` variable to locate the license file:

```
LM_LICENSE_FILE="C:\\PRISM_INST_DIR\\Licenses\\License.dat"
```

B.2 pRISM+ Variables for the UNIX Environment

The following table contains descriptions of general-purpose shell environment variables.

[Table B-2](#) contains descriptions of the environment variables used by `psosmake`, the make facility pRISM+ calls by default.

TABLE B-1 General-Purpose Shell Environment Variables

Variable	Description
<code>\$PRISM_INST_DIR</code>	Set to the directory path where pRISM+ is installed.
<code>\$SNIFF_DIR</code>	Set to the directory path where SNIFF+ is installed
<code>\$PSS_ROOT</code>	Set to the directory path where pSOSystem is installed
<code>\$DIABLIB</code>	Set to the directory path for the directory where the Diab Data compiler suite is installed.
<code>\$LM_LICENSE_FILE</code>	<p>A list of files (full path name separated by a colon). The list must contain files that have the license keys for ESp, SNIFF+, and the compiler and debugger. (The information in the files is used by the FlexLM license manager to allocate a license to the user for these tools.)</p> <p>For more information about these files, see the installation publications for SNIFF+ 3.0.2 and the <i>SingleStep User Guide</i>.</p>
<code>\$LD_LIBRARY_PATH</code>	<p>Specifies an ordered list of directories.</p> <p>The information is used by the host operating system to search for shared libraries used by executables.</p>

Table B-2 contains a list of the variables required by `gmake`.

NOTE: Use the pRISM+ Manager to set the BSP you want to use. See the **pRISMSpace Settings** dialog in the on-line help.

TABLE B-2 Variables Required by `gmake`

Variable	Description
<code>\$PSS_BSP</code>	<p>Specifies the directory of the board support package for your target board.</p> <p>You must set this variable to <code>\$PSS_ROOT/bsps/board_name</code>, where <i>board_name</i> is a board support package provided by Integrated Systems.</p> <p>BSPs are located in the directory <code>\$PSS_ROOT/bsps</code> and have names that correspond to the boards they support.</p>

C

pRISM+ Supported Host/Target Connections

pRISM+ for pSOSystem offers many ways to communicate to your target. This appendix provides the requirements for your applications and hardware and host configuration for each communication option.

In this appendix, you will learn about the parameters and options that need to be set before you can compile your application. **You will also see what hardware and host configurations are required to use a particular communication mode.**

This appendix describes the following communication configurations:

- *Using a Serial Connection on page C-1*
- *Using an Ethernet Connection on page C-4*
- *Using a Communication Server Remotely on page C-7*
- *Using the TFTP Server on page C-9*

C.1 Using a Serial Connection

This section provides the necessary information on how to use pRISM+ with a serial connection to communicate to the target board. It is recommended that you use the pRISM+ Tutorial in [Chapter 3](#) before developing your own application. This section refers back to the pRISM+ Tutorial.

C.1.1 Building a pSOSystem Application

Sys_conf.h Settings

To configure your application to communicate to the pRISM+ tools through a serial connection, you must set the following parameters in your application's `sys_conf.h` file.

TABLE C-1 `sys_conf.h` File Settings

Parameters	Settings
SC_PROBE	YES
SC_PROBE_DEBUG	YES
SC_PMONT	YES (optional)
SC_SD_DEBUG_MODE	Storage or host/serial
SD_DEF_BAUD	Default is 9600
SC_DEV_SERIAL	Serial channel is in the form of a port number and a driver number.
SC_RBUG_PORT	Should not be the same value as SC_APP_CONSOLE and, if using pMONT+ and ESsp, PM_DEV

C.1.2 Configuring Target Environment

Set up your hardware connection as shown in the following figure.

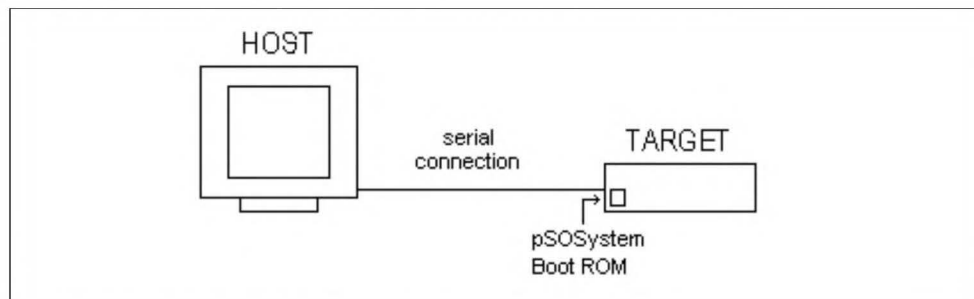


FIGURE C-1 Serial Hardware Configuration

The building and installation of the boot ROMS are defined in [Appendix A](#).

C.1.3 Configuring Target Communications Parameters

To boot your application, refer to [Section 3.8, *Configuring the Target Communications Parameters on page 3-20*](#). When booting your pSOSystem, you must remember three things:

1. Set your communication mode to 2, as shown in the example below:

For each of the following questions, press <Return> to keep the value in braces, or you can enter a new value.

How should the board boot?

1. pROBE+ standalone mode
2. pROBE+ waiting for host debugger via a serial connection
3. pROBE+ waiting for host debugger via a network connection
4. Run the TFTP bootloader

Which one do you want? [1]

Enter 2.

2. Set your baud rate.
3. When you have finished configuring and booting your application, you must disconnect from your HyperTerminal or `tip` session.

C.1.4 Configuring Host Tools Connection with the Target

Refer to [Section 3.9, *Adding a Target Board to the pRISM+ Target List on page 3-23*](#) for corresponding figures.

1. From pRISM+ Manager, select **Target** → **List**. The **Target List** dialog is displayed.
2. Click the **Add** button. The **Add Target** dialog is displayed.
3. Enter a name for your target and click **OK**. The **Target Properties** dialog is displayed.
4. In the **Target Properties** dialog, do the following:
 - a. Verify that **Server Selection** is set to **Use Local Communications server**.
 - b. Choose **Serial** in both the **pROBE Target Connection** and **pMONT Target Connection** areas.
 - c. In the **Port Name** field of both the **pROBE Target Connection** and **pMONT Target Connection** areas, enter the serial port names to be used for these connections.

Typically on a PC they can be `COM1` or `COM2`. On a Solaris machine they can be `/dev/ttya` or `/dev/ttyb`.

- d. In the **Baud Rate** field of both the **pPROBE Target Connection** and **pMONT Target Connection** areas, set the baud rate. The default is 9600.
- e. Click **OK** to accept the information.
5. Click **Select**. This registers the target as the current target for the pRISMSpace.
6. Click **Close** to close the **Target List** dialog.

C.1.5 Using pRISM+ Tools

All the pRISM+ Tools that are in your pRISM+ Development Environment are available to you use when using a serial connection. For additional information on the pRISM+ Tools, refer to the corresponding chapters or to [Chapter 3, *Quick Start with a Tutorial*](#).

C.2 Using an Ethernet Connection

In [Chapter 3, *Quick Start with a Tutorial*](#), you used an Ethernet connection. This section provides the necessary information on how to use pRISM+ with an Ethernet connection to communicate to the target board. It is recommended that you use the pRISM+ Tutorial in [Chapter 3](#) before developing your own application. This section refers back to the pRISM+ Tutorial.

C.2.1 Building a pSOSystem Application

Sys_conf.h Settings

To configure your application to communicate to the pRISM+ tools through an Ethernet connection, you must set the following parameters in your application's `sys_conf.h` file.

TABLE C-2 `sys_conf.h` File Settings

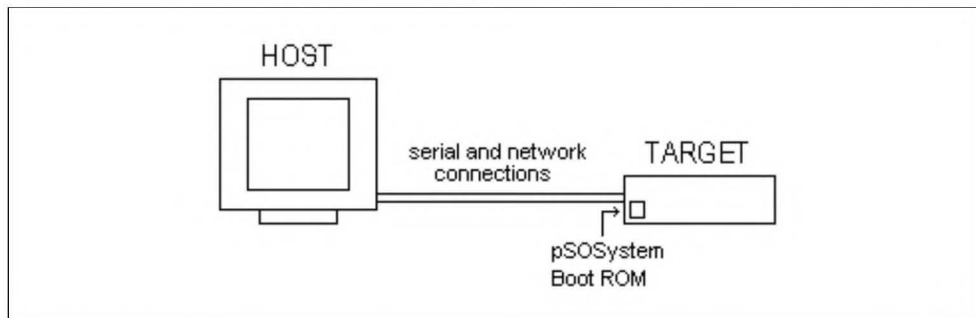
Parameters	Settings
SC_PROBE	YES
SC_PROBE_DEBUG	YES
SC_PMON	YES (optional)
SC_SD_DEBUG_MODE	Storage or host/network

TABLE C-2 `sys_conf.h` File Settings (Continued)

Parameters	Settings
SD_DEF_BAUD	Whatever the board supports
SC_DEV_SERIAL	To a non-zero value
SC_RBUG_PORT	Should not be the same value as SC_APP_CONSOLE and SC_PROBE_CONSOLE
SC_PNA or SC_PNET	YES

C.2.2 Configuring Target Environment

In order to download your application to the target, you must set up your hardware connection as shown in the following figure.

**FIGURE C-2** Ethernet Hardware Configuration

The building and installation of the boot ROMS are defined in [Appendix A](#).

C.2.3 Booting pSOSystem

To boot your application, refer to [Section 3.8, *Configuring the Target Communications Parameters on page 3-20*](#). When booting your pSOSystem, you must remember the following:

- Set your communication mode to 3, as shown in the example below:

For each of the following questions, press <Return> to keep the value in braces, or you can enter a new value.
How should the board boot?

```
1. pROBE+ standalone mode
2. pROBE+ waiting for host debugger via a serial connection
3. pROBE+ waiting for host debugger via a network connection
4. Run the TFTP bootloader
Which one do you want? [1]
```

Enter 3.

- Set your baud rate

C.2.4 Configuring Host Tools Connection with the Target

Refer to [Section 3.9, Adding a Target Board to the pRISM+ Target List on page 3-23](#) for corresponding figures.

1. From pRISM+ Manager, select **Target** → **List**. The **Target List** dialog is displayed
2. Click the **Add** button. The **Add Target** dialog is displayed.
3. Enter a name for your target and click **OK**. The **Target Properties Dialog** is displayed.
4. In the **Target Properties** dialog, do the following:
 - a. Verify that **Server Selection** is set to **Use Local Communications server**.
 - b. Choose **Network** in both the **pROBE Target Connection** and **pMONT Target Connection** areas.
 - c. In the **Network Name** field of both the **pROBE Target Connection** and **pMONT Target Connection** areas, enter either the name of your target (if you are correctly configured for DNS) or the IP address of your target.
 - d. Click **OK** to accept the information.
5. Click **Select**. This registers the target as the current target for the pRISMSpace.
6. Click **Close** to close the **Target List** dialog.

C.2.5 Using pRISM+ Tools

All the pRISM+ Tools that are in your pRISM+ Development Environment are available to you use when using an Ethernet connection. For additional information on the pRISM+ Tools, refer to the corresponding chapters or to [Chapter 3, Quick Start with a Tutorial](#).

C.3 Using a Communication Server Remotely

This section describes how to use a remote communication server.

C.3.1 Building a pSOSystem Application

To use the pRISM+ communication server remotely, your application must use the parameter settings defined for an Ethernet connection. See [Section C.2, Using an Ethernet Connection on page C-4](#).

C.3.2 Configuring Target Environment

In most cases the target host is connected to your PC or workstation. If the target host is connected to another PC or workstation you can still use that target board. (See [Figure C-3](#).) This section will describe how you can use a target host that is connect to another system.

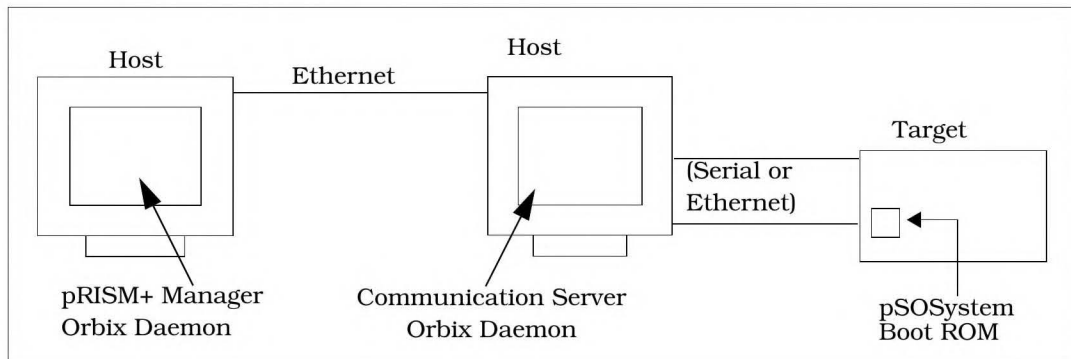


FIGURE C-3 Hardware Connection

1. From the pRISM+ Manager, select **File** → **proj1**. This project was created during the tutorial in [Chapter 3](#).
2. From the pRISM+ Manager toolbar, click **Target** → **List**. The **Target List** dialog box will display.
3. Click **Add**. The **Add Target** dialog will appear.
4. In the **Add Target** dialog, enter the name of your target in the **Target Name** field. The target name can be any name. In this instance use `targ2` for a target name. The **Properties for Target** dialog will appear. See [Figure C-4 on page C-8](#).

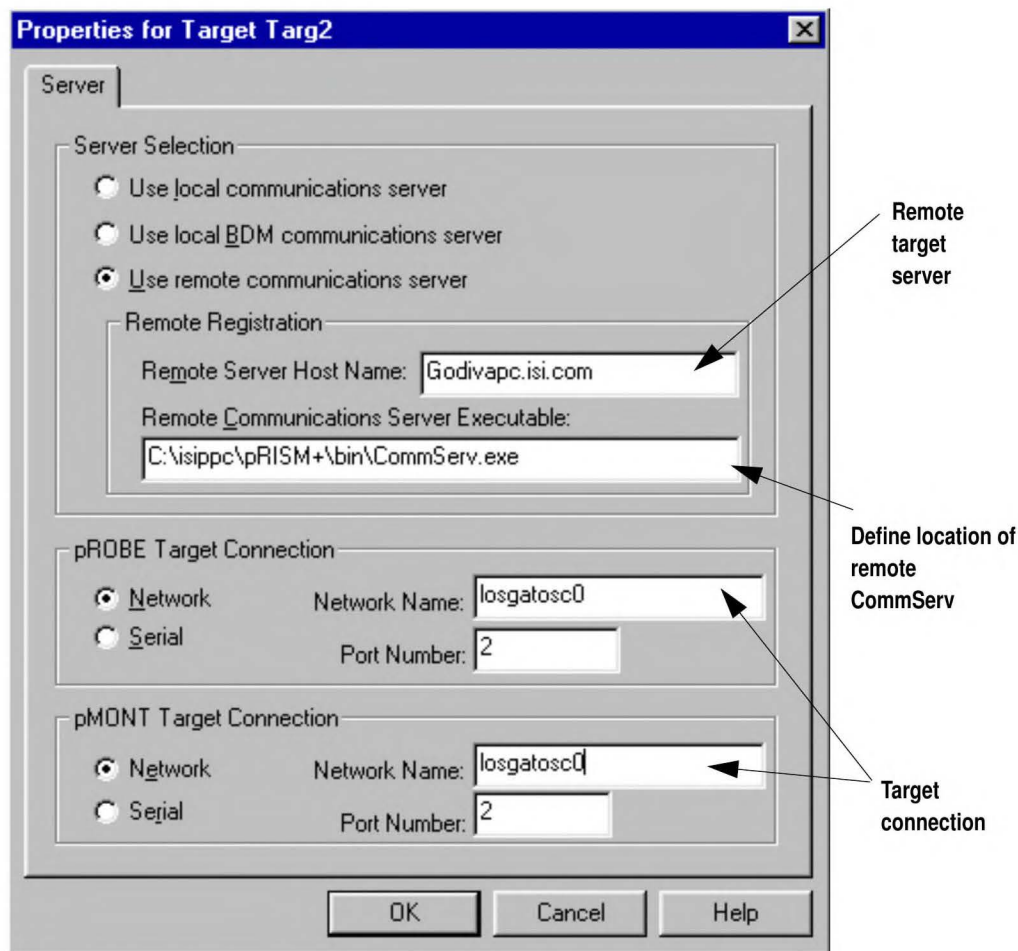


FIGURE C-4 Properties for Target Dialog

NOTE: The target server is only case-sensitive if it is connected to a UNIX workstation.

5. Define the target's properties in the dialog box.
 - a. In the **Server Selection** area, select **Use remote communications server**.

- b. In the **Network Name** field of both the **pROBE Target Connection** and **pMONT Target Connection** areas, enter either the name of your target (if you are correctly configured for DNS) or the IP address of your target.
 - c. In the **Remote Server Host Name** field of the **Remote Registration** area, enter the name of the PC or workstation that is connected to your target.
 - d. In the **Remote Communications Server Executable** field of the **Remote Registration** area, enter the path of the Communication Server. This communication server must be on the same system as the remote target server. The remote target server was defined in [step c](#).
 - e. Click **OK** to accept the changes.
6. In the **Target List** dialog, click **Close**.
7. On the remote target host server, start the **orbixd** daemon.
8. From the pRISM+ Manager, select **Target** → **target2**. This will launch the Communication Server on the remote host you defined in [step 5](#).

You are now ready to download your application to the remote target board.

C.3.3 Booting pSOSystem

When booting your pSOSystem, you need to use the same options described for an Ethernet connection. See [Section C.2, Using an Ethernet Connection on page C-4](#).

C.3.4 Using pRISM+ Tools

When using pRISM+ Tools, you can use the same types of tools described for an Ethernet connection. See [Section C.2, Using an Ethernet Connection on page C-4](#).

C.4 Using the TFTP Server

This section provides the necessary requirements in order to use the TFTP Server for Windows only. We recommend you use the pRISM+ Tutorial in [Chapter 3](#) before developing your own application. This section refers back to the pRISM+ Tutorial.

The TFTP Server for Windows supplied by ISI implements the common Trivial File Transfer Protocol (RFC 1350). The TFTP server allows a tftp client (typically a target system or a diskless node) to download a file (a boot image for a target system or diskless node). It only supports TFTP Read Request (RRQ), to transfer file to a target

(which runs the client side of the protocol). It does not support Write Request (WRQ) to transfer files to the server.

The TFTP server can be accessed by selecting **Start** → **Programs** → **pRISM+ 2.0<target_CPU>** → **Utilities** → **TFTP Server**. The `tftpd.exe` file is invoked and a windows application appears.

C.4.1 Building a pSOSystem Application

Target Hardware Requirement

TFTP Server is an independent application that does not depend on the target. However to use the TFTP server to download the boot image, the BootROM should have TFTP (client) support. If the TFTP is not built into the BootROM you will not be able to use the TFTP server option to boot the target.

pRISM+ Tools Supported

The pRISM+ Tools that are in your pRISM+ Development Environment are available for you to use when using this downloading mechanism. For additional information on the pRISM+ Tools, refer to the corresponding chapters or to the [Quick Start with a Tutorial](#) chapter.

C.4.2 Sys_conf.h Settings

TFTP Server does not require special setting in the `sys_conf.h` file. TFTP Server only requires that the BootROM has the TFTP support. If the TFTP is not built into the BootROM you will not be able to use the TFTP server option.

C.4.3 Configuring Target Environment

In this section you learn about the target environment requirements when using TFTP Server.

BootROM Settings

TFTP Server only requires that the target board BootROM has the TFTP support. If the TFTP is not built into the BootROM you will not be able to use the TFTP server option. Refer to [Appendix A](#) for additional information on how to create a bootROM using the TFTP application.

