

INTEGRITY Development Guide



Green Hills Software
30 West Sola Street
Santa Barbara, California 93101
USA
Tel: 805-965-6044
Fax: 805-965-6343
www.ghs.com

DISCLAIMER

GREEN HILLS SOFTWARE MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Green Hills Software reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Green Hills Software to notify any person of such revision or changes.

Copyright © 1983-2015 by Green Hills Software. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from Green Hills Software.

Green Hills, the Green Hills logo, CodeBalance, GMART, GSTART, Slingshot, INTEGRITY, and MULTI are registered trademarks of Green Hills Software. AdaMULTI, Built With INTEGRITY, EventAnalyzer, G-Cover, GHnet, GHnetLite, Green Hills Probe, Integrate, ISIM, PathAnalyzer, Quick Start, Resource-Analyzer, Safety Critical Products, SuperTrace Probe, TimeMachine, TotalDeveloper, velOSity, and μ -velOSity are trademarks of Green Hills Software.

All other company, product, or service names mentioned in this book may be trademarks or service marks of their respective owners.

Book compiled on February 28, 2018

RTOS Document Id: 54497

Contents

1	Introduction	13
1.1	The INTEGRITY Real-Time Operating System	13
1.2	About This Manual	13
1.3	INTEGRITY 11.7 Document Set	14
1.3.1	MULTI Document Set	15
1.3.2	Conventions Used in Green Hills Documentation	16
2	Getting Started with INTEGRITY	19
2.1	Starting the MULTI Launcher	21
2.2	Creating MULTI Workspaces for Installed BSPs	23
2.3	Building Installed BSPs	25
2.4	Running the Kernel in ISIM and Connecting with rtserv2	27
2.5	Using INTEGRITY Tutorials	29
2.5.1	POSIX Demo	29
2.5.2	Pizza Demo	35
2.6	Target Board Setup	48
2.7	Booting an INTEGRITY Kernel	49
3	INTEGRITY Installation and Directory Structure	51
3.1	INTEGRITY Directory Structure	53
3.2	INTEGRITY Include Directory	54
3.3	target Directory	54
3.3.1	Common Library Source Directories	55
3.4	BSP Source Directory	55
3.4.1	default.gpj	57

3.5	libs Directory	59
3.6	bin Directory	59
4	Using MULTI to Develop INTEGRITY Applications	61
4.1	MULTI Integrated Development Environment	61
4.1.1	MULTI Launcher	61
4.1.2	MULTI Project Manager	62
4.1.3	Freeze-Mode Debug Server	63
4.1.4	Run-Mode Debug Server	63
4.1.5	INTEGRITY Simulator (ISIM)	64
4.1.6	Compiler, Toolchain, and Object File Utilities	64
4.2	MULTI Tools for Finding Performance Bottlenecks	65
4.2.1	MULTI Profile Window	65
4.2.2	ResourceAnalyzer	65
4.2.3	MULTI EventAnalyzer	65
4.2.4	Trace Data	66
5	Building INTEGRITY Applications	67
5.1	INTEGRITY Application Types	67
5.2	Creating an INTEGRITY Top Project	68
5.2.1	Default INTEGRITY Top Project	70
5.3	Monolith INTEGRITY Application Project	72
5.3.1	Creating a Monolith INTEGRITY Project with the New Project Wizard . .	72
5.3.2	Default Monolith INTEGRITY Project	73
5.3.3	Building a Monolith INTEGRITY Application Project	74
5.4	Dynamic Download INTEGRITY Application Project	75
5.4.1	Creating a Dynamic Download Application with the New Project Wizard .	75
5.4.2	Default Dynamic Download INTEGRITY Project	76
5.4.3	Building a Dynamic Download INTEGRITY Application Project	76
5.5	Virtual AddressSpace Project	78
5.5.1	Virtual AddressSpace Program	78
5.6	KernelSpace Project	79
5.6.1	Creating a KernelSpace Project with the New Project Wizard	79

5.6.2	Default KernelSpace Project	81
5.6.3	Building a KernelSpace Project	82
5.6.4	Creating a KernelSpace Project from Scratch	83
5.7	Integrate Configuration File	84
5.7.1	Object Number Usage	84
5.8	Linker Directives File	87
5.8.1	Creating Custom Linker Directives Files	87
5.8.2	default.ld Example	89
5.8.3	Modifying Linker Constants	91
5.8.4	Specifying Memory Regions in the Linker Directives File	92
5.9	Shared Libraries	94
5.9.1	Available Shared Libraries	95
5.9.2	Sharing Libraries Across Multiple INTEGRITY Applications	97
5.10	Additional Options for Building Applications	99
5.10.1	Using Checksum and SHA-1 Utilities for Data Consistency	99
5.10.2	Using INTEGRITY Version Constants to Conditionalize Code	103
5.10.3	Using makefiles	104
6	Configuring Applications with the MULTI Project Manager	107
6.1	Configuring Monolith Projects with the NPW	108
6.2	Modifying Monolith Projects with the Project Manager	112
6.2.1	Settings for Monolith Dialog	112
6.2.2	Converting a Monolith to a Dynamic Download	113
6.3	Configuring Dynamic Download Projects with the NPW	115
6.4	Modifying Dynamic Download Projects with the Project Manager	118
6.4.1	Settings for Dynamic Download Dialog	118
6.4.2	Converting a Dynamic Download to a Monolith	119
6.5	Configuring KernelSpace Projects with the NPW	119
6.6	Modifying KernelSpace Projects with the Project Manager	122
6.6.1	Converting a KernelSpace Project to a Monolith	123
6.7	Adding Items to INTEGRITY Projects	125
6.7.1	Select Item to Add Dialog	125
6.7.2	Adding and Configuring Virtual AddressSpaces with the Project Manager	126

6.7.3	Adding OS Modules with the Project Manager	126
6.8	Copying Examples with the MULTI Project Manager	128
6.8.1	Troubleshooting Copy/Paste in the Project Manager	130
6.9	Converting and Upgrading Projects with the MULTI Project Manager	131
7	Common Application Development Issues	133
7.1	Creating and Destroying INTEGRITY Tasks	133
7.1.1	Creating Tasks Dynamically via INTEGRITY API	134
7.1.2	Creating Tasks Dynamically via POSIX API	135
7.1.3	Creating Tasks Statically via Integrate	135
7.1.4	Using Exit() vs. exit()	137
7.1.5	Automatically Created Tasks	138
7.2	Application Memory Configuration	140
7.2.1	MemoryPool Configuration	140
7.2.2	ExtendedMemoryPool Configuration	141
7.2.3	Heap Configuration	142
7.2.4	Stack Configuration	144
7.2.5	INTEGRITY Shared Memory Configuration	148
7.3	Virtual to Physical Mapping for Code and Data	151
7.3.1	Virtual to Physical Mapping for Monolith Applications	151
7.3.2	Virtual to Physical Mapping for Dynamic Download Application	152
7.4	Application Development Restrictions	153
7.4.1	Restrictions on Hand-Written Assembly Code	153
7.4.2	Restrictions on MULTI Builder and Driver Options	153
7.4.3	MULTI Builder and INTEGRITY Documentation Restrictions	153
7.4.4	Restrictions on KernelSpace Application Code	154
7.5	Clearing Overruns and Handling Multiple Events Simultaneously	155
7.6	Weak Symbols	157
8	ISIM - INTEGRITY Simulator	159
8.1	Introduction to ISIM	159
8.2	ISIM Features	160
8.3	ISIM Target BSPs	160

8.4	Running ISIM	161
8.5	ISIM Socket Emulation	162
8.5.1	ISIM Socket Port Remapping	162
8.6	ISIM Caveats	163
9	Freeze-Mode Debugging	165
9.1	Freeze-Mode Debugging Using Probes	165
9.2	Debugging Out of RAM	166
9.3	Debugging Out of ROM/Flash	167
9.4	Freeze-Mode Debugging Using Host I/O	169
9.4.1	Redirecting the Console over Host I/O	170
10	Connecting with INDRT2 (rtserv2)	171
10.1	Introduction to rtserv2 and INDRT2	171
10.1.1	Communication Media	172
10.2	Building in Run-Mode Debugging Support	173
10.2.1	libdebug.a Configuration	173
10.3	Connecting to rtserv2	174
10.3.1	Using the Run-Mode Partner	174
10.3.2	Using Probe Run Mode	175
10.3.3	Connecting to rtserv2 Using the MULTI Connection Organizer	175
10.3.4	Connecting to rtserv2 Using a Custom Connection Command-Line	179
10.3.5	Connecting to Multiple ISIM Targets	180
10.3.6	Disconnecting from rtserv2	181
11	Run-Mode Debugging	183
11.1	Run-Mode Debugging Limitations	183
11.1.1	Number of Debug Entities	185
11.1.2	Task and AddressSpace Naming	186
11.2	Using the MULTI Debugger with rtserv2	186
11.2.1	Controlling Target Settings	188
11.3	Using the Target Pane	189
11.4	Using the I/O Pane	190
11.4.1	Host I/O	190

11.5 Using Breakpoints with Run-Mode Debugging	192
11.5.1 Using Software Breakpoints with rtsserv2	192
11.5.2 Using Hardware Breakpoints with rtsserv2	193
12 Dynamic Downloading	195
12.1 Introduction to Dynamic Downloading	195
12.2 LoaderTask	196
12.2.1 Download Area Requirements	196
12.3 Initiating a Dynamic Download	197
12.4 Dynamic Download Status	198
12.5 Unloading or Reloading an Application	199
13 Profiling with rtsserv2	201
13.1 Using the MULTI Profile Window	201
13.1.1 Program Counter (PC) Samples	202
13.1.2 Block Coverage Profiling	204
13.1.3 Call Count Data	207
13.2 Generating Profile Data Programmatically	210
13.3 Profiling Overhead and Limitations	211
14 Object Structure Aware Debugging	213
14.1 Introduction to OSA Debugging	213
14.2 Using the OSA Explorer	214
14.2.1 OSA Explorer Task Tab	215
14.2.2 OSA Explorer Connection Tab	217
14.2.3 OSA Explorer Activity Tab	218
14.2.4 OSA Explorer Semaphore Tab	218
14.2.5 OSA Explorer Memory Region Tab	219
14.2.6 OSA Explorer Link Tab	220
14.2.7 OSA Explorer Clock Tab	221
14.2.8 OSA Explorer IODevice Tab	221
14.2.9 OSA Explorer Object Tab	222
14.2.10 OSA Explorer Resource Tab	222
14.3 Using the OSA Object Viewer	224

14.3.1	Common Features of Object View Windows	225
14.3.2	Viewing AddressSpace Objects	228
14.3.3	Viewing MemoryRegion Objects	232
14.3.4	Viewing Task Objects	233
14.3.5	Viewing Semaphore Objects	237
14.3.6	Viewing Clock Objects	241
14.3.7	Viewing Connection Objects	243
14.3.8	Viewing Activity Objects	244
14.3.9	Viewing IODevice Objects	247
14.3.10	Viewing Link Objects	247
15	Run-time Error Checking	249
15.1	Using Run-Time Error Checks	249
15.1.1	Assignment Bounds	251
15.1.2	Array Bounds	252
15.1.3	Case Label Bounds	252
15.1.4	Divide by Zero	253
15.1.5	Nil Pointer Dereference	253
15.1.6	Write to Watchpoint	253
15.2	Using Run-time Memory Checks	254
15.2.1	Memory Leak	255
15.2.2	Access Beyond Allocated Area	256
15.2.3	Bad Free	257
15.3	Detecting Stack Overflow	258
15.3.1	Automatic Stack Overflow Detection	258
15.3.2	Instrumented Stack Checking	258
15.4	Overhead of Run-time Error Checking	260
16	Using the MULTI ResourceAnalyzer	261
16.1	Introduction to the MULTI ResourceAnalyzer	261
16.2	Working with the ResourceAnalyzer	263
16.2.1	System Total and Selected Task/AddressSpace Graphical Panels	265
16.2.2	Data Table	266

17 Core File Debugging	269
17.1 System Core Dumps	269
17.2 Application Core Dumps	270
17.3 Multiple Core Dumps	270
17.4 Debugging Core Dumps	271
17.5 Advanced Core Dumping	271
18 Debug Agent Commands	273
18.1 Supported Debug Agent Commands	273
19 Security Issues	279
19.1 Security and INTEGRITY Kernel	279
19.2 Security and the INDRT2 Debug Agent	280
19.3 Security and Networking Software	280
19.4 Security and ResourceManagers	280
19.5 Security and the EventAnalyzer	281
19.6 Multi-Level Security	281
20 Synchronous Callbacks	283
20.1 Using Callbacks with INTEGRITY	283
20.2 Initializing a Callback	284
20.3 Callback Handlers	285
20.4 Placing a Callback	285
20.5 Callbacks and KernelSpace Device Drivers	286
20.6 Callbacks and IODevices	286
20.7 Callbacks and Timers	287
20.8 Callbacks and Message Queues	287
20.9 Callback Restrictions	287
20.9.1 ResourceManager and Callbacks	287
20.9.2 Callbacks and Stack Size Configuration	288
20.9.3 Restrictions on BSPs and Drivers	288
21 Developing Symmetric Multiprocessing (SMP) Applications	291
21.1 Introduction to SMP	291

21.1.1	SMP and Shared Libraries	292
21.2	Understanding Task Scheduling	292
21.2.1	Kernel Lock Contention	292
21.2.2	Processor Binding	292
21.3	Managing Shared Data	294
21.3.1	Task Priority Does Not Provide Mutual Exclusion	294
21.3.2	Shared Memory Consistency	294
21.3.3	SMP Atomic Operations	297
21.4	Debugging SMP Systems	299
21.4.1	SMP Debugging Tools	299
21.4.2	SMP Run-mode Debugging Limitations	299
22	Troubleshooting	301
22.1	Running a Dynamically Downloadable Image	301
22.2	Dynamic Download from Debugger Failed	302
22.3	Debugging Problems Downloading with the Virtual Loader	303
22.4	MULTI / rtserver Loss of Communication to the Board	303
22.5	Undefined Symbol main()	304
22.6	INTEGRITY Violation Before main()	304
22.7	No Kernel Banner	305
22.8	Not Enough Target RAM Memory Allocation Errors	305
22.9	Cannot Find Target File Error	305
22.10	Task Defined in Integrate Configuration File Cannot Exit	306
22.11	MULTI Project Manager Reports Incorrect Component Types for Legacy Projects	307
22.12	Checksum Warning	307
22.13	Stack Overflow	308
22.14	CheckSuccess at Boot Time	308
22.15	CheckSuccess: CannotRaiseTaskInterruptPriorityLevelBelowCurrentLevel	310
22.16	Performance Bottlenecks	310
Index		312

Chapter 1

Introduction

This chapter contains:

- The INTEGRITY Real-Time Operating System
- About This Manual
- INTEGRITY Document Set

1.1 The INTEGRITY Real-Time Operating System

The INTEGRITY real-time operating system is a secure, high reliability system intended for use in mission critical embedded systems. The INTEGRITY real-time operating system uses hardware memory protection to isolate and protect itself and user tasks from incorrect operation caused by accidental errors or malicious tampering. Its object-oriented design allows verification of the security and integrity of data, communications, individual components, and the system as a whole. Its strict adherence to provable resource requirements allows an embedded system designer to guarantee resource availability.

Unlike other memory protected operating systems, the INTEGRITY real-time operating system does not sacrifice real-time performance for security and protection. It is first and foremost a real-time operating system.

1.2 About This Manual

The *INTEGRITY Development Guide* describes the important files in an INTEGRITY installation and how to navigate through them in order to build system libraries (for standard source or kernel source installations) and user applications. This guide also describes how to perform application or run-mode debugging as well as kernel or freeze-mode debugging. This guide is meant to be a central resource for learning how to make the most of the Green Hills MULTI environment as it pertains to software development for INTEGRITY.

1.3 INTEGRITY 11.7 Document Set

INTEGRITY documentation is available in the following formats:

- Printed books (select books are not available in print).
- Online help, accessible from MULTI's **Help** menu.

This method is generally the best option because the MULTI Help Viewer has powerful search capabilities and options for displaying online help. For more information see “The Help Viewer Window” in the “Introduction” chapter of *MULTI: Getting Started*.

- PDF documents for viewing in your favorite PDF viewer.

PDF versions of documents are located in *install_dir\manuals*. This directory may also contain pdf-only manuals that are not included in the online help system or available in printed format.

The INTEGRITY 11.7 documentation set includes the following manuals:

- **INTEGRITY Installation Guide** — provides information that is specific to installing and using the INTEGRITY real-time operating system. The *Installation Guide* provides directions for installing the INTEGRITY binary distribution, describes how to install optional items and licensed source code components, and explains how to connect to your target and run the INTEGRITY kernel.
- **INTEGRITY Development Guide** — describes the important files in an INTEGRITY installation and how to navigate the files to build system libraries (for standard source or kernel source installations) and user applications. The guide also describes how to perform run-mode application debugging as well as freeze-mode kernel debugging. The *Development Guide* is meant to be a central resource where the user can learn to maximize the benefits of the Green Hills MULTI Environment as it pertains to software development for INTEGRITY.
- **INTEGRITY Kernel User's Guide** — describes the major INTEGRITY RTOS concepts and provides the core INTEGRITY API description. The INTEGRITY API includes functions for manipulating all of the INTEGRITY Object types, including Tasks, Semaphores, Connections, MemoryRegions, IODevices, Activities, and AddressSpaces.
- **INTEGRITY Kernel Reference Guide** — contains reference material for the *INTEGRITY Kernel User's Guide*. It contains complete information about INTEGRITY kernel function calls, including a functional description, restrictions, arguments, and return values.

- **INTEGRITY Libraries and Utilities User's Guide** — contains information about a variety of system services and libraries, such as the INTEGRITY shell, INTEGRITY file system, POSIX (an alternate API to the standard INTEGRITY API), and device drivers.
- **INTEGRITY Libraries and Utilities Reference Guide** — contains reference material for the *Libraries and Utilities User's Guide*. This book contains function calls for a variety of system services libraries such as POSIX, file system, and device drivers. Functional description, restrictions, arguments, and return values are provided for each function.
- **INTEGRITY Networking Guide** — provides information about network configuration, network and sockets libraries, and the GHnet v2 TCP/IP stack, including the dual-mode IPv4/IPv6 stack. Also includes information about GHnet v2 optional services, such as the web server.
- **INTEGRITY BSP User's Guide** — describes the functions that must be defined and called when porting INTEGRITY to a new board. It also includes important information regarding how to write device drivers, install interrupt service routines, and other low-level software. A single ASP (Architecture Support Package) accommodates all the boards that are powered by the same processor family.
- **Integrate User's Guide** — contains information about using the Integrate utility. Integrate allows you to create multiple virtual AddressSpaces, each containing a variety of possible Objects such as Tasks, Semaphores, and Connections. INTEGRITY supports the common method of creating such objects dynamically via kernel calls, with Integrate you have the option of statically specifying a variety of RTOS objects and relationships, saving code development time.
- **EventAnalyzer User's Guide** — provides information about using the MULTI EventAnalyzer to collect and analyze data. The EventAnalyzer helps programmers understand the complex interaction of resources in a real-time operating system by providing a graphical representation of system activities. With the EventAnalyzer, system events occurring within microseconds can easily be isolated and analyzed. This system analysis tool effectively gives programmers the ability to stop time in order to scrutinize system behavior.
- **INTEGRITY Implementation-Dependent Behavior Guide** — provides information about implementation-dependent behavior. Because implementation-dependent behavior can change between minor INTEGRITY releases, you should use this manual for general guidelines, but be aware that details may change without notice.

1.3.1 MULTI Document Set

In addition to the INTEGRITY document set, the Green Hills toolchain comes with a comprehensive set of documentation to aid developers using Green Hills compilers, toolchain,

and the MULTI Integrated Development Environment (IDE). These books are available as printed manuals, as **pdf** files, and through MULTI’s **Help** menu.

INTEGRITY manuals focus on the tools that an INTEGRITY user is most likely to use and often refer to the MULTI manuals. Programmers (especially those who are just starting to use MULTI for the first time) will find the MULTI manuals to be useful companions. While the INTEGRITY manuals focus on high level tools and commonly used commands, the MULTI manuals provide detailed information about each tool. Some of the information in MULTI manuals is not pertinent to INTEGRITY users, or superseded by methods described in INTEGRITY manuals.

The primary documents in the MULTI document set are:

- **MULTI: Getting Started** — provides an introduction to the MULTI Integrated Development Environment (IDE) and leads you through a simple tutorial.
- **MULTI: Licensing** — describes how to obtain, install, and administer Green Hills licenses.
- **MULTI: Managing Projects and Configuring the IDE** — describes how to create and manage projects and how to configure the MULTI IDE.
- **MULTI: Building Applications** — describes how to use the compiler driver and the tools that compile, assemble, and link your code. Also describes the Green Hills implementation of supported high-level languages.
- **MULTI: Configuring Connections** — describes how to configure connections to your target.
- **MULTI: Debugging** — describes how to set up your target debugging interface for use with MULTI and how to use the MULTI Debugger and associated tools.
- **MULTI: Debugging Command Reference** — explains how to use Debugger commands and provides a comprehensive reference of Debugger commands.
- **MULTI: Scripting** — describes how to create MULTI scripts. Also contains information about the MULTI-Python integration.

1.3.2 Conventions Used in Green Hills Documentation

Green Hills documentation uses the following typographical conventions to present information.

Bold Type indicates:

- Filenames or pathnames
- Commands
- Options
- Window titles
- Program names
- Button names
- Menu names or menu items
- Field names

Italic Type indicates:

- Titles of books
- Text that the user should replace with an appropriate argument, command, filename, or other value
- New terms or emphasis

`Monospace Font` indicates:

- Code samples
- Text that should be entered exactly as presented

Chapter 2

Getting Started with INTEGRITY

After INTEGRITY and the MULTI Integrated Development Environment have been successfully installed, every installed BSP must be built before new projects can be created or a kernel can be booted. In addition, if new BSPs or source packages are added to the INTEGRITY installation at a later time, or the MULTI tools used with the INTEGRITY distribution are modified, all installed BSPs must be built again to ensure that all libraries are up-to-date, evaluation time limits are removed, and shared libraries are linked with the appropriate run-time libraries provided with the toolchain. This chapter focuses on using the INTEGRITY Simulator to get started, but the basic steps for getting started also apply to hardware targets.

This chapter contains the following sections:

- Starting the MULTI Launcher
- Creating MULTI Workspaces for Installed BSPs
- Building Installed BSPs
- Running the Kernel in ISIM and Connecting with rtserv2
- Using INTEGRITY Tutorials
- Target Board Setup
- Booting an INTEGRITY Kernel

This chapter contains step-by-step instructions for downloading a kernel using the INTEGRITY Simulator (ISIM) as the target. ISIM is an ideal method to get up and running quickly because there are so few contingencies. The ISIM instructions provide a good example because the process of booting an INTEGRITY kernel is very simple and will be the same for every user. This process is summarized in the “Running the Kernel in ISIM and Connecting with rtserv2” section of this chapter, and described in greater detail in the “ISIM - INTEGRITY Simulator” chapter of this manual.

After you have connected to ISIM you can go through tutorials in the “Using INTEGRITY Tutorials” section of this chapter. The tutorials provide step-by-step instructions for building new applications and running example applications that are included in the installation. They are an

excellent way to quickly understand how to develop applications with the INTEGRITY RTOS and the MULTI Integrated Development Environment (IDE).