C.4.4 Configuring Host Environment

In this section you learn about the host environment requirements when using TFTP Server.

pRISM+ Host Setting

Before you can download your compiled executable code using the TFTP Server, you must configure the TFTP Server settings. See the section [Configure](#) on page C-11.

TFTP Server Commands

When invoked, the program opens a window and waits for the user to select a command from the **Tftpd** menu. A status line is displayed at the bottom of the window that displays the status of the server. The **Tftpd** menu supports the following commands:

Start

Starts the TFTP server. You must select this command to start the TFTP server. Before you can start the server you must configure it by selecting the **Configure** command from the menu.

Stop

This command is used to stop the TFTP server. Before stopping, the server waits for all existing client sessions to terminate. Once all sessions are closed, the TFTP server is stopped. To start the server again, you need to select the **Start** command.

Configure

Configures the TFTP server. You must issue this command before you can run the server. The following configuration entries are required.

TABLE C-3 Tftpd Settings Description

Options	Description
Home Directory	Home directory of the tftp server. This is the root directory for all download requests. You will have to copy your application boot image into this directory before the target can download the boot image.
Number Of Clients	This is the maximum number of simultaneous tftp clients supported.

TABLE C-3 Tftpd Settings Description (Continued)

Options	Description
Logging Desired	Specifies whether the tftp server's operation needs to be logged. This also includes error messages.
Log File Name	Specifies the filename into which the messages will be logged.
Verbose Logging	Controls the verbosity of log messages. If turned on, more messages are logged.

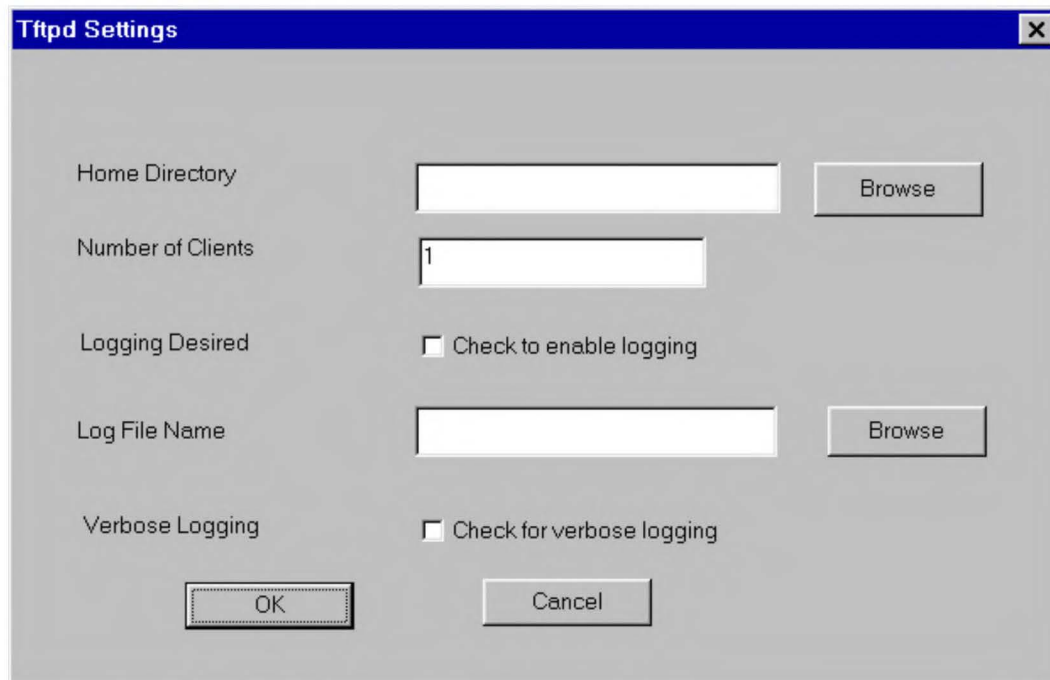


FIGURE C-5 Tftpd Settings Dialog Box

Exit

This command is used to exit the TFTP server program. When issued, the program simply exits. It does not wait for any existing clients to terminate.

C.4.5 Using the TFTP Server Connection

In this section you learn about how to configure your connection to use the TFTP Server, how to download your application, and how to connect to the pRISM+ Manager so you can use the pRISM+ Tools.

Configuring the TFTP Server

1. Select **Start** → **Programs** → **pRISM+ 2.0<target_CPU>** → **Utilities** → **TFTP Server**. The TFTP Server dialog box will display.
2. In the **TFTP Server** window, select **Tftpd** → **Configure**.
3. In the **TFTP Settings** dialog box, fill in all the fields in the dialog box. See [Figure C-5 on page C-12](#).

Once you defined the TFTP settings you can begin the downloading process.

Downloading your Application

You have defined the TFTP Server settings; you can begin the downloading process.

1. Select **Start** → **Programs** → **pRISM+ 2.0<target_CPU>** → **pROBE+ Console (COM1).ht** or **pROBE+ Console (COM2).ht**. The HyperTerminal window will display.
2. In the **TFTP Server** window, select **Tftpd** → **Start**. This starts the server.

Booting pSOSystem

1. Power on your target board or reset your target.
2. To boot your application, refer to [Section 3.8, *Configuring the Target Communications Parameters* on page 3-20](#).

When booting your pSOSystem, you must remember these things:

1. Set the communication mode to 4, as shown in the example below:

For each of the following questions, press <Return> to keep the value in braces, or you can enter a new value.

How should the board boot?

1. pROBE+ standalone mode
2. pROBE+ waiting for host debugger via a serial connection
3. pROBE+ waiting for host debugger via a network connection
4. Run the TFTP bootloader

Which one do you want? [1]

Enter 4.

2. Set the IP address of the target board.
3. Set the IP address of the Server host where the TFTP server is running.
4. Set the filename of host image.
5. If the target and host reside on different IP subnets, set the default gateway address to reach the host from the target.

A status message is displayed: TFTP download completed, transferring control to the download code.

Connecting to the pRISM+ Manager

1. Add your target board to the pRISM+ target list. Refer to [Section 3.9, *Adding a Target Board to the pRISM+ Target List* on page 3-23](#) for directions.
2. From pRISM+ Manager, select the target you defined in the previous step.
3. From the pRISM+ Manager toolbar, select **Target** → **Connect**. This connects your downloaded application to the pRISM+ Tools. You are now able to use all your pRISM+ Tools.

All the pRISM+ Tools that are in your pRISM+ Development Environment are available to you use when using this connection. For additional information on the pRISM+ Tools, refer to the corresponding chapters or to [Chapter 3, *Quick Start with a Tutorial*](#).

D

pRISM+ Shell Commands

The pRISM+ Shell, based on TCL, is part of the pRISM+ Development Environment. Unless otherwise noted, the pRISM+ Shell supports all functionality supported by TCL. This chapter describes all the pRISM+ Shell commands and their syntax; it also describes enhancements to the TCL commands that pRISM+ Shell supports.

For more information about the TCL language, see the scriptics web site at www.scriptics.com, or consult one of the numerous books available on the subject.

D.1 Overview

The pRISM+ Shell commands are functionally grouped in two categories: CommSrv (Communication Server) based commands and DbgSrv (Debug Server) based commands.

- The DbgSrv-based commands invoke and communicate with the target through the debug server.
- The CommSrv-based commands invoke and communicate with the target through the communication server.

The majority of pRISM+ Shell commands are CommSrv-based commands. If you are accustomed to using pROBE+ console commands, see [Section D.3](#) for a table comparing pRISM+ Shell commands to pROBE+ commands.

D

D.2 Communication Server- and Debug Server-Based Commands

This section documents the pRISM+ Shell commands in alphabetical order. Each entry provides a description, syntax, and examples of the commands.

Some pRISM+ Shell commands are shortcut equivalents to regular pRISM+ Shell commands with longer names; for example, command `cb` is a shortcut for the command `breakpoint clear`. These shortcut command names match the names of pROBE+ commands with identical (or similar) functionality.

This list summarizes the pRISM+ Shell commands and their basic functionality:

boot — *boot the operating system*
breakpoint — *manage instruction breakpoints*
cb — *clear breakpoints (shortcut)*
cn — *connect to a target (shortcut)*
comm — *display or set communication parameters*
condvar — *display information about conditional variables*
connect — *connect to a target*
csabout — *display CommSrv information*
db — *define a breakpoint (shortcut)*
dcn — *disconnect from a target (shortcut)*
debugger — *set and show debug session settings*
di — *disassemble instructions (shortcut)*
disassemble — *disassemble instructions*
disconnect — *disconnect from a target*
dl — *download a file from the host (shortcut)*
dm — *display memory (shortcut)*
dr — *display register (shortcut)*
dsession — *manipulate, open, or load the target through the pRISM+ Shell*
ev — *evaluate variable (shortcut)*
evaluate — *evaluate local and global variables*
evt — *set events (shortcut)*
fl — *display and set pROBE+ flags (shortcut)*
fm — *fill memory (shortcut)*
go — *continue execution of foreground tasks or halted application*
halt — *stop execution of target application or all foreground tasks*
he — *display summary of all pRISM+ Shell commands (shortcut)*
help — *display help for all the available pRISM+ Shell commands*
il — *display and set pROBE+ interrupt level (shortcut)*
init — *initialize pSOS+ on the target (shortcut)*
initialize — *initialize pSOS+ on the target*
lb — *list all breakpoints (shortcut)*

log — log packets to a log file (shortcut)
memory — allocate, deallocate, read, fill, and write ranges of memory
mod — set debugging mode (shortcut)
mutex — display information about mutual-exclusion objects
osbreakpoint — manage all operating-system-specific breakpoints
partition — display information about partitions
pm — patch memory (shortcut)
pr — patch register (shortcut)
probe — display and set pROBE+ flags and interrupt level
psos — make a pSOS+ system call
*q** — query-related commands (shortcuts)
queue — display information about queues
quit — close the session and exit the pRISM+ Shell window
region — display information about regions
register — manage task-specific and shared registers
sc — make a pSOS+ system call (shortcut)
semaphore — display information about semaphores
session — manipulate, open, and load the target through the pRISM+ Shell
sf — display stack frame information (shortcut)
stackfrm — display stack frame information
*t** — task-related commands (shortcuts)
target — manage target definitions
task — manage task operations
tsd — display task-specific data
version — display pRISM+ Shell version

boot boot the operating system

boot

Description

The `boot` command, normally used at the beginning of a target debug session, causes the operating system to be booted. By default, the debugger uses the entry point of the executable file for the boot address.

Examples

- To boot the operating system, using the entry point of the executable file for the boot address:

```
boot
```

See Also

[*initialize on page D-38*](#)

breakpoint

manage instruction breakpoints

```
breakpoint help
breakpoint show
breakpoint set (line|function) location_specifiers [count number] [disable]
breakpoint set address bp_address [ task (task_ID|*|isr) ]
breakpoint (clear|enable|disable) (bp_index|all)
```

Description

The `breakpoint` command displays syntax information; displays the status of all breakpoints; sets a breakpoint on a source-code line, a function, or an address; and enables, disables, or clears breakpoints.

Usage

breakpoint help

Displays the syntax of the `breakpoint` command.

breakpoint show

Displays all the breakpoints and the status of each (enabled or disabled).

```
breakpoint set line line_number source_file exe_file
[task (task_ID|*|isr)] [count number] [disable]
```

Sets a breakpoint on line *line_number* in the file *source_file* in the specified executable *exe_file* for the task *task*.

The `task` option can specify a task ID number, a "*" for "any task", or an ISR number. If you specify a task ID or an ISR, the breakpoint is specific to only that task or ISR. If you omit a `task`, any task or ISR can hit the breakpoint.

The `count` option specifies the number of times the line of code must execute before the breakpoint occurs. If you do not specify a `count number`, the breakpoint breaks the first time the line is reached.

The `disable` option specifies that the breakpoint is to be set but also disabled.

```
breakpoint set function function_name source_file exe_file
[task (task_ID|*|isr)] [count number] [disable]
```

Sets a breakpoint on function *function_name* in the file *source_file* in the specified executable *exe_file*.

D

The `task` option can specify a task ID number, a "*" for "any task", or an ISR number. If you specify a task ID or an ISR, the breakpoint is specific to only that task or ISR. If you omit a `task`, any task or ISR can hit the breakpoint.

The `count` option specifies the number of times function *function_name* must execute before the breakpoint occurs. If you do not specify a `count number`, the breakpoint breaks the first time the function is reached.

The `disable` option specifies that the breakpoint is to be set but also disabled.

breakpoint set address *bp_address* [task (*task_ID*|*|*isr*)]

Sets a breakpoint on the specified address. The `task` option can specify a task ID number, a "*" for "any task", or an ISR number. If you specify a task ID or an ISR, the breakpoint is specific to only that task or ISR. If you omit a `task`, any task or ISR can hit the breakpoint.

breakpoint clear (*bp_index*|all)

Removes a specified breakpoint or all breakpoints.

breakpoint enable (*bp_index*|all)

Activates the specified breakpoint or all breakpoints.

breakpoint disable (*bp_index*|all)

Deactivates the specified breakpoint or all breakpoints.

Examples

- To set a breakpoint at the 8th execution of line 404 of the source file `demo.c`, used in the executable file `ram.elf`, enter this command:

```
breakpoint set line 404 demo.c ram.elf count 8
```

- To set and disable a breakpoint at the entry point of function `process_data` in file `data.c`, used in executable file `ram.elf`, enter this command:

```
breakpoint set function process_data data.c ram.elf disable
```

- To set a breakpoint at address `0x0033BF4`, enter this command:

```
breakpoint set address 0x0033BF4
```

- To disable the breakpoint whose index is 2, enter this command:

```
breakpoint disable 2
```

- To clear the breakpoint whose index is 3, enter this command:

```
breakpoint clear 3
```

See Also

[cb](#) on page D-8

[db](#) on page D-14

cb clear breakpoints (shortcut)

cb (*bp_index*|all)

Description

The **cb** command clears a specified breakpoint, or all breakpoints. This command is a shortcut for the `breakpoint clear` command and the `osbreakpoint clear` command.

Usage

cb *bp_index*

Clears the breakpoint whose index number is *bp_index*.

cb all

Clears all breakpoints.

Examples

- To clear one breakpoint, where 5 is the breakpoint index:

cb 5

- To clear all breakpoints:

cb all

See Also

[breakpoint](#) on page D-5

[osbreakpoint](#) on page D-45

cn connect to a target (shortcut)

```
cn [ hot ] [ target_name ]
```

Description

The `cn` command opens both a CommSrv session and a DbgSrv session for debugging. This is a shortcut for the `connect` command.

Usage

```
cn hot
```

Connects to the default target, which is running.

```
cn [ hot ] target_name
```

Establishes contact to target *target_name* through the pRISM+ Shell. Use this command first, before you enter any other CommSrv commands. If you specify `hot`, the target is running; otherwise, the target is halted.

If the connection is through an Ethernet, *target_name* is the target's network name or its IP address. If the connection is serial, *target_name* is the target's serial port number (or name) and baud rate, separated by a comma.

Examples

- To connect to a running default target:

```
cn hot
```

- To connect through an Ethernet to a halted target and to a running target:

```
cn seant3
cn hot 152.216.226.158
```

- To connect through a serial port to a halted target and to a running target:

```
cn COM1,9600
cn hot /dev/ttya,19200
```

comm display or set communication parameters

```
comm [ timeout acktimeout retries ]
```

Description

The `comm` command, given without arguments, displays the current communication parameters of the CommSrv-to-pROBE+ connection. Given with arguments, the `comm` command sets the parameters as specified.

Usage

comm

Shows the current settings for communication parameters of the CommSrv-to-pROBE+ connection.

comm *timeout acktimeout retries*

Sets the specified communication parameters.

- *timeout* specifies how long the CommSrv must wait (in milliseconds) for a response to a request before it re-sends the request.
- *acktimeout* specifies how long the CommSrv must wait (in milliseconds) for an acknowledgement to a request before re-sending the request. In most cases you should adjust the *acktimeout* value before modifying the *retries* or *timeout* value(s).
- *retries* defines the number of times the CommSrv will re-send the same request if an acknowledgement is not received.

Examples

- To show the current communication parameters:

```
comm
```

- To set communication parameters: `timeout = 6000 ms`, `acktimeout = 300 ms`, `retries = 6 attempts`:

```
comm 6000 300 6
```

condvar

display information about conditional variables

```
condvar help
condvar show [ (condvar_ID|'condvar_name') ]
```

Description

The `condvar` command displays information about the conditional variables in the application.

Usage

```
condvar help
```

Displays the syntax of the `condvar` command.

```
condvar show [ (condvar_ID|'condvar_name') ]
```

Without arguments, `condvar show` displays a summary of all active conditional variables in the application. The display includes conditional-variable names and IDs, pSOS+m access, type of queue used, deferred signals, associated mutex name and ID, and task queue length.

If you specify a conditional variable by ID or name, this command displays the status (name, ID, access, queue type, etc.) for that variable.

Examples

- To show information about all the conditional variables:

```
condvar show
```

- To show detailed information about specified conditional variables:

```
condvar show 0x00170000
condvar show 'CV_1'
```

D

connect connect to a target

```
connect [ hot ] [ target_name ]
```

Description

The `connect` command opens both a CommSrv session and a DbgSrv session for debugging.

This is equivalent to a `session open` command followed by a `dsession open` command.

Usage

```
connect hot
```

Connects to the default target, which is running.

```
connect [ hot ] target_name
```

Establishes contact to target *target_name* through the pRISM+ Shell. Use this command first, before you enter any other CommSrv commands. If you specify `hot`, the target is running; otherwise, the target is halted.

If the connection is through an Ethernet, *target_name* is the target's network name or its IP address. If the connection is serial, *target_name* is the target's serial port number (or name) and baud rate, separated by a comma.

Examples

- To connect to a running default target:

```
connect hot
```

- To connect through an Ethernet to a halted target and to a running target:

```
connect seant3  
connect hot 152.216.226.158
```

- To connect through a serial port to a halted target and to a running target:

```
connect COM1,9600  
connect hot /dev/ttya,19200
```

csabout

display CommSrv information

```
csabout help
csabout (version|license)
```

Description

The `csabout` command displays information about the communication server in the application.

Usage

```
csabout help
```

Displays the syntax of the `csabout` command.

```
csabout version
```

Displays the version number of the communication server.

```
csabout license
```

Displays the license information of the communication server; for example:

```
This server is licensed!
Floating license
Total license 2
Inuse license 1
```

Examples

- To display the CommSrv version number:

```
csabout version
```

- To display the CommSrv license information:

```
csabout license
```


db define a breakpoint (shortcut)

```
db address [ (*|isr|task_ID) ]
db se system_call parameter origin
db di (task_ID|'task_name'|*)
db ti (ticks|date_and_time)
```

Description

The `db` command defines a breakpoint to be an instruction break, a pSOS+ service break, a dispatch break, or a timed break (relative or absolute).

Usage

```
db address [ (*|isr|task_ID) ]
```

Sets a breakpoint on an instruction at *address*. You can set the breakpoint for all tasks ("*"), an ISR, or a task specified by its task ID. This command is a shortcut for the `breakpoint set address` command.

```
db se system_call parameter origin
```

Defines a pSOS+ service-call break, which stops execution when the application makes a qualifying system call into the pSOS+ kernel.

This command is a shortcut for the `osbreakpoint set syscall` command. See the definition of *osbreakpoint* on [page D-45](#) for a discussion of the *parameter* and *origin* components of the `db se system_call` command.

Refer to [Table D-1 on page D-47](#) for a complete list of the qualifying pSOS+ system calls and their parameters.

```
db di (task_ID|'task_name'|*)
```

Sets a dispatch breakpoint on the task specified by *task_ID* or *task_name*, or on any task (if you specify "*"). This command is a shortcut for the `osbreakpoint set dispatch` command.

```
db ti (ticks|date_and_time)
```

Sets a relative timer breakpoint or an absolute timer breakpoint. If you specify *ticks*, the `db ti` command sets a relative timer breakpoint to occur *ticks* clock ticks after the target has started running. If you specify *date_and_time*,

`db ti` sets an absolute timer breakpoint to occur at the specified date and time. Argument *date_and_time* takes this form:

date_num-month-year hours:minutes:[seconds]

where valid values for each *date_and_time* component are as follows:

<i>date_num</i>	01 – 31	<i>hours</i>	01 – 24
<i>month</i>	JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC	<i>minutes</i>	0 – 59
<i>year</i>	0 – 9999	<i>seconds</i>	0 – 59

The *seconds* component is optional; if not specified, it defaults to 0.

Examples

- To set a breakpoint at address 0x10438c:
`db 0x10438c`
- To set a system-call breakpoint at `q_send` for any task:
`db se q_send * *`
- To set a dispatch breakpoint for task 0x160000:
`db di 0x160000`
- To set a relative timed break to occur 100 clock ticks after the target has started running:
`db ti 100`
- To set an absolute timed break:
`db ti 01-AUG-1999 8:30:21`

See Also

[breakpoint](#) on page D-5

[osbreakpoint](#) on page D-45

dcn disconnect from a target (shortcut)

dcn

Description

The `dcn` command closes both the CommSrv session and the DbgSrv session. This command is a shortcut for the `disconnect` command.

Examples

- To disconnect the current connected target:

```
dcn
```

See Also

[disconnect](#) on page D-21

[dssession](#) on page D-25

[session](#) on page D-72

debugger

set and show debug session settings

```
debugger help
debugger show
debugger set (timeout|retries|acktimeout) number
debugger include filename
```

Description

The `debugger` command modifies the debugger object, which represents attributes that are global to all debug sessions. These attributes are generally set once and remain unchanged during the rest of the session.

Usage

debugger help

Displays the syntax of the `debugger` command.

debugger show

Shows the current settings for communication parameters of the CommSrv-to-pROBE+ connection.

debugger set acktimeout *milliseconds*

Specifies how long the CommSrv must wait (in milliseconds) for an acknowledgement to a request before re-sending the request. In most cases you should adjust the `acktimeout` value before modifying the `retries` or `timeout` value(s).

debugger set retries *decimal_number*

Defines the number of times the CommSrv will re-send the same request if an acknowledgement is not received.

debugger set timeout *milliseconds*

Specifies how long the CommSrv must wait (in milliseconds) for a response to a request before it re-sends the request.

debugger include *filename*

Reads in and executes a pRISM+ Shell command file.

Examples

- To show the debug session settings:

```
debugger show
```

- To set the `timeout` parameter to 4000 milliseconds:

```
debugger set timeout 4000
```

- To set the `acktimeout` parameter to 100 milliseconds:

```
debugger set acktimeout 100
```

- To set the `retries` parameter to 3:

```
debugger set retries 3
```

- To include a source file:

```
debugger include /homes/hair/ShellTest.tcl  
debugger include c:\homes\hair\ShellTest.tcl
```

di disassemble instructions (shortcut)

di *address* [*number*]

Description

The **di** command disassembles instructions, starting at the specified address. The default number of instructions to be disassembled is 10; to disassemble a different amount, you must specify the *number* argument.

Usage

di *address*

Disassemble 10 instructions, starting from address *address*.

di *address number*

Disassemble *number* instructions (where *number* is a decimal integer), starting from address *address*.

Examples

- To disassemble 10 instructions (the default amount), starting from address 0x10438c:

```
di 0x10438c
```

- To disassemble 20 instructions, starting from address 0x10438c:

```
di 0x10438c 20
```

See Also

[disassemble on page D-20](#)

disassemble disassemble instructions

```
disassemble help
disassemble address start_addr [ number ]
disassemble line start_line end_line filename
```

Description

The `disassemble` command disassembles instructions starting at a specified address or over a specified range of source-file lines. The default number of instructions to be disassembled is 10.

Usage

```
disassemble help
```

Displays the `disassemble` command syntax and available options.

```
disassemble address start_addr [ number ]
```

Disassembles *number* instructions starting from address *start_addr*.

```
disassemble line start_line end_line filename
```

Disassembles instructions from line *start_line* to line *end_line* of source file *filename*.

Examples

- To disassemble 10 instructions from start address 0x10438c

```
disassemble address 0x10438c
```
- To disassemble 20 instructions from start address 0x104350

```
disassemble address 0x104350 20
```
- To disassemble instructions from line 414 to line 416 in file `demo.c`

```
disassemble line 414 416 demo.c
```

See Also

[di](#) on page D-19

disconnect disconnect from a target

disconnect

Description

The `disconnect` command closes both the `CommSrv` session and the `DbgSrv` session. This command is equivalent to a `session close` command followed by a `dssession close` command.

Examples

- To disconnect the current connected target:

```
disconnect
```

See Also

[connect](#) on page D-12

[dcn](#) on page D-16

[dssession](#) on page D-25

[session](#) on page D-72

dl download a file from the host (shortcut)

```
dl filename [ [ all ] |symbol|image ]
```

Description

The `dl` command loads/downloads the specified file *filename* from the host to the DbgSrv or the target, as appropriate. This is a shortcut for the `dssession load` command.

Options are to do the following; load the symbol table into the DbgSrv and download the executable image to the target; load the symbol table into the DbgSrv only; download the executable image to the target only.

Usage

```
dl filename [ all ]
```

Load the symbol table of file *filename* into the DbgSrv and download the executable image of file *filename* from the host to the target. This is the default.

```
dl filename symbol
```

Load the symbol table of *filename* into the DbgSrv, only.

```
dl filename image
```

Download the executable image of *filename* from the host to the target, only.

Examples

- To load the symbol table of file `ram.elf` to the DbgSrv and download its executable image to the target:

```
dl c:/isi/users/me/apps/pdemo/ram.elf
dl \isi\users\me\apps\pdemo\ram.elf
```

- To load the symbol table of the same `ram.elf` file to the DbgSrv, only:

```
dl c:/isi/users/me/apps/pdemo/ram.elf symbol
dl \isi\users\me\apps\pdemo\ram.elf symbol
```

dm display memory (shortcut)

dm [*.width*] (*address*|*start_addr*..*stop_addr*)

Description

The **dm** command displays memory units in the specified width (unit size); either for 0x40 bytes starting at the specified address, or over the specified range. The **dm** command is a shortcut for the `memory read` command.

Usage

dm [*.width*] *address*

Displays 64 (0x40) bytes of memory starting at address *address*. Without a *.width* argument, each unit of memory is one byte wide (the default). With a *.width* argument, each unit of memory has width *width*, where valid values for *width* are *c* (for char, one byte, the default), *s* (for short, two bytes), and *l* (for long, four bytes).

dm [*.width*] *start_addr*..*stop_addr*

Displays a range of memory, starting at address *start_addr* and ending at address *stop_addr*. Without a *.width* argument, the memory is displayed in one-byte units, the default. With a *.width* argument, the memory is displayed in units of width *width*, where valid values for *width* are *c* (for char, one byte, the default), *s* (for short, two bytes), and *l* (for long, four bytes). The two dots (*.* *.*) are required to indicate a range of memory.

Examples

- To display memory in short (two-byte) units, from address 0x1040 for the default 64 (0x40) bytes (to address 0x1080):

```
dm.s 0x1040
```

- To display memory in char (one-byte) units (the default) from address 0x1040 to address 0x1090:

```
dm 0x1040..0x1090
```

dr display register (shortcut)

```
dr ( [ general ] | fpu) [ task_ID ]  
dr (mmu|control)
```

Description

The `dr` command displays the names and numbers of the general or floating-point registers of a task, or of the MMU or control registers for the entire application. This command is a shortcut for the `register show category` command.

Usage

```
dr [ general ] [ task_ID ]
```

Displays the general register values. This is the default. If no *task_ID* is specified, the contents of the registers are displayed relative to the default task.

```
dr fpu [ task_ID ]
```

Displays the floating-point (FPU) register values. If no *task_ID* is specified, the contents of the FPU registers are displayed relative to the default task.

```
dr (mmu|control)
```

Displays the MMU or control register values. (MMU and control registers are shared by all tasks).

Examples

- To display the floating-point unit register values of task 0x160000:

```
dr fpu 0x160000
```

- To display MMU register values of the target application:

```
dr mmu
```

See Also

[register](#) on page D-68

dsession manipulate, open, or load the target through the pRISM+ Shell

```
dsession help  
dsession (open|close)  
dsession reset  
dsession load file_name [ all|symbol|image ]
```

Description

The `dsession` command manipulates, opens, and loads the target through the pRISM+ Shell. This command can also initiate a target debug session, which begins when you open a connection to a specified target.

Usage

dsession help

Displays the `dsession` command syntax and available options.

dsession open

Establishes contact to an existing predefined target through the pRISM+ Shell. Use this command first, before entering any other commands.

dsession close

Closes the existing debug session. Use this command before exiting the debugger shell.

dsession reset

Reestablishes contact to an existing predefined target through the pRISM+ Shell.

dsession load *filename* [all|symbol|image]

Loads/downloads the specified file *filename* to the DbgSrv or the target, as appropriate. Options are to do the following:

- `all` — load the symbol table into the DbgSrv and download the executable image to the target
- `symbol` — load the symbol table into the DbgSrv only
- `image` — download the executable image to the target only

Examples

- To open a debug session for the DbgSrv, enter this command; it will return a session number:

```
dsession open
```

- To close a debug session for the DbgSrv, enter this command before exiting the debugger shell:

```
dsession close
```

- To load the symbol table of file ram.elf to the DbgSrv and download its executable image to the target:

```
dsession load c:/isi/users/me/apps/pdemo/ram.elf  
dsession load isi\users\me\apps\pdemo\ram.elf
```

- To load the symbol table of the same ram.elf file to the DbgSrv, only:

```
dsession load c:/isi/users/me/apps/pdemo/ram.elf symbol  
dsession load \isi\users\me\apps\pdemo\ram.elf symbol
```

- To reset the target connection, enter this command:

```
dsession reset
```

See Also

[session on page D-72](#)

ev evaluate variable (shortcut)

ev *var_name frame_number [task_ID]*

Description

The **ev** command evaluates the named variable in the specified stack frame of a task (default or specified). The **ev** command is a shortcut for the **evaluate** command.

Usage

ev *var_name frame_number*

Evaluates variable *var_name* in frame *frame_number* of the default task.

ev *var_name frame_number task_ID*

Evaluates variable *var_name* in frame *frame_number* of the task *task_ID*.

Examples

- To evaluate variable *i* in frame 0 of the task whose ID is 0x1b0000:

```
ev i 0 0x1b0000
```

- To evaluate variable *i* of frame 0 in the default task:

```
ev i 0
```

See Also

[evaluate on page D-28](#)

evaluate evaluate local and global variables

evaluate help

evaluate *var_name* frame *frame_number* [task *task_ID*]

Description

The `evaluate` command evaluates local and global variables. Use this command with the `stackfrm` command.

Usage

evaluate help

Displays syntax for the `evaluate` command.

evaluate *var_name* frame *frame_number* [task *task_ID*]

Evaluates variable *var_name* in frame *frame_number* of the specified task *task_ID*, or of the default task (if no *task_ID* specified).

Examples

- This example shows how to use the `evaluate` command:

- a. Set the default task:

```
task set 0x00170000
```

- b. Show the stack frames for the default task:

```
stackfrm show
```

- c. Evaluate variable `i` of frame 0 in the default task (as set in the preceding step [a.](#)):

```
evaluate i frame 0
```

See Also

[ev](#) on page D-27

[stackfrm](#) on page D-77

evt set events (shortcut)

evt *event_code_mask*

Description

You can receive information from CommSrv about certain events occurring in the target, such as an instruction break, creation or deletion of an object, an output request, and so on. Each event is represented by an event code, which can be ORed together to generate an event mask.

To receive information about an event, you must first register with CommSrv for that event (that is, “set” the event) with the `evt` command.

When you have registered for an event, CommSrv reports occurrences of that event in the pRISM+ Shell window. This is a shortcut for the `target set event` command. See the description of *target* on [page D-80](#) for a list of event codes.

Examples

- To register for target I/O request events:

```
evt 0x1800
```

fl display and set pROBE+ flags (shortcut)

fl [*flag* (on|off)]

Description

The **fl** command displays and sets pROBE+ flags. This is a shortcut for the `probe set flag flag_type` command.

Usage

fl

Displays the pROBE+ flags and their current status.

fl *flag* (on|off)

Sets the specified pROBE+ flag either ON or OFF. The available pROBE+ flags are `nopage`, `nomanb`, `nodots`, `rbug`, and `smode`. For an explanation of the meaning of these flags, see the entry for the [probe](#) command on [page D-53](#).

Examples

- To display pROBE+ flags:

```
fl
```

- To set pROBE+ flags to ON or OFF as needed:

```
fl nomanb on
fl nodots off
fl nopage off
fl smode on
fl rbug off
```

See Also

[probe](#) on [page D-53](#)

fm fill memory (shortcut)

fm [*.width*] *start_addr..stop_addr data*

Description

The **fm** command fills a range of memory with a specified value. This is a shortcut for the **memory fill** command. The two dots (*. .*) are required to indicate a range of memory.

Usage

fm *start_addr..stop_addr data*

Fills a range of memory in one-byte units with the value of *data*, starting at address *start_addr* and ending at address *stop_addr*.

fm.*width* *start_addr..stop_addr data*

Fills a range of memory with the value of *data*, starting at address *start_addr* and ending at address *stop_addr*. The value of *data* is zero-extended to 1, 2, or 4 bytes, as specified by *width*. Valid values for *width* are *c* (for char, one byte, the default), *s* (for short, two bytes), and *l* (for long, four bytes).

Examples

- To fill address range 0x10 to 0x20 with the one-byte value 0x7b:

```
fm.c 0x10..0x20 0x7b
```

See Also

[dm](#) on page D-23

[memory](#) on page D-41

go continue execution of foreground tasks or halted application

go

Description

In system debug mode (SDM), the `go` command continues execution of your halted target application.

In task debug mode (TDM), this command continues the execution of all foreground tasks.

Examples

- To run the halted target program:

```
go
```

See Also

[halt on page D-33](#)

halt stop execution of target application or all foreground tasks

halt

Description

In system debug mode (SDM), the `halt` command stops execution of the target application.

In task debug mode (TDM) the `halt` command causes all foreground tasks to stop executing. Use `halt` only if the foreground tasks are currently executing. Once the foreground tasks are halted, the setting for the default task changes.

Examples

- To stop the running target:

```
halt
```

See Also

[go on page D-32](#)

he display summary of all pRISM+ Shell commands (shortcut)

he

Description

The `he` command prints a list of all pRISM+ Shell commands and a brief description of what each command does. This command is a shortcut for the `help` command.

Examples

- To display all the pRISM+ Shell commands:

```
he
```

See Also

[*help on page D-35*](#)

help display help for all the available pRISM+ Shell commands

```
help
help [ command_name ]
```

Description

The `help` command provides information about pRISM+ Shell commands, their purpose and syntax.

Usage

```
help
```

Displays a list of all pRISM+ Shell commands and a brief description of what each command does.

```
help command_name
```

Displays information about the pRISM+ Shell command *command_name*, including its syntax and options, and gives examples of how the command is used.

Examples

- To display all the pRISM+ Shell commands:

```
help
```
- To display the syntax and options of the `breakpoint` command:

```
help breakpoint
```

il display and set pROBE+ interrupt level (shortcut)

```
il [ i_level ]
```

Description

The `il` command displays the current pROBE+ interrupt level and sets the pROBE+ interrupt level to level *i_level*.

Usage

```
il
```

Displays the current pROBE+ interrupt level. This is a shortcut for using the `probe show` command.

```
il i_level
```

Sets the pROBE+ interrupt level to level *i_level*. This is a shortcut for the `probe set ilevel i_level` command.

- On the MIPS processor, the interrupt level can range from 0 to 1.

For more information about the available interrupt levels, see the *Programmer's Reference* manual.

Examples

- To set the pROBE+ interrupt level to 1 (one):

```
il 1
```

See Also

[probe](#) on page D-53

init initialize pSOS+ on the target (shortcut)

init

Description

The `init` command initializes or re-initializes pSOS+ on the target. This command is a shortcut for the `initialize` command.

Examples

- To initialize your downloaded application:

```
init
```

See Also

[boot on page D-4](#)

[initialize on page D-38](#)

initialize initialize pSOS+ on the target

initialize

Description

The `initialize` command initializes or re-initializes pSOS+ on the target.

Examples

- To initialize your downloaded application:

```
initialize
```

See Also

[*boot* on page D-4](#)

[*init* on page D-37](#)

lb list all breakpoints (shortcut)

lb

Description

The `lb` command lists all current breakpoints and their status. This is a shortcut for the `breakpoint show` command and the `osbreakpoint show` command.

Examples

- To list all breakpoints:

```
lb
```

See Also

[*breakpoint* on page D-5](#)

[*cb* on page D-8](#)

[*db* on page D-14](#)

[*osbreakpoint* on page D-45](#)

log log packets to a log file (shortcut)

log (*log_file*|end)

Description

The `log` command turns on and off the logging of packets exchanged between pROBE+ and CommSrv to a specified log file. This is a shortcut for the `session log` command.

Usage

log *log_file*

Starts the logging of packets to the log file *log_file*.

log end

Turns off the logging of packets started with a previous `log log_file`.

Examples

- To generate this log file, datapkt.txt:

```
Send: QUERY_RQT Type: AllRegions, Request: NEW, Node: -1
Packet Dump:
  0:  0C 84 00 00 00 00 00 00  FF FF FF FF  .....
Recv: QUERY_RPY status: pROBE_OK, More?: FALSE, Partial?: FALSE, NItems: 2
Packet Dump:
  0:  8C 00 01 00 00 00 00 02  52 4E 23 30 00 00 00 00  .....RN#0.
 16:  00 30 00 80 00 7F FF 00  00 00 01 00 00 76 D4 00  .0.....
 32:  00 76 AA 00 00 00 00 00  00 00 00 00 52 4D 45 4D  .v.....R
 48:  00 22 00 00 00 AE A5 00  00 00 08 00 00 00 00 80  ."......
 64:  00 00 07 00 00 00 07 00  00 00 00 00 00 00 00 00  .....
```

enter this series of commands:

```
log datapkt.txt
qr
log end
```

See Also

[session on page D-72](#)

memory allocate, deallocate, read, fill, and write ranges of memory

```
memory help
memory allocate ulong_units
memory deallocate address
memory read address [ width (1|2|4) ] [ count number ]
memory fill address value data [ width (1|2|4) ] [ count number ]
memory write address value data [ data ... ] [ width (1|2|4) ]
```

Description

The `memory` command allocates, deallocates, reads, fills, and writes ranges of memory.

Usage

`memory help`

Displays the syntax of the `memory` command.

`memory allocate ulong_units`

Sets aside *ulong_units* (a decimal integer > 0) of memory, where each unit is the size of an unsigned long, so the application can use the memory later. The `memory allocate` command returns the start address of the allocated block.

`memory deallocate address`

Frees a block of previously allocated memory, starting at address *address*.

`memory read address [width (1|2|4)] [count number]`

The `memory read` command reads the contents of a region of memory, starting at address *address*. If you specify a *width* argument *W* and a *count* argument *number*, the memory is read as *number* *W*-byte-sized units. Default values for *W* and *number* are 4 and 1, respectively.

`memory fill address value data [width (1|2|4)] [count number]`

The `memory fill` command fills a region of memory with a specified value, starting at address *address*.