After the tutorials have been mastered, the **examples** directory contains additional INTEGRITY applications that demonstrate commonly used aspects of the INTEGRITY API and other layered software components included with the installation. For more information about how to use these examples, see “Copying Examples with the MULTI Project Manager” in the “Configuring Applications with the MULTI Project Manager” chapter of this manual.

Connecting to an actual board requires additional configuration steps not covered in the ISIM instructions. Some of these issues are described in the “Target Board Setup” and “Booting an INTEGRITY Kernel” sections later in this chapter. Additional information is in the **.notes** file for each BSP.

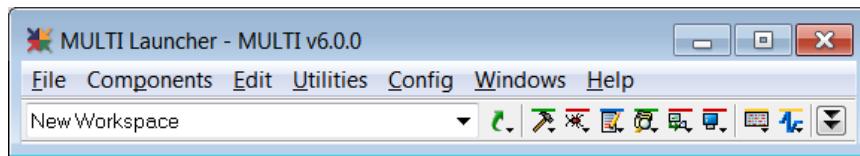
2.1 Starting the MULTI Launcher

The MULTI Launcher provides a convenient way to launch frequently used tools, create new files and projects, access recently used files and projects, and manage workspaces. The Launcher's **Windows** menu also provides a centralized window manager you can use to access any of your open MULTI windows.

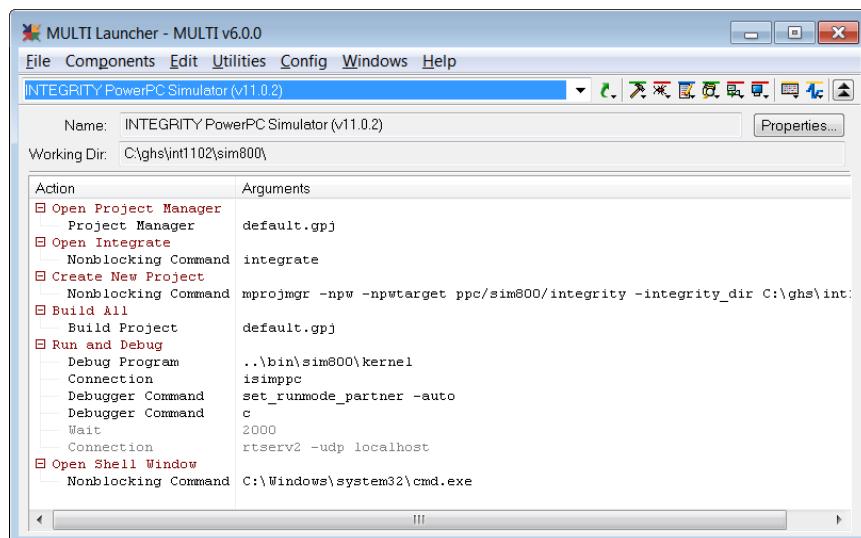
To start the MULTI Launcher:

1. Select **Start⇒Programs⇒Green Hills Software⇒MULTI *release name*⇒MULTI**.

The MULTI Launcher will open.



2. If the detail pane is hidden, click the **Show Detail Pane** () button.



The main MULTI components can be accessed with buttons on the Launcher toolbar:

— Runs a shortcut or an action sequence in the current workspace. Also allows you to create a new workspace or create or edit a shortcut.

— Opens the Project Manager on a recent or new project.

— Opens the Debugger on a recent or new executable.



— Opens the Editor on a recent or new file.



— Opens the Checkout Browser on a recent or new checkout.



— Opens the Connection Organizer or a recent or new target connection.



— Opens a Serial Terminal using a recent or new connection.



— Opens the EventAnalyzer (licensed separately).



— Opens the ResourceAnalyzer (licensed separately).



— Closes the MULTI Launcher (UNIX only by default).



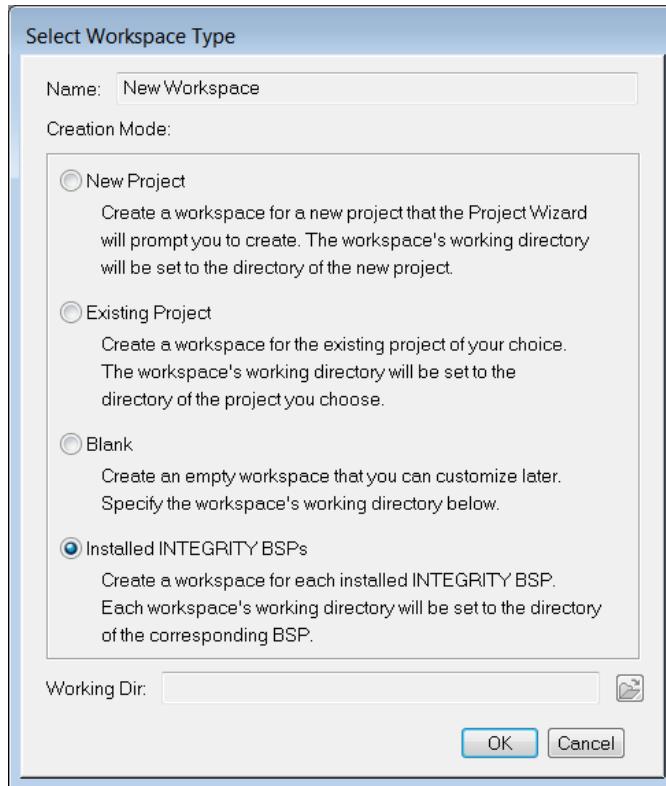
— Shows/hides the detail pane of the Launcher.

2.2 Creating MULTI Workspaces for Installed BSPs

The MULTI Launcher can automatically create default workspaces for BSPs you have installed. These workspaces add shortcuts to the Launcher that perform common tasks such as opening the Project Manager, building all INTEGRITY source code for the BSP, creating new projects, and opening Integrate. For simulators, MULTI creates Launcher shortcuts that automatically launch an INTEGRITY simulator (ISIM) and run an INTEGRITY kernel within it. This section provides instructions for creating these default workspaces. After you have created a workspace, you can modify it to suit your needs, or create additional workspaces as needed.

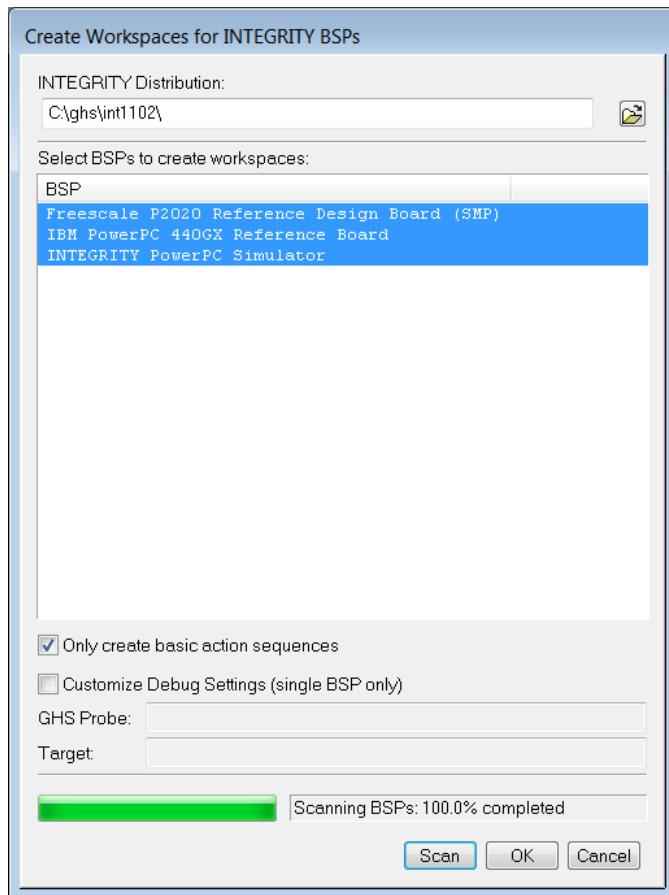
For more information about using the MULTI Launcher and creating workspaces, see the “Managing Workspaces and Shortcuts with the Launcher” chapter of *MULTI: Managing Projects and Configuring the IDE*.

1. From the MULTI Launcher, select **File⇒Create Workspace**. The Select Workspace Type dialog will open.



2. Select the **Installed INTEGRITY BSPs** radio button and click **OK**.
3. Enter the location of your INTEGRITY installation in the **INTEGRITY Distribution** text box (if it is not already set correctly) and click **OK**. The MULTI Launcher scans the directory and displays all detected BSPs.

4. Highlight all of the BSPs for which workspaces should be created. Be sure to highlight the INTEGRITY simulator BSP you will be using for the remainder of this section. When finished, click **OK**.

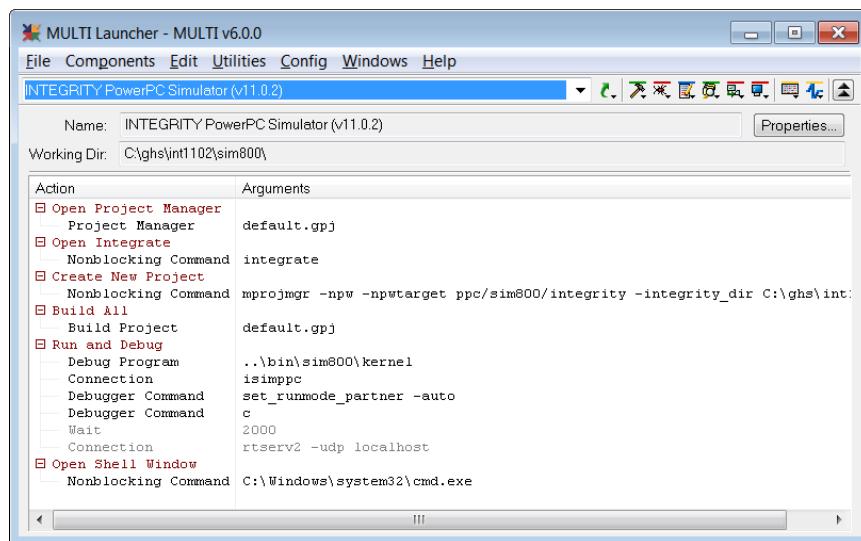


The MULTI Launcher creates default workspaces for each BSP you selected. The workspace names are derived from the name of the board. For example, the INTEGRITY PowerPC simulator workspace is named **INTEGRITY PowerPC Simulator**.

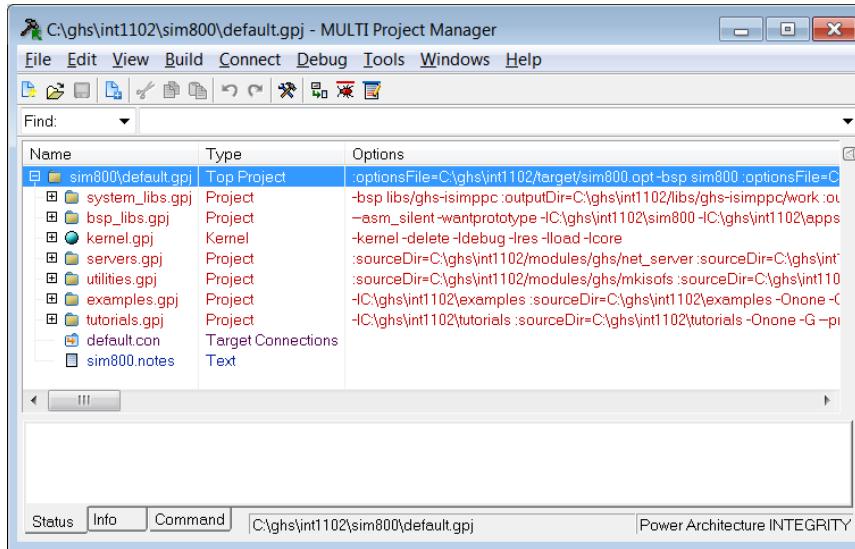
2.3 Building Installed BSPs

After installing INTEGRITY, you must rebuild all installed BSPs before creating new projects or booting a kernel. The instructions in this section use the Simulator BSP, but these same steps must be followed for all installed BSPs after installation, and anytime you install a new version of MULTI tools or add components to your installation. Failing to rebuild a BSP can result in later project build failures (missing libraries), inconsistent behavior between static and shared libraries, and other problems. Building the BSP is also a prerequisite for performing the tutorials in the “Using INTEGRITY Tutorials” section of this chapter. If you are new to using MULTI to create INTEGRITY applications, we recommend that you perform the tutorials.

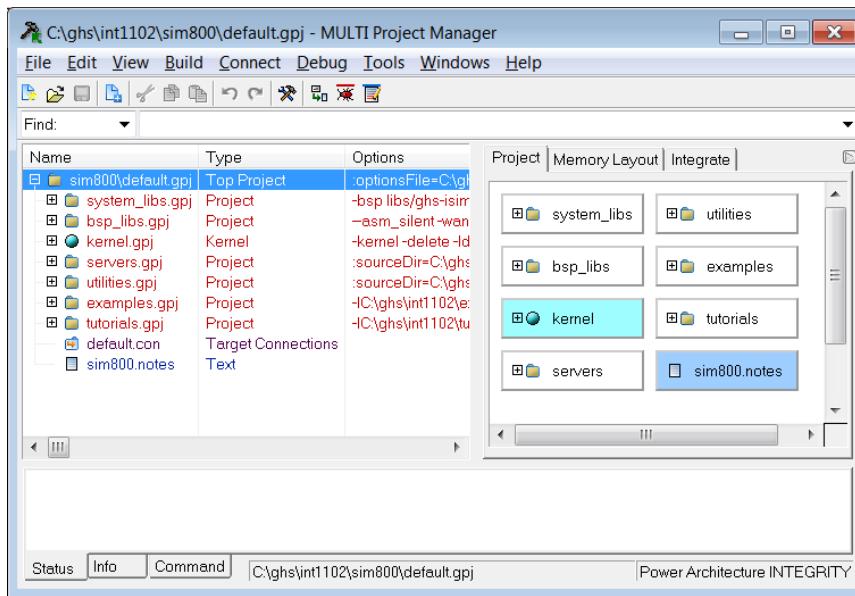
If the MULTI Launcher is not already open, use the instructions in “Starting the MULTI Launcher” to open MULTI.



1. Use the Workspace selection drop-down menu to select the BSP you want to build.
2. From the Launcher, double-click the **Open Project Manager** action to open **default.gpj**. The MULTI Project Manager window will open with the default view as follows:



3. Select **View⇒Show All Views** or click the **Show View Tabs** arrow on the right border to display Advanced Views tabs in the right pane.



4. If you have installed new MULTI tools or INTEGRITY components, you must clean the build of any installed BSPs before proceeding. To do this, select **Build**⇒**Clean default.gpj**.
 5. Select **Build**⇒**Build Top Project default.gpj**. This builds **default.gpj** for the BSP, including the examples and tutorials for the simulator.
 6. Repeat this process for all installed BSPs.

2.4 Running the Kernel in ISIM and Connecting with rtserver2

This section describes the steps needed to boot an INTEGRITY kernel in the INTEGRITY Simulator (ISIM) and connect to the running kernel with the run-mode debugger, **rtserver2**.

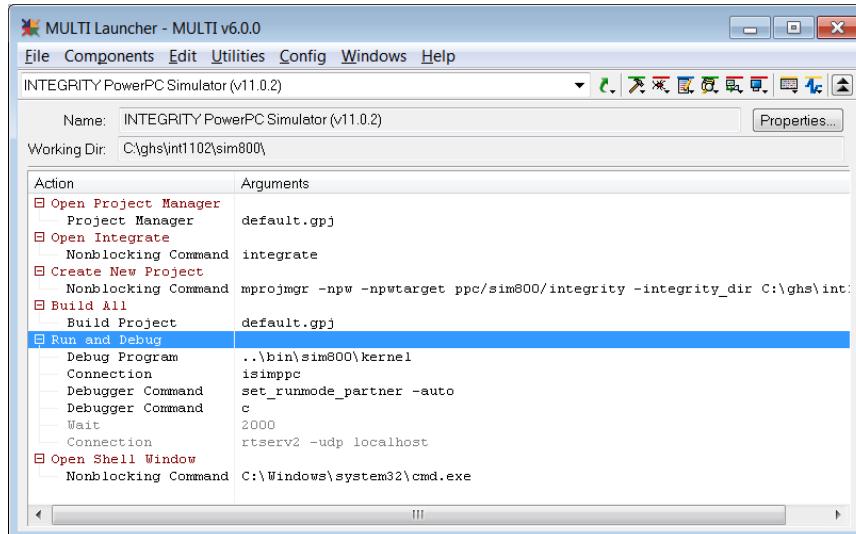
ISIM is a simulation environment for the INTEGRITY real-time operating system. It enables programmers to develop and test embedded INTEGRITY-based applications when target hardware is not yet manufactured, or in supply too limited to accommodate the entire programming team. Unlike conventional RTOS simulators that run as native UNIX or PC applications, ISIM simulates the same code that runs on the target. Thus, application size characteristics are known during simulation. In addition, the exact same compilers and development tools are used for both simulation and for actual hardware development. This increases the value of the development tools investment and the productivity of the programming team. ISIM provides complete INTEGRITY simulation, including virtual memory and memory protection. If you are unfamiliar with the MULTI IDE, we recommend that you begin by using the INTEGRITY Simulator (ISIM) as your target.

The MULTI Launcher adds a very powerful action sequence into INTEGRITY simulator workspaces: **Run and Debug**. This action sequence automatically performs the following steps:

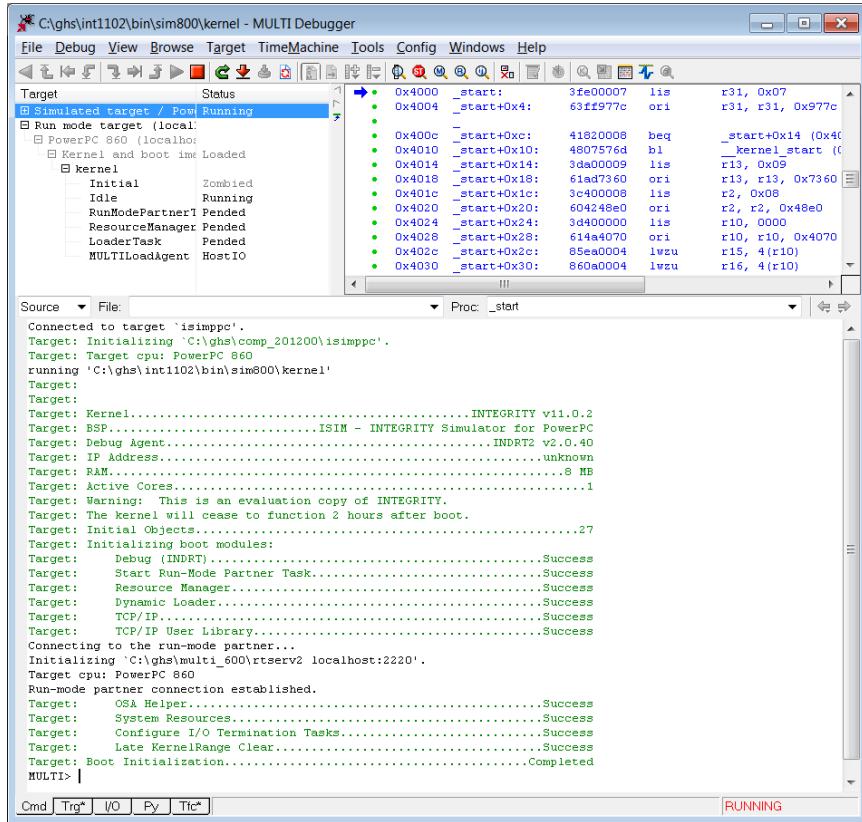
- Launches the MULTI Debugger on an INTEGRITY kernel.
- Connects to the appropriate INTEGRITY simulator.
- Instructs the Debugger to download the kernel to the simulator and start it running.
- Launches **rtserver2**, the debug server, and instructs it to connect to the running kernel.

These instructions assume that you have already created the MULTI Launcher workspace for the simulator of your choice as described in the “Creating MULTI Workspaces for Installed BSPs” section earlier in this chapter.

1. On the Launcher’s Action list, double-click **Run and Debug**.



Upon successful connection to the ISIM target, the run-mode debug Tasks will appear in the MULTI Debugger Window:



2. The first time you open the MULTI Debugger, the Target list is located at the top of the window, with the source pane below. INTEGRITY users should move the target list to the left side of the Debugger window (as shown in the picture above). To do this, click the **Move target list to left** button (), located to the right of the column headings. MULTI remembers the location of the Target list across sessions.

You have now completed the process. An INTEGRITY kernel is running on ISIM and you have connected the MULTI Debugger to the running kernel. If you are new to INTEGRITY, it is highly recommended that you complete the tutorials in the following section before attempting to create your own application.

2.5 Using INTEGRITY Tutorials

This section provides information about the following tutorials designed to introduce you to INTEGRITY:

- POSIX Demo
- Pizza Demo

These tutorials step through demos of existing applications contained in the INTEGRITY installation. If you are new to INTEGRITY, it is highly recommended that you complete these tutorials before attempting to create your own application.

To use the tutorials, you must have an INTEGRITY kernel running that has the **Debugging** library (**libdebug.a**) linked. **libdebug.a** is required to establish an rtserver2 debug connection and to dynamically download these applications. The tutorial applications are set up for dynamic downloading, but you can also reconfigure them as monolith INTEGRITY application projects.

The tutorials are located in **default.gpj**⇒**tutorials.gpj**. These instructions assume that you have already completed the following steps:

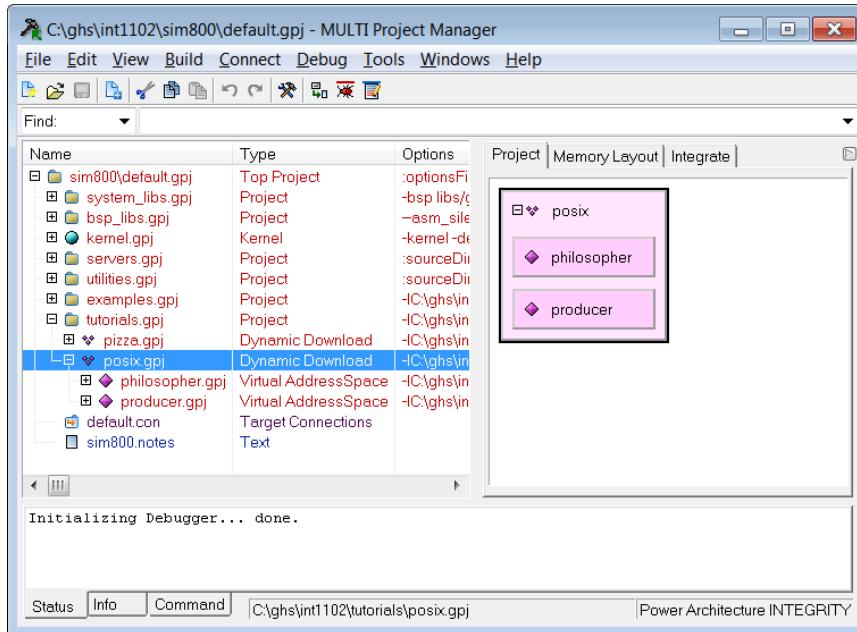
1. Create the MULTI Launcher workspace for the simulator of your choice as described in the “Creating MULTI Workspaces for Installed BSPs” section.
2. Build the examples in **default.gpj** as described in the “Building Installed BSPs” section.
3. Have an INTEGRITY kernel running on ISIM and connected to the MULTI Debugger as described in “Running the Kernel in ISIM and Connecting with rtserver2”. The same kernel and INDRT2 (rtserver2) connection can be used for all of the tutorials.