The value of the *data* argument is zero-extended to 1, 2, or 4 bytes, as specified by the *width* qualifier (4 is the default width). The zero-extended *data* argument is duplicated *number* times (1 is the default value of *number*).

```
memory write address value data [ data ... ] [ width (1|2|4) ]
```

The `memory write` command modifies the contents of a memory address by writing the data element(s) to memory, starting at address *address*. The value of each *data* argument is zero-extended to 1, 2, or 4 bytes, as necessary, as specified by the *width* qualifier (4 is the default width).

NOTE: If a *data* element is larger than the specified *width* value, CommSrv truncates it and takes the least significant bits. For example, if you attempt to write 0x1234 to a byte, CommSrv writes it as 0x34.

Examples

- To read memory contents starting from memory address 0x003F1BC4 and reading four units (*count* 4) of 1-byte data (*width* 1):

```
memory read 0x003F1BC4 width 1 count 4
```

- To write to memory starting at memory address 0x003F1BC4, writing data items that are each two bytes wide (*width* 2), consisting of data item 0x1234 and data item 0x5678:

```
memory write 0x003F1BC4 value 0x1234 0x5678 width 2
```

- To fill memory, starting at address 0x003F1BC4, with the value 0x55AA66BB — which is a long word (*width* 4) — and insert the fill value two times (*count* 2):

```
memory fill 0x003F1BC4 value 0x55AA66BB width 4 count 2
```

- To allocate a block of memory the size of two unsigned long values, and return the start address of the allocated block:

```
memory allocate 2
```

- To deallocate a block of previously allocated memory, starting at the block's start address 0x00231B16:

```
memory deallocate 0x00231B16
```

mod set debugging mode (shortcut)

mod (**tdm**|**sdm**)

Description

The **mod** command specifies the debugging mode (*task debug mode* or *system debug mode*). This is a shortcut for the **session set mode** command.

Usage

mod tdm

Sets the debug mode to task debug mode.

mod sdm

Sets the debug mode to system debug mode.

Examples

- To set the debugging mode to task debug mode:

mod tdm

See Also

[session on page D-72](#)

mutex display information about mutual-exclusion objects

```
mutex help  
mutex show [ (mutex_ID|'mutex_name') ]
```

Description

The `mutex` command displays information about the mutual exclusion objects in the application.

Usage

```
mutex help
```

Displays the syntax of the `mutex` command.

```
mutex show (mutex_ID|'mutex_name') )
```

Without arguments, `mutex show` displays a summary of all active mutual exclusion objects in the application. The display includes conditional-variable names and IDs, pSOS+m access, nest lock, type of queue used, priority inverters, ceil priority, task name, task ID, node, hold count, and task queue length.

If you specify a mutex object by ID or name, this command displays the status (name, ID, access, queue type, etc.) for that object.

Examples

- To display information about all mutex objects in the application:

```
mutex show
```

- To display detailed information about mutex 0x00130000:

```
mutex show 0x00130000
```

osbreakpoint manage all operating-system-specific breakpoints

```
osbreakpoint help
osbreakpoint show
osbreakpoint clear [ osbp_index|session|all ]
osbreakpoint set dispatch [ (task_ID|'task_name'|*) ]
osbreakpoint set time time_and_date
osbreakpoint set ticks number
osbreakpoint set syscall system_call origin parameter
```

Description

The `osbreakpoint` command manages all operating-system-specific breakpoints. Currently, this set consists of pSOS+ specific breakpoints such as service-call, dispatch, and timer breaks.

Usage

osbreakpoint help

Displays the syntax and options of the `osbreakpoint` command.

osbreakpoint show

Displays all the currently set OS breakpoints. The display is grouped by OS breakpoint type (dispatch, timer, or service-call). Each OS breakpoint is assigned a unique *OS breakpoint index* number.

osbreakpoint clear [*osbp_index*|*session*|all]

Removes an OS breakpoint from the breakpoint table. You can remove an individual breakpoint (by *osbp_index*), all the OS breakpoints for the current session, or all OS breakpoints for the target application.

The pRISM+ Shell does not display an error message if the specified OS breakpoint does not exist in the breakpoint table.

osbreakpoint set dispatch [(*task_ID*|'*task_name*'|*)]

Sets a dispatch breakpoint on the default task (if no task ID or name specified), or on the specified task.

A *dispatch breakpoint* is a breakpoint that stops execution of the target application if one of the following occurs:

- The task is pre-empted.
- The task blocks.
- The task becomes the running task.

osbreakpoint set time *time_and_date*

Sets an absolute timer breakpoint to occur at the specified time and date. Argument *time_and_date* takes this form:

hours minutes [seconds] [month date_num year]

where valid values for each *time_and_date* component are as follows:

<i>hours</i>	1 – 24	<i>date_num</i>	1 – 31
<i>minutes</i>	0 – 59	<i>month</i>	JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
<i>seconds</i>	0 – 59	<i>year</i>	0 – 9999

The *seconds* component is optional; it defaults to 0 if not given. If you do not specify the three date components, *osbreakpoint* uses the current date.

osbreakpoint set ticks *number*

Sets a relative timer breakpoint to occur *number* clock ticks after the target has started running.

osbreakpoint set syscall *system_call* *origin* *parameter*

Defines a pSOS+ service-call break, which stops execution when the application makes a qualifying system call into the pSOS+ kernel.

The origin

The *origin* specifies that the service break is to be further qualified by the entity executing code when the service break is hit. The *origin* can be one of the following: a task name, a task ID, the *** character, or the *isr* option.

origin* = '*task_name*' | *task_ID

Specifies that the service break must be hit by this specified task to stop execution. The task can be identified by either a task name or task ID.

origin = *

Specifies that the service break can be hit by any task or ISR (in system debug mode) or any debug task (in task debug mode) to stop execution.

origin = isr

Specifies that the service break must be hit by any interrupt service routine (ISR) to stop execution. This *origin* is valid only in system debug mode.

The parameter

The *parameter* specifies the function-related parameter that further qualifies a break on a specific system call. For each system call, the *parameter* value is one of the following types, or no parameter at all, as specified in the table on the next page:

- a conditional variable name or ID (*Condvar*)
- a device number (*Device*)
- a mutual-exclusion object name or ID (*Mutex*)
- a name (four letters enclosed in single quotes, such as 'SRCE' or 'IO_2')
- a partition name or ID (*Partition*)
- a queue name or ID (*Queue*)
- a region name or ID (*Region*)
- a semaphore name or ID (*Semaph*)
- a task name or ID (*Task*)

Tables D-1 and D-2 identify the pSOS+ system calls on which a service break can occur and, for each call, list the qualifying parameter type (**Parm Type**). For detailed information about these system calls, see the *pSOS+ System Calls* manual. The calls in Table D-2 are new for pSOS+ version 2.5.

TABLE D-1 pSOS+ System Calls for `osbreakpoint set syscall` Command

System Call	Parm Type	System Call	Parm Type	System Call	Parm Type
as_catch	<i>None</i>	q_aurgent	<i>Queue</i>	sm_delete	<i>Semaph</i>
as_notify	<i>None</i>	q_avsend	<i>Queue</i>	sm_ident	<i>Name</i>
as_return	<i>None</i>	q_avurgent	<i>Queue</i>	sm_p	<i>Semaph</i>
as_send	<i>Task</i>	q_broadcast	<i>Queue</i>	sm_v	<i>Semaph</i>

TABLE D-1 pSOS+ System Calls for `osbreakpoint set syscall` Command (Continued)

System Call	Parm Type	System Call	Parm Type	System Call	Parm Type
<code>de_close</code>	<i>Device</i>	<code>q_create</code>	<i>Name</i>	<code>t_create</code>	<i>Name</i>
<code>de_ctrl</code>	<i>Device</i>	<code>q_delete</code>	<i>Queue</i>	<code>t_delete</code>	<i>Task</i>
<code>de_init</code>	<i>Device</i>	<code>q_ident</code>	<i>Name</i>	<code>t_getreg</code>	<i>Task</i>
<code>de_open</code>	<i>Device</i>	<code>q_receive</code>	<i>Queue</i>	<code>t_ident</code>	<i>Name</i>
<code>de_read</code>	<i>Device</i>	<code>q_send</code>	<i>Queue</i>	<code>t_mode</code>	<i>None</i>
<code>de_write</code>	<i>Device</i>	<code>q_urgent</code>	<i>Queue</i>	<code>t_restart</code>	<i>Task</i>
<code>errno_addr</code>	<i>None</i>	<code>q_vbroadcast</code>	<i>Queue</i>	<code>t_resume</code>	<i>Task</i>
<code>ev_asend</code>	<i>Task</i>	<code>q_vcreate</code>	<i>Name</i>	<code>t_setpri</code>	<i>Task</i>
<code>ev_receive</code>	<i>None</i>	<code>q_vdelete</code>	<i>Queue</i>	<code>t_setreg</code>	<i>Task</i>
<code>ev_send</code>	<i>Task</i>	<code>q_vident</code>	<i>Name</i>	<code>t_start</code>	<i>Task</i>
<code>k_fatal</code>	<i>None</i>	<code>q_vreceive</code>	<i>Queue</i>	<code>t_suspend</code>	<i>Task</i>
<code>k_terminate</code>	<i>None</i>	<code>q_vsend</code>	<i>Queue</i>	<code>tm_cancel</code>	<i>None</i>
<code>m_ext2int</code>	<i>None</i>	<code>q_vurgent</code>	<i>Queue</i>	<code>tm_evafter</code>	<i>None</i>
<code>m_int2ext</code>	<i>None</i>	<code>rn_create</code>	<i>Name</i>	<code>tm_evevery</code>	<i>None</i>
<code>pt_create</code>	<i>Name</i>	<code>rn_delete</code>	<i>Region</i>	<code>tm_evwhen</code>	<i>None</i>
<code>pt_delete</code>	<i>Partition</i>	<code>rn_getseg</code>	<i>Region</i>	<code>tm_get</code>	<i>None</i>
<code>pt_getbuf</code>	<i>Partition</i>	<code>rn_ident</code>	<i>Name</i>	<code>tm_set</code>	<i>None</i>
<code>pt_ident</code>	<i>Name</i>	<code>rn_retseg</code>	<i>Region</i>	<code>tm_tick</code>	<i>None</i>
<code>pt_retbuf</code>	<i>Partition</i>	<code>sm_av</code>	<i>Semaph</i>	<code>tm_wkafter</code>	<i>None</i>
<code>pt_sgetbuf</code>	<i>None</i>	<code>sm_create</code>	<i>Name</i>	<code>tm_wkwhen</code>	<i>None</i>
<code>q_asend</code>	<i>Queue</i>				

TABLE D-2 pSOS+ Version 2.5 System Calls for `osbreakpoint set syscall` Command

System Call	Parm Type	System Call	Parm Type	System Call	Parm Type
<code>co_register</code>	<i>None</i>	<code>dnt_remove</code>	<i>None</i>	<code>q_notify</code>	<i>Queue</i>
<code>co_unregister</code>	<i>None</i>	<code>ioj_bind</code>	<i>None</i>	<code>q_notify</code>	<i>Queue</i>
<code>cv_abroadcast</code>	<i>Condvar</i>	<code>ioj_bindany</code>	<i>None</i>	<code>sm_notify</code>	<i>Semaph</i>
<code>cv_asignal</code>	<i>Condvar</i>	<code>ioj_getent</code>	<i>None</i>	<code>t_addvar</code>	<i>Task</i>
<code>cv_broadcast</code>	<i>Condvar</i>	<code>ioj_lock</code>	<i>None</i>	<code>t_delvar</code>	<i>Task</i>
<code>cv_create</code>	<i>Name</i>	<code>ioj_unlock</code>	<i>None</i>	<code>tm_getticks</code>	<i>None</i>
<code>cv_delete</code>	<i>Condvar</i>	<code>mu_create</code>	<i>Name</i>	<code>tsd_create</code>	<i>Name</i>
<code>cv_ident</code>	<i>Name</i>	<code>mu_delete</code>	<i>Mutex</i>	<code>tsd_delete</code>	<i>None</i>
<code>cv_signal</code>	<i>Condvar</i>	<code>mu_ident</code>	<i>Name</i>	<code>tsd_getval</code>	<i>None</i>
<code>cv_wait</code>	<i>Condvar</i>	<code>mu_lock</code>	<i>Mutex</i>	<code>tsd_ident</code>	<i>Name</i>
<code>dnt_add</code>	<i>None</i>	<code>mu_setceil</code>	<i>Mutex</i>	<code>tsd_setval</code>	<i>None</i>
<code>dnt_find</code>	<i>None</i>	<code>mu_unlock</code>	<i>Mutex</i>		

Examples

- To set a dispatch OS breakpoint:

```
osbreakpoint set dispatch 0x00010000
osbreakpoint set dispatch 'IDLE'
```

- To set an absolute timer breakpoint at 14:12:10 on July 4, 1999:

```
osbreakpoint set time 14 12 10 JUL 4 1999
```

- To set a relative timer breakpoint to break after 100 ticks have elapsed:

```
osbreakpoint set ticks 100
```

- To set a system-call breakpoint to break when the program calls the pSOS+ system call `q_send` (for any queue and from any task or ISR):

```
osbreakpoint set syscall q_send * *
```

Refer to [Table D-1 on page D-47](#) for the list of supported system calls.

- To show information about all the OS breakpoints set:

```
osbreakpoint show
```

- To clear the OS breakpoint whose index number is 3:

```
osbreakpoint clear 3
```

- To clear all OS breakpoints for the current session:

```
osbreakpoint clear session
```

- To clear all OS breakpoints:

```
osbreakpoint clear all
```

See Also

[breakpoint](#) on page D-5

[cb](#) on page D-8

[db](#) on page D-14

[lb](#) on page D-39

partition

display information about partitions

partition help

partition show [(*partition_ID*|'*partition_name*')]

Description

The `partition` command displays information about the partitions of your current application.

Usage

partition help

Displays the syntax of the `partition` command.

partition show

With no argument, `partition show` displays a summary of all active partitions in the application. The display includes partition names and IDs, buffer size, access (local or global), whether the delete override (DO) bit is set, the number of total buffers, the number of free buffers, and the starting address. The display is similar to the output of the `QP` command in `pROBE+`.

partition show [(*partition_ID*|'*partition_name*')]

Given a *partition_ID* or *partition_name*, this command displays information about the specified partition.

Examples

- To display information about partitions 0x00150000 and PTN1:

```
partition show 0x00150000
partition show 'PTN1'
```

See Also

[psos](#) on page D-55
[queue](#) on page D-63
[region](#) on page D-66
[semaphore](#) on page D-71
[task](#) on page D-82

pm patch memory (shortcut)

pm[*.width*] *address value*

Description

The **pm** command changes the value of a location in memory to a new specified value.

Usage

pm.*width* *address value*

Replaces the contents of a *width*-sized region of memory, starting at address *address*, with the specified value *value*. Valid values for *width* are *c* (for char, one byte, the default), *s* (for short, two bytes), and *l* (for long, four bytes).

Examples

- To change the value of four bytes at address 0x1020 to 0x12345678:

```
pm.l 0x1020 0x12345678
```

See Also

[fm](#) on page D-31

[memory](#) on page D-41

pr patch register (shortcut)

```
pr reg_num value ([general]|fpu) [ task_ID ]  
pr reg_num value (mmu|control)
```

Description

The `pr` command changes the value of a specified register to a new specified value. To find the *reg_num* number associated with a register name, use the `dr` command or the `register show` command.

Usage

```
pr reg_num value ([general]|fpu) [ task ]
```

Changes the contents of the general or floating-point register *reg_num* for task *task* to the new value *value*. If no *task_ID* is specified, the specified register of the default task is changed.

```
pr reg_num value [ (mmu|control) ]
```

Changes the contents of the MMU or control register *reg_num* (shared by all tasks) to the new value *value*.

Examples

- To change the contents of general register 6 of the default task to the value 0x10:

```
pr 6 0x10
```

See Also

[register](#) on page D-68

probe

display and set pROBE+ flags and interrupt level

```
probe help
probe show
probe set flag flag_type (on|off)
probe set ilevel i_level
```

Description

The `probe` command sets pROBE+ flags and interrupt level.

Usage

probe help

Displays the `probe` command syntax and available options.

probe show

Displays the current settings of all pROBE+ flags and the interrupt level.

probe set flag *flag_type* (on|off)

Sets a pROBE+ flag to ON or OFF. These are valid values for *flag_type*:

- `nodots` — prevents output of the periods normally sent to the console by the `dl` command
- `nopage` — disables paging
- `rbug` — directs the pROBE+ target agent to operate in distributed debug mode
- `nomanb` — specifies whether you can halt a running application (`off` means you can halt it, `on` means you cannot)
- `smode` — directs the pROBE+ target agent to run in silent debug mode

probe set ilevel *i_level*

Sets the pROBE+ interrupt level to level *i_level*.

On the MIPS processor, the interrupt level can be 0 or 1.

For more information about the available interrupt levels, see the *Programmer's Reference* manual.

Examples

- To show pROBE+ flag settings and interrupt levels:

```
probe show
```

- To set pROBE+ flags to ON or OFF as needed:

```
probe set flag nomanb on
probe set flag nodots off
probe set flag nopage off
probe set flag smode on
probe set flag rbug off
```

- To set the pROBE+ interrupt level to 1:

```
probe set ilevel 1
```

See Also

[*fl* on page D-30](#)

[*il* on page D-36](#)

[*psos* on page D-55](#)

psos make a pSOS+ system call

```
psos help
psos show (date|version|multiprocessor)
psos show (devicenametable|iojumptable)
psos show table table_type
psos show object [ obj_type ]
psos call system_call
```

Description

The `psos` command displays information about the date, version number, target multiprocessors, tables, and pSOS+ objects; you can also make pSOS+ system calls with this command.

Usage

psos help

Displays the `psos` command syntax and available options.

psos show (date|version|multiprocessor)

Displays the date, the version numbers of installed pSOS+ components (such as pROBE+, pMONT, and so on), and the node number and sequence number of each target multiprocessor.

psos show (devicenametable|iojumptable)

Displays information about the pSOS+ device name table or I/O jump table, respectively.

psos show object [*obj_type*]

Displays information about all the pSOS+ objects currently active in the target application (with no *obj_type* specified) or about the specified *obj_type*. Valid values for *obj_type* are task, partition, region, queue, semaphore, mutex, and condvar.

psos show table *table_type*

Displays information about the specified pSOSSystem configuration table *table_type*. Valid values for *table_type* are node, psos, probe, prepc, phile, pna, pse, pmont, or multiprocessor.

psos call system_call parameter

Makes a pSOS+ call directly from the pRISM+ Shell to manually stimulate or simulate portions of an application. Due to potential race conditions, only a subset of pSOS+ calls are supported.

Tables D-3 and D-4 identify the supported pSOS+ system calls and, for each call, list the requisite parameters (by type). The calls in Table D-4 are new for pSOS+ version 2.5. For detailed information about these system calls, see the *pSOSystem System Calls* manual.

TABLE D-3 pSOS+ System Calls for `psos call` Command

System Call	Parameter 1	Parameter 2	Parameter 3	Parameter 4	Parameter 5
as_send	tID	signal			
ev_send	tID	evnts			
pt_getbuf	ptID				
pt_retbuf	ptID	bufaddr			
q_broadcast	qID	msg_buf[0]	msg_buf[1]	msg_buf[2]	msg_buf[3]
q_receive	qID				
q_send	qID	msg_buf[0]	msg_buf[1]	msg_buf[2]	msg_buf[3]
q_urgent	qID	msg_buf[0]	msg_buf[1]	msg_buf[2]	msg_buf[3]
q_vbroadcast	qID	msg_buf	msg_len		
q_vreceive	qID				
q_vsend	qID	msg_buf	msg_len		
q_vurgent	qID	msg_buf	msg_len		
sm_p	smID				
sm_v	smID				
t_resume	tID				
t_setpri	tID	newprio			
t_setreg	tID	regnum	reg_value		
t_suspend	tID				
tm_set	date	time	ticks		
tm_tick					

TABLE D-4 pSOS+ Version 2.5 System Calls for `psos call` Command

System Call	Parameter 1	Parameter 2	Parameter 3
<code>as_notify</code>	<code>evnts</code>		
<code>cv_signal</code>	<code>cvID</code>		
<code>cv_broadcast</code>	<code>cvID</code>		
<code>q_notify</code>	<code>qID</code>	<code>tID</code>	<code>evnts</code>
<code>q_vnotify</code>	<code>qID</code>	<code>tid</code>	<code>evnts</code>
<code>sm_notify</code>	<code>smID</code>	<code>tID</code>	<code>evnts</code>

In tables D-3 and D-4 , the specified parameters have the following meanings:

<code>cvID</code>	Conditional-variable ID	<code>regnum</code>	Register number
<code>date</code>	Date (dd-mm-yyy)	<code>reg_value</code>	Register value
<code>evnts</code>	Bit-encoded events	<code>signal</code>	Bit-encoded signal list
<code>msg_buf</code>	Message buffer	<code>smID</code>	Semaphore ID
<code>msg_len</code>	Length of message	<code>ticks</code>	Number of elapsed clock ticks
<code>newprio</code>	New priority	<code>tID</code>	Task ID
<code>ptID</code>	Partition ID	<code>time</code>	Time (hh:mm[ss])
<code>qID</code>	Queue ID		

Examples

- To display multiprocessor information:

```
psos show multiprocessor
```

- To display all the pSOS+ objects:

```
psos show object
```

- To display all the pSOS+ task objects:

```
psos show object task
```

- To display the pSOS+ node configuration table:

```
psos show table node
```

- To display the pSOS+ pREPC+ configuration table:

```
psos show table prepc
```

- To send an asynchronous signal of bit-encoded value 0x12 to task 0x000B0000:

```
psos call as_send 0x000B0000 0x12
```

- To send an event signal of bit-encoded value 0x13 to task 0x000B0000:

```
psos call ev_send 0x000B0000 0x13
```

- To send an urgent message to a variable-length queue (whose ID is 0x00250000) where 0x803ea864 is the message buffer start address and 2 is the message length:

```
psos call q_vurgent 0x00250000 0x803ea864 2
```

Refer to [Table D-3 on page D-56](#) and [Table D-4 on page D-57](#) for a complete list of the supported pSOS+ system calls.

See Also

[partition](#) on page D-50

[probe](#) on page D-53

[queue](#) on page D-63

[region](#) on page D-66

[semaphore](#) on page D-71

[task](#) on page D-82

q*

query-related commands (shortcuts)

```

qc [ table_type ]
qcs
qcv [ (condvar_ID|'condvar_name') ]
qd
qdnt
qioj
qmu [ (mutex_ID|'mutex_name') ]
qo [ obj_type ]
qp [ (partition_ID|'partition_name') ]
qq [ (queue_ID|'queue_name') [ nodata] ]
qr [ (region_ID|'region_name') ]
qs [ (semaphore_ID|'semaphore_name') ]
qsv
qt [ (task_ID|'task_name') ]
qtsd
qtv [ (task_ID|'task_name') ]
qv

```

Description

The **q*** commands are shortcuts for lengthier pRISM+ Shell commands. See the [Usage](#) section for details about which **q*** shortcut command matches which pRISM+ Shell command.

Usage

```
qc [ table_type ]
```

Queries a configuration table. With no argument, **qc** queries the node configuration table. Given a *table_type* argument, **qc** displays information about configuration table *table_type*, which can be any of the following: *psos*, *probe*, *prepc*, *phile*, *pna*, *pse*, *pmont*, or *mpc* (multiprocessor).

This command is a shortcut for the *psos show table table_type* command.

```
qcs
```

Queries the CommSrv version and license numbers; shortcut for the *csabout version* command.

qcv [(*condvar_ID*|'*condvar_name*')]

Queries conditional variable(s). With no argument, **qcv** displays a summary of all active conditional variables in the application. Given a *condvar_ID* or *condvar_name* argument, **qcv** displays information about the specified conditional variable. This command is a shortcut for the **condvar show** command.

qd

Queries the date; shortcut for the **psos show date** command.

qdnt

Queries the device name table; shortcut for the **psos show devicenametable** command.

qioj

Queries the I/O jump table; shortcut for the **psos show iojumptable** command.

qmu [(*mutex_ID*|'*mutex_name*')]

Queries mutual exclusion objects. With no argument, **qmu** displays a summary of all active mutual exclusion (mutex) objects in the application. Given a *mutex_ID* or *mutex_name* argument, **qmu** displays information about the specified mutex object. This command is a shortcut for the **mutex show** command.

qo [*obj_type*]

Queries pSOS+ objects. With no argument, **qo** displays a summary of all objects in the application. Given an *obj_type* argument, **qo** displays information about specified object *obj_type*, which can be any of the following: task, queue, semaphore, region, partition, mutex, or condvar.

This command is a shortcut for the **psos show object** command.

qp [(*partition_ID*|'*partition_name*')]

Queries partition(s). With no argument, **qp** displays a summary of all partitions in the application. Given a *partition_ID* or *partition_name* argument, **qp** displays information about the specified partition.

The **qp** command, without arguments, is a shortcut for the **psos show object partition** command or the **partition show** command.

qq [(*queue_ID*|'*queue_name*') [*nodata*]]

Queries queue(s) in the application. With no argument, **qq** displays a summary of all active queues in the application. Given a *queue_ID* or *queue_name* argument, **qq** displays detailed information about the queue.

Given the *nodata* option, **qq** displays only the specified queue's ID number, size, and address; without the *nodata* option, **qq** also displays the first 16 bytes of the queue's message contents. This command is a shortcut for the `queue show` command.

qr [(*region_ID*|'*region_name*')]

Queries region(s) in the application. With no argument, **qr** displays a summary of all active regions in the application. Given a *region_ID* or *region_name* argument, **qr** displays detailed information about the region. This command is a shortcut for the `region show` command.

qs [(*semaphore_ID*|'*semaphore_name*')]

Queries semaphore(s) in the application. With no argument, **qs** displays a summary of all active semaphores in the application. Given a *semaphore_ID* or *semaphore_name* argument, **qs** displays detailed information about the semaphore. This command is a shortcut for the `semaphore show` command.

qsv

Queries the pRISM+ Shell version number; shortcut for the `version` command.

qt [(*task_ID*|'*task_name*')]

Queries task(s) in the application. With no argument, **qt** displays a summary of all active tasks in the application. Given a *task_ID* or *task_name* argument, **qt** displays detailed information about the task. This command is a shortcut for the `task show` command.

qtsd

Queries the task-specific data (TSD), such as task name, task ID, size, Nindex, allocation, and so on; shortcut for the `tsd show` command.

qtv [(*task_ID*|'*task_name*')]

Queries task variables for the specified task *task_ID* or *task_name*. This command is a shortcut for the `task variable` command.

qv

Queries the version numbers of all the pSOSystem components available on the target. This command is a shortcut for the `psos show version` command.

Examples

- To query the node configuration table:

```
qc
```

- To query the pROBE+ configuration table:

```
qc probe
```

- To query all objects:

```
qo
```

- To query the task object:

```
qo task
```

- To query partition PTN1:

```
qp 'PTN1'
```

- To query the queue whose ID number is 0x1E0000 and display only its ID number, size, and address (no message contents):

```
qq 0x1E0000 noata
```

- To query the task variables for the task whose ID number is 0x160000:

```
qtv 0x160000
```

See Also

[condvar](#) on page D-11

[mutex](#) on page D-44

[partition](#) on page D-50

[psos](#) on page D-55

[region](#) on page D-66

[task](#) on page D-82

[tsd](#) on page D-84

[version](#) on page D-85

queue display information about queues

queue help

queue show [(*queue_ID* | '*queue_name*') [*data|nodata*]]

Description

The `queue` command displays information about the queues in the application.

Usage

queue help

Displays the `queue` command syntax and available options.

queue show

Without arguments, `queue show` displays the current settings of all the active queues in the application. The display includes the queue names and IDs, length of the task queue, length of the message queue, maximum message queue length, status of the buffer pool, type of queue, and whether a queue is a variable-length message queue.

queue show [(*queue_ID* | '*queue_name*') [*data|nodata*]]

If you specify a pSOS+ queue object (by ID number or name), the settings of that particular object are displayed. You can also specify `data` or `nodata` for a given queue object. The `nodata` option additionally displays the queue's ID number, size, and address. The `data` option additionally displays the queue's ID number, size, and address, and also the first 16 bytes of the message.

Examples

- To show all queue information:

queue show

- To show detailed information for queue 0x000E0000:

queue show 0x000E0000

- To show detailed information for queue 0x000E0000 with no message contents displayed:

```
queue show 0x000E0000 nodata
```

See Also

region on page D-66

task on page D-82

semaphore on page D-71

partition on page D-50

psos on page D-55

quit close the session and exit the pRISM+ Shell window

quit

The `quit` command closes the session and exits the pRISM+ Shell window.

region display information about regions

```
region help
region show [ (region_ID|'region_name') ]
```

Description

The `region` command displays information about the active regions in the application.

Usage

```
region help
```

Displays the `region` command syntax and available options

```
region show
```

Without arguments, `region show` displays a summary of all active regions in the application. The display includes the region names and IDs, starting address of each region, length of each region, unit size in each region, number of free bytes, the largest contiguous size, the length of the task wait queue, whether the delete override (DO) bit is set, and the type of queue used.

```
region show [ (region_ID|'region_name') ]
```

If you specify a valid region object (by ID number or name), the pRISM+ Shell displays detailed information about that region. In addition to all the information displayed for `region show`, this command displays the contents of the specified region's task wait queue and a detailed breakdown of memory usage with the region. The display is similar to the output of the `QR <region>` command in pROBE+.

Examples

- To show a summary of all active regions:

```
region show
```

- To show detailed information for region 0x00220000:

```
region show 0x00220000
```


See Also

partition on page D-50

psos on page D-55

queue on page D-63

semaphore on page D-71

task on page D-82

register manage task-specific and shared registers

```
register help
register show [number reg] [category ([general]|fpu)] [task task_ID ]
register show [number reg] category (mmu|control)
register set reg value [category ([general]|fpu)] [task task_ID ]
register set reg value category (mmu|control)
```

Description

The `register` command displays and modifies the value of a specified register in a task.

Usage

register help

Displays the syntax of the `register` command.

```
register show [number reg] [category general] [task task_ID]
register show [number reg] category fpu [task task_ID]
```

The `register show` command, without arguments, displays the contents of all the general registers of the default task. Registers are separated into categories so a more manageable group can be displayed at one time.

- To display the contents of register *reg*, specify the number *reg* argument. To find the *reg_num* number associated with a register name, use the `dr` command or the `register show` command.
- To display the floating-point registers, give the `category fpu` option.
- To display general or floating-point registers for a specific task, also specify a `task task_ID` argument.
- To display memory-management unit (MMU) or control registers, which are shared by all tasks, give the `category mmu` or `category control` option. When the MMU and control registers are displayed, any task specifier given is ignored.

```
register set reg value [category ([general]|fpu)] [task task_ID]
register set reg value category (mmu|control)
```

The `register set reg value` command, without additional arguments, sets the contents of the default task's general register `reg` to the value `value`.

- To set the `value` of floating-point register `reg`, give the `category fpu` option.
- To set general or floating-point register `reg` for a specific task, also specify a `task task_ID` argument.
- To set memory-management unit or control register `reg`, give the `category mmu` or `category control` option. When the MMU and control registers are set, any task specifier given is ignored.

Examples

- To show the contents of general registers related to task 0x00160000:

```
register show task 0x00160000
```

This command is equivalent to the following set of commands:

```
task set 0x00160000
register show
```

NOTE: The `task set` command sets the default task, and the `register show` command displays the registers for that default task.

- To show the contents of floating-point registers related to the default task:

```
register show category fpu
```

- To show the contents of floating-point register 2 related to task 0x00160000:

```
register show number 2 category fpu task 0x00160000
```

- To set the contents of task 0x00150000's floating-point register 3 to the value 0x12345678:

```
register set 3 0x12345678 category fpu task 0x00150000
```

See Also

[dr](#) on page D-24

[pr](#) on page D-52

SC make a pSOS+ system call (shortcut)

sc system_call

Description

The `sc` command executes a pSOS+ system call. This is a shortcut for the `psos call` command.

Usage

sc system_call

Executes a pSOS+ system call. Depending on the system call, you might also need to specify one or more system-call parameters. Refer to [Table D-3 on page D-56](#) and [Table D-4 on page D-57](#) for a complete list of the supported pSOS+ system calls and their requisite parameters.

Examples

- To suspend the task whose task ID is 0x160000:

`sc t_suspend 0x160000`
- To send an asynchronous signal of bit-encoded value 0x12 to task 0x000B0000:

`sc as_send 0x000B0000 0x12`
- To send an urgent message to a variable-length queue (with ID 0x00250000) where 0x803ea864 is the message buffer start address and 2 is the message length:

`sc q_vurgent 0x00250000 0x803ea864 2`
- To acquire a semaphore token where 0x000A0000 is the semaphore ID:

`sc sm_p 0x000A0000`

See Also

[psos on page D-55](#)

semaphore

display information about semaphores

```
semaphore help
semaphore show [ (sem_ID | 'sem_name') ]
```

Description

The `semaphore` command displays information about the semaphores in the application.

Usage

```
semaphore help
```

Displays the `semaphore` command syntax and available options.

```
semaphore show (sem_ID | 'sem_name') 
```

Without arguments, `semaphore show` displays a summary of all active semaphores in the application. The display includes the semaphore names and IDs, the current count number, the task queue length, and the type of queue used. The display is similar to the output of the `QS` command in pROBE+.

If you specify a valid semaphore object (by ID number or name), the pRISM+ Shell displays detailed information about that semaphore. In addition to all the information described for `semaphore show`, this command displays the contents of the semaphore's task wait queue. The display is similar to the output of the `QS <semaphore>` command in pROBE+.

Examples

- To show detailed information for semaphore 0x000B0000:

```
semaphore show 0x000B0000
```

See Also

[partition](#) on page D-50
[psos](#) on page D-55
[queue](#) on page D-63
[region](#) on page D-66
[task](#) on page D-82

session manipulate, open, and load the target through the pRISM+ Shell

```
session help
session open [hot] [ target_name ]
session close
session reopen
session show [statistics]
session set [mode] (tdm|sdm)
session (add|delete) (task_ID|'task_name')
session log (log_file|end)
```

Description

The `session` command manipulates, opens, and loads the target through the pRISM+ Shell. This command also initiates a target debug session, which begins when you open a connection to a specified target.

Usage

session help

Displays the `session` command syntax and available options.

session open [target_name]

Establishes contact to an existing halted predefined target through the pRISM+ Shell. Use this command first, before you enter any other CommSrv commands. If the connection is through an Ethernet, *target_name* is the target's network name or its IP address. If the connection is serial, *target_name* is the target's serial port number (or name) and baud rate, separated by a comma.

If you do not specify a *target_name*, the default target you selected in the pRISM+ Manager is used.

session open hot [target_name]

Establishes contact to the specified running target.

session reopen

Reestablishes contact to an existing predefined target through the pRISM+ Shell. Use this command during the same debug session when the communication link to the pROBE+ target agent breaks. For example, if a cable came loose

and then was reconnected, you would use this command to re-establish communication.

session close

Disconnects contact from the target. Given the `session close` command, the pRISM+ Shell terminates the communication channel to the target.

session show [statistics]

Displays information about the target debug session, including the processor type on the target, the name of any log file, the names of executable files downloaded, and the current debug mode.

If you specify the `statistics` option, this command also displays communication statistics, such as the number of packets sent.

session set [mode] (tdm|sdm)

Sets the debug mode to TDM (*task debug mode*) or SDM (*system debug mode*).

session add (task_ID|'task_name')

Specifies a task to add to the set of tasks being debugged (that is, the foreground tasks). This command can be used only when the debug mode is TDM.

session delete (task_ID|'task_name')

Specifies a task to remove from the set of tasks being debugged (that is, the foreground tasks). This command can be used only when the debug mode is TDM.

session log (log_file|end)

Manages the packet log file (a log of packets exchanged between CommSrv and the target). Command `session log log_file` starts logging to file `log_file`, while `session log end` stops the logging.

Examples

- To connect through an Ethernet to a halted target and to a running target:

```
session open seant3
session open hot 152.216.226.158
```


- To connect through a serial port to a halted target and to a running target:

```
session open COM1,9600  
session open hot /dev/ttya,19200
```

- To reopen the current debug session (re-establish the connection after the communication channel has been lost):

```
session reopen
```

- To close the current debug session:

```
session close
```

- To display information about the current debug session:

```
session show
```

- To set the debug mode to TDM (Task Debug Mode):

```
session set tdm
```

- To set the debug mode to SDM (System Debug Mode):

```
session set mode sdm
```

- To add a task (ID = 0x00010000 or name = CHAR) to the debug list:

```
session add 0x00010000  
session add 'CHAR'
```

- To delete a task from the debug list:

```
session delete 0x00010000  
session delete 'NUMS'
```

- To open a log file to record packet exchange information between CommSrv and the target:

```
session log log_1020.txt
```

- To end the current logging session:

```
session log end
```

- The commands `session log filename` and `session log end` should always be used in pairs. For example, enter this series of commands:

```
session log datapkt.txt
region show
session log end
```

to generate this log file, `datapkt.txt`:

```
Send: QUERY_RQT Type: AllRegions, Request: NEW, Node: -1
```

```
Packet Dump:
```

```
0: 0C 84 00 00 00 00 00 00 FF FF FF FF .....
```

```
Recv: QUERY_RPY status: pROBE_OK, More?: FALSE, Partial?: FALSE, NItems: 2
```

```
Packet Dump:
```

```
0: 8C 00 01 00 00 00 00 02 52 4E 23 30 00 00 00 00 .....RN#0.
16: 00 30 00 80 00 7F FF 00 00 00 01 00 00 76 D4 00 .0.....
32: 00 76 AA 00 00 00 00 00 00 00 00 00 00 52 4D 4D .v.....R
48: 00 22 00 00 00 00 AE A5 00 00 00 08 00 00 00 80 .".....
64: 00 00 07 00 00 00 07 00 00 00 00 00 00 00 00 00 .....
```

See Also

[log](#) on page D-40

sf display stack frame information (shortcut)

```
sf [ task_ID ]
```

Description

The `sf` command displays the stack frame for the default task or for a specified task. This is a shortcut for the `stackfrm show` command.

Usage

```
sf [ task_ID ]
```

The `sf` command displays a summary of all stack frames in the application. If you specify a *task_ID*, the command displays a summary of information about the stack frame for that task.

Examples

- To display the stack frame of the task whose ID is 0x1b0000:

```
sf 0x1b0000
```

stackfrm display stack frame information

```
stackfrm help
stackfrm show [ task (task_ID) ]
```

Description

The `stackfrm` command displays information about the stack frames.

Usage

```
stackfrm help
```

Displays the syntax of the `stackfrm` command.

```
stackfrm show [ task (task_ID) ]
```

The `stackfrm show` command displays a summary of all stack frames in the application. If you specify a *task_ID*, the command displays a summary of information about the stack frames for that task.

Examples

- To show stack frame information for the default task and task 0x00160000, respectively:

```
stackfrm show
stackfrm show task 0x00160000
```

t* task-related commands (shortcuts)

```
tadd (task_ID|'task_name')
tdef [ (task_ID|'task_name') ]
tdel (task_ID|'task_name')
tin string [ task_ID ]
```

Description

The **t*** commands are shortcuts for lengthier pRISM+ Shell commands.

See the [Usage](#) section for details about which **t*** shortcut command matches which pRISM+ Shell command.