2.5.1 POSIX Demo

The **POSIX Demo** application demonstrates basic task management and inter-task communications using the POSIX threads (pthread) API. This demo contains the “Dining Philosophers Demo” and the “Producer/Consumer Demo” which are described in this section.

To run the demo:

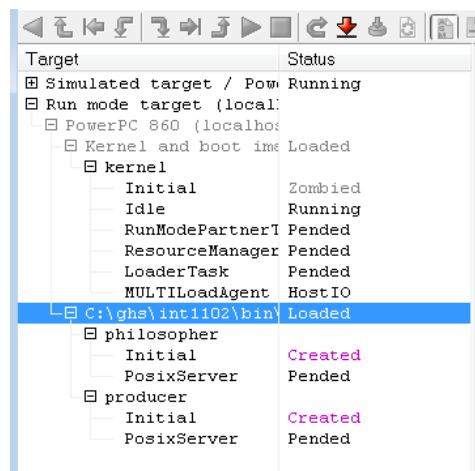
1. Make sure an INTEGRITY kernel is running on the simulator and an INDRT2 (rtserver2) connection has been established.
2. Start the demo from the MULTI Project Manager by right-clicking **posix.gpj** and selecting **Load Module posix** to download the **posix** module.



Or, from the MULTI Debugger, select **Target**⇒**Load Module**⇒ **Load Module**, and use the file chooser to navigate to *rtos_install_dir\bin\bspname*, and select **posix.ael**.

3. Make sure **Debug**⇒**Target Settings**⇒**Stop On Task Creation** is disabled in the MULTI Debugger. This causes newly spawned tasks to remain unattached, which you want for purposes of this tutorial because you are not going to debug them individually. (If you want to examine the programs more closely later, you can rerun with **Stop On Task Creation** enabled to debug the individual tasks).

After the download is complete, the following AddressSpaces display in the Target list:



- **philosopher** — contains a demonstration of the dining philosophers problem (and solution).

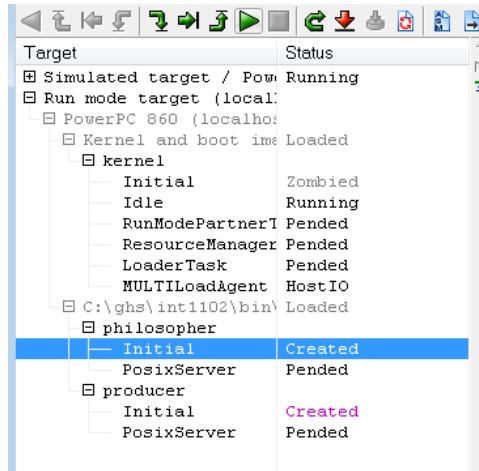
- **producer** — contains a demonstration of the classic producer/consumer problem (and solution).

Each of the AddressSpaces contains a task named **Initial**, otherwise known as the Initial Task.

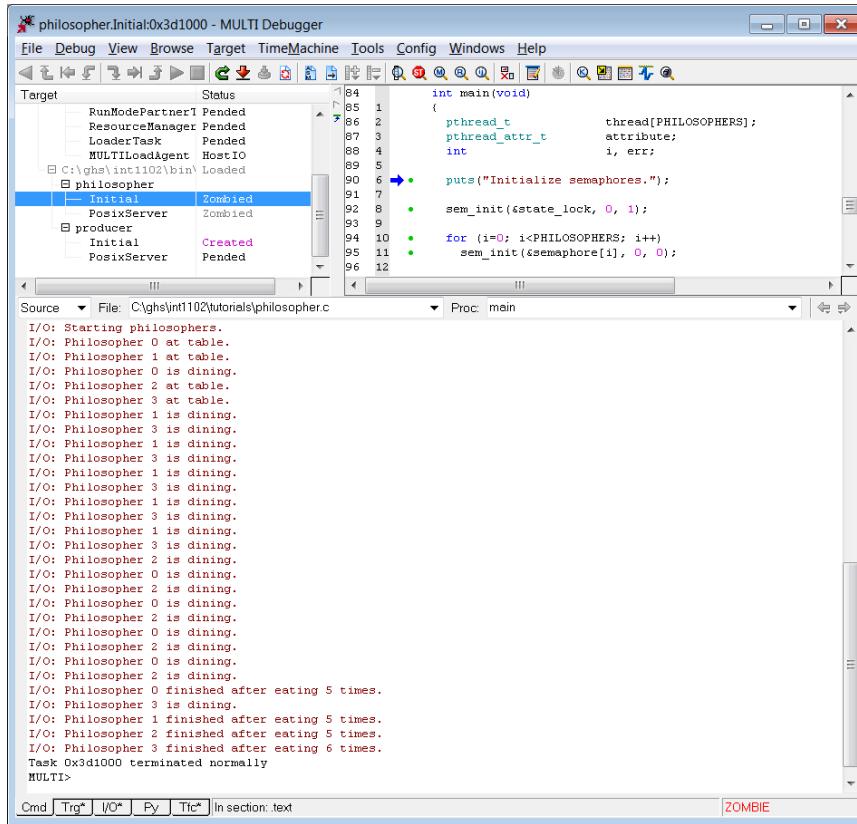
2.5.1.1 Dining Philosophers Demo

The dining philosophers problem, a classic in computer science, was proposed and solved by Dijkstra, using semaphores. There are N philosophers and N forks (or chopsticks if preferred). Any philosopher must pick up two forks in order to dine. This means that only $N/2$ philosophers can dine at once. The problem is that each philosopher might pick up the fork to his/her left and then try to pick up the fork to his/her right, only to find that it is not available. If each philosopher waits for his/her second fork to become available, then none of them will ever eat. The situation that results is known as *deadlock*, which is to be avoided. The philosopher AddressSpace demonstrates a solution to this problem using POSIX threads and semaphores.

1. In the MULTI Debugger Target list, select the Initial Task of the **philosopher** AddressSpace.



2. Click **Go** in the new Debugger window. Output from the Initial Task and the **philosopher** tasks should look similar to the following:

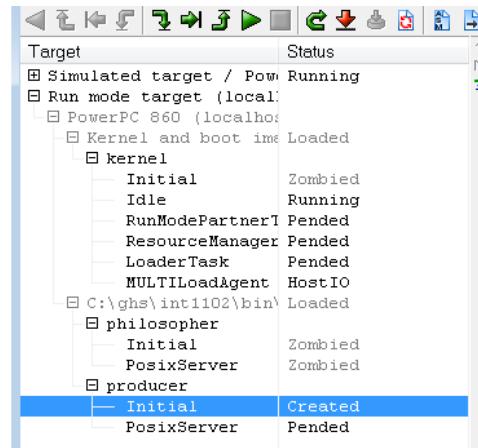


As the demo proceeds, notice the philosopher threads as they are spawned and appear in the MULTI Debugger Target list. The **PosixServer** task for this AddressSpace is also shown. When the Initial Task terminates, the demonstration is complete.

2.5.1.2 Producer/Consumer Demo

The producer/consumer problem is found often in multi-threaded software. In general, there can be some producers and some consumers of some resource. As an example, the producers may be generating information that they store in a queue, and the consumers may retrieve and act on the data that they find in the queue. This type of situation often occurs with communication systems, such as TCP/IP stacks. The problem arises when more than one agent attempts to access the global queue of information. If there is no synchronization, more than one producer could write to the same location in the queue, thus losing data. Another problem could arise if more than one consumer attempts to read from the same location. This implementation uses POSIX thread mutexes and condition variables to synchronize accesses.

1. Select Task Initial of the **producer** AddressSpace in the MULTI Debugger Target list.



2. In the Debugger window, click **Go**. The producer Initial task spawns the producer and consumer threads. Output from the Initial task as well as the producer and consumer threads is displayed in the command pane and should look similar to the following:

The screenshot shows the Integrity Debugger's main window during execution. The top menu bar includes File, Debug, View, Browse, Target, TimeMachine, Tools, Config, Windows, and Help. The Source tab is active, displaying the C code for the 'producer' task. The code includes comments for initializing locks and condition variables, and puts() statements for logging. The command pane at the bottom shows the following output:

```

I/O: Producing item 0.
I/O: Producing item 1.
I/O: Consuming item 0. There are 0 items left.
I/O: Producing item 2.
I/O: Consuming item 1. There are 0 items left.
I/O: Producing item 3.
I/O: Consuming item 2. There are 0 items left.
I/O: Producing item 4.
I/O: Consuming item 3. There are 0 items left.
I/O: Producing item 5.
I/O: Consuming item 4. There are 0 items left.
I/O: Producing item 6.
I/O: Consuming item 5. There are 0 items left.
I/O: Producing item 7.
I/O: Consuming item 6. There are 0 items left.
I/O: Producing item 8.
I/O: Consuming item 7. There are 0 items left.
I/O: Producing item 9.
I/O: Consuming item 8. There are 0 items left.
I/O: Consuming item 9. There are 0 items left.
I/O: The producer thread exited with status 0.
I/O: The consumer thread exited with status 0.
Task 0x496000 terminated normally
MULTI>

```

The status bar at the bottom right indicates 'ZOMBIE'.

As the demo proceeds, notice the producer and consumer threads as they are spawned and appear in the Task Manager. The **PosixServer** task for this AddressSpace is also shown in the Task Manager.

3. Before proceeding to load new applications, unload the application from the target with one of the following methods:

- Right-click the **posix** line in the Target list and select **Unload Module**.
- Select **Target⇒Unload Module** and select the **posix** module to unload.
- Type `unload posix` in the Target pane.

2.5.2 Pizza Demo

The **Pizza Demo** application demonstrates basic inter-task and inter-AddressSpace communications using the INTEGRITY API. Stepping through the demo also demonstrates how you can debug multiple tasks simultaneously to effectively determine how these communications affect the system. The **Pizza Demo** involves four virtual AddressSpaces, with work accomplished by the Initial Task in each AddressSpace. We will refer to each Initial Task by its AddressSpace name because the Initial Task is the only task that executes in each AddressSpace of this demo. Thus, the four tasks in the system are as follows:

- PhoneCompany Task
- Information Task (411)
- PizzaHut Task
- Engineer Task

The flow of control for the entire application is as follows:

- Engineer calls Information (411) and asks for the phone number for Pizza Hut.
- Engineer uses number to call Pizza Hut and order a pizza. Engineer also gives his address and phone number.
- Pizza Hut calls the Engineer back to verify the order.
- Pizza Hut makes the pizza and delivers it.
- Engineer eats the pizza.

The Phone Company Task manages all the telephone Connections involved. The PhoneCompany AddressSpace begins with one Connection to each of the three other Pizza Demo AddressSpaces. The PhoneCompany Task's flow of control is as follows:

```
loop forever

    Wait for someone to dial their phone
    Identify the Caller
    Identify the Callee
    Create a Connection
    Pass one end of the Connection to the Caller
    Pass the other end of the Connection to the Callee

end loop
```

The Information AddressSpace begins with one Connection to the PhoneCompany AddressSpace. Flow of control for the Information task is as follows:

```
loop forever

    Wait for someone to call
    Send "What listing, please?"
```

```
Receive listing request
Send phone number for listing
Hangup Telephone

end loop
```

The PizzaHut AddressSpace begins with one Connection to the PhoneCompany AddressSpace.
Flow of control for the PizzaHut task is as follows:

```
loop forever

    Wait for someone to call
    Send "Pizza Hut, may I take your order?"
    Receive Order
    Send query "What's your address?"
    Receive Address
    Send query "What's your number?"
    Receive PhoneNumber
    Hangup Telephone
    Dial PhoneNumber on Telephone
    Wait for telephone connection
    Receive Greeting
    Send query "Did you order a pizza?"
    Receive Response
    Hangup Telephone
    if Response is "Yes" then

        Make pizza Order
        Deliver pizza to Address

    end if

end loop
```

The Engineer AddressSpace begins with one Connection to the PhoneCompany AddressSpace.
Flow of control for the Engineer task is as follows:

```
loop forever

    Work
    Dial "411" on Telephone
    Wait for telephone connection
    Receive Greeting
    Send "Pizza Hut"
    Receive Response
    Hangup Telephone
    Dial Response on Telephone
    Receive Connection
    Receive Greeting
    Send "Large pepperoni pizza, no cheese"
    Receive Response
    loop while Response is not "What's your address?"
```

```
Send "What?"
Receive Response

end loop
Send "30 West Sola Street"
Receive Response
loop while Response is not "What's your number?"

    Send "What?"
    Receive Response

end loop
Send "965-6044"
Hangup Telephone
loop forever

    Wait for someone to call Telephone
    Greet caller with "Green Hills Software"
    Receive Response
    if Response is "Did you order a pizza?"

        Send "Yes"
        Hangup Telephone
        exit inner loop

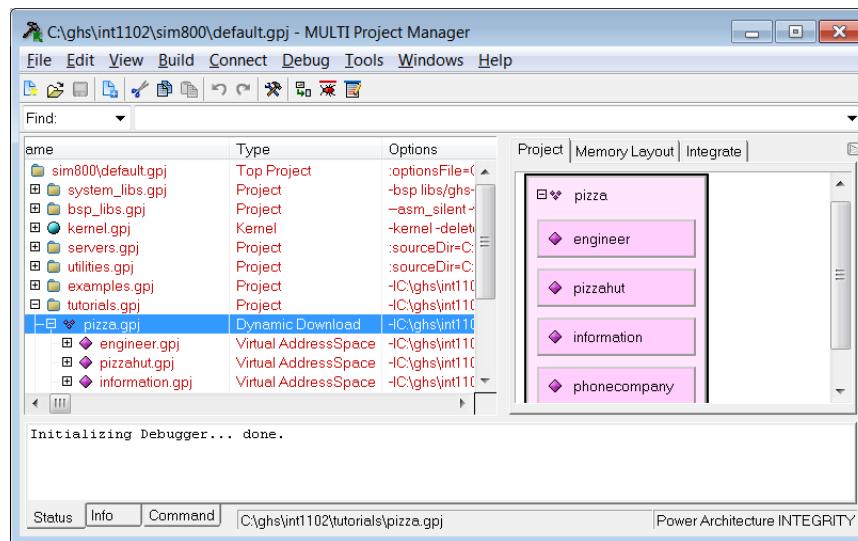
    end if
    Hangup Telephone

end loop
Wait for pizza
Eat pizza
Sleep

end loop
```

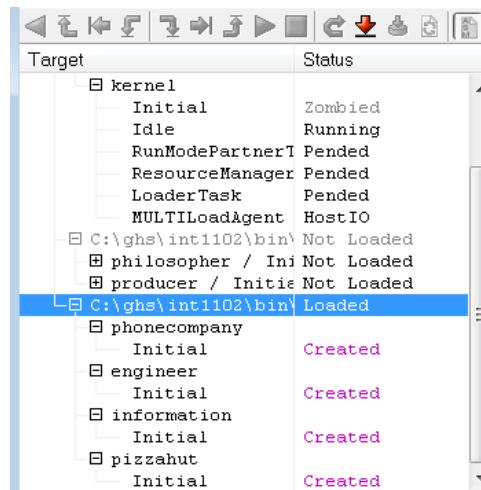
2.5.2.1 Start the Demo

1. Make sure the **pizza** application has been built, the INTEGRITY kernel is running, and an rtser2 connection has been established. For instructions, see “Building Installed BSPs” and “Running the Kernel in ISIM and Connecting with rtser2” earlier in this chapter.
2. Start the demo from the MULTI Project Manager by right-clicking **pizza.gpj** and selecting **Load Module pizza** to download the **pizza** module.



Or, from the MULTI Debugger, select **Target⇒Load Module⇒ Load Module**, and use the file chooser to navigate to *rto*_s*_install_dir\bin\bspname*, and select **pizza.ael**.

After the download is complete, the following AddressSpaces will display in the Target list.



- **phonecompany**
- **engineer**
- **information**
- **pizzahut**

Each AddressSpace contains a single task named **Initial**. These are the AddressSpace Initial Tasks.

3. In the MULTI Debugger Target list, select the Initial task in the **engineer** AddressSpace. The Source Pane will display the main() function for the Engineer task.

```

engineer.Initial:0x397000 - MULTI Debugger
File Debug View Browse Target TimeMachine Tools Config Windows Help
Target Status
kernel Initial Zombied
Idle Running
RunModePartner1 Pended
ResourceManager Pended
LoaderTask Pended
MULTIloadagent HostIO
C:\ghs\int102\bin\Not Loaded
philosopher / Ini Not Loaded
producer / Initis Not Loaded
C:\ghs\int102\bin\Loaded
phonecompany Initial Created
information Initial Created
pizzahut Initial Created
engineer Initial Created
    
```

```

17 #include "ipc.h"
18 #include <string.h>
19
20
21
22 /* The Engineer task */
23 void main(void)
24 {
25     Connection Telephone;
26     Connection C;
27     char *Response = NULL;
28
29     count = 0;
30     GetConnection(10, &Telephone);
31
32     while (1) {
33         Work();
34
35         C = Dial(Telephone, "411");
36         Response = ReceiveString(C);
37
38         SendString(C, "Pizza Hut");
39         Response = ReceiveString(C);
40
41         C = Dial(Telephone, Response);
42         Response = ReceiveString(C);
43         SendString(C, "Large pepperoni pizza, no cheese");
44
45         while (!equal(Response, "What's your address?"))
46             SendString(C, "What?");
47         Response = ReceiveString(C);
        
```

4. Click the break dot on file line 35 of code (`C = Dial(Telephone, "411");`), then click **Go**. The break dot will be replaced with a stop sign. Execution will resume and the breakpoint will be hit.

```

engineer.Initial:0x397000 - MULTI Debugger
File Debug View Browse Target TimeMachine Tools Config Windows Help
Target Go on Selected Items (F5)
kernel Initial Zombied
Idle Running
RunModePartner1 Pended
ResourceManager Pended
LoaderTask Pended
MULTIloadagent HostIO
C:\ghs\int102\bin\Not Loaded
philosopher / Ini Not Loaded
producer / Initis Not Loaded
C:\ghs\int102\bin\Loaded
phonecompany Initial Created
information Initial Created
pizzahut Initial Created
engineer Initial Created
    
```

```

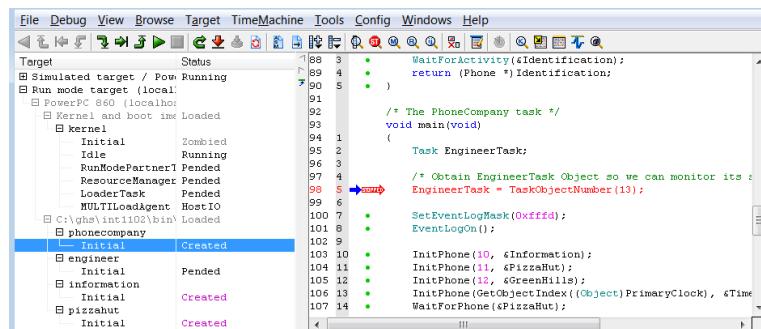
26
27
28
29
30
31
32
33
34
35 C = Dial(Telephone, "411");
36
37
38
39
40
41
42
43
44
45
46
47
        
```

5. Remove the breakpoint on line 35 and click **Next**.

The call to the Dial() function will not return because the Engineer task is Pended waiting for the PhoneCompany task to establish a phone connection with the Information task.

2.5.2.2 PhoneCompany Task

1. In the MULTI Debugger Target list, select the Initial task in the **phonecompany** AddressSpace.



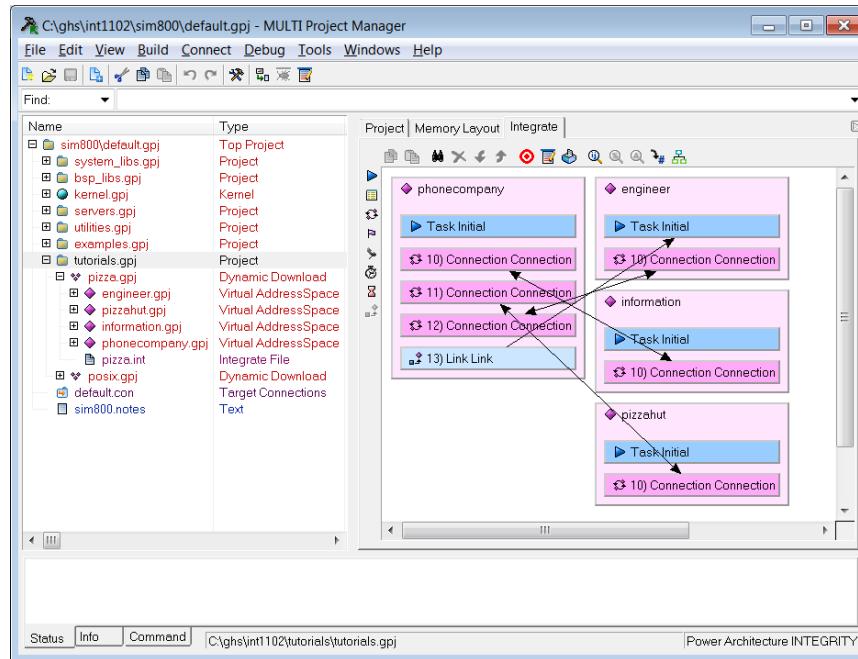
```

File Debug View Browse Target TimeMachine Tools Config Windows Help
Target Status
Simulated target / Pow Running
Run mode target (local)
PowerPC 860 (localhost)
Kernel and boot img Loaded
Initial Idle Running
ResourceManager Pended
LoaderCode Pended
MemoryLoadAgent HostIO
C:\ghs\int1102\bin\PhoneCompany
PhoneCompany Initial Created
engineer Initial Pended
information Initial Created
pizzahut Initial Created
WaitForActivity(Identification);
return (Phone *) Identification;
}
/* The PhoneCompany task */
void main(void)
{
    Task EngineerTask;
    /* Obtain EngineerTask Object so we can monitor its :
     * EngineerTask = TaskObjectNumber(10);
    */
    SetEventLogMask(0xffff);
    EventLogOn();
}
InitPhone(10, &Information);
InitPhone(11, &PizzaHut);
InitPhone(12, &GreenHills);
InitPhone(GetObjectIndex((Object)PrimaryClock), &Time
WaitForPhone(&PizzaHut);
}

```

The MULTI Debugger Source Pane will now display the main() function for the PhoneCompany task. At startup, the PhoneCompany task has Connections to all other AddressSpaces. In turn, the other AddressSpaces have Connections to the PhoneCompany. These Connections were set up by **pizza.int**, the Integrate configuration file used for the pizza application.

2. To see the various INTEGRITY Connections that have been specified, select **pizza.gpj** in the MULTI Project Manager and select the **Integrate** tab in the right-pane. Or, right-click **pizza.int** and select **Graphically Edit** for the full Integrate GUI.