Usage

```
tadd (task_ID|'task_name')
```

Adds the specified task to the debugger list when in task-debug mode; shortcut for the `session add` command.

```
tdef [ (task_ID|'task_name') ]
```

Displays or sets the default task. Without arguments, `tdef` displays the default task. This is a shortcut for the `task default` command.

Given a task ID or name, `tdef` sets the default task to the specified task. This is a shortcut for the `task set` command.

```
tdel (task_ID|'task_name')
```

Deletes the specified task from the debugger list when in task-debug mode; shortcut for the `session delete` command.

```
tin string task_ID
```

Inputs a string from the pRISM+ Shell to a task. The input is given to the default task unless you specify a `task_ID`. Use this command when a task calls `db_input` to get input. This is a shortcut for the `task input` command.

Examples

- Add to the debug list the task whose ID is 0x160000 and the task whose name is IO2_, respectively:

```
tadd 0x160000
tadd 'IO2_'
```

- Show the default task

```
tdef
```

- Set the default task to the task whose ID is 0x160000:

```
tdef 0x160000
```

- Delete from the debug list the task whose ID is 0x160000 and the task whose name is IO2_, respectively:

```
tdel 0x160000
tdel 'IO2_'
```

- Input string "ABcd" to the target to task 0x1b0000.

```
tin ABcd 0x1b0000
```

See Also

[session on page D-72](#)

[task on page D-82](#)

target manage target definitions

```
target help
target set event [ event_code_mask ]
```

Description

You can receive information from CommSrv about certain events occurring in the target, such as an instruction break, creation or deletion of an object, an output request, and so on.

Each event is represented by an event code, which can be ORed together to generate an event mask. These are the events defined for each event code:

0x00000000	No event		
0x00000001	Instruction Break	0x00001000	Output Request
0x00000002	Memory Access Break	0x00002000	Experiment Data
0x00000004	Service Call Break	0x00004000	End Experiment
0x00000008	Kernel Break	0x00008000	Perfmer Data
0x00000010	Exception (Exc) Break	0x00010000	Stack Problem
0x00000020	Background Exc Break	0x00020000	Object Create
0x00000040	TDM-off Break	0x00040000	Object Delete
0x00000080	Manual Break	0x00080000	Application Restart
0x00000100	Fatal Error Break	0x00100000	Invocation Complete
0x00000200	Unexpected Break	0x00200000	Target Load Module
0x00000400	ASM Step Break	0x00400000	Target Unload Module
0x00000800	Input Request	0xFFFFFFFF	All events

To receive information about an event, you must first register with CommSrv for that event (that is, “set” the event) with the `target set event` command.

When you have registered for an event, CommSrv reports occurrences of that event in the pRISM+ Shell window.

Usage

target help

Displays the `target` command syntax and available options.

target set event [*event_code_mask*]

Registers target events for which you want to receive notification. The *event_code_mask* is one event code or a combination of event codes ORed together.

If you have previously registered for events and are no longer interested in some of them, changing the *event_code_mask* causes CommSrv to immediately search through all saved events and remove those that are no longer of interest.

Examples

- To register target Input Request (0x00000800) events and Output Request (0x00001000) events:

```
target set event 0x00001800
```

task manage task operations

```
task help
task default
task (show|set|variable) [ (task_ID|'task_name') ]
task input string [ (task_ID|'task_name') ]
```

Description

The `task` command manages task operations, including setting the default task.

Usage

task help

Displays the `task` command syntax and available options.

task default

Displays the task ID of the default task. The default task is set when you invoke the `session open` command.

task show [(task_ID|'task_name')]

With no argument, `task show` displays a summary of all tasks in the application. This summary display includes the task names and task IDs, priorities, mode, status, suspension state, and, if the task is blocked, the reason for the blockage. The display is similar to the output of the `QT` command in pROBE+.

Given a specific task ID or name, the command displays detailed information about that task, if it exists. This task-specific display includes the values of all software registers; initial pc, sp, priority, and mode; ASR address and mode; pending events and ASR; and outstanding timers. The display is similar to the output of the `QT <task>` command in pROBE+.

task set [(task_ID|'task_name')]

Sets the default task, overriding the previous setting of the default task.

task input *string* [(task_ID|'task_name')]

Inputs a string from the pRISM+ Shell to a task.

The input is given to the default task unless you specify another task by ID or name. Use this command when a task calls `db_input` to get input.

```
task variable [ (task_ID|'task_name') ]
```

Displays task variables for the specified task or, if no task ID or name is specified, for the default task.

Examples

- To show information about all tasks:

```
task show
```

- To show detailed information about task 0x000A0000

```
task show 0x000A0000
```

- To set task 0x00010000 as the default task for debugging:

```
task set 0x00010000
```

- To show the default task ID:

```
task default
```

- To input the string "ABCD" from the pRISM+ Shell to the default target:

```
task input ABCD
```

- To show variables of task 0x00020000:

```
task variable 0x00020000
```

See Also

[*partition* on page D-50](#)

[*psos* on page D-55](#)

[*queue* on page D-63](#)

[*region* on page D-66](#)

[*semaphore* on page D-71](#)

tsd display task-specific data

```
tsd help
tsd show
```

Description

The `tsd` command displays task-specific data for the default task, such as task name, task ID, size, Nindex, allocation, and so on.

Examples

- To display all task-specific data for the default task of the target application:

```
tsd show
```

version display pRISM+ Shell version

version

Description

The `version` command displays the version number of the pRISM+ Shell.

Example

- To display version information about the pRISM+ Shell:

```
version
```

See Also

[q*](#) on page D-59

D.3 Comparison of pROBE+ and pRISM+ Shell Commands

What the Command Does	pROBE+ Command	pRISM+ Shell Command(s)	pRISM+ Shell Shortcut
Memory Commands			
Display memory	DM	<i>memory</i> read	dm
Patch memory	PM	<i>memory</i> write	pm
Fill memory	FM	<i>memory</i> fill	fm
Search memory	SM	N/A	--
Move memory	MM	N/A	--
Compare memory	CM	N/A	--
Disassemble memory	DI	<i>disassemble</i>	di
Assemble into memory	AS	N/A	--
Download S-record file from host	DL	<i>dl</i>	dl
Verify download from host	VL	N/A	--
Upload to host	UL	N/A	--
Register Commands			
Display registers	DR	<i>register</i> show	dr
Patch register	PR	<i>register</i> set	pr
Display offset registers	DO	N/A	--
Display floating-point registers	DF	<i>register</i> show fpu	dr
Breakpoint Commands			
Set breakpoints	DB	<i>breakpoint</i> set <i>osbreakpoint</i> set	db
Show breakpoints	LB	<i>breakpoint</i> show <i>osbreakpoint</i> show	lb
Remove breakpoints	CB	<i>breakpoint</i> clear <i>osbreakpoint</i> clear	cb
Execution Commands			
Start the execution	GO	<i>go</i>	<i>go</i>
Initialize pSOS+ kernel	GS	<i>initialize</i>	--
Go until pSOS+ exit	GX	N/A	--
Stepping	ST	N/A	--

What the Command Does	pROBE+ Command	pRISM+ Shell Command(s)	pRISM+ Shell Shortcut
Query Commands			
Query component version	QV	<i>psos</i> show version	qv
Query configuration tables	QC	<i>psos</i> show table	qc
Query date and time	QD	<i>psos</i> show date	qd
Query object tables	QO	<i>psos</i> show object	qo
Query partitions	QP	<i>partition</i> show	qp
Query queues	QQ	<i>queue</i> show	qq
Query regions	QR	<i>region</i> show	qr
Query semaphores	QS	<i>semaphore</i> show	qs
Query tasks	QT	<i>task</i> show	qt
Profiling Commands			
Clear profile data	CP	N/A	--
List profile data	LP	N/A	--
Miscellaneous Commands			
Help; display list of commands	HE	<i>help</i>	he
Enter host communication mode	HO	N/A	--
Make pSOS+ system call	SC	<i>psos</i> call	sc
Evaluate constant	EC	N/A	--
Set pROBE+ flags	FL FL "f" ON FL "f" OFF	<i>probe</i> show <i>probe</i> set flag <i>f</i> on <i>probe</i> set flag <i>f</i> off	fl fl <i>f</i> on fl <i>f</i> off
Set pROBE+ interrupt level	IL IL "val"	<i>probe</i> show <i>probe</i> set ilevel <i>val</i>	il il <i>val</i>

D.4 TCL Commands

This section describes the built-in TCL commands used to construct pRISM+ Shell commands. These commands need only be used if you want to extend the provided set of commands. These commands are not needed for typical interactive use of the shell.

The pRISM+ Shell allows you to access and use specialized TCL commands. These new commands have been developed to allow you to incorporate TCL scripts to interact with the CommSrv and DbgSrv via CORBA.

The built-in extensions to the TCL shell that allow access to the CommSrv and DbgSrv are known collectively as *TCLCorba*.

TCLCorba binds the CORBA interfaces defined by pRISM+ to the TCL language. These interfaces are only used if you want to extend the command set.

NOTE: These commands are for advanced users who understand TCL and can develop their own TCL scripts.

These are the pRISM+ TCL commands:

- The *type* command opens a connection to an Interface Repository (IR).
- The *vinfo* command allows a TCLCorba script to construct an object reference to a CORBA service.
- The *bind* command allows a TCLCorba script to bind an object reference to a CORBA service.
- The *set* command is an enhanced version of the TCL set command.
- The *new* command creates a new instance of a CORBA object.
- The *delete* command removes a CORBA object.
- The *toString* command converts an instance of a basic type CORBA.
- The *invoke* command sends a request to CORBA service.
- The *slength* command provides the length of the sequence.

type opens a connection to the IR

```
type open (ir hostname) | (TypeStore filename)
type save TypeStore filename
```

Description

The `type` command opens a connection to the Interface Repository. Type provides the connection of the database you require to use the remaining pRISM+ Shell TCL commands to run a TCL script with pRISM+. TCLCorba extends the `type` environment of TCL by extracting this information from the Interface Repository. The syntax is:

To invoke one or more CORBA services, TCLCorba needs to know about the interfaces provided by the services and about the IDL types defined by these interfaces. It obtains this information from the Interface Repository (IR) as follow:

```
type open ir hostname
```

Where *hostname* is the name of the system where the IR services is located. The `open` option retrieves all the type information contained in the specified IR.

This process may be very time consuming. TCLCorba provides a command to save the information from the IR in a local storage and retrieve it later:

```
type save TypeStore filename
```

Where *filename* is a name of a local file. All type store file names have the `.ts` extension.

vinfo allows a TCLCorba script to construct an object reference to a CORBA service

```
vinfo type [typeName]|interface [interfaceName]
```

Description

Use vinfo command to check which database types and interface types known to TCLCorba are available. The syntax is:

The following command displays all the types known to TCLCorba:

```
vinfo typename
```

To specify a particular type known to TCLCorba, use the following command:

```
vinfo type typeName
```

If *typeName* is specified as an argument to vinfo type, only information about that type is shown.

The following command displays all the interfaces known to TCLCorba:

```
vinfo interface
```

If *interfaceName* is specified as an argument to vinfo interface, only information about that interface is shown.

bind allows a script to bind an object reference to CORBA services

bind **ObjectReference**

Description

ObjectReference is one of the following:

- *hostname* is the name of the system where the CORBA services is located.
- *serverName* is the name of the CORBA services. In IDL, server name is the name of the IDL interface.
- *marker* is the marker for that particular server.
- *interfaceMarker* is the marker for that particular interface.

More information about CORBA object references can be obtained from Iona or other CORBA vendors.

set extended form of the built-in TCL set command

set (tcl_var|tcl_corba_object) value

Description

This `set` command is an extended form of the built-in TCL `set` command. TCLCorba script can invoke `set` to assign values to TCL variables or to TCLCorba basic-type objects.

NOTE: The ability to assign values to TCLCorba complex-type objects is not yet implemented.

new creates a TCLCorba object

new *typename*

Description

typename is the type name of the object. *typename* must be known to the instance of TCLCorba running. To verify if this type is available use `vinfo type typename`.

delete removes a previously created TCLCorba object

delete obj_ref

toString converts a basic-type TCLCorba object into a printable form

toString *obj_ref*

You can also use “*” as a short-cut of toString.

invoke sends a request to CORBA services

invoke *obj_ref* *operation* [*args*]*

obj_ref is obtained from a previous use of *bind*.

operation is the name of an operation defined by the interface *bind* to by *obj_ref*.

args is one or more argument to the request.

slength

returns the current and maximum length of a TCLCorba sequence or array object

slength *array|sequence obj_ref*

E

pSOSystem Source Projects

E.1 Generic pSOSystem Projects

<code>include.shared</code>	Project for pSOSystem include files: <code>\$PSS_ROOT/include</code> and subdirectories.
<code>sys_os.shared</code>	Project for system library: <code>sys/os</code> directory.
<code>configs_std.shared</code>	Project for pSOSystem configuration files: <code>configs/std</code> directory. Projects for <code>sys/os/src/dir_name</code> :
<code>sysclass.shared</code>	Project for C++ library: <code>sys/libc/src/sysclass</code> directory.
<code>profiler.shared</code>	Project for profiler library: <code>sys/libc/src/profiler</code> directory.

E.2 Drivers Project

<code>enetdmpi_drv.shared</code>	Project for drivers/enetdmpi.
<code>lap_drv.shared</code>	Project for drivers/lap.
<code>modem_drv.shared</code>	Project for drivers/modem.
<code>otcp_drv.shared</code>	Project for drivers/otcp.
<code>ppp_drv.shared</code>	Project for drivers/ppp
<code>slip_drv.shared</code>	Project for drivers/slip.
<code>x25_drv.shared</code>	Project for drivers/x25.

E.3 Bsp Projects

<code>bsp_src.shared</code>	Project for BSP sources: <code>bsps/bsp_name/src</code> directory.
<code>bsp.shared</code>	Project for each <code>bsps/bsp_name</code> .

E.4 Sample Application Projects

<code><app_name>.shared</code>	Project for each sample <code>apps/app_name</code>
--------------------------------------	--

E.5 Sample Application Projects

Following projects are added as subprojects to the application projects:

- `include.shared`
- `sys_os.shared`
- `configs_std.shared`
- `bsp_src.shared`
- `bsp.shared`
- Projects from the drivers and `sys/libc/src` if referred by the application.

E.6 VPATH

Makefiles implement workspaces overriding using the make VPATH facility. VPATH is a way to specify list of directories to the make that it should search for dependency files.

E.6.1 gnu gmake and VPATH

For gmake, VPATH define specifies the directory list. Directory names are separated by colons or blanks. The search can be qualified using 'vpath'. For example:

```
VPATH = $(PSS_ROOT)/apps/hello:/tmp/apps/hello
```

Specifies to gmake to look for any dependency in `$(PSS_ROOT)/apps/hello` and `/tmp/apps/hello` when not found in the current directory.

```
vpath %.h $(PSS_ROOT)/apps/hello:/tmp/apps/hello
```

Specifies to make to look for any .h dependency in \$(PSS_ROOT)/apps/hello and /tmp/apps/hello when not find in the current directory.

E.6.2 \$< Macro

For VPATH to work correctly \$< should be use in all the compilation rules. \$< expands to prerequisite file with the directory name, wherever the file is found. For example, say

```
VPATH=$(PSS_ROOT)/apps/hello
```

and you are building under \$(HOME)/psosppc_pwe/apps/hello. The rule for making root.o should be written as follows:

```
root.o: root.c
```

```
$(CC) $(COPTS) -o root.o $<
```

\$< is expanded to \$(PSS_ROOT)/apps/hello/root.c by make when root.c is not found in the local directory.

E.6.3 Compiler Option -o:

Some compilers, in absence of -o option, generate .o file in the source file directory instead of the current directory. DIAB does that when compiling .s files.

```
Init.o: init.s
$(AS) $(AOPTS) -o init.o $<
```

In the above rule \$< may expand to file coming from some distant directory. Without -o compiler would generate .o file in that directory.

E.6.4 Compiler Option -I@:

This option specify compiler to strictly follow include directory order given with -I directives. In absence of this option, compiler treats source file directory as the current directory for **include filename** directives. This is required for the case when a included file is overridden but not the file including it. For example, assume root.c file includes sys_conf.h using **include sys_conf.h** directive. The SWS contains both these files. The developer has a modified copy of sys_conf.h in his PWS, which should hide the sys_conf.h in SWS. Since the source files is coming from the SWS, if -I@ is not used, sys_conf.h from the SWS will be used.

E.6.5 Use of Relative Path for Overriding

Relative paths should be used in the makefiles for overriding to work. For example, assume SWS points to the PSS_ROOT and user is building hello in his PWS, if rule is written as follows

```
sysinit.o: $(PSS_ROOT)/configs/std/sysinit.c
$(CC) $(COPTS) -o sysinit.o $<
```

sysinit.c file will always come from SWS \$(PSS_ROOT)/configs/std directory even if a copy exist in the PWS. The rule should be modified to have relative path when using SNIFF+, i.e.

```
sysinit.o: ../../configs/std/sysinit.c
$(CC) $(COPTS) -o sysinit.o $<
```

make would look for ../../configs/std/sysinit.c first. If not found it will get the file from \$PSS_ROOT/configs/std.

With SNIFF+, PSS_ROOT (and PSS_BSP, wherever applicable) is redefined to a relative path **inside the makefiles**.

E.6.6 Generating Include and Link Paths

Since VPATH is a make feature and is not supported by the compiler/linker/archiver, complete include and link path have to be generated using the VPATH to pass it to the compiler/linker. For example, when building in PWS using include path -I. -I../../include and VPATH set to \$PSS_ROOT/apps/hello, the following include path should be generated and passed to the compiler for overriding to work:

```
-I. -I$(PSS_ROOT)/apps/hello/. -I../../include -I$(PSS_ROOT)/apps/hello/../../include
```

E.6.7 Object and .opt files Overriding

Since Compiler/linker/archiver do not understand VPATH, object and .opt files are to be generated in the local PWS and cannot be overridden.

E.6.8 With or Without SNIFF+

The makefiles are written to work with or without SNIFF+. In absence of SNIFF+, makefiles work as tradition makefiles, i.e., any pSOSystem application or BSP can be build under PSS_ROOT or outside PSS_ROOT by just defining PSS_ROOT and PSS_BSP environment variables.

With SNIFF+ these makefiles support overriding of workspaces.

E.6.9 macros.incl File

SNIFF+ generates macros.incl file that has definition for SNIFF_ShSW macro. This macro contains list of workspace directories. This file is included by \$(SNIFF_MAKE_CMD).mk file. \$(SNIFF_MAKE_CMD).mk uses SNIFF_ShSW to generate VPATH.

E.6.10 Problems Using Recursive Make

Because PSS_ROOT (and PSS_BSP) are defined to a relative path in the snf_gnu.mk file, recursive make rules can cause problems. For example, the relative PSS_ROOT value for apps/loader is ../../ and for apps/loader/loadable is ../../... Now if make for loadable is called from the loader makefile, it would import the PSS_ROOT definition ../../ from the loader makefile which would be incorrect. The correct value of PSS_ROOT and PSS_BSP is passed to sub-make using the RECURSIVE_MAKE_SETUP define statement.

E.6.11 Check_vpath Target

Before building any targets, check_vpath target is made. This is invoked with the -e option to make. The check_vpath target generates the definition of PSS_SNIFF_ShSWS using the absolute value of PSS_ROOT. This macro is used to specify the VPATH value to make. This is required because PSS_ROOT is redefined to a relative path in the makefile and VPATH needs the absolute PSS_ROOT value. This target also adds EXP_PSS_ROOT, EXP_PSS_BSP, and BSP_BASE define statements. EXP_PSS_ROOT and EXP_PSS_BSP contain the absolute value of PSS_ROOT and PSS_BSP, respectively, which can be used during normal make when PSS_ROOT and PSS_BSP are changed to have a relative path. BSP_BASE is defined to the base of PSS_BSP directory.