By including **pizza.int** in the **pizza.gpj** project, the MULTI Project Manager builds the application in configuration file mode, using **pizza.int** to specify the initial state of the system. For more information, see the “Integrate Configuration File” section in the “Building INTEGRITY Applications” chapter of this manual.

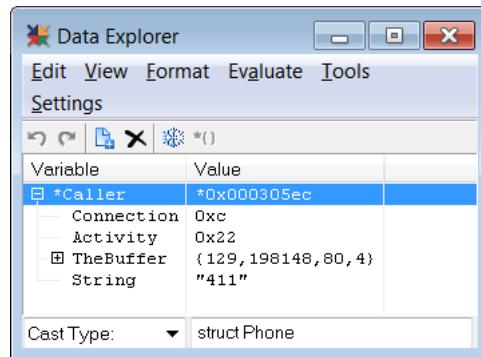
3. Set a breakpoint on line 143, then click **Go**.

```

File Debug View Browse Target TimeMachine Tools Config Windows Help
Target Go on Selected Items (F5) 128 36
Simulated target / Powr running 130 37
  Simulated target (local: 131 38
    PowerPC600 (localhost: 132 39
      Board Boot Loader 133 40
        kernel 134 41
          Initial Zombie 135 42
          Idle Running 136 43
          RunModePartnerT Pended 137 44
          ResourceManager Pended 138 45
          LeaderTask Pended 139 46
          MULTILoadAgent HostIO 140 47
          C:\ghs\int1102\bin\Loaded 141 48
        phonecompany 142 49
          Initial Created 143 50
            engineer 144 51
            pizzaHut 145 52
            information 146 53
          Initial Created 147 54
          Initial Created 148 55
  
```

The assembly code shows a series of task initializations and a conditional block starting at line 143. Line 143 contains a red dot indicating it is a breakpoint.

4. When the breakpoint is reached, double-click the variable named **Caller** on line 120. A Data Explorer will open displaying the contents of the structure pointed to by the **Caller** variable.



The last member of this structure, **String**, has the value "411". This is the first message passed from the Engineer to the PhoneCompany.

The operation of the PhoneCompany task between lines 143 and 146 provides an example of dynamic Connection creation. It creates Connections that enable the Engineer, Pizza Hut, and Information to speak to each other over the “phone”:

```

129 36
130 37
131 38
132 39
133 40
134 41
135 42
136 43
137 44
138 45
139 46
140 47
141 48
142 49
143 50
144 51
145 52
146 53
147 54
148 55
  
```

The assembly code shows the execution path through various tasks and their initializations. A red dot marks the breakpoint at line 143.

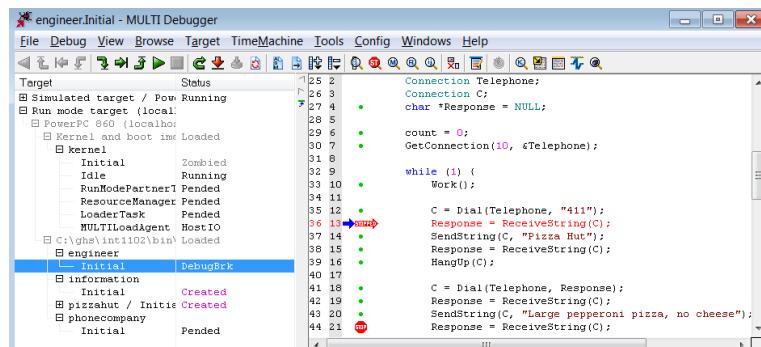
- On line 143, the PhoneCompany creates two new Connections.

- On line 144, the PhoneCompany passes one end of the Connection pair to the caller, in this case the Engineer.
 - On line 145, the PhoneCompany passes the other end of the Connection pair to the callee, in this case Information. Now the Engineer task and the Information task can communicate over this Connection.
5. Remove the breakpoint on line 143 by clicking the stop sign. The stop sign will go away, but the program counter arrow remains.
 6. Close the Caller Data Explorer.
 7. In the Target list, select the Initial task in the **phonecompany** AddressSpace and select **File⇒Detach from Process** to detach the PhoneCompany task.

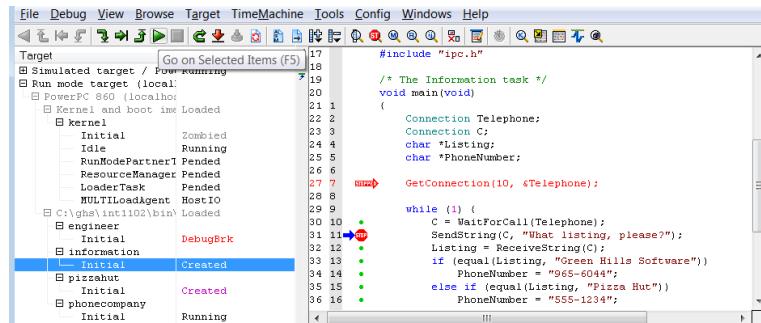
Detaching a task sets it running. The PhoneCompany will continue to handle Connections as the demo proceeds. The PhoneCompany task can be reattached and debugged by selecting the entry in the Target list again, but the remainder of the demo focuses on the other tasks.

2.5.2.3 Information Task

1. Go back to the Engineer task in the Target window. The Engineer should now be stopped after the Dial() call, and the Connection C, is one end of the Connection pair created by the PhoneCompany.



2. In the Target list, select the Initial task in the **information** AddressSpace to view it in the MULTI Debugger Source Pane.
3. Place a breakpoint on file line 31 (SendString(C, "What listing, please?");), then click **Go**.



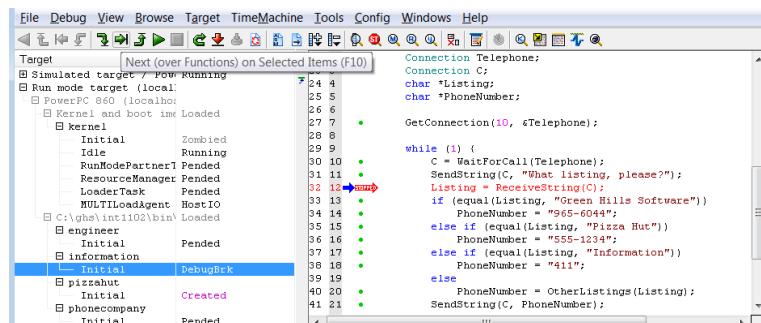
4. When the breakpoint is reached, click the stop sign to remove it.
5. Click **Next**. The Information task will not return from the `SendString` call(), as it is Pended waiting for the Engineer to communicate the desired listing.

Note that Information holds the other end of the Connection pair created by the PhoneCompany. Because the PhoneCompany did not create any Links to these Connections before it sent them off to the Engineer and Information Tasks, the PhoneCompany is now completely excluded from any messages sent across this Connection pair. Messages sent over these Connections are transmitted point-to-point, completely independent of the state of the PhoneCompany.

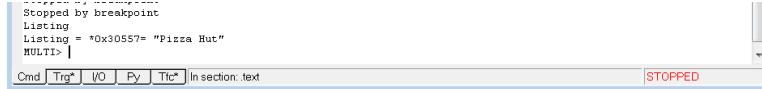
6. Go back to the **engineer** and click **Next** twice to step over the call to `SendString`() on line 37.

This call will complete when Information receives the message on the other end of the Telephone Connection.

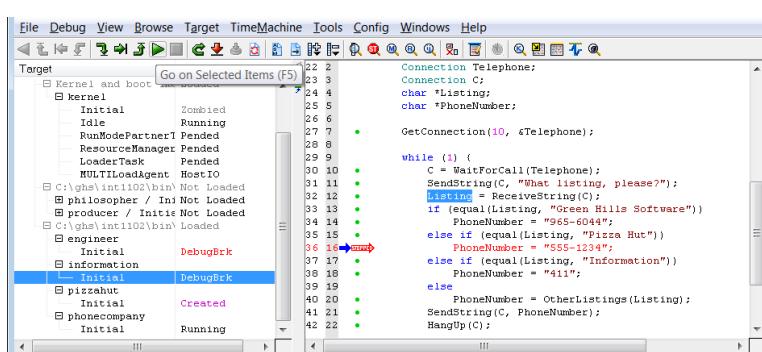
7. Go back to **information** and notice that the `SendString`() has completed, then click **Next** over the `ReceiveString`() call.



8. Click the word **Listing** on line 32. The value will be output to the Debugger command pane similar to the following:



9. Click **Next** until you are stopped at line 36. Information is about to communicate the requested PhoneNumber back to the Engineer.
10. Click **Go**.



The Information Task will continue to process phone number requests during the remainder of the demo.

11. Go back to the Engineer and click **Next** over the ReceiveString() call.

The Response is successfully returned from the Information task, and now the Engineer is ready to call Pizza Hut.

12. Click the word Response on line 36 to see the value output to the Debugger command pane, similar to the following:

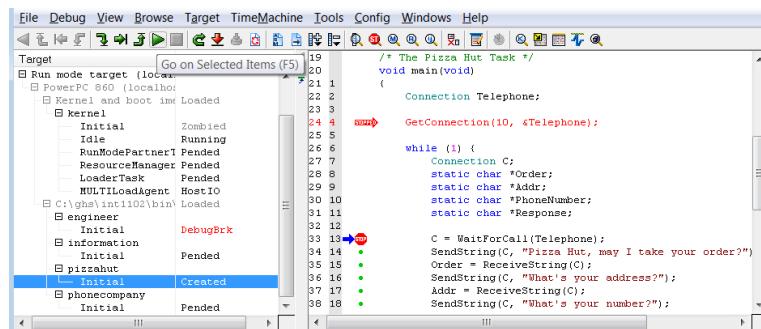


2.5.2.4 PizzaHut Task

1. On the Engineer task, click **Next** twice to have the Engineer dial Pizza Hut.

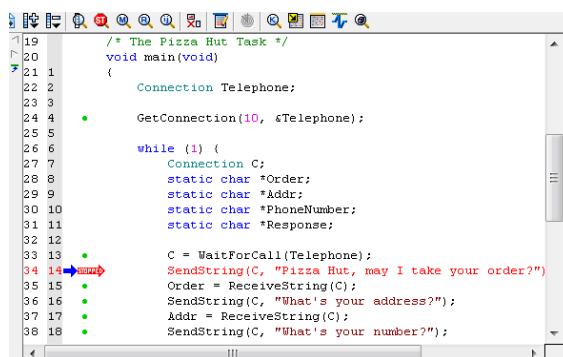
The ReceiveString() call will not return until the phone is answered by the PizzaHut task with the appropriate greeting.

2. Debug the PizzaHut task by selecting the Initial task of the **pizzahut** AddressSpace in the MULTI Debugger Target list and setting a breakpoint on file line 33 (C = WaitForCall(Telephone);). Then, click **Go**.



3. When the breakpoint is reached, click the stop sign to remove it and click **Next** over the `WaitForCall()` on line 33, answering the phone.
4. Click **Next** again over the `SendString()` call.

Pizza Hut is asking for the Engineer's order.



5. In the Engineer, click **Next** over the `SendString()` call on line 43.

The order for a large pepperoni pizza with no cheese is sent to Pizza Hut.

6. In PizzaHut, click **Next** over the `ReceiveString()` call.
7. Click **Order** to see the Engineer's order.

The PizzaHut task asks for the Engineer's address and phone number and calls the Engineer back.

8. Continue alternating **Next** operations on the PizzaHut task and the Engineer task.

Communication proceeds as PizzaHut verifies the Engineer's order.

2.5.2.5 Engineer Task

1. After PizzaHut has verified the Engineer's order, place a breakpoint in the EatPizza() function of the Engineer task (line 71).
2. Click **Go** to set the Engineer running.

```

59 36     C = WaitForCall(Telephone);
60 37     SendString(C, "Green Hills Software");
61 38     Response = ReceiveString(C);
62 39     if (equal(Response, "Did you order a pizza?"))
63 40         SendString(C, "Yes");
64 41         HangUp(C);
65 42     break;
66 43 }
67 44     HangUp(C);
68 45 }
69 46
70 47     WaitForPizza();
71 48     EatPizza();
72 49     SleepForABit();
73 50 }
74 51 }
75
76     void Work()
77 1
78 2     char buf[24];

```

3. Select the **pizzahut** Initial Task and click **Go** to set PizzaHut running.

When the Engineer's EatPizza() breakpoint is hit, we are near the end of the first cycle through the Pizza Demo.

4. In the Engineer task, click **Next**.

The Engineer eats the pizza (the **eaten** variable is incremented), and a message is displayed in the command pane similar to the following:

```

Stopped by breakpoint
I/O: YUMMY, I've eaten 1 pizza!
MULTI>

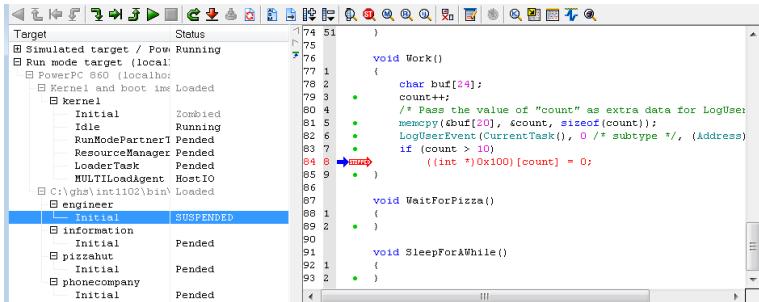
```

5. Continue the Engineer task until the EatPizza() breakpoint is again hit.

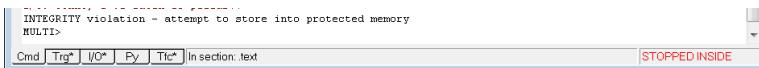
A second pass of the demo has occurred.

6. Remove the EatPizza() breakpoint, then click **Go**.

The Engineer should stop in the Work() function at line 84:



An intentional error has been inserted into the code. The erroneous fragment attempts to store to an address outside the Engineer's AddressSpace. INTEGRITY has successfully detected the violation and averted disaster. The following message is displayed in the command pane:



2.5.2.6 Close the Pizza Demo

When you are finished with the Pizza tutorial, unload **pizza** application from the target with one of the following:

- Right-click the **pizza** line in the Target list and select **Unload Module**.
- Select **Target⇒Unload Module** and select the **pizza** module to unload.
- Type `unload pizza` in the Target pane.

2.6 Target Board Setup

The instructions used in earlier sections of this chapter used an INTEGRITY Simulator (ISIM) to run the kernel on a simulated target. However, the process of loading your own application to an actual board is different than it is for ISIM.

To boot a kernel on a board, you may need to use a board-specific boot loader or ROM monitor, or you may be able to use a hardware debugger (for example, the Green Hills Probe) with an associated debug server (e.g., mpbserv). For information about booting an INTEGRITY kernel or Monolith on your board, see the ***bspname.notes*** file for the BSP you are using.

To boot a kernel on an x86 or x64 PC target, you must have already set up the target to run INTEGRITY as its operating system. For information about setting up an x86 or x64 PC target to run INTEGRITY, see the *INTEGRITY GILA Boot Loader Guide*.

To develop code for an INTEGRITY system on an actual board, you will need:

- An INTEGRITY installation present on the host computer being used for development.
- A board set up and ready for use as described in this section.

Before attempting communication with the board, see the “Network Configuration” chapter of the *INTEGRITY Networking Guide*.

A typical board setup includes:

- A serial connection to the board from the user’s host computer — the serial connection can be used to display RTOS diagnostics information and for Task debugging. Sometimes the serial connection is required in order to communicate with a board’s factory installed firmware (if any).
- An Ethernet connection to the board — Ethernet can be used for Task debugging (most BSPs support this functionality) and for TCP/IP communication (provided with some INTEGRITY installations). If Ethernet is used, the system administrator must provide the board with a valid IP address for the local network.
- Other Hardware — depending on the type of freeze-mode debugging solution for a particular board, some hardware setup may be necessary. For example, on the Freescale CDS 8548, the suggested freeze-mode debug solution is a Green Hills Probe that is connected to the board’s debug port. For more information, consult the “Freeze-Mode Debugging” chapter of this manual.
- BSP Specific Configuration — consult the ***bspname.notes*** file for the particular BSP you are using. This file contains important information about how to set up the board and load a kernel to it.
- x86 and x64 Specific Configuration — consult the *INTEGRITY GILA Boot Loader Guide* for additional information about setting up an x86 or x64 PC target.

Note: In the event that the board is either not yet available or in limited supply, ISIM enables programmers to develop and test embedded INTEGRITY-based applications without an actual board.

2.7 Booting an INTEGRITY Kernel

After a hardware connection has been established with the target board, an INTEGRITY kernel must be booted. The instructions used in earlier sections of this chapter used the MULTI Launcher **Run and Debug** action to boot a kernel on an INTEGRITY Simulator (ISIM). However, the process of booting a kernel on an actual board is different and depends on which board you are using, and whether the kernel was built for RAM or ROM.

- On some boards, you use factory-installed firmware and TFTP to boot the kernel over the network.
- On other boards, you boot the kernel through MULTI by using **mpserv** and a Green Hills Probe.
- On an x86 or x64 PC target, you use the GILA boot loader to boot the kernel.
- Other boards and/or custom designs may use entirely different booting methods.

Consult the ***bspname.notes*** file for the particular reference BSP you are using. This file often contains important information about how to set up the board and boot a kernel on it.

Each reference BSP provided in the INTEGRITY Installation includes a default kernel. You can either boot the default kernel into RAM on the target board, or you can build a custom kernel. For more information, consult the “Building INTEGRITY Applications” chapter of this manual.

The default kernel is located in the **bin\bspname** directory. For example, if the INTEGRITY Installation is located in the **c:\ghs\intversion** directory, then the default kernel for the ibm440 BSP would be **c:\ghs\intversion\bin\ibm440\kernel**. For more information, see the “INTEGRITY Installation and Directory Structure” chapter of this manual.

If you use a loader that only works with files in a binary image (**.bin**) format, the Green Hills **elfloader** utility allows you to use ELF images with these loaders. Loaders that work only with files in the **.bin** format (such as PPCBug) can be wasteful because the **.bin** format includes a full section of zeros for any **bss** sections that precede the final section of the image. The **bss** sections of an ELF format file take up no space, and as a result, ELF produces smaller executables. For more information, see “elfloader” in the “Alternate Download Methods: elfloader and bootloader” chapter of the *INTEGRITY Libraries and Utilities User’s Guide*.

Chapter 3

INTEGRITY Installation and Directory Structure

This chapter provides information about the different types of INTEGRITY installations, the INTEGRITY installation directory structure, and important directories and files.

This chapter contains the following sections:

- INTEGRITY Directory Structure
- INTEGRITY Include Directory
- target Directory
- BSP Source Directory
- libs Directory
- bin Directory

There are three different types of INTEGRITY installations:

Binary Installation

A binary installation contains object files, executables, and libraries pre-built for a particular BSP. With a binary installation alone, users can build and run INTEGRITY applications for only the reference BSPs that are shipped by Green Hills. Binaries for these reference BSPs are built with debugging enabled, console messages enabled, and compiler optimizations enabled. This provides a good mix of optimized performance and debugging capability.

Standard Source Installation

A standard source installation is a superset of the binary installation. In addition to the binaries required to build applications for reference BSPs, it also includes the source files for the reference BSPs so you can modify the BSP or port to another board of an already supported processor family. All of the code for the drivers and board specific support are included.

Kernel Source Installation

The kernel source installation is a superset of the standard source installation. The kernel source installation also includes the source files for the entire INTEGRITY kernel and layered system software such as the INDRT2 debug agent. In addition to the binaries and BSP source files for the INTEGRITY reference BSPs, a kernel source installation also includes source files for INTEGRITY ASPs. However, a kernel source installation does not contain source code to separately licensed source components such as the file system or the INTEGRITY target shell.

ASP stands for Architecture Support Package. The ASP contains support for processor-specific items, such as caches, interrupts, and memory protection. The PowerPC 750 is one example of a supported INTEGRITY ASP.

BSP stands for Board Support Package. The BSP contains everything needed to support a particular board running INTEGRITY. The BSP also contains support for such board-level functionality as timers, Ethernet, serial, and other devices. The name of the BSP is used as part of many directories and filenames to distinguish them from other BSPs and to point out that the files are BSP-specific. The term *bspname* is used throughout the INTEGRITY document set to refer to your selected BSP and the BSP source and build directories.

An INTEGRITY installation can contain many other optional components that are licensed separate from the kernel. Examples of such components include the file system, the INTEGRITY shell, and the GHnet v2 TCP/IP stack. These components can be licensed and used with standard source or kernel source installations, and in most cases, with binary installations also. For more information about separately licensed source components, see “Installing Encrypted and Other Optional Components” in the *INTEGRITY Installation Guide*.

3.1 INTEGRITY Directory Structure

This section provides information about the INTEGRITY installation directory structure, in terms of directories, files, and programs used for creating and building INTEGRITY applications.

An INTEGRITY installation consists of a base binary installation, and any number of source and licensed binary packages installed on top of the base installation. All INTEGRITY installations have the same overall structure, but some files and directories will be missing unless the corresponding source code packages have been installed.

The “Top Level” or “root” of the INTEGRITY installation is assumed to be the directory: **c:\ghs\intversion**. All descriptions of the installation structure and directories are in reference to this top level directory.

The top level directory contains a number of subdirectories. The following are some of the important subdirectories:

- **asp** — contains support for processor-specific items.
- **bspname** — source directory for a specific BSP, this directory will actually be named to indicate the installed BSP. See “BSP Source Directory” for more information.
- **bin** — contains BSP-specific subdirectories that are used to store the files generated as a result of building a particular BSP. See “bin Directory” for more information.
- **INTEGRITY-include** — contains important include files. See “INTEGRITY Include Directory” for more information.
- **intlib** — language-dependent run-time library support
- **kernel** — contains files used to build the INTEGRITY kernel.
- **libs** — contains subdirectories for built libraries. See “libs Directory” for more information.
- **target** — contains CPU-specific files, default BSP description files and options for the BSP, and subdirectories with New Project Wizard files. The **target\libs** directory contains files used to build libraries. See “target Directory” for more information.

Additional directories exist depending on which source packages have been installed. Documentation for these directories is available in comments in the source files themselves.

Standard source installations have a **modules\ghs\bpsrc** directory which contains board-specific source files that are modified when porting to a new BSP. For detailed information, see the *BSP User’s Guide*.

INTEGRITY.ld is one of the important files in the top level INTEGRITY directory. This is the default linker directives file used by the MULTI Project Manager when creating a virtual AddressSpace. INTEGRITY’s virtual AddressSpaces have true virtual addressing, so the specific locations of code and data in the virtual AddressSpace can usually be the same across different target processors and boards. As a result, **INTEGRITY.ld** resides in the top level INTEGRITY directory, rather than in each BSP’s build directory. For more information about **INTEGRITY.ld**, see the “Virtual AddressSpace Project” and “Linker Directives File” sections in the “Building INTEGRITY Applications” chapter of this manual.

3.2 INTEGRITY Include Directory

The **INTEGRITY-include** directory is a subdirectory of the top level directory. The **INTEGRITY-include** directory contains the file **INTEGRITY.h**. User programs that call the INTEGRITY API directly or use INTEGRITY types should include this file as follows:

```
#include <INTEGRITY.h>
```

The **INTEGRITY-include** directory also contains other system header files used by programmers. Important examples are:

- **<INTEGRITY_version.h>** — for INTEGRITY version constants
- **<dirent.h>** — for directory entry constants and data structures
- **<fcntl.h>** — for open flag definitions
- **<pthread.h>** — for POSIX pthread support
- **<unistd.h>** — for common file system related constants
- **<sys\mnttab.h>** — for mount table constants
- **<sys\socket.h>** — for BSD-compatible sockets support
- **<sys\stat.h>** — for constants related to the stat file system call
- **<sys\time.h>** — for struct timeval definition and related function prototypes

3.3 target Directory

The MULTI Project Manager uses a build target to set the target CPU type and the **bspname**, which are used to find libraries, linker directives files, and default BSP description files appropriate for the reference BSP. The target build file also contains important options for that BSP. The target build file is specified by opening the appropriate **default.gpj** from your installation directory, selecting the appropriate Workspace in the MULTI Launcher, or with the **-bsp** driver option. When using MULTI, make sure the lower right corner of the MULTI Project Manager displays the correct target.

The **target** directory contains several levels of target build files:

- The per-BSP file, named **bspname.bld**.
- A CPU option file, which is included in **bspname.bld** (for example **intppc750.bld**). This file may include other increasingly general architecture option files.
- The architecture option file, named **integrity.bld**, which has options common to all INTEGRITY targets.

For more information about build targets, see *MULTI: Building Applications* for your target.

3.3.1 Common Library Source Directories

target\libs contains CPU-specific subdirectories containing project files used to build libraries and common link map files which are used across multiple BSPs. For example, some PowerPC common library directories are:

- **ibm-ppc440x5** — contains libraries that are automatically used by the MULTI Project Manager when building programs for the processors with the PowerPC 440x5 core.
- **ibm-ppc440x5-fp** — contains libraries that are used for processors with the PowerPC 440x5 core that support hardware floating point.
- **fsl-e600** — contains libraries used for the processors with the e600 core, and MPC74xx processors.
- **ghs-isimppc** — contains libraries used for the INTEGRITY Simulator for PowerPC.

The MULTI Project Manager automatically uses the appropriate library directory depending on the CPU type that is used to build the application.

Sometimes you may need to locate the common library source directory for a given BSP, for example, in order to examine the common link map files. To do this, you need to view the BSP's options files, which are located in the **target** directory. Each BSP's **default.gpj** file references an **.opt** file. This file will either:

- Contain a line with the following:
`:sourceDir=$(__OS_DIR)\target\libs\common_library_dir.`

For example, **sim800\default.gpj** references **sim800.opt**, which lists
`$(__OS_DIR)\target\libs\ghs-isimppc` as the common library source directory.

- Include another **.opt** file, which in turn will list the directory.

For example, **ibm440\default.gpj** does not list a **sourceDir** in its immediate **ibm440.opt** file, but it includes **intppc440gp.opt**, which lists
`$(__OS_DIR)\target\libs\ibm-ppc440x4` as the common library directory.

3.4 BSP Source Directory

BSP stands for Board Support Package. The name of the BSP is used as part of many directories and filenames to distinguish them from other BSPs and to point out that the files are BSP-specific. The term **bspname** is used throughout the INTEGRITY document set to refer to your selected BSP.

The BSP source directory is used to build the kernel and other programs. It is a subdirectory of the top-level directory, and is referred to as the **bspname** directory throughout INTEGRITY documentation. Some examples of supported **bspnames** include:

- **omap35x-logicpd** — LogicPD Zoom OMAP35x Development Kit (ARM Cortex A8)
- **p4080ds** Freescale P4080 Development System
- **pcx86** — PC x86-based board (x86)
- **sim800** — INTEGRITY Simulator for PowerPC systems

The BSP source directory (*bspname*) contains a variety of files that are used during the development process. The complete list depends on the BSP, but some of the important files are:

default.gpj

The Top Project in the *bspname* directory is **default.gpj**. The results of building **default.gpj** are placed in **bin\bspname**, **libs\bspname**, and **bin\commonlib**. For more information, see “**default.gpj**” later in this chapter.

default.ld

The default linker directives file used when linking an INTEGRITY KernelSpace program (unless it has been explicitly overridden). Each **default.ld** is written for a particular board’s memory map and is the file that typically needs to be modified for a new BSP. This file will typically #include a CPU-specific common linker directives file **ram_sections.ld**, which contains all of the common linker sections and is located in the Common Library Source directory corresponding to a given BSP. In general, **default.ld** is not modified for an existing reference BSP, instead a new **default.ld** is created for each new BSP port due to differences in board memory maps. For more information see “Linker Directives File” in the “Building INTEGRITY Applications” chapter.

flash.ld

Similar to **default.ld**, except that the section map is appropriate for booting the kernel out of flash or ROM. This file will typically #include a CPU-specific common linker directives files **ram_sections.ld** and **rom_sections.ld**, which contain all of the common linker sections and are located in the Common Library Source directory corresponding to a given BSP. For some BSPs, MULTI’s built-in flash burning capability can be used to burn the program built with **flash.ld** into flash. For other BSPs, the factory-installed firmware has an ASCII command language that can be used to download and burn the kernel into flash. See the BSP’s **.notes** and **flash.ld** files for additional information. Your BSP will not have this file if booting from flash is not supported or possible. For more information see “Linker Directives File” in the “Building INTEGRITY Applications” chapter

libbsp.gpj

The subproject that contains the BSP source code specific to the reference BSP. A new **libbsp.gpj** is created for each new BSP. This subproject is located in the **bsp_libs.gpj⇒/bsp.gpj** project.

default.bsp

The default BSP description file used by the Integrate utility. This file specifies, among other things, the range of physical memory used for virtual memory mappings. There is a

default.bsp created for each BSP. Even though Integrate allows it, a BSP description file should not generally be specified within an Integrate configuration file because the MULTI Project Manager will pass the name of the appropriate BSP description file to Integrate as an option. A BSP description file can be customized by the user and added to an INTEGRITY application project. If it is added to the project, the MULTI Project Manager will use it for Integrate instead of **default.bsp**. For more information about the creation and use of Integrate configuration files and BSP description files, see the *Integrate User's Guide*.

bspname.notes

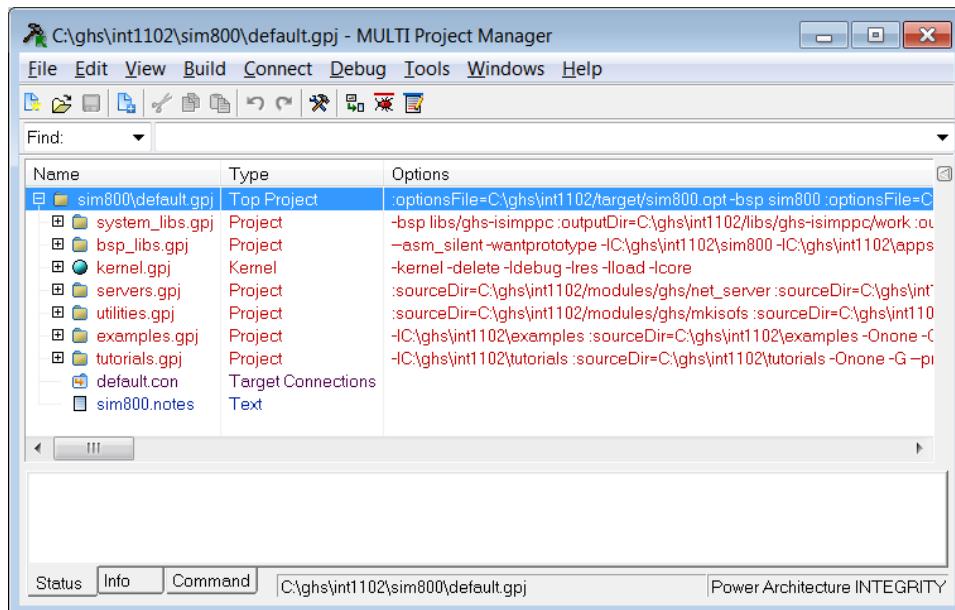
BSP-specific release notes. This file contains information specific to the BSP, including information about INTEGRITY Clocks and IODevices that are supported by the BSP, supported drivers, Interrupt Vectors, instructions for booting the kernel and flashing the image, and other board configuration details and nuances about a particular BSP port.

bspname.mbs

(or **bspname.dbs**) A setup script that is run prior to every download to ensure that your target is clean, stable, and properly configured. Newer target setup script files have **.mbs** extensions and use MULTI commands and scripting conventions. Older target setup scripts have **.dbs** extensions. The **bspname.notes** file explains which debugging solutions are supported for each target. If a debugging solution is not supported, a script may not be present. You can use the **Target Setup Script** field of the Connection Editor to create a connection method that will automatically run the script. For more information, see “Connecting to rtserver2” later in this book.

3.4.1 **default.gpj**

The Top Project in the **bspname** directory is **default.gpj**. The contents of **default.gpj** vary depending on the BSP. The following projects are typically seen from the **default.gpj** Top Project level. The resulting executables will be located in: **bin\bspname** (with exceptions noted below).



system_libs.gpj

Contains subprojects used to build the INTEGRITY system libraries. The resulting libraries will be located in: **libs\commonlib**

bsp_libs.gpj

Contains subprojects used to build the INTEGRITY libraries dependent on your specific BSP. The resulting libraries will be located in: **libs\bspname**

kernel.gpj

An example of a KernelSpace project. The **kernel.gpj** project can be used to create an INTEGRITY KernelSpace program. It links in libraries created from **system_libs.gpj** and **bsp_libs.gpj**. This kernel can be downloaded and run in RAM on the reference board. This KernelSpace program is used as the default kernel by the Integrate utility when creating an INTEGRITY application consisting of a combination of this kernel and user-created virtual AddressSpaces. **kernel.gpj** contains one constituent file: **global_table.c**, which links in libraries that add additional system software to the kernel, such as:

- **Debugging library (libdebug.a)**
- **ResourceManager library (libres.a)**
- **Dynamic Load library (libload.a)**
- **Core File Collection library (libcore.a)**

Note: All the libraries used in **kernel.gpj** are not necessarily desirable for a custom KernelSpace project, and other libraries can be added.

servers.gpj

Contains servers that can be downloaded as stand-alone INTEGRITY applications or used as programs in larger INTEGRITY applications.

utilities.gpj

Contains stand-alone INTEGRITY applications that can be downloaded to perform utility functions, such as formatting the file system.

examples.gpj

Contains small pieces of sample code that demonstrate the use of the INTEGRITY API and other INTEGRITY concepts such as the Integrate configuration file. It is helpful to run through these examples to get a feel for developing INTEGRITY applications and how to accomplish some common real-time operating system functions in INTEGRITY. For documentation of these examples, see the comments in their source code and related **readme.txt** files.

tutorials.gpj

Contains example applications that can be downloaded and run in RAM on the target board. It is helpful to step through these tutorials to get a feel for the environment. For more information about these tutorials, see the “Using INTEGRITY Tutorials” section of this manual.

3.5 **libs** Directory

The **libs** directory contains pre-built libraries that ship with INTEGRITY or are the result of building libraries with MULTI. The **libs** directory contains subdirectories for Common Libraries (**libs\commonlib**), and BSP-specific subdirectories (**libs\bspname**). INTEGRITY attempts to use libraries in **libs\bspname**, first; if a particular library cannot be located in this directory, INTEGRITY uses the library from **libs\commonlib** instead.

The source files for Common Libraries are generally located in the **target\libs** directory. To locate the common library source directory for a given BSP, see the instructions in the “Common Library Source Directories” section.

3.6 **bin** Directory

The **bin** directory is not part of the INTEGRITY installation, instead it is created the first time you build a BSP. The **bin** directory contains BSP-specific subdirectories that are used to store the files generated as a result of building a particular BSP.

After building **default.gpj** for a particular BSP, the contents of the **bin** directory will include INTEGRITY applications, modules and Virtual AddressSpace projects.

Chapter 4

Using MULTI to Develop INTEGRITY Applications

MULTI is a complete Integrated Development Environment (IDE) designed especially for embedded systems engineers to assist them in compiling, optimizing, debugging, and editing embedded applications. This chapter provides an overview of the MULTI IDE and the tools available for developing INTEGRITY applications.

- MULTI Integrated Development Environment
- MULTI Tools for Finding Performance Bottlenecks

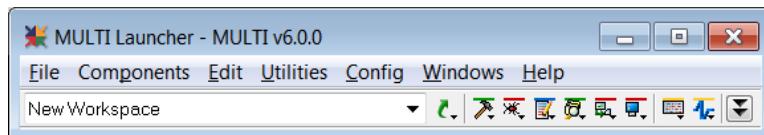
4.1 MULTI Integrated Development Environment

MULTI is installed separately from INTEGRITY. It is usually installed in **c:\ghs\multiversion** and contains the following components:

- MULTI Launcher (**multi**)
- MULTI Project Manager (**mprojmgr**)
- Freeze-Mode Debug Server (**mpserv**)
- Run-Mode Debug Server (**rtserv2**)
- INTEGRITY Simulators (**isim***)
- Compiler, Toolchain, and Object File Utilities

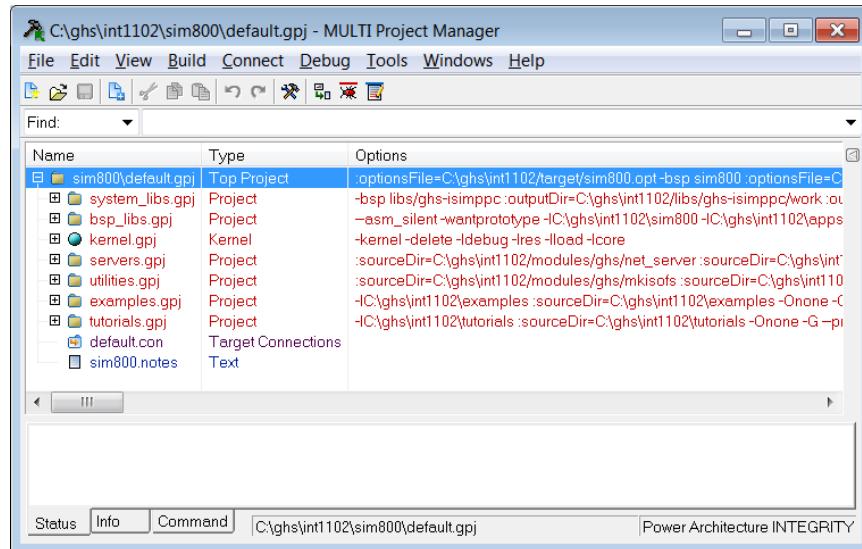
4.1.1 MULTI Launcher

The MULTI Launcher is the gateway to the MULTI IDE. From the Launcher, you can quickly launch any of the primary MULTI tools, create target connections, access open windows, and manage MULTI workspaces.

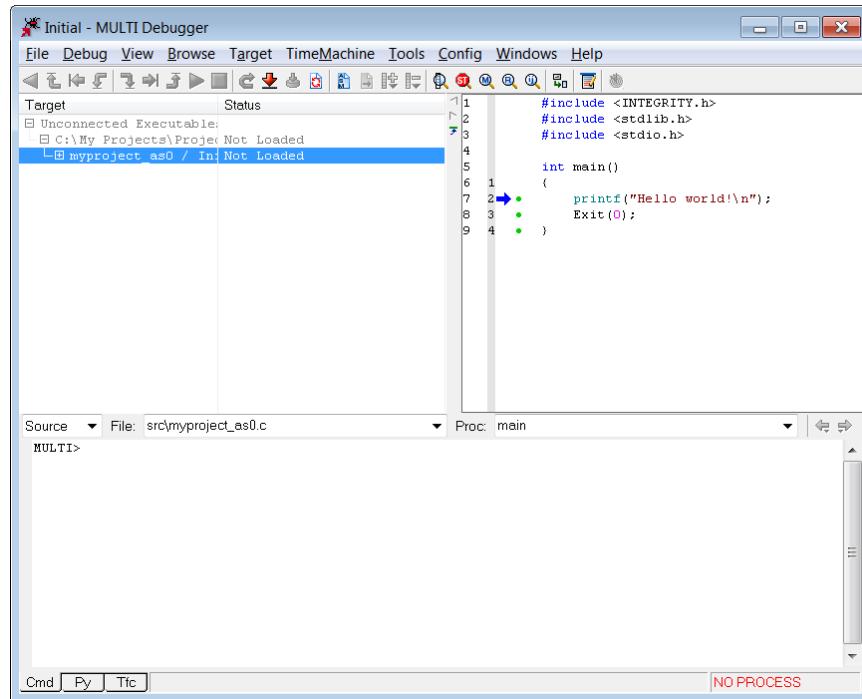


4.1.2 MULTI Project Manager

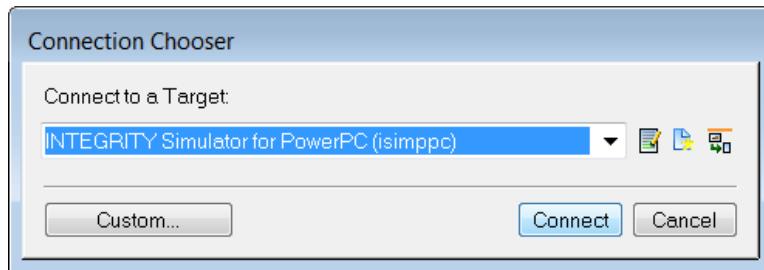
The MULTI Project Manager is a graphical interface used for managing and building kernel binaries and user applications.



The MULTI Debugger is a source-level debugger. You can open the Debugger from the MULTI Launcher or MULTI Project Manager.



The Connection Chooser is used to create connections to target systems. It is available by clicking the **Connect to Target** button in the MULTI Launcher, or the **Connect** button in the MULTI Project Manager or MULTI Debugger.



MULTI can be started with any of the following methods:

- Invoke the **mstart** executable.
- Click the MULTI icon on your desktop or Taskbar.
- Select MULTI in your **Start** menu.

For detailed information about using the MULTI IDE and its constituent parts, such as the MULTI Debugger, see the MULTI documentation.

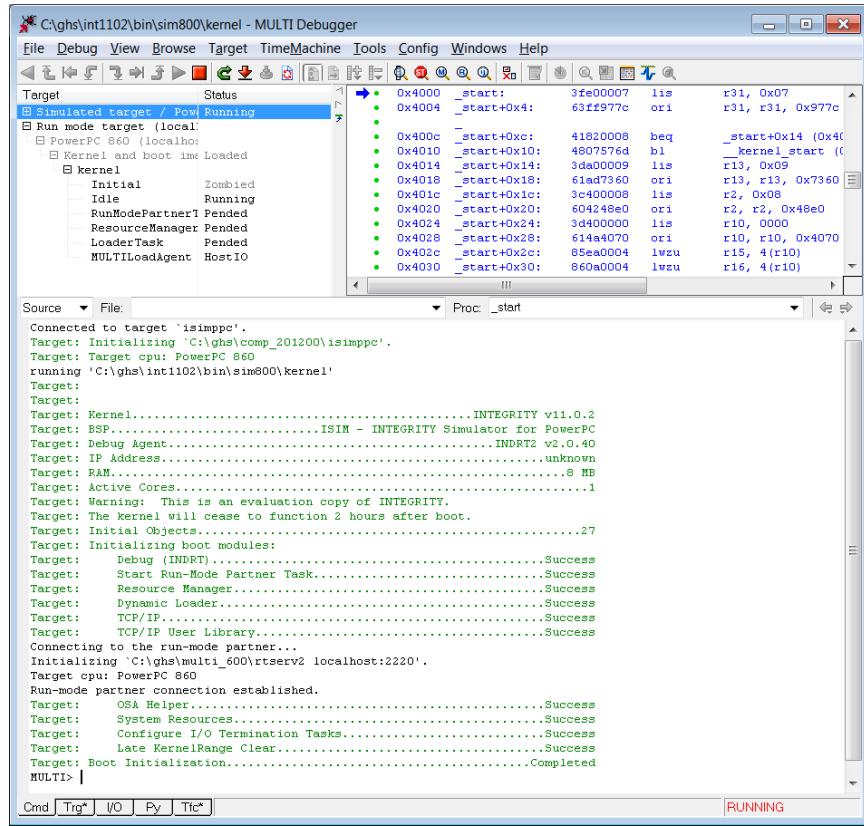
4.1.3 Freeze-Mode Debug Server

A MULTI freeze-mode debug server can be used for RAM downloading of an INTEGRITY system and freeze-mode debugging. Freeze-mode debugging can be used to debug an INTEGRITY kernel and low-level interrupt service routines.

mpserv is the preferred MULTI freeze-mode debug server for INTEGRITY when using a Green Hills Probe. **mpserv** connects to a Green Hills Probe, which communicates with the target board's on-chip debugging port. ISIM is the freeze-mode debug server for a simulated target. For more information, see the “Freeze-Mode Debugging” chapter of this manual.

4.1.4 Run-Mode Debug Server

The MULTI debug server, **rtserv2**, is used to connect to a running INTEGRITY system to perform run-mode debugging. rtserv2 displays all the tasks in the system in the MULTI Debugger Target list. Any user task can be selected and individually debugged with the MULTI Debugger.



rtsserv2 enables advanced features such as Dynamic Downloading of virtual AddressSpaces, and using the MULTI Profile window to view profiling data for an INTEGRITY kernel or individual user tasks.

For more information, see the “Connecting with INDRT2 (rtsserv2)”, “Run-Mode Debugging”, and “Profiling with rtsserv2” chapters of this manual.

4.1.5 INTEGRITY Simulator (ISIM)

ISIM is the generic name for the family of INTEGRITY simulators. The following executables are used to run ISIM for the following applications:

- **isimppc** — for PowerPC based applications
- **isimarm** — for ARM applications

For more information, see the “ISIM - INTEGRITY Simulator” chapter of this manual.

4.1.6 Compiler, Toolchain, and Object File Utilities

In addition to the host-based executables, the MULTI directory contains the Green Hills compilers, toolchain, include files, libraries, and utilities that are used to build kernel libraries and user applications. The MULTI Project Manager automatically invokes many of these programs during the build process. For detailed information, see *MULTI: Managing Projects and Configuring the IDE* and *MULTI: Building Applications* for your target.

4.2 MULTI Tools for Finding Performance Bottlenecks

MULTI provides several tools to help you find and eliminate performance bottlenecks in your system. These tools can be used separately, or in conjunction with each other. You can use these tools to track exactly where bottlenecks occur and identify sections of code that should be optimized or rewritten.

For more information about common problems, and suggested solutions, see the “Troubleshooting” chapter of this manual.

4.2.1 MULTI Profile Window

The MULTI Profile window can analyze performance on a per-function level within a selected AddressSpace. The MULTI Profile window can be used with INTEGRITY on either instrumented code, or on non-instrumented code (with the assistance of INDRT2). On instrumented code, the profiling provides an exact analysis of time expenditure and call chains in the application. On non-instrumented code, profiling performs Monte Carlo analysis of where time is being spent in the application.

For information about using the MULTI Profile window, see the “Profiling with rtserve2” chapter of this manual, and the “Collecting and Viewing Profiling Data” chapter of *MULTI: Debugging*.