E.6.12 Gnu Make

The standard gnu make command is invoked by psosmake on Unix. On PC platforms gnu make sources are modified to convert '\ ' to '/' for pSOSystem related environment variables.

E

E.7 pLUGINS+ Scripts

There are two types of plugins scripts: scripts which are used for creating SNIFF+ projects for pSOSystem+ and scripts which integrate pmanager with SNIFF+/pSOSystem+.

E.7.1 Scripts to Create SNIFF+ Projects for pSOSystem+

These scripts exist in the `$PSS_ROOT/bin/source/plugins/scripts` directory. They run on Unix under the Bourne Shell and on Windows under the mks shell. These scripts are shipped with pSOSystem+ and can be used to facilitate the creation of SNIFF+ projects that have a complex code directory structure. These scripts (with some changes, if needed) automate the creation of a SNIFF+ project for the inclusion of the project's code. A project created using these scripts works on Unix as well as Windows platforms.

`plugins_create_proj`

Given a file list, relative to `PSS_ROOT`, this script creates a SNIFF+ project. It creates the main SNIFF+ project in the directory where the first file of the list exists. Since SNIFF+ does not support files from different directories in a single project, this script creates projects for every directory in the list. These projects are added to the main project as subproject.

By default, the PDFS file of the main project has the name 'basename `MAIN_PROJ_DIR`'.shared. The remainder of the PDFS files have the name 'base-name `project_dir`'_<TAG>.shared, where *TAG* is the basename of the main project directory. An alternative name for the main project can be specified using `-n <main_proj_name>` option, where *main_proj_name* is the name of the main project without the extension. An alternative *TAG* name can also be specified using `-t <TAG>` option.

By default, the lists of files are read from the `.sniffpl.lst` file from the current directory. Alternatively, `-f <filename>` option can be used to specify a filename.

Option `-m` can be used to generate make support files (`*.incl` files) for the main project.

Usage: `plugin_create_proj [-f <filename>] [-n <main_proj_name>] [-t <TAG>] [-m]`

The script assumes that SNIFF+ is running (session0).

When creating SNIFF+ projects for the pSOSystem, the directory attribute of the 'Project Description' FileType in the preferences file is set to `sniffprj`. As a result, project PDFS files are created under the `sniffprj` subdirectory of the main project directory.

plugins_create_app

This script uses the `plugins_create_proj` and `plugins_add_target` scripts and creates a SNIFF+ project for a pSOSystem+ application. It first creates the main project using the `plugins_create_prj` script. It then adds generic pSOSystem+ projects and bsp projects to the main project and if referred to by the application, it also adds drivers projects.

This script modifies the application project to use the `PSS_BSP` environment variable to refer to bsp projects `bsp.shared` and `bsp_src.shared`. The application project is also modified to add the build targets (`ram.hex`, for example).

The `plugins_create_all` script is used to create pSOSystem application projects. To use `plugins_create_app` in stand-alone mode to create an application project the following must apply:

- SNIFF+ should already be running (session0)
- The script should be invoked from the application dir
- The application directory should contain the `.sniffpl.lst` file.
- The `plugins_create_app` script and scripts used by it should be in the path

Usage: `plugins_create_app`

plugins_create_all

This is the master script which creates SNIFF+ projects for the entire pSOSystem+. The following must be setup before you start this script:

- The environment variables `PSS_ROOT`, `PSS_BSP` and `SNIFF_DIR` are set. `PSS_BSP` can be set to any valid bsp. This is used when creating application projects. The references to the BSP is changed to use `$PSS_BSP`.
- The `$SNIFF_DIR/bin` directory is in the path.
- `PSS_ROOT` contain the `workingenvs` directory.
- `PSS_ROOT` contains `.sniffpl.lst` files.

This script creates the following SNIFF+ projects:

TABLE E-1 SNIFF+ Projects

Project	Description
Generic pSOSystem Projects	
include.shared	Project for \$PSS_ROOT/include and its subdirectories. No file list needed. Should have automatic add remove property for .h files.
sys_os.shared	Project for sys/os directory. Should have automatic add remove for .s and .o files. No file list needed.
configs_std.shared	Project for configs/std. No file list needed.
sysclass.shared	Project for sys/libc/src/sysclass. No file list needed.
profiler.shared	Project for sys/libc/src/profiler. No file list needed.
NOTE: The list of directories under sys/os/src is not hard coded. This script looks for subdirectories under \$PSS_ROOT/sys/libc/src and creates projects for every one of them.	
Drivers Projects	
enetdmpi.shared	Project for drivers/enetdmpi. No file list needed.
lap.shared	Project for drivers/lap. No file list needed.
modem.shared	Project for drivers/modem. No file list needed.
otcp.shared	Project for drivers/otcp. No file list needed.
ppp.shared	Project for drivers/ppp. No file list needed.
slip.shared	Project for drivers/slip. No file list needed.
x25.shared	Project for drivers/x25. No file list needed.
NOTE: The list of drivers is not hard coded. This scripts looks for subdirectories under the \$PSS_ROOT/drivers directory and creates projects for every one of them.	

TABLE E-1 SNiFF+ Projects (Continued)

Project	Description
Bsp Projects	
<code>bsp_src.shared</code>	For each <code>bsps/<bsp_name>/src</code> . Created using <code>plugins_create_prj</code> .
<code>bsp.shared</code>	Project for each <code>bsps/<bsp_name></code> . No file list needed.
App Projects	
<code><app_name>.shared</code>	Project for each <code>apps/<app_name></code> . Created using <code>plugins_create_app</code> script. Application projects should be created at the end of the script because they refer to the projects created above.

When a project contains files only from a single directory and all the files from that directory go into the project, no file list is needed to create the project. In this case a SNiFF+ project created by default, will be accurate.

The `include.shared` and `sys_os.shared` files need to have the automatic add and remove property for files because files under these directories depend upon what components of pSOSystem are installed. There are two ways to achieve this:

- Create these projects like any other project and change them using `sed/awk` (`sniffaccess` does not provide a way to change these attributes) or,
- Use the specialized preferences file. This method has a simpler approach and is the method used by the scripts.

The preferences files used for creating projects for pSOSystem+ differs from what is shipped with pSOSystem+ for the customer.

These difference are:

- `MakeFileSupport` in these files is set to `FALSE`.
- Default working environment is set to `SSWE:pSOSystem-ppc`, the directory where the projects are created.
- For `include` and `sys/os` directories a different preferences file is used.
- Directory attribute of Project Description FileType is set to `sniffprj`, which is where all SNiFF+ projects are generated.

This script copies the appropriate `umenupref` to `$HOME/.sniffdir` on the Unix platform or `$SNIFF_DIR/Preferences/$LOGNAME` on the PC platform before creating any projects.

To implement an override of the working environment, `main`, (makefile as oppose to `.mk` files) makefiles includes a `SNiFF+` generated `macros.incl` file. `SNiFF+` generates `*.incl` files only if the project has the `MakeFileSupport` attribute set. Option `-m` is used by `plugins_create_proj` to set this attribute of the main project. For the projects not created using `plugins_create_proj`, this option is set by `plugins_create_all`.

The following scripts are used by `plugins_create_all`:

- `plugins_add_target`
- `plugins_create_proj`
- `plugins_create_app`

Usage: `plugins_create_all`

`plugins_clean_all`

This script deletes all the `SNiFF+` projects and `.sniffdir` from `$PSS_ROOT`. It can be used to restore the pSOSystem to its previous state in the event that `plugins_create_all` script fails to complete properly.

Usage: `plugins_clean_all`

E.7.2 Integration scripts:

These scripts integrate `pmanager` with `SNiFF+`/pSOSystem+.

`plugins_open_proj`

This script is invoked by the `pmanager` when user selects **File→New→SNiFF+→‘Start from the pSOSystem sample application’**. This script is also invoked when the `SNiFF+` button is pressed for an existing `pRISMspace`. It opens an existing `SNiFF+` project in the user's `PWE`. It takes the project directory name as the input parameter and looks for the `SNiFF+` project file under this directory using the following order:

- `pss_main/sniffprj/pss_main.shared`
- `sniffprj/`basename dirname`.shared`

- sniffprj/bsp_src.shared

Usage: `plugins_open_proj <dir_name>`

plugins_create_uproj

This script is invoked by the pmanager when user selects **File→New→SNIFF+→‘Start from an existing code base’**. It creates a SNIFF+ project (recursively) for the given directory. The directory should be a subdirectory of \$PSS_USER_SSWE. This script then opens the project in user's PWE. This script also adds `relinkable_obj_name` as a relinkable target for the SNIFF+ project.

Usage: `plugins_create_uproj <prj_dir> [relinkable_obj_name]`

plugins_convert_proj

This script is invoked when the user selects "plug-ins->"convert to pSOSystem sample application" from the SNIFF+ PE window. It converts a SNIFF+ project to the pSOSystem application project. It performs the following:

- Creates the `pss_main` directory under the project directory and copies the template `sys_conf.h`, `drv_conf.c` and `makefile` under `pss_main` from the `$PSS_ROOT/apps/snf_tmpl` directory.
- Extracts the relinkable object name from the project description file.
- Modifies the `pss_main` makefile to define `PSS_APOBJS` to a relinkable object and adds a rule for it.
- Creates a SNIFF+ project for `pss_main` and adds pSOSystem generic projects to it as subprojects. It also adds the project being converted as the subproject of this project.

Usage: `plugins_convert_proj <proj_name>`

plugins_add_target

This script modifies a SNIFF+ project to add pSOSystem target names. These targets are then displayed by SNIFF+ under **PE→Target→Make** menu. The list of targets are hard coded in this script and should be changed whenever there is a change in targets defined in `config.mk`.

Usage: `plugins_add_target <project_name>`

plugins_create_bsp

This script creates SNIFF+ projects for a custom BSP. These projects are required to integrate custom BSPs into pRISM+. The BSP should be located under `$PSS_ROOT/bmps` and should follow the standard pSOSystem BSP format, (`<custom_bsp>/src` source format). Before you invoke this command you would need to create a file containing a list of files which make this BSP. The default name of this file is `.sniffpl.lst` and should exist under `<custom_bsp>/src`.

First file name in this file should be from `<custom_bsp>/src`. An example can be found in one of the standard BSPs. This script creates `bsp_src.shared` under `$PSS_ROOT/bmps/<custom_bsp>/src/sniffprj` and `bsp.shared` under `$PSS_ROOT/bmps/<custom_bsp>/sniffprj` and assumes that the pRISM+ environment setup the makefile derived from pSOSystem BSP makefile. It uses `plugins_create_proj` from `$PSS_ROOT/bin/source/plugins/scripts` and `plugins_add_target` from `$PSS_ROOT/bin`

Usage: `plugins_create_bsp <bsp_dir> [-f <file_list_file>]`

Where `bsp_dir` is `$PSS_ROOT/bmps/<custom_bsp>`

plugins_make_copy

For every file type in SNIFF+ a custom menu command can be added, which can be invoked by right clicking on the file in PE. This script is invoked to make a local copy of a file from SSWE to PWE when the user selects the "Make a local copy" command from this custom menu.

Usage: `plugins_make_copy <src> <dst>`

plugins_edit_file

This script opens a file in the SNIFF+ source editor. It assumes that SNIFF+ is running with session0 and one of the open projects contains this file. This script is typically used with SDS to replace its editor with SNIFF+ SE. User should alias `_edit` to

On Unix: `alias _edit 'plugins_edit_file \!:2 \!:1'`

On PC: `alias _edit 'sh -f plugins_edit_file \!:2 \!:1'`

This can be done in SDS command window or in the `sstep.ini` file. Output of this script (such as errors) is displayed in the SDS `cmd` window.

Usage: `plugins_edit_file <filename> [linenumber]`

plugins_pwizard

solaris/plugins_pwizard, hpux/plugins_pwizard, or win32/plugins_pwizard.bat are invoked by SNIFF+ as a result of a double click on the sys_conf.h file. This script invokes pRISM+ wizard on the sys_conf.h file.

Usage: plugins_pwizard *<full path to sys_conf.h>*

psosmake

This is a wrapper to the actual make command and is host specific. There are three different scenarios in which make is called:

- make using pSOSystem makefile in a Non-SNIFF+ environment
- make using pSOSystem makefile with SNIFF+
- make using SNIFF+ generated makefile.

This wrapper, depending upon the make situation, calls the appropriate make commands.

Usage:


In SNIFF+ environment: psosmake SNIFF_MAKE *[target_name]*

In non SNIFF+ environment: psosmake *[target_name]*



Glossary

apps	A pSOSystem directory that contains a number of subdirectories with sample pSOSystem applications.
browser	A tool that is used to view data, but not make changes to it. A complete pRISM+ package has four SNiFF+ browsers. Each browser is designed for a specific task.
BSP (Board-Support Package)	The hardware-specific code in the pSOSystem software; contained in the <code>\$PSS_ROOT/bsps</code> directory.
build	The process of creating an executable program and installing it on a target system. The build steps are usually described in make files that are executed by programs like make. A build involves translations of source files and the construction of binary files by compilers, linkers, and other tools.
communication server	An application that runs on the host machine and is responsible for interaction with the target agent.
CORBA (Common Object Request Broker Architecture)	CORBA is a middle-ware specification created by the Object Management Group (OMG), a group of 500 leading companies in the software industry. CORBA allows applications to communicate with one another no matter where they are located or who has designed them



CORBA bus (ORB)	The middle-ware that establishes the client/server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its methods, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments, and seamlessly interconnects multiple object systems.
editor	A tool that is used to both view and change data. pRISM+ contains the SNiFF+ Project Editor, and two source code editors, the SNiFF+ Source Code Editor and emacs.
host system	The system on which the source application program resides.
JTAG (Joint Test Action Group)	A mechanism for controlling a target processor. JTAG is based on the IEEE 1149.1 standard and was originally designed as a way to perform in-circuit testing during manufacturing. The current version is used for debugging embedded applications.
make	The program that reads make files and drives the build process.
MLIB	A communication protocol used by the pRISM+ Communication Server and the pMONT target agent.
multitasking	The breaking up of a program into several tasks. Each task has its own system resources and competes with the other tasks in the system for CPU time.
multi threading	The situation wherein a single process has several threads of execution. Each thread inherits its environment from its parent and shares the resources of the parent process. A process can have several threads, whereas a task does not have any threads.
object file	A derived file that is generated from source code using the build process.

Object Query Language (OQL)	The Object Database Management Group's (ODMG) object language specification. OQL provides full object query capabilities and contains almost all the SQL-92 query language as a subset. The Object Management Group (OMG) is working with the ODMG to create a single query language for objects.
Object Request Broker	A CORBA-based service provided by an object bus that lets clients invoke methods on remote objects either statically or dynamically. The ORB included in pRISM+ is Orbix from IONA Technologies.
OQL	See Object Query Language.
ORB	See Object Request Broker and CORBA bus (ORB).
pRISMSpace	The directory in which pRISM+ stores all the files for your application project. You must create a pRISMSpace when you start a pRISM+ project.
project	The main organizational element in SNIFF+. A project consists of files, attributes, and subprojects, and is described by a project description file (PDF). Project hierarchies can be built around projects and subprojects, which are also projects on their own.
project description file (PDF)	A file that describes a project's attributes, structure, and contents. A PDF is a structured ASCII file that is created, saved, and opened by SNIFF+.
pSOSystem	An operating system used on embedded controllers. Its code consists of read-only object libraries, include files and source files.
pSOSystem directory tree	The central location of pSOSystem on the host system. It contains the shared pSOSystem code so that multiple users can have access to it.
pSOSystem environment	A standard set of services for the application code. It usually contains the pSOS+ kernel and the following companion software elements: pROBE+, pNA+, pHILE+, device drivers, interrupt handlers, and configuration tables to customize the pSOSystem environment for a particular target system.
RBUG	The communication protocol used by the pRISM+ Communication Server and the pROBE+ target agent.

repository	Storage for persistent data.
services	Services store and provide information that can be used by any tool or service. Components that use the services are known as the clients of the service. Services are primarily implemented as servers and by background processes without a user interface.
symbol	A named language construct in source code.
symbol repository	The information base for a development project. The symbol repository contains information about the declaration, definition, and use of named program elements such as classes, methods, variables, and functions. Each project has its own symbol table that is filled in by the parser. Symbol repositories are kept in memory and persistently stored to disk. pRISM+ has two symbol repositories: The SNIFF+ Symbol Table is for static data and the pRISM+ Repository is for runtime data.
sys_conf.h	The pSOSystem configuration file. It is an include file that must reside in the working directory.
target agent	The software and/or hardware that is responsible for controlling the state of the target being debugged.
target system	The system on which the embedded operating system and the compiled embedded application reside.
tool	Tools visualize and manipulate information provided by the system services. All tools have a user interface. Tools can also provide some services that can be used by other tools.
version	A particular revision and an element of the version tree of a file. A version is created by checking in a working file. Versions are checked out as working files.
version control	The process of managing and administering versions of files. The SNIFF+ Project Editor is the main tool for version control in pRISM+.
version tree	The hierarchical structure in which all versions of a file are organized. A version tree has one main branch and can have several sub-branches. The version tree is typically stored in a repository file.

working directory	A directory in which you build a pSOSystem executable image. You can locate your working directory under <code>\$PSS_ROOT</code> .
workspace	A directory tree that contains projects and working files. There are two kinds of workspaces: private and shared. A shared workspace is accessed among several developers in a team and is overridden by their private workspaces. Shared workspaces can be split into shared source and object workspaces in order to separate platform-independent from platform-dependent files. Shared workspaces can override other shared workspaces, resulting in multiple levels of overriding workspaces. The common part of overridden workspaces must have the same directory structure. The workspace variables indicate the locations of workspaces.