4.2.2 ResourceAnalyzer

The ResourceAnalyzer is a good way to start analyzing system performance. The ResourceAnalyzer is a graphical tool that allows the user to dynamically examine how both CPU time and memory are allocated between the various Tasks and AddressSpaces in the system. This can help narrow down a performance problem to an individual Task in an individual AddressSpace.

For more information, see the “Using the MULTI ResourceAnalyzer” chapter in this manual.

4.2.3 MULTI EventAnalyzer

The EventAnalyzer is a graphical tool that can be used at the highest level to decipher how the various pieces of the system interact. It displays the Tasks in the system on the y-axis, time on the x-axis, and charts what happens in the system over time, including:

- Tasks that are running, ready, or pended, and the time the tasks entered their current status.
- Tasks that are making kernel calls, which kernel calls they are making, and the time the kernel calls are made.
- Hardware interrupts that are coming in, and the time they come in.

For information about the MULTI EventAnalyzer, see the *EventAnalyzer User’s Guide*.

4.2.4 Trace Data

With supported hardware, MULTI can collect and display instruction-by-instruction trace data of what is happening in the system over any given stretch of execution. For more information, see “Analyzing Trace Data with the TimeMachine Tool Suite” in *MULTI: Debugging*.

Chapter 5

Building INTEGRITY Applications

This chapter provides information about building INTEGRITY applications and covers the following topics:

- INTEGRITY Application Types
- Creating an INTEGRITY Top Project
- Monolith INTEGRITY Application Project
- Dynamic Download INTEGRITY Application Project
- Virtual AddressSpace Project
- KernelSpace Project
- Integrate Configuration File
- Linker Directives File
- Shared Libraries
- Additional Options for Building Applications

5.1 INTEGRITY Application Types

Note: Beginning with INTEGRITY 11.4, several kernel libraries have been deprecated and will not be supported in a future release. For future compatibility, these kernel libraries should be replaced with OS modules or Dynamic Downloads that can be added to INTEGRITY Monoliths or downloaded onto a system running the INTEGRITY kernel.

There are three types of INTEGRITY application projects:

- **Monolith INTEGRITY Application Project** — A monolith image is a stand-alone executable, suitable for running directly on a target, that contains a kernel and virtual AddressSpaces. Use a Monolith INTEGRITY Application Project to take advantage of the safety and security of INTEGRITY’s memory mapping for application code without requiring run-time download. For more information, see “Monolith INTEGRITY Application Project” later in this chapter.

- **Dynamic Download INTEGRITY Application Project** — A Dynamic Download image consists of one or more virtual AddressSpace projects that are downloaded via MULTI onto an INTEGRITY kernel already running on your target. Use a Dynamic Download INTEGRITY Application Project to bundle related virtual AddressSpaces without linking in a kernel project, so it can be downloaded independently. For more information, see “Dynamic Download INTEGRITY Application Project” later in this chapter.
- **KernelSpace Project** — A kernel is a stand-alone executable linked with INTEGRITY libraries suitable for running directly on a target. Use a KernelSpace project when only KernelSpace is needed, or when further virtual AddressSpaces may be downloaded at run time. For more information, see “KernelSpace Project” later in this chapter.

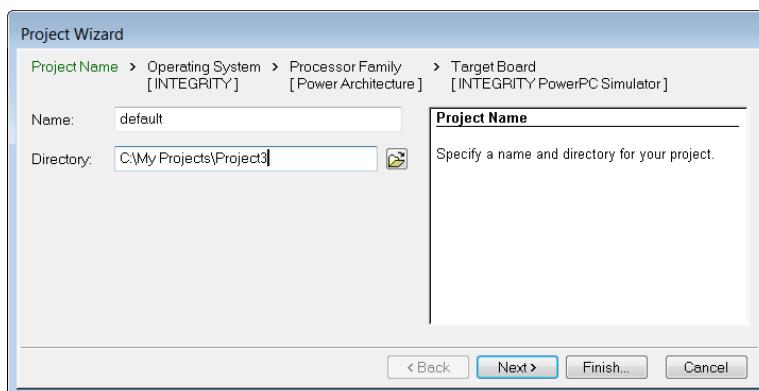
The remainder of this chapter provides details about all three of these types of projects, and instructions on how to create and build them.

5.2 Creating an INTEGRITY Top Project

Before you begin development, create an INTEGRITY Top Project using the New Project Wizard. A Top Project contains all of your code and defines any executables you plan on building.

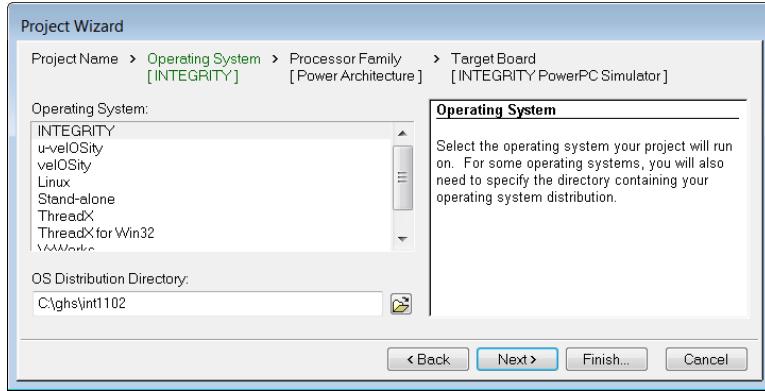
1. In the MULTI Launcher, click the **Launch Project Manager** button () and select **Create Project** from the drop-down, or in the MULTI Project Manager select **New Top Project**.

The first Project Wizard screen will open as shown below.



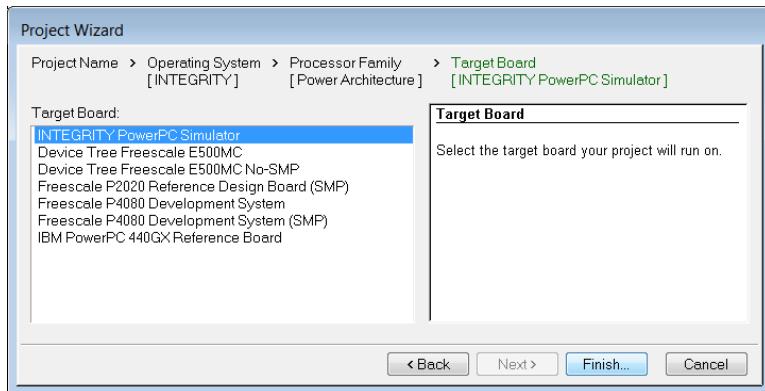
2. Specify the name of the Top Project and the directory where it will be created. This should either be an empty directory, or a new directory. When done, click **Next**.

The next Project Wizard screen will open as shown below.

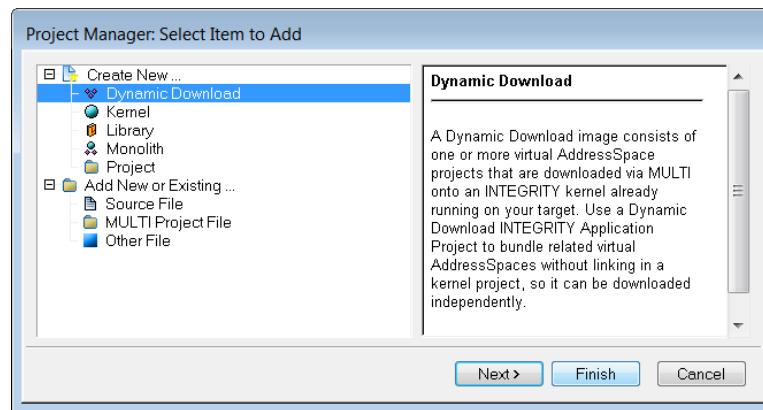


3. In the **Operating System** field, select **INTEGRITY**. Make sure the correct INTEGRITY install directory displays in the **OS Distribution Directory** field. When done, click **Next**.
4. Select the Processor Family for your target. When done, click **Next**. If you only have INTEGRITY installed for one processor family, this screen will be skipped.

The next Project Wizard screen will open as shown below.



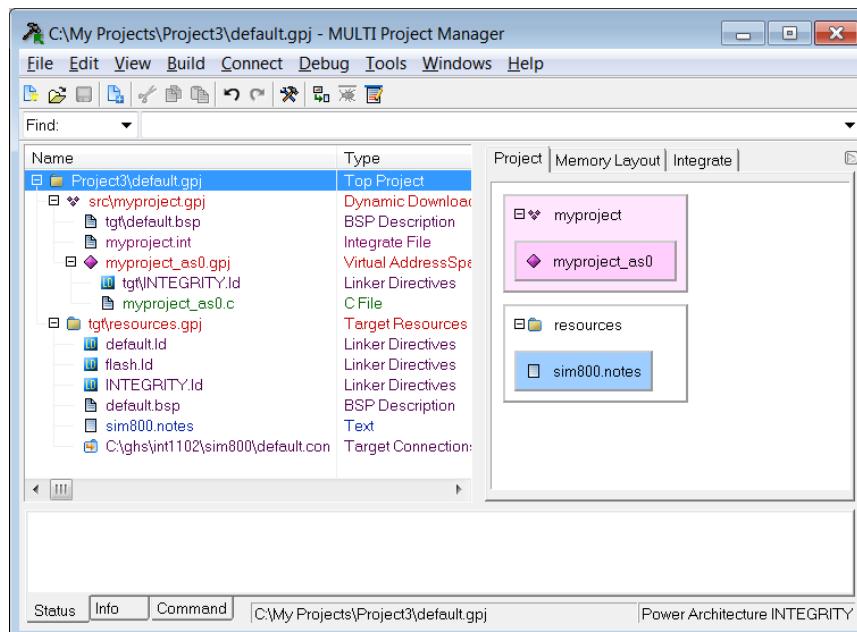
5. In the **Target Board** field, select the board for your target.
6. Click **Finish**. The MULTI Project Manager will open with your new project and the **Select Item to Add** screen will display as shown below.



7. Select an item to add. You can click **Finish** to use default settings and view the project in the MULTI Project Manager, or you can click **Next** for additional project settings.

5.2.1 Default INTEGRITY Top Project

When you use the New Project Wizard to create an INTEGRITY Top Project, it creates a project named **default.gpj** (or a name specified on the first screen of the Project Wizard) that has the selected build target and contains the following files:



- **tgt\resources.gpj** — Contains useful files associated with the selected BSP, such as linker directives files. Any projects created by using the New Project Wizard that are added to this INTEGRITY Top Project will reference these files by default.
- **default.con** — Target Connections file that contains sample connection methods for rtserver2 and any debug servers appropriate for the target. Note that this file is actually

located in the BSP Source Directory, so it may be preferable not to edit this file if the installation is shared.

- **default.ld** — Linker directives file used to link a kernel for RAM.
- **flash.ld** — Linker directives file used to link a kernel for ROM to RAM copy. Only available for certain BSPs.
- **INTEGRITY.ld** — Linker directives file used to link a virtual AddressSpace.
- **default.bsp** — BSP description file used to describe your BSP to the Integrate utility.
- ***bspname.notes*** — A copy of the BSP notes file from the INTEGRITY distribution. Contains useful information about the board.

5.3 Monolith INTEGRITY Application Project

A Monolith INTEGRITY application project creates an INTEGRITY system consisting of a KernelSpace project and a number of virtual AddressSpace projects. The result of building a Monolith INTEGRITY application project is an **ELF** executable.

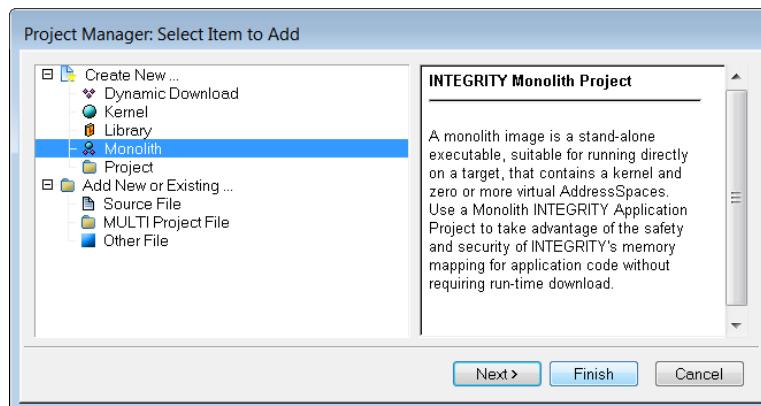
A Monolith INTEGRITY application project must have its project type set to **INTEGRITY Application**.

The recommended method for creating a Monolith is to use the MULTI Project Manager's New Project Wizard.

5.3.1 Creating a Monolith INTEGRITY Project with the New Project Wizard

To create a Monolith INTEGRITY Application Project using the New Project Wizard:

1. From the MULTI Launcher or MULTI Project Manager, create a new INTEGRITY Top Project. (For instructions, see “Creating an INTEGRITY Top Project”.)
2. On the **Select Item to Add** screen, select **Monolith**.



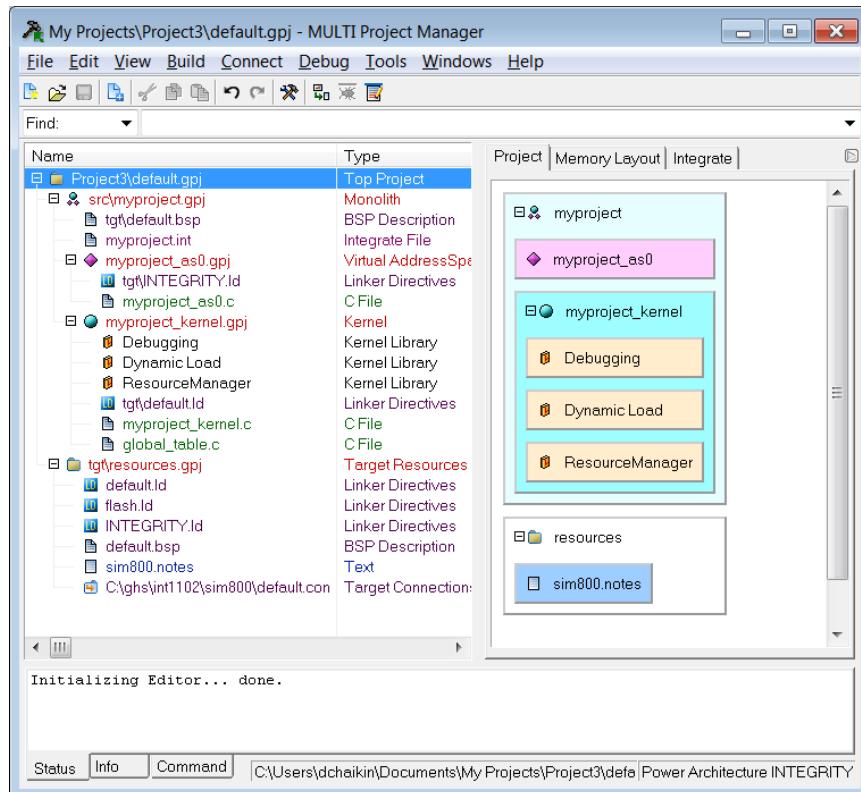
3. Click **Finish** to use default settings and view the project in the MULTI Project Manager. This creates a project named **myproject.gpj** in your Top Project.

Alternately, you can click **Next** to set additional configuration options. For complete information about configuring your project, see “Configuring Monolith Projects with the NPW” in the “Configuring Applications with the MULTI Project Manager” chapter.

You can use these same instructions to add a Monolith INTEGRITY Project to an existing project. In the MULTI Project Manager, select the item you want to add the Monolith Project to (generally the INTEGRITY Top Project) and click **Add Item** to open the **Select Item to Add** screen.

5.3.2 Default Monolith INTEGRITY Project

When you use the New Project Wizard to create a Monolith INTEGRITY Application project, it creates a project named **myproject.gpj** that has the selected build target and contains the following files:



- **src\myproject.gpj** — Monolith INTEGRITY Application Project. The name of the stand-alone image suitable for running on target is **myproject**. This build file contains:
 - **tgt\default.bsp** — A reference to the copy of the **default.bsp** from the specified BSP that comes from the resources file in the INTEGRITY Top Project. Customize as desired.
 - **myproject.int** — Integrate File describing the monolith application. Right-click to edit.
 - **myproject_as0.gpj** — Virtual AddressSpace Project (for more information, see “Virtual AddressSpace Project”). This build file contains:
 - * **myproject_as0.c** — Contains a main() entry point for user code. Customize as desired.
 - * **tgt\INTEGRITY.id** — A reference to the copy of the **INTEGRITY.id** from the specified BSP. This comes from the resources file in the INTEGRITY Top Project. Customize as desired.

- **myproject_kernel.gpj** — KernelSpace project. This build file contains:
 - * **global_table.c** — Contains INTEGRITY specific defines (see “global_table.c”).
 - * **myproject_kernel.c** — Contains a main() entry point for user code. Customize as desired.
 - * **tgt\default.ld** — A reference to the copy of the **default.ld** from the specified BSP. This comes from the resources file in the INTEGRITY Top Project. Customize as desired.

5.3.3 Building a Monolith INTEGRITY Application Project

To build a Monolith INTEGRITY application:

1. In the MULTI Project Manager, highlight **src\myproject.gpj**.
2. Click the **Build** () button.

When building the Monolith INTEGRITY application project, the MULTI Project Manager first builds all the constituent virtual AddressSpace projects into virtual AddressSpace programs (see “Virtual AddressSpace Project”), and the KernelSpace project into a KernelSpace program.

kernel is used as the default KernelSpace program if a KernelSpace project is not specified within the INTEGRITY application project.

In the final stage of the build, the MULTI Project Manager automatically invokes the Integrate back-end utility (**intex**). This creates the composite INTEGRITY application, which consists of the KernelSpace and the virtual AddressSpaces.

5.4 Dynamic Download INTEGRITY Application Project

A Dynamic Download INTEGRITY Application Project creates an INTEGRITY system consisting of a number of virtual AddressSpace projects. No kernel is included. The result of building a Dynamic Download INTEGRITY application project is an ELF executable that can be downloaded to a kernel that is already running on the board.

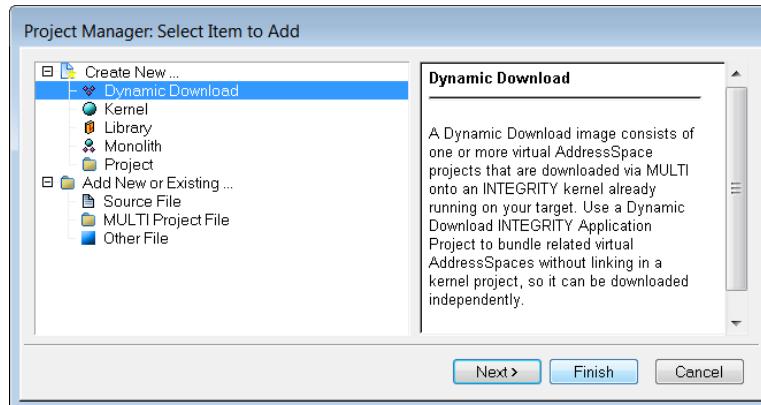
A Dynamic Download INTEGRITY application project must have its project type set to **INTEGRITY Application** and have the option **-dynamic** set.

The recommended method for creating a Dynamic Download INTEGRITY application project is to use the New Project Wizard in the MULTI Project Manager.

5.4.1 Creating a Dynamic Download Application with the New Project Wizard

To create a Dynamic Download Application with the New Project Wizard:

1. From the MULTI Launcher or MULTI Project Manager, create a new INTEGRITY Top Project. (For instructions, see “Creating an INTEGRITY Top Project”.)
2. On the **Select Item to Add** screen, select **Dynamic Download**.



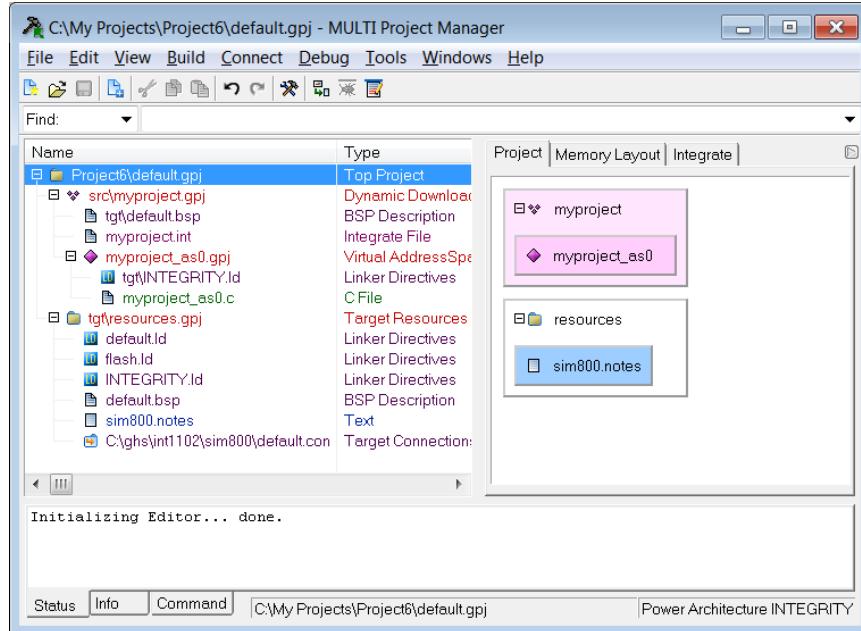
3. Click **Finish** to use default settings and view the project in the MULTI Project Manager. This creates a project named **myproject.gpj** in your Top Project.

Alternately, you can click **Next** to specify additional configuration options. For complete information about configuring your project, see “Configuring Dynamic Download Projects with the NPW” in the “Configuring Applications with the MULTI Project Manager” chapter.

You can use these same instructions to add a Dynamic Download application to an existing project. In the MULTI Project Manager, select the item you want to add the Dynamic Download to (generally the INTEGRITY Top Project) and click **Add Item** to open the **Select Item to Add** screen.

5.4.2 Default Dynamic Download INTEGRITY Project

When you use the New Project Wizard to create a Dynamic Download INTEGRITY Application project, it creates a project named **myproject.gpj** that contains the selected build target and the following files:



- **src\myproject.gpj** — Dynamic Download INTEGRITY Application Project. The name of the image suitable for downloading to a kernel that is already running on the board is **myproject**. This build file contains:
 - **tgt\default.bsp** — A reference to the copy of the **default.bsp** from the specified BSP that comes from the resources file in the INTEGRITY Top Project. Customize as desired.
 - **myproject.int** — Integrate File describing the Dynamic Download application. Right-click to edit.
 - **myproject_as0.gpj** — Virtual AddressSpace Project (for more information, see “Virtual AddressSpace Project”). This build file contains:
 - * **myproject_as0.c** — Contains a main() entry point for user code. Customize as desired.
 - * **tgt\INTEGRITY.id** — A reference to the copy of the **INTEGRITY.id** from the specified BSP. This comes from the resources file in the INTEGRITY Top Project. Customize as desired.

5.4.3 Building a Dynamic Download INTEGRITY Application Project

To build a Dynamic Download INTEGRITY application:

1. In the MULTI Project Manager, highlight **src\myproject.gpj**.
2. Click the **Build** () button.

When building the Dynamic Download INTEGRITY application project, the MULTI Project Manager first builds all the constituent virtual AddressSpace projects into virtual AddressSpace programs (see “Virtual AddressSpace Project” later in this chapter). In the final stage of the build, the MULTI Project Manager automatically invokes the Integrate back-end utility (**intex**). This creates the composite INTEGRITY application, which consists of the virtual AddressSpaces.