Index

A

analysis
 post-mortem, ES_p 10-5
analysis tool
 ES_p 1-7
 Object Browser 1-7
application
 downloading
 SearchLight Debugger 8-2
application code 2-5
application stack size 10-5
applications
 sample 2-11
apps directory 2-11
ARM debugger environment variables B-2
ASEV task 14-5
autoinit 14-4

B

begin trace events, ES_p 10-4
bind command, pRISM+ shell D-91
boards
 IBM 403GA/GC A-3
 IDT79S440 A-10
 IDT79S500 A-14
 LSI4101 A-16
board-support package 2-5, 2-7

boot command, pRISM+ shell D-4
break
 on high-level language statements 9-1
breakpoint
 setting
 SearchLight Debugger 8-17
breakpoint command, pRISM+ shell D-5
BSP 2-7
 adding custom 15-3
bsps directory 2-7
buffer list, using
 pRISM+ Editor 5-11
buffer management 10-3
 ES_p 10-3
 Halt on Buffer Full 10-3
 Transmit 10-3
 warnings 14-2
 Wraparound 10-3
build command 4-6

C

CAD-UL
 environment variables B-4
call stack
 examining
 SearchLight Debugger 8-20
cb (clear breakpoints) command D-8

cmode parameter	14-2	delete	D-94
cn (connect to target) command	D-9	db (define breakpoint)	D-14
code		dcn (disconnect)	D-16
application	2-5	di (disassemble)	D-19
environment, hardware-specific	2-5	disassemble	D-20
system configuration	2-5	disconnect	D-21
code parameter	14-2	dl (load or download)	D-22
Code window	9-9	dm (display memory)	D-23
color settings		dr (display registers)	D-24
changing	15-13	dssession	D-25
comm (communication parameters)		ev (evaluate variable)	D-27
command, pRISM+ shell	D-10	evaluate	D-28
commands		evt (set events)	D-29
breakpoint		fl (flags)	D-30
SearchLight Debugger	8-17	fm (fill memory)	D-31
build	4-6	go (run)	D-32
go, pROBE+	14-5, 14-6	halt	D-33
pRISM+ shell		he (help)	D-34
pRISM+ Shell	13-2	help	D-35
pSOS-aware		il (interrupt level)	D-36
pRISM+ Shell	13-1	init	D-37
SearchLight Debugger single step	8-4	initialize	D-38
commands, pRISM+ shell		invoke	D-96
bind	D-91	lb (list breakpoints)	D-39
boot	D-4	log	D-40
breakpoint	D-5	memory	D-41
comm (communication parameters)	D-10	mod (debugging mode)	D-43
condvar	D-11	mutex	D-44
connect to target	D-12	new	D-93
cb (clear breakpoints)	D-8	osbreakpoint	D-45
cn (connect to target)	D-9	partition	D-50
csabout	D-13	pm (patch memory)	D-51
debugger	D-17	pr (patch register)	D-52
		probe	D-53

psos	D-55	pMONT+	14-2
q* (query shortcut)	D-59	system	2-12
queue	D-63	configuration table	
quit	D-65	ESp	10-5
region	D-66	configuration tables	
register	D-68	Node Configuration Table	14-4
sc (system call)	D-70	configuration table, query	D-59
semaphore	D-71	configuring pMONT	14-1
session	D-72	connect to target command	D-12
set	D-92	corrupted stacks	
sf (stack frame)	D-76	boundaries	10-5
slength	D-97	CPU trap entry points	14-6
stackfrm	D-77	Cross-Compiler Suite	
t* (task-related)	D-78	Diab Data	1-8
target	D-80	csabout command, pRISM+ shell	D-13
task	D-82	CSV files	
toString	D-95	Object Browser	11-7
tsd (task-specific data)	D-84	custom BSP	
type	D-89	adding	15-3
version	D-85	customize	
vinfo	D-90	pRISM+ Environment	15-5
commands, pROBE+		toolbar	4-7
pRISM+ shell	D-86	customizing	
Communication		pRISM+ Shell	13-14
pRISM+ Shell	13-3	C++ language support	8-1, 9-2
Communication Server		D	
definitions	1-10	data collection	
pRISM+ shell	D-13, D-17	refining	
compilers		ESp	10-3
Diab Data	1-8	data parameter	14-2
conditional variable, pRISM+ shell	D-11	dataSize parameter	14-2
condvar command, pRISM+ shell	D-11	db (define breakpoint) command	D-14
configuration		dcn (disconnect) command	D-16
multiple users	15-7		

deadlocks		directory, working	2-6
checking		disassemble command	D-20
Object Browser	11-5	disconnect command	D-21
Debug Server		dl (load or download) command	D-22
definitions	1-10	dm (display memory) command	D-23
pRISM+ shell	D-25, D-28	download	
debugger command, pRISM+ shell	D-17	pRISM+ Shell	13-5
debuggers		dr (display registers) command	D-24
SearchLight	1-8	drv_conf.c	14-4
SingleStep	1-9	dssession command, pRISM+ shell	D-25
debugging modes			
high level, assembly language	9-9	E	
definitions	1-8, 1-9	Embedded System Profiler	
Communication Server	1-10	See Also ESp	1-7
Debug Server	1-10	Enable Checking	10-5
Diab Data Compiler	1-8	end trace events	
ESp	1-7	delay	10-5
Object Browser	1-7	environment variables	2-13
pRISM+	1-1	changing	15-12
pRISM+ Configuration Wizard	1-5	UNIX	B-5
pRISM+ Editor	1-6	Windows	B-1
pRISM+ Shell	1-8	error checking	
pSOSystem	1-3, 2-1	pRISM+ Wizard	7-6
RTA Suite	1-9	ESp	
SNiFF+	1-6	begin trace events	10-4
delete command, pRISM+ shell	D-94	buffer management	10-3
dev parameter	14-2	configuration table	10-5
di (disassemble) command	D-19	definitions	1-7
Diab Data Compiler		end trace events	10-4
definitions	1-8	event specification	10-4
environment variables	B-2, B-5	events	
dialog.c file	2-6	ignore	10-5
directories		log	10-5
host	2-14	log_event	14-6

memory usage	14-7
placing user-defined events	10-2
post-mortem	10-5
prerequisites	10-2
refining data collection	10-3
transmit	10-3
wraparound option	10-4
ESp communication with pMONT	14-5
Ethernet connection	
booting pSOS+	C-4
host tools configuration	C-4
how to	C-4
pRISM+ tools	C-4
sys_conf.h settings	C-4
ev (evaluate variable) command	D-27
evaluate command, pRISM+ shell	D-28
event logging	14-6
events	
begin trace, ESp	10-4
end trace, ESp	10-4
ESp	10-5
precedence to ignore	10-5
specification	10-3
ESp	10-4
specification overhead	10-4
user-defined	
placing, ESp	10-2
event_data parameter	14-7
evt (set events) command	D-29
executable image	2-5, 2-6

F

features	
pRISM+	1-1
pRISM+ Editor	3-6, 5-1
pRISM+ Manager	4-1
pRISM+ Wizard	7-1
SearchLight Debugger	8-1
file	
dialog.c	2-6
include	2-5
source	2-5
sys.lib	2-7
sys_conf.h	2-6, 2-7
.map	2-11
files	
adding to project, pRISM+ Editor	5-7
configuration	2-6
copying	
pRISM+ Editor	5-6
creating, pRISM+ Editor	5-5
driver configuration	2-11
drv_conf.c	14-4
error checking, pRISM+ Editor	5-8
makefile	2-8
saving, pRISM+ Editor	5-6
sys_conf.h	14-2
Find dialog box	
using, SearchLight Debugger	8-17
fl (flags) command	D-30
fm (fill memory) command	D-31
format	
Intel Extended Hexadecimal	2-10
Motorola S-record	2-10

G

- go command [14-6](#)
- go (run) command, pRISM+ shell [D-32](#)
- gs command [14-5](#)

H

- halt command, pRISM+ shell [D-33](#)
- Halt on Buffer Full overhead [10-4](#)
- hardware-specific environment code [2-5](#)
- he (help) command [D-34](#)
- help command, pRISM+ shell [D-35](#)
- host
 - directories [2-14](#)

I

- IBM 403GA/GC boards [A-3](#)
- IDT79S440 boards [A-10](#)
- IDT79S500 boards [A-14](#)
- il (interrupt level) command [D-36](#)
- include files [2-5](#)
- init command [D-37](#)
- initialize command, pRISM+ shell [D-38](#)
- installation
 - memory considerations [15-8](#)
- installations
 - multiple [15-5](#)
- InstallDriver [14-4](#)
- installing a driver [14-4](#)
- Intel
 - Extended Hexadecimal format [2-10](#)
- interface
 - pRISM+ Wizard [7-2](#)
- invoke command, pRISM+ shell [D-96](#)
- IP address [2-6](#)

K

- kernels
 - pSOS+ [2-2](#)
 - pSOS+m [2-2](#)

L

- launch
 - pRISM+ [3-3](#)
- lb (list breakpoints) command [D-39](#)
- libraries
 - including custom, pRISM+ Editor [5-10](#)
- library
 - pSOSystem [2-14](#)
 - system [2-7](#)
- Link Map Analyzer
 - description, RTA [12-1](#)
- log command, pRISM+ shell [D-40](#)
- log_event() [10-2, 14-6](#)
 - return value [14-6](#)
- LSI4101 boards [A-16](#)

M

- makefile [2-8](#)
- makefile browser, pRISM+ Editor [5-1](#)
- makefile view, pRISM+ Editor [5-3](#)
- makefiles
 - adding, pRISM+ Editor [5-3, 5-10](#)
 - removing, pRISM+ Editor [5-11](#)
- memory
 - usage
 - ESp [14-7](#)
 - Object Browser [14-7](#)
 - target agent [14-7](#)
 - viewing, SearchLight Debugger [8-12](#)

memory command, pRISM+ shell [D-41](#)
 memory considerations
 installation [15-8](#)
 memory leaks
 finding, Object Browser [11-4](#)
 memory requirements, pMONT [14-2](#)
 MKS Toolkit
 environment variables [B-3](#)
 mod (debugging mode) command [D-43](#)
 modes
 pRISM+ Wizard [7-4](#)
 monitoring
 target requirements
 pMONT+ [14-2](#)
 Motorola
 S-record format [2-10](#)
 mutex command, pRISM+ shell [D-44](#)

N

navigating Files window
 SearchLight Debugger [8-15](#)
 new command, pRISM+ shell [D-93](#)
 Next command
 using, SearchLight Debugger [8-8](#)
 Node Configuration Table [14-4](#)

O

Object Browser
 checking for deadlocks [11-5](#)
 checking for priority inversion [11-5](#)
 CSV files [11-7](#)
 definitions [1-7](#)
 examining messages in queue [11-5](#)

examining tasks waiting for messages [11-6](#)
 finding memory leaks [11-4](#)
 log_event [14-6](#)
 memory usage [14-7](#)
 monitoring stack problems [11-4](#)
 Prerequisites [11-3](#)
 object libraries [2-5](#)
 optimized code [9-2](#)
 OS Breakpoint command
 removing, SearchLight Debugger [8-12](#)
 setting, SearchLight Debugger [8-9, 8-28](#)
 osbreakpoint command, pRISM+ shell [D-45](#)
 output parameters [2-10](#)

P

partition command, pRISM+ shell [D-50](#)
 peak stack usage [10-5](#)
 pHILE+
 file system manager [2-2](#)
 pm (patch memory) command [D-51](#)
 PMCM task [14-5](#)
 PMON task [14-5](#)
 pMONT
 autoinit [14-4](#)
 cmode parameter [14-2](#)
 code parameter [14-2](#)
 configuration [14-1](#)
 data parameter [14-2](#)
 dataSize parameter [14-2](#)
 dev parameter [14-2](#)
 list of related topics [14-1](#)
 memory requirements [14-2](#)

system call	14-2	SNiFF+	3-12
target behavior	14-1	using	3-4
tasks	14-5	pRISM+	
tmFreq	14-2	architecture	1-2
tmRead	14-2	definitions	1-1
tmReset	14-2	documentation	1-10
traceBuff parameter	14-2	environment variables	B-4, B-5
traceBuffSize parameter	14-2	features	1-1
pMONT Configuration Table	14-2	launching	3-3
pMONT+		pRISM+ Configuration Wizard	
behavior on target	14-5	definitions	1-5
configuring	14-2	pRISM+ Editor	
monitoring target requirements	14-2	adding files to projects	5-7
pNA+		adding makefiles	5-10
network manager	2-2	copying files	5-6
pr (patch register) command	D-52	creating files	5-5
pREPC+		definitions	1-6
ANSI C standard library	2-2	error checking files	5-8
Prerequisites		features	3-6, 5-1
Object Browser	11-3	including custom libraries	5-10
prerequisites		makefile browser	5-1
ESp	10-2	makefile view	5-3
using pRISM+	3-2	makefiles	5-3
printing		program editor	5-4
issues, UNIX	15-13	removing makefiles	5-11
priority inversion		saving files	5-6
checking for, Object Browser	11-5	source view	5-3
pRISMspace		using	3-5
creating	4-3	using buffer list	5-11
pRISM+ Manager	4-3	pRISM+ Manager	
pRISMspace Settings	4-5	definitions	1-4
pRISMspace Wizard		features	4-1
pRISM+ Editor	3-5	pRISMspace	4-3

pRISM+ Shell			
communication timeouts	13-3		
customizing	13-14		
definitions	1-8		
downloading application	13-5		
features	13-1		
levels of service	13-1		
pSOS objects	13-3		
pSOS-aware commands	13-1		
queue command	13-7		
SearchLight Debugger	13-5		
Tcl script examples	13-8		
TCL scripts	13-2		
timeouts	13-3		
pRISM+ shell			
Communication Server	D-13, D-17		
conditional variable	D-11		
Debug Server	D-25, D-28		
pROBE+ commands	D-86		
pRISM+ shell commands			
bind	D-91		
boot	D-4		
breakpoint	D-5		
cb (clear breakpoints)	D-8		
cn (connect to target)	D-9		
comm (communication parameters)	D-10		
condvar	D-11		
connect to target	D-12		
csabout	D-13		
db (define breakpoint)	D-14		
dcn (disconnect)	D-16		
debugger	D-17		
delete	D-94		
di (disassemble)	D-19		
disassemble	D-20		
disconnect	D-21		
dl (load or download)	D-22		
dm (display memory)	D-23		
dr (display registers)	D-24		
dssession	D-25		
ev (evaluate variable)	D-27		
evaluate	D-28		
evt (set events)	D-29		
fl (flags)	D-30		
fm (fill memory)	D-31		
go (run)	D-32		
halt	D-33		
he (help)	D-34		
help	D-35		
il (interrupt level)	D-36		
init	D-37		
initialize	D-38		
invoke	D-96		
lb (list breakpoints)	D-39		
log	D-40		
memory	D-41		
mod (debugging mode)	D-43		
mutex	D-44		
new	D-93		
osbreakpoint	D-45		
partition	D-50		
pm (patch memory)	D-51		
pr (patch register)	D-52		
probe	D-53		
psos	D-55		
queue	D-63		
quit	D-65		

q* (query shortcut)	D-59	pROBE+ commands	
region	D-66	pRISM+ shell	D-86
register	D-68	Profiler	
sc (system call)	D-70	description, RTA	12-1
semaphore	D-71	program	
session	D-72	execution tracking	9-1
set	D-92	program editor	
sf (stack frame)	D-76	pRISM+ Editor	5-4
slength	D-97	project editors	
stackfrm	D-77	See Also pRISM+ Editor	1-6
target	D-80	See Also SNIFF+	1-6
task	D-82	project settings	3-7
toString	D-95	projects	
tsd (task-specific data)	D-84	new, pRISM+	3-4
type	D-89	protocols	
t* (task-related)	D-78	list	2-3
using in pRISM+ Shell window	13-2	pRPC+	
version	D-85	remote procedure call library	2-2
vinfo	D-90	psos command, pRISM+ shell	D-55
pRISM+ Wizard		pSOS IO jump table	14-4
error checking	7-6	pSOS Object	
features	7-1	pRISM+ Shell	13-3
interface	7-2	pSOSystem	
modes	7-4	architecture	2-1
See Also pRISM+ Configuration Wizard	1-5	components	2-1, 2-2
probe command, pRISM+ shell	D-53	definitions	1-3, 2-1
pROBE+		environment	2-3
behavior on target	14-8	facilities	2-3
configuring	14-8	overview	2-1
go command	14-6	root directory	2-5
gs command	14-5	pSOS+	
target agent	14-7	real-time multitasking kernel	2-2

pSOS+m			
multiprocessor multitasking			
kernel	2-2		
PSS_APOBJS	2-8		
PSS_BSP	2-8		
PSS_DRVOBJS	2-8		
PSS_ROOT			
directory	2-6, 2-7, 2-8, 2-11		
environment variable	2-5		
PSS_ROOT/bsps	2-8		
Q			
queue			
examining messages			
Object Browser	11-5		
pRISM+ Shell	13-7		
queue command, pRISM+ shell	D-63		
quit command, pRISM+ shell	D-65		
q* (query shortcut) commands	D-59		
R			
region command, pRISM+ shell	D-66		
register command, pRISM+ shell	D-68		
Registers			
viewing, SearchLight Debugger	8-14		
remote communication connection			
host tools configuration	C-7		
remote communication server connection			
how to	C-7		
pRISM+ tools	C-7		
sys_conf.h settings	C-7		
RTA			
Link Map Analyzer description	12-1		
Profiler description	12-1		
Run-Time Error Checker			
description	12-1		
RTA Suite			
definitions	1-9		
Run-time Analysis Suite			
See Also RTA Suite	1-9		
run-time analysis tool			
ESp	1-7		
Object Browser	1-7		
Run-Time Error Checker			
description, RTA	12-1		
S			
sc (system call) command	D-70		
SDM			
See Also System Debug Mode	8-4		
SearchLight Debugger			
accessing	8-2		
commands			
breakpoint	8-17		
next	8-8		
OS Breakpoint	8-9, 8-12, 8-28		
step	8-5		
stepi	8-7		
downloading application	8-2		
examining call stack	8-20		
examining system objects	8-22		
features	8-1		
Files window, navigating	8-15		
Find dialog box	8-17		
pRISM+ Shell	13-5		
setting breakpoint	8-17		
single step commands	8-4		
starting	8-2		

System Debug Mode	8-4	source editors	
Task Debug Mode, using	8-25	See Also pRISM+ Editor	1-6
TDM		See Also SNIFF+	1-6
See Also Task Debug Mode	8-25	source files	2-5
viewing memory	8-12	source view	
viewing registers	8-14	pRISM+ Editor	5-3
SearchLight debugger	1-8	stack problems	
definitions	1-8	monitoring, Object Browser	11-4
semaphore command, pRISM+ shell	D-71	stackfrm command, pRISM+ shell	D-77
serial connection		startup	
booting pSOS+	C-1	dialog.c file	2-6
host tools configuration	C-1	start-up	
how to	C-1	dialog	2-6
pRISM+ tools	C-1	Step command	
sys_conf.h settings	C-1, C-9	using, SearchLight Debugger	8-5
serial driver	14-4	Stepi command	
server		using, SearchLight Debugger	8-7
Debug Server	1-10	system calls	
servers		pMONT	14-2
Communication Server	1-10	system configuration code	2-5
session command, pRISM+ shell	D-72	System Debug Mode	
set command, pRISM+ shell	D-92	SearchLight Debugger	8-4
sf (stack frame) command	D-76	system library	2-7
SingleStep		System Objects	
debugger environment variables	B-3	examining, SearchLight	
SingleStep debugger	1-9	Debugger	8-22
definitions	1-9	sys_conf.h file	2-6, 2-7
slength command, pRISM+ shell	D-97		
SNIFF+		T	
definitions	1-6	target	
environment variables	B-3, B-5	system	2-5
using	3-12	target agent	
		log_event	14-6
		memory usage	14-7

pMONT+ configuration	14-2	tmFreq parameter	14-2, 14-3
pROBE+	14-7	tmRead parameter	14-2, 14-3
pROBE+ configuration	14-8	tmReset parameter	14-2, 14-3
target behavior		toolbar	
pMONT	14-1	customize	4-7
target command, pRISM+ shell	D-80	toString command, pRISM+ shell	D-95
target configuration directory	4-9	trace events	
target definition	14-5	begin, ES _p	10-4
target requirements, pMONT	14-1	end, ES _p	10-4
task command, pRISM+ shell	D-82	traceBuff parameter	14-2, 14-3
Task Debug Mode		traceBuffSize parameter	10-5, 14-2, 14-3
SearchLight Debugger, using	8-25	tracking of program execution	9-1
tasks		transmit	
ASEV	14-5	ES _p	10-3
PMCM	14-5	Transmit buffer overhead	10-3
PMON	14-5	tsd (task-specific data) command, pRISM+ shell	D-84
tasks waiting		type command, pRISM+ shell	D-89
examining, Object Browser	11-6	t* (task-related) commands	D-78
Tcl Script		U	
pRISM+ Shell		user events	10-2
examples	13-8	user_event_id parameter	14-7
TCL scripts		V	
pRISM+ Shell	13-2	variables	
Tcl shell		ARM debugger	B-2
See Also pRISM+ Shell	1-8	CAD-UL	B-4
TCL/CORBA		Diab Data compiler	B-2, B-5
pRISM+ Shell	13-1	environment	2-13
TFTP Server connection		UNIX	B-5
booting pSOS+	C-9	Windows	B-1
host tools configuration	C-9	MKS Toolkit	B-3
how to	C-9	pRISM+	B-4, B-5
pRISM+ tools	C-9		
timeouts			
pRISM+ Shell	13-3		

SingleStep debugger	B-3
SNiFF+	B-3, B-5
version command, pRISM+ shell	D-85
vinfo command, pRISM+ shell	D-90
W	
warnings	14-2
window	
Code	9-9
working directory	2-6, 2-7
wraparound	
ESp	10-4
Wraparound buffer overhead	10-4