When you use the MULTI Project Manager to create the project, the **-dynamic** option is automatically enabled. This option is not necessary if an Integrate Configuration File (**.int**) is included in the project. If included, the Integrate configuration file of the INTEGRITY application project should specify the following Kernel/EndKernel directives:

```
Kernel
      Filename          DynamicDownload
EndKernel
```

(For information about specifying a Kernel/EndKernel directive, see the “Kernel Section” chapter of the *Integrate User’s Guide*.)

The dummy DynamicDownload name informs **intex** that the application is going to be downloaded dynamically to an already running INTEGRITY kernel and that there is actually no KernelSpace program being included in the application.

The program resulting from building the Dynamic Download project is appropriate for dynamic downloading with rtserv2.

5.5 Virtual AddressSpace Project

In the MULTI Project Manager, each virtual AddressSpace in an INTEGRITY application is specified by means of a virtual AddressSpace project.

A complete INTEGRITY application is a collection of virtual AddressSpaces, together with the kernel and its AddressSpace. In the MULTI Project Manager, a project of type **Program** that does not have the **-kernel** option set represents a virtual AddressSpace.

For more information, see “Adding and Configuring Virtual AddressSpaces with the Project Manager” in the “Configuring Applications with the MULTI Project Manager” chapter.

5.5.1 Virtual AddressSpace Program

The result of building a virtual AddressSpace project is a fully linked **ELF** executable, called a virtual AddressSpace program. For example, when you build the virtual AddressSpace project **aspace1.gpj**, the result is the virtual AddressSpace program **aspace1**. This program contains only the code and data for this single virtual AddressSpace, **aspace1**.

By default, the linker directives file **INTEGRITY.ld** is used by the MULTI Project Manager when linking a virtual AddressSpace program. **INTEGRITY.ld** ensures that the text and data segments start on page-aligned boundaries. If needed, **INTEGRITY.ld** can be overridden by user-created section maps by explicitly adding a different section map to the program project for the virtual AddressSpace. For more information, see “Linker Directives File” later in this chapter.

User code for a virtual AddressSpace program begins with the standard C/C++ function `main()`. When Ada is used, the Green Hills Ada run time begins in a function called `main()`, but user code follows the standard Ada model. The entry point of the program is actually `_start`, which initializes the virtual AddressSpace so that the C/C++ run-time environment and the INTEGRITY API can be used. `_start` transfers control to `main()`, the start of user code for each virtual AddressSpace.

The thread of control executing into `main()` is known as the Initial Task. The name of the Initial Task as seen in the MULTI Debugger Target list is **Initial**, grouped with any other tasks that may be in the same AddressSpace.

5.6 KernelSpace Project

Note: Beginning with INTEGRITY 11.4, several kernel libraries have been deprecated and will not be supported in a future release. For future compatibility, these kernel libraries should be replaced with OS modules or Dynamic Downloads that can be added to INTEGRITY Monoliths or downloaded onto a system running the INTEGRITY kernel.

A KernelSpace project is a stand-alone executable program that can be downloaded to RAM by loading from MULTI, or via network boot (if supported by your BSP or board monitor). If properly linked, the program can be booted directly from flash. **flash.ld** is an example of a linker directives file that is added to a KernelSpace project so that it will be built for flash.

KernelSpace is another name for the kernel's AddressSpace. **kernel.gpj** is an example of a KernelSpace project. The KernelSpace program resulting from building **kernel.gpj** is the program **kernel**. This program contains:

- the kernel
- the ASP
- the BSP
- INDRT2 (if **libdebug.a** is linked)
- other system tasks such as the optional LoaderTask for dynamic loads

The KernelSpace program may also contain user code that is executing in KernelSpace.

A KernelSpace program does not include any virtual AddressSpaces. When virtual AddressSpaces are included along with the kernel, the result is a Monolith INTEGRITY application, which is built from a Monolith INTEGRITY application project (see “Monolith INTEGRITY Application Project” earlier in this chapter).

In order to dynamically create tasks to execute in KernelSpace, you can add user code to the KernelSpace project. This code begins with the standard C/C++ function `main()`. When Ada is used, the Green Hills Ada runtime begins in a function called `main()`, but user code follows the standard Ada model. The KernelSpace Initial Task will transfer control to `main()` after some KernelSpace initialization has been completed.

Although performance-critical user code can be placed in KernelSpace, the preferred method is to place user code in a virtual AddressSpace. This way, the kernel is protected from malfunctions in the user code, and the user code is protected from malfunctions in other AddressSpaces.

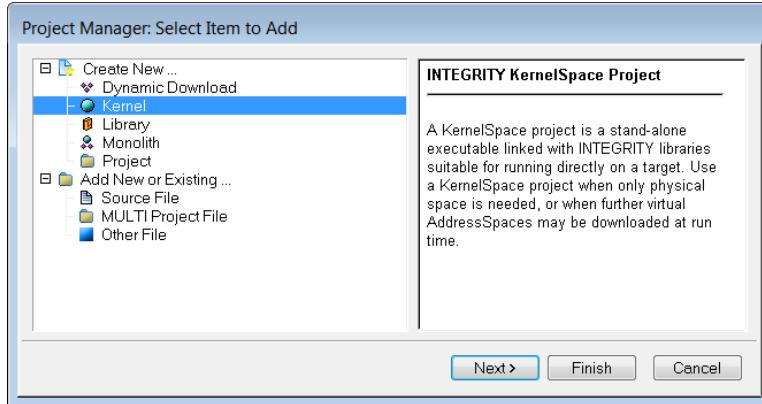
There are two ways to create a custom KernelSpace project. The recommended method is to use the New Project Wizard. The other method is to create custom KernelSpace projects from scratch. Details about both methods are provided in the following sections.

5.6.1 Creating a KernelSpace Project with the New Project Wizard

To create a KernelSpace Project using the New Project Wizard:

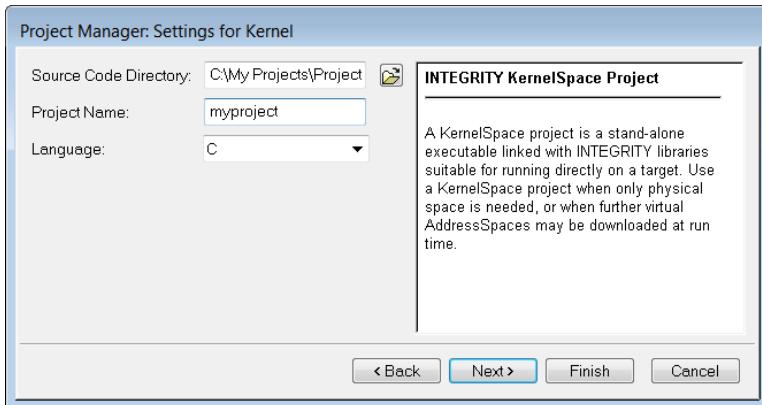
1. From the MULTI Launcher or MULTI Project Manager, create a new INTEGRITY Top Project. (For instructions, see “Creating an INTEGRITY Top Project”.)

2. On the **Select Item to Add** screen, select **Kernel**.

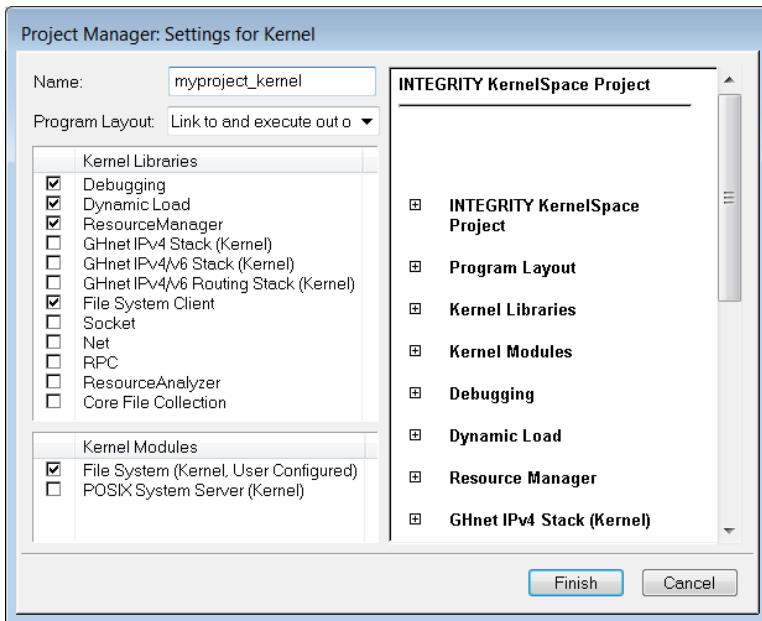


3. Click **Finish** to use default settings and view the project in the MULTI Project Manager. This creates a project named **myproject_kernel.gpj** in your Top Project.

- Alternately, you can click **Next** to specify a name and directory for the project.



- You can click **Next** again to add Kernel Libraries to your project.

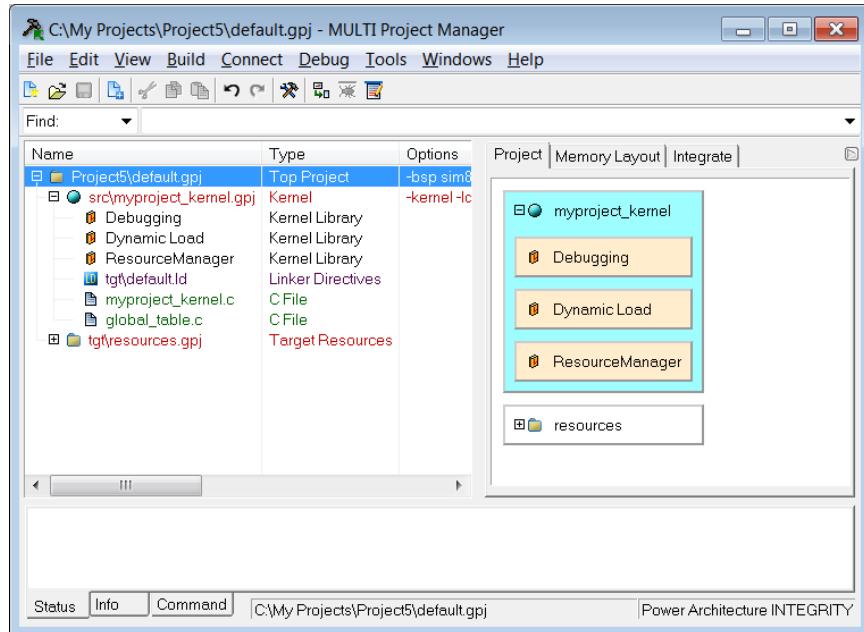


For complete information about configuring your project, see “Configuring KernelSpace Projects with the NPW” in the “Configuring Applications with the MULTI Project Manager” chapter.

You can use these same instructions to add a KernelSpace Project to an existing project. In the MULTI Project Manager, select the item you want to add the KernelSpace Project to (generally the INTEGRITY Top Project) and click **Add Item** to open the **Select Item to Add** screen.

5.6.2 Default KernelSpace Project

When you use the New Project Wizard to create a KernelSpace project, it creates a project named **myproject_kernel.gpj** that has the selected build target and contains the following files:



- **myproject_kernel.c** — Contains a main() entry point for user code. Customize as desired.
- **global_table.c** — Contains INTEGRITY specific defines (see the following section, “global_table.c”).
- **tgt\default.Id** — A reference to the **default.Id** from the specified BSP. Refers to the resources file in **tgt\resources.gpj** in the INTEGRITY Top Project. Customize as desired.

5.6.2.1 global_table.c

global_table.c is located in the **kernel** directory of the INTEGRITY Installation. Any KernelSpace project that will be combined with virtual AddressSpaces into a Monolith INTEGRITY application must include this source file because the Integrate utility stores the boot table header into data defined in this file. If virtual AddressSpaces will not be used **global_table.c** can be omitted and the BootTable code will not be included in the kernel.

This file also contains a data structure for setting network configuration. For more information, see “Setting Network Configuration in the KernelSpace Project” in the “Network Configuration” chapter of the *INTEGRITY Networking Guide*.

5.6.3 Building a KernelSpace Project

To build the KernelSpace project:

1. In the MULTI Project Manager, highlight **src\myproject_kernel.gpj**.
2. Click the **Build** () button.

5.6.4 Creating a KernelSpace Project from Scratch

To create a KernelSpace Project from scratch:

1. Bring up the MULTI Project Manager on an existing top-level build file with the target file set. For example:

```
cd INTEGRITY_Install_Directory/bspname; multi default.gpj
```

Alternately, you can browse your INTEGRITY install directory, open the *bspname* subdirectory, then open **default.gpj**.

2. Right-click the top-level build file and select **Add File Into**.
3. In the file chooser, type the kernel project name, for example: `myproject_kernel.gpj`.

Because the **Type** for **myproject_kernel.gpj** is already **Program**, it does not need to be modified.

4. Right-click **myproject_kernel.gpj** and select **Set Build Options**.
5. On the **All Options** tab, select **Target⇒Operating System**.
6. In the Build Options pane, right-click **Kernel Project**, and select **On**. This will set the **-kernel** option in the build file.
7. Close the Build Options window. You will be prompted to save your changes.

5.7 Integrate Configuration File

Integrate is a system design tool used to lay out the initial state of AddressSpaces, Tasks, and other kernel Objects. Integrate has two parts:

- **integrate** — a graphical front-end for creating Integrate configuration files.
- **intex** — a command-line utility that reads Integrate configuration files (either written manually, or created by the integrate front end) and constituent AddressSpace executables to create a final boot image that can be loaded onto the target board.

intex is used to combine user-created virtual AddressSpaces, and an INTEGRITY kernel (KernelSpace), into an INTEGRITY Monolith, which is a composite executable appropriate for downloading or booting. It is also used to combine user-created virtual AddressSpaces into an image that can be dynamically downloaded to a running INTEGRITY system. The MULTI Project Manager automatically invokes **intex** when building an INTEGRITY application project, although **intex** can also be run manually.

An Integrate configuration file enables you to use some of the more advanced features of Integrate. For example, you can specify that Objects such as Connections are automatically created during bootup (instead of being created at run time by user application code in the Initial Task, or by other tasks executing in the AddressSpace).

It is the system designer's responsibility to ensure that the Integrate configuration file describes the system accurately. The *Integrate User's Guide* provides details about creating configuration files, as well as some of the more advanced topics related to this kind of static system description.

For working examples of Integrate, see the **examples** and **tutorials** directories. For information about using Integrate, see the *Integrate User's Guide*.

Using the examples from earlier in this chapter, the Integrate configuration file **myapplication.int** would look like the following:

```
Kernel
    Filename      DynamicDownload
EndKernel

AddressSpace
    Filename      myproject_as0
    Language      C
EndAddressSpace
```

As an alternative to executing the integration from the MULTI Project Manager, you can run **integrate** from the command line. For information, see the “Alternative Modes of Operation” chapter in the *Integrate User's Guide*.

5.7.1 Object Number Usage

Integrate configuration files can be used to statically specify INTEGRITY Objects owned by an AddressSpace. For example, in order to create a Connection between two virtual AddressSpaces, a Connection Object for each AddressSpace must be specified:

```

AddressSpace      aspace1
    Filename        aspace1
    Language         C
    Object 10
        Connection      aspace2
        OtherObjectNumber 10
    EndObject
EndAddressSpace

AddressSpace      aspace2
    Filename        aspace2
    Language         C
    Object 10
        Connection      aspace1
        OtherObjectNumber 10
    EndObject
EndAddressSpace

```

In this example, when booted, AddressSpaces aspace1 and aspace2 will both have one end of a Connection to each other, thereby enabling communication across the two AddressSpaces.

The Object Number (10 in the example above) is used for subsequent INTEGRITY API calls that refer to Connections, for example:

```
SynchronousSend(ConnectionObjectNumber(10), ...);
```

- In a virtual AddressSpace, the minimum specifiable Object number is 10. Object indexes 1-9 are reserved. Attempting to specify an Object number in the 1-9 range will cause undefined results.
- For KernelSpace, user-defined Object numbers must be greater than the value specified for `InitialKernelObjects` in the BSP description file used for the INTEGRITY application. For example, if the BSP description file for the application contains the following:

```

...
InitialKernelObjects          18
...

```

A corresponding Integrate configuration file specifying Object numbers in KernelSpace must use numbers greater than 18:

```

Kernel
    Object 19
        Connection      aspace1
    EndObject
EndKernel
...

```

Attempting to specify an Object number between 1 and the `InitialKernelObjects` value will cause an error.

If the BSP has been customized, the `InitialKernelObjects` field may not be large enough for the BSP. Upon startup, the kernel banner output indicates how many Objects are created by the kernel. If an Object was specified between the **.bsp** file `InitialKernelObjects` number and the run time number from the banner, the boot up code will fail and display an error message indicating that you need to edit the **.bsp** file and rebuild the project.

5.8 Linker Directives File

default.Id and **INTEGRITY.Id** are the default linker directives files used when building an INTEGRITY application. These linker directives files contain comments that describe the various sections in the link map. **default.Id** is used for the kernel AddressSpace, and **INTEGRITY.Id** is used for virtual AddressSpaces.

You can specify alternate linker directives files by adding the name of a different file to the program project. For example, some INTEGRITY BSPs provide the file **flash.Id**, which is an alternate to **default.Id** appropriate for building a flash-based kernel.

ram_sections.Id and **rom_sections.Id** are common linker directives files. A set of these files typically exists for each supported CPU type and is included by the BSP's **default.Id** or **flash.Id** files. The files contain all of the common linker sections for a given CPU type and their contents rarely, if ever, need to be modified. These files reside in the Common Library Source directory corresponding to a given BSP. For more information see the "Common Library Source Directories" section.

ram_sections.Id contains all the common linker sections that will reside in RAM memory. This file is included by **default.Id** as well as **flash.Id** files.

rom_sections.Id is only used by **flash.Id** and similar files designed for building flash-based kernels. The file contains sections that go into ROM memory and is used in conjunction with the corresponding **ram_sections.Id** file. This file will only exist for CPUs that support booting from flash.

INTEGRITY.Id is the default linker directives file used for virtual AddressSpaces.

INTEGRITY.Id ensures that the text and data segments start on page-aligned boundaries. The **INTEGRITY.Id** located in the top level INTEGRITY directory should never be modified. When you use the MULTI Project Manager to create or modify an INTEGRITY project, a reference copy of **INTEGRITY.Id** is automatically placed in the project's **tgt** directory.

5.8.1 Creating Custom Linker Directives Files

At times, it may be necessary to modify or create a custom linker directives file for a particular project. For example, the size of the run-time heap may need to be increased or decreased.

To modify or create a new **default.Id** or **flash.Id** linker directives file:

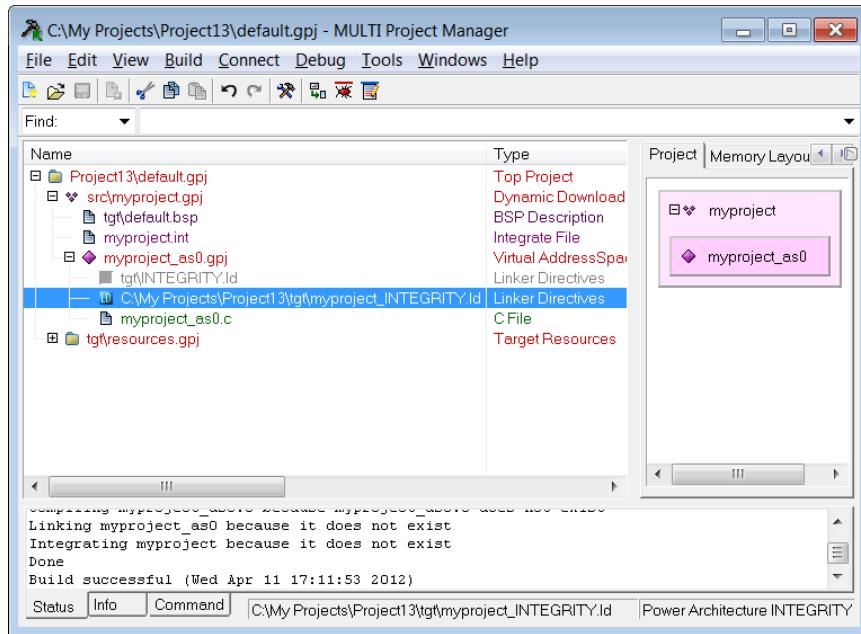
1. Copy **default.Id** or **flash.Id** to a new file, for example **myproject_default.Id**. Starting with the default file as a template ensures that necessary sections will not be mistakenly left out in the new file.
2. Add the customized linker directives file to the project's kernel AddressSpace.
3. For information about modifying the heap size for KernelSpace, see "Heap Size Configuration" in the "Common Application Development Issues" chapter.

The **INTEGRITY.Id** file in your top level INTEGRITY directory should never be modified. If you need to add user-created section maps or make other changes to the linker directives file for a

virtual AddressSpace, you should create and use your own custom version of **INTEGRITY.ld**. Some of the simple ways to do this are by creating a new **.ld** file, or by making changes to the reference copy of **INTEGRITY.ld** in the project's directory, as described in the following instructions.

To create a new custom linker directives file for a virtual AddressSpace project:

1. Create a new **.ld** file, for example **myproject_INTEGRITY.ld**.
2. In the new linker directives file, include **INTEGRITY.ld** from the top level INTEGRITY directory.
3. Add any custom sections to the new **.ld** file.
4. Add the customized **.ld** file to the virtual AddressSpace project and comment out the original **INTEGRITY.ld** from the project.



To modify the reference copy of **INTEGRITY.ld**:

1. Build or modify your project using the MULTI Project Manager. A reference copy of **INTEGRITY.ld** will be placed in your virtual AddressSpace project.
2. In the Project Manager, **Edit** the reference copy of **INTEGRITY.ld** file as needed.
3. Save your changes. The custom **INTEGRITY.ld** file will only be used in this Top Project.

Note: A change to a linker directives file has no effect until the program that uses this file is re-linked.

5.8.2 default.Id Example

default.Id and **flash.Id** are both linker directives files. These files will contain DEFAULTS, MEMORY, and SECTIONS blocks. Most of the common linker sections will be included with **ram_sections.Id** and **rom_sections.Id** files, which typically do not ever need to be modified. The following is an example **default.Id**:

```

DEFAULTS
{
    __INTEGRITY_DebugBufferSize      = 0x10000
    __INTEGRITY_DebugEntities       = 128
    __INTEGRITY_DebugEventLogSize   = 0x20000
    __INTEGRITY_HeapSize            = 0x90000
    __INTEGRITY_StackSize           = 0x4000
    __INTEGRITY_DownloadSize        = 0x1a0000
    __INTEGRITY_MaxCoreSize         = 0x200000
    __INTEGRITY_BootAreaSize        = 0x800000
    __INTEGRITY_FlashBufSize        = 0x10000
}

MEMORY
{
    ram_memory      :      ORIGIN = 0x00000000, LENGTH = 128M
    flash_memory    :      ORIGIN = 0xf0000000, LENGTH = 0
    ethaddr_memory  :      ORIGIN = 0xfffffe00, LENGTH = 256
}

SECTIONS
{
    .ramstart          : > ram_memory
    #include "ram_sections.ld"
    .ramend            align(0x1000)          : > ram_memory

    .romstart          : > flash_memory
    .romend            : > .

    // Flash memory location where MAC addresses are stored
    .ethaddr          pad(0x100)             : > ethaddr_memory
}

```

The corresponding **ram_sections.Id** file may then contain:

```

.bootarea                  : { . += isdefined(boot_elf) ? 
                                __INTEGRITY_BootAreaSize : 0; } > ram_memory
.PPC.EMB.sdata0            : > .
.PPC.EMB.sbss0  CLEAR      : > .
.vector                   align(0x10000) : > .
.text                      : > .
.loadertext               : > .
.syscall                  : > .
.execname                 : > .
.sdata2                   align(16)    : > .
.rodata                    align(16)    : > .
#ifndef FLASHABLE_KERNEL

```

```
.resettext          : > .
.resetvector        : > .
.resetrodata         : > .
.fixaddr           : > .
.fixtype            : > .
.secinfo             : > .
.robase              : > .
.initrodata          : > .

#endif // !FLASHABLE_KERNEL
.sdatabase      align(8)      : > .
.sdata          align(8)      : > .
.sbss           : > .
.data            : > .
.dbsockemu     align(4)      : > .
.bss             : > .
.earlybss        NOCLEAR : > .
.debugbuffer      NOCLEAR : { . += isdefined(_logchar) ?
                           __INTEGRITY_DebugBufferSize : 0; } > .
.entities       align(4) CLEAR : { . += isdefined(INDRT_ServiceEvents) ?
                           __INTEGRITY_DebugEntities : 0; } > .
.eventlog        align(16) NOCLEAR : { . += isdefined(INDRT_ServiceEvents) ?
                           __INTEGRITY_DebugEventLogSize : 0; } > .
.heap            align(16) pad(__INTEGRITY_HeapSize)    NOCLEAR      : > .
.stack           align(16) pad(__INTEGRITY_StackSize)   NOCLEAR : > .
.kstack           align(16) pad(0x4000)    NOCLEAR : > .
.cstack           align(16) pad(0x4000)    NOCLEAR : > .
.download         NOCLEAR : { . += isdefined(__INTEGRITY_LoaderTask) ?
                           __INTEGRITY_DownloadSize : 0; } > .
.mr_rwe_download align(0x1000) NOCLEAR : {
                           . += isdefined(VirtualLoaderHelperInit) ?
                           __INTEGRITY_DownloadSize : 0; } > .
.mr__core         align(0x1000) NOCLEAR : {
                           . += isdefined(DumpKernelCore) ?
                           (min(__INTEGRITY_MaxCoreSize,(sizeof(.kstack)+
                           sizeof(.bss)+sizeof(.sbss)+sizeof(.data)+
                           sizeof(.sdata)+0x4000))) : 0; . = align(0x1000); } > .
.mr___flashbuf   align(0x1000) pad(__INTEGRITY_FlashBufSize) NOCLEAR : > .
```

- The section `.vector` contains the machine vectors for interrupt handlers.
- The sections `.ramstart`, `.ramend`, `.romstart`, `.romend` are zero-size sections that mark the beginning and end of the RAM and FLASH/ROM areas used for kernel code and data (plus any user code and data linked in KernelSpace). This example section map is appropriate only for a RAM resident application, so `.romstart` and `.romend` are at the same location with no program sections within their range. (**flash.id** is provided for some BSPs and is appropriate for a flashable setup). The sections must be in the link map because of symbol references.
- The section `.execname` is a special section filled in by the Green Hills linker. It contains the name of the image resulting from this link and is used by the kernel to inform the Debugger of the executable to use when a Task from this application is attached.

- `.rodata` is the regular const data used by the kernel, libraries, and user application.
- The section `.initrodata` is constant data used specifically during BSP memory initialization.
- The section `.debugbuffer` is used to store serial console output.
- The section `.heap` represents dynamically allocated memory for the KernelSpace.
- The section `.stack` represents the area of stack used by the KernelSpace Initial Task. If the user links a `main()` routine with the kernel, `main()` is the entry point for the Initial Task.
- The section `.kstack` represents the area of stack used by the kernel and its interrupt handlers.
- The section `.mr___flashbuf` is used to store flash sectors when patching. It is used by the Flash IODevice and by the `ncs` command, which saves network configuration to flash.
- The other sections are standard sections for a link map, which can be customized. For example, you can increase the size of the `.heap` or `.stack`, or add user-defined sections.
- Before changing any of the constants provided in the `DEFAULTS` block, make sure you understand what these values do and how they will affect the setup of the system. Typically, `__INTEGRITY_HeapSize` and `__INTEGRITY_DownloadSize` may need to be adjusted, but the rest of the constants usually do not. For more information, see the “Modifying Linker Constants” section that follows.

Note: The CROM attribute must not be used. The INTEGRITY boot code does not support decompression of compressed sections and the resulting image will not boot.

5.8.3 Modifying Linker Constants

At times, you may need to modify linker constants defined in `default.ld`. Modifying the `__INTEGRITY_HeapSize` and `__INTEGRITY_DownloadSize` constants is the most common scenario, while other constants are less likely to need modification. Before changing linker constants, make sure you understand what these values do and how they will affect the setup of the system.

To change the value of a linker constant:

1. In the Project Manager, right-click the project and select **Edit⇒Set Build Options**.
2. On the **All Options** tab, expand the **Linker** category and select **Symbols**.

3. In the Build Options pane, double-click **Define Linker Constant (-C)**.
4. In the Edit List Option dialog, enter:
`--INTEGRITY_Constant=value.`
5. Click **OK** to close the dialog, and **Done** to close the Build Options window.

Note: A change to a linker directives file has no effect until the program that uses this file is re-linked.

5.8.4 Specifying Memory Regions in the Linker Directives File

Linker sections in the kernel's linker directives file may also be used to specify named memory regions that should be added to the BSP's memory table. These memory regions can either be registered with the ResourceManager or reserved using the **Section** keyword in an Integrate configuration file.

To specify and use a named memory region:

1. Create a line in the kernel linker directives file with the following format:

```
.mr_[attributes]_[name] address size
```

This registers a memory region with the ResourceManager under the name of *name* and the default system password (!systempassword). If you plan on reserving this memory region using the **Section** keyword in the Integrate configuration file, you must specify the **u** attribute as described below.

- *attributes* is an optional string that specifies the properties of the memory region to be created. The following *attributes* can be specified:
 - **r** — readable
 - **w** — writable
 - **e** — executable
 - **v** — volatile
 - **i** — I/O Coherent
 - **c** — I/O Coherent (synonymous with **i**)
 - **u** — unnamed. Provides a method of ensuring the memory range created in the BSP memory table has no name and will not be added to the ResourceManager at startup. This allows you to use the **Section** keyword.

If no attributes are specified, or only the **u** attribute is set, the memory region will default to being readable and will inherit the writable and executable attributes if indicated by the linker.

- address and size correspond to the address and size given to the linker section. These must be aligned to kernel page size boundaries because MemoryRegions must be aligned this way.

For example, to create a memory region named `myregion` located at address `0xFF000000`, with size `0x1000` bytes, and with the default properties, place the following line in the kernel's link file:

```
.mr_myregion 0xFF000000 pad(0x1000) :
```

To give the memory region readable, writable, and executable properties, use the following:

```
.mr_rwe_myregion 0xFF000000 pad(0x1000) :
```

Note: For backwards compatibility, the underscore separating the optional attribute string from the name may be omitted, provided that the name does not contain underscore characters.

2. After a named memory region has been added to the memory table, do one (but not both) of the following:

- Reserve the memory region in an integrate configuration (`.int`) file using the `Section` keyword in a `MemoryRegion` Object section. (For more information, see the “Object Section” chapter of the *Integrate User’s Guide*).
- Register the memory regions with the `ResourceManager`. This occurs automatically at boot time after rebuilding your application. (For more information, see the “`ResourceManagers`” chapter of the *INTEGRITY Libraries and Utilities User’s Guide*.)

Note that no checking is performed to ensure that the memory region specified is valid on the target. The linker section used to generate a memory region should be located at a valid address on the target. For example, you could create the section between `.ramstart` and `.ramend` to guarantee that the created memory region comes from the kernel's reserved RAM memory.

Also note that if you specify a memory region outside of the kernel's reserved RAM memory (after `.ramend` or before `.ramstart`), special care must be taken to ensure that the Integrate utility and other memory reservation mechanisms do not also try to use that region of memory for their own purposes. Specifically for use with Integrate, the address range occupied by the memory region should not be contained within one of the `MinimumAddress/MaximumAddress` pairs read by `intex`, as those pairs describe the memory that `intex` may use for its own purposes. See the “Reserving Memory Regions” section of the *INTEGRITY BSP User’s Guide* for additional information and restrictions.

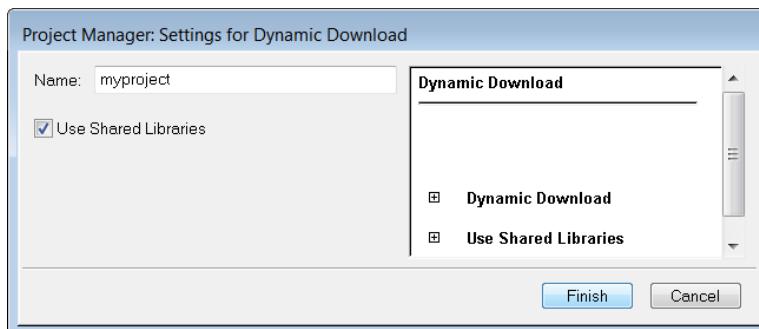
5.9 Shared Libraries

Shared libraries are provided for some BSPs. INTEGRITY shared libraries are fully linked ELF executables that are mapped into multiple virtual AddressSpaces so that the code can be shared. The libraries are linked at fixed virtual addresses, typically at high values. When you use shared libraries, you must ensure that any linker directives file used for a virtual AddressSpace project does not map code or data that overlaps with the shared libraries.

Shared libraries are not supported on SMP systems. For backward compatibility, integrate will recognize shared libraries and put a separate copy of any shared libraries in each AddressSpace, but the libraries will not actually be shared. At your earliest convenience, you should convert any projects running on SMP systems to use static libraries instead.

To enable or disable Shared Libraries:

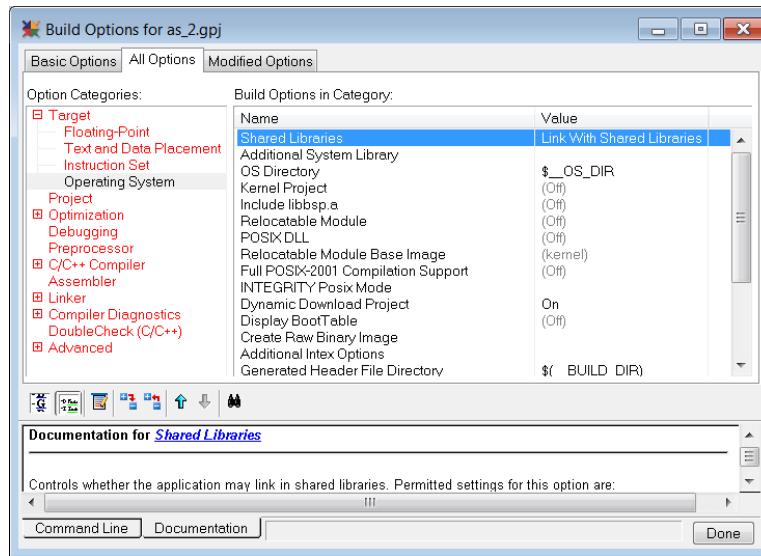
1. In the MULTI Project Manager, right-click your INTEGRITY Application **myproject.gpj** and select **Configure**. The Settings window will open.



2. Select or clear the **Use Shared Libraries** check box as desired.

Generally all virtual AddressSpaces should use shared libraries if one virtual AddressSpace does. However, if you need to modify shared library behavior for specific virtual AddressSpaces, you can modify the **-non_shared** option in the **.gpj** files as follows:

1. Right-click **.gpj** corresponding to the AddressSpaces that you want to modify and select **Set Build Options**.
2. On the **All Options** tab, select **Target⇒Operating System**.



3. In Build Options pane, right-click **Shared Libraries** and select **No Shared Libraries**, **Link With Shared Libraries**, or **Inherit from parent** as desired, then click **Done**.

To determine what addresses are used by shared libraries, use the Green Hills **gdump** utility. For example:

```
gdump -map c:\ghs\int10\ads85xx\libINTEGRITY.so
.text    0x17c0000    0x9114  (SHT_PROGBITS / Execinstr, Alloc)
...
...
```

This reserves virtual addresses from 0x17c0000 to the end of the map for **libINTEGRITY.so** in the virtual AddressSpace.

Note: In general you should not change the link addresses of shared libraries. If you do, choose a page-aligned address.

5.9.1 Available Shared Libraries

The following shared libraries are provided for INTEGRITY. Creating additional shared libraries requires custom development efforts. For assistance, contact Green Hills Technical Support for information about custom development availability.

- **libINTEGRITY.so** — Contains the INTEGRITY API layer and some language-independent run-time support such as Host I/O. **libINTEGRITY.so** is shared in each virtual AddressSpace of an INTEGRITY application that is using Shared Libraries.
- **libc.so** — Contains everything needed to support the C Language, including stdio and math functionality. **libc.so** is shared in each virtual AddressSpace of an INTEGRITY application that is using Shared Libraries.
- **libadaint.so** — Contains everything needed to support the Ada language run-time environment. **libadaint.so** is shared in each virtual AddressSpace of an INTEGRITY application that is using shared libraries and has its Language set to **Ada**.

- **libscxx.so** — Contains the C++ run-time library support for Standard C++ mode. This is the default mode when using C++. If C++ Exception Handling is being used, use **libscxx_e.so** instead.
- **libecxx.so** — Contains the C++ run-time library support for Extended Embedded C++ mode. This mode adds Template support to the Embedded C++ standard. **libecxx.so** has a much smaller footprint than **libscxx.so** and should be used whenever the C++ feature set employed permits it. If C++ Exception Handling is being used, use **libecxx_e.so** instead.
- **libscxx_e.so** — Used instead of **libscxx.so** when C++ Exception Handling is in use. **Note:** An AddressSpace using this library requires a stack size greater than 0x4000 for the library to initialize properly.
- **libecxx_e.so** — Used instead of **libecxx.so** when C++ Exception Handling is in use.

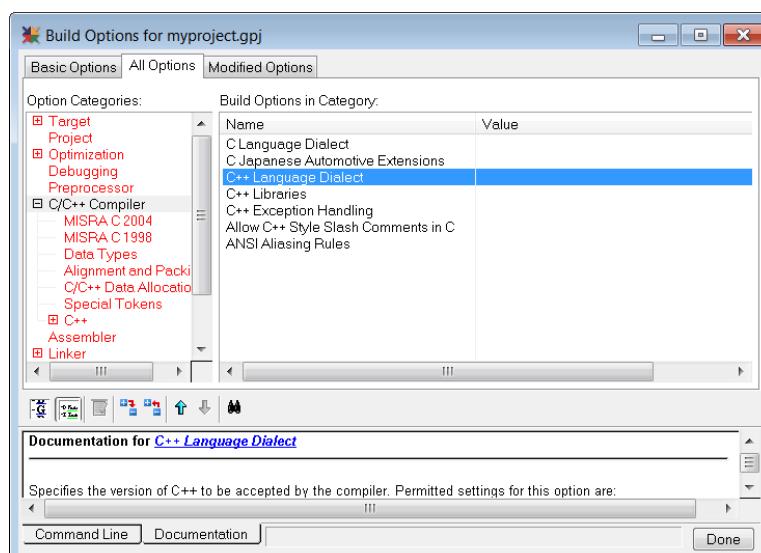
Note: Shared libraries are not supported for Standard Embedded C++.

In addition to the above libraries, INTEGRITY provides shared libraries for use with POSIX. For more information, see “Setting Up POSIX with Shared Libraries” in the *INTEGRITY Libraries and Utilities User Guide*.

5.9.1.1 Setting Options for C++ Shared Libraries

Shared libraries are supported for Standard C++ and Extended Embedded C++, but are not supported for Standard Embedded C++. To use Shared Libraries with C++, you must build your application with appropriate settings for C++ Language Dialect, C++ Libraries, and C++ Exception Handling. To set C++ options:

1. Right-click **myproject.gpj** and select **Set Build Options**.

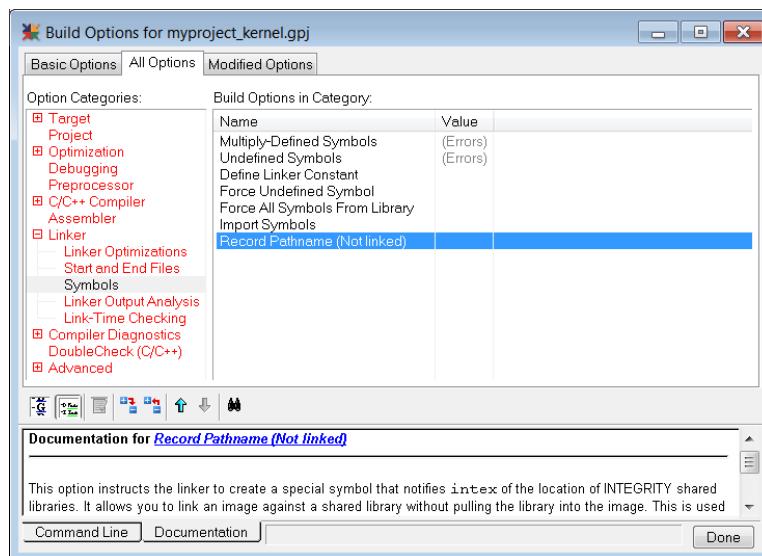


2. On the **All Options** tab, select **C/C++ Compiler**.
3. In the Build Options pane, right-click **C++ Language Dialect**, **C++ Libraries** or **C++ Exception Handling**, to set options that are appropriate for your application and shared library usage.

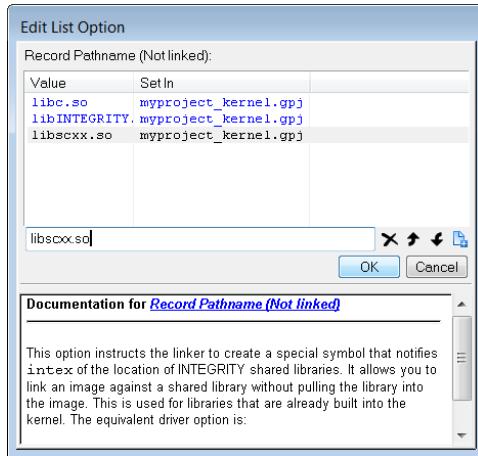
5.9.2 Sharing Libraries Across Multiple INTEGRITY Applications

Libraries can be shared across multiple applications. In order to share libraries across applications, you need to set additional build options, and modify the **.int** files. For example, to share **libc.so**, **libINTEGRITY.so** and **libscxx.so** across multiple applications:

1. Open a Monolith INTEGRITY application in the Project Manager.
2. Set Build Options for the Kernel. To do this:
 - (a) Right-click the Kernel's **.gpj** file and select **Set Build Options** to open the Build Options window.
 - (b) On the **All Options** tab, select the **Linker⇒Symbols** category.



- (c) Double-click **Record Pathname**. The Edit List Option dialog will open.



- (d) Enter `libc.so` in the text field, then press **Enter**. Repeat this process for `libINTEGRITY.so`, and `libsccxx.so`.
3. Include the desired shared libraries in your Monolith image. To do this, right-click the Monolith project's `.int` file and select **Edit**. The libraries can be designated in a virtual AddressSpace, or in the kernel, as follows.

```
Kernel
  Filename      kernel
  Library       libINTEGRITY.so
  Library       libc.so
  Library       libsccxx.so
EndKernel
```

4. For any Dynamic Download applications that need to share the libraries, specify `UnmappedLibrary` in the `.int` file, as follows:

```
Kernel
  Filename      DynamicDownload
EndKernel

AddressSpace
  Filename      myproject
  Language      C
  UnmappedLibrary libINTEGRITY.so
  UnmappedLibrary libc.so
  UnmappedLibrary libsccxx.so
EndAddressSpace
```

5. Upon download, the text section for each shared library will be mapped to the library included in the Monolith image. Make sure you download to a kernel that includes the libraries, otherwise the shared libraries will remain unmapped and this application will not be able to run.

5.10 Additional Options for Building Applications

This section provides information about optional features that can be enabled when building INTEGRITY applications, how to conditionalize code for different versions of INTEGRITY, and how to use makefiles with command-line drivers.

5.10.1 Using Checksum and SHA-1 Utilities for Data Consistency

INTEGRITY provides a checksum utility and a SHA-1 utility for data consistency. Both are implemented similarly, the tools calculate and embed the desired values in the final image, and during INTEGRITY boot up, one of the first actions is to verify that the contents of memory match what is expected, in both ROM and RAM when appropriate.

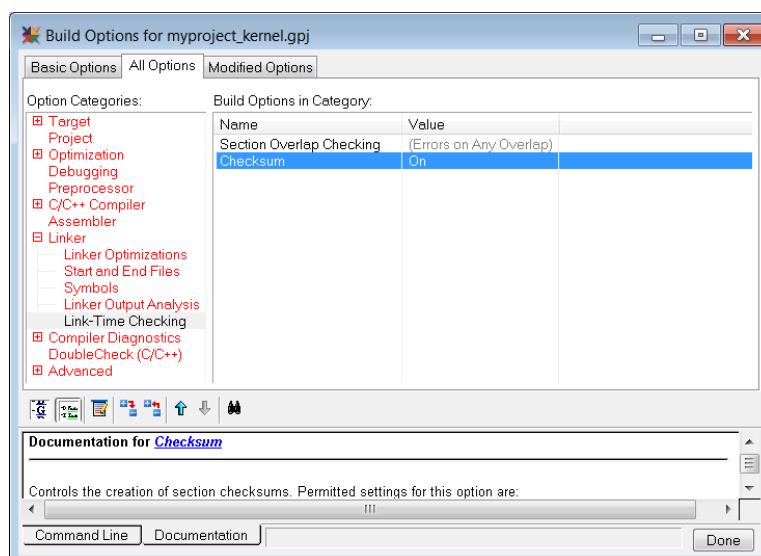
Note: Breakpoints can cause the CRC or SHA-1 comparison to fail. This includes breakpoints that may automatically be set by the MULTI Debugger, such as the breakpoint used by some Debugger versions to implement the Automatic Run-mode Partner Connection. See your MULTI documentation for details about this limitation.

5.10.1.1 Invoking CRC on the Kernel and Virtual AddressSpaces

The linker contains a feature that calculates a CRC for text and data sections in the kernel. When this feature is enabled, the kernel recalculates this CRC and compares it to the value calculated by the linker. If the values match, it confirms the integrity of the kernel image in memory. A discrepancy between the two CRCs indicates a memory failure.

To enable the CRC, in the KernelSpace project **myproject_kernel.gpj**:

1. Right-click **myproject_kernel.gpj** and select **Set Build Options**.
2. On the **All Options** tab, select **Linker⇒Link-Time Checking**.
3. In the Build Options pane, right-click **Checksum**, and set to **On**.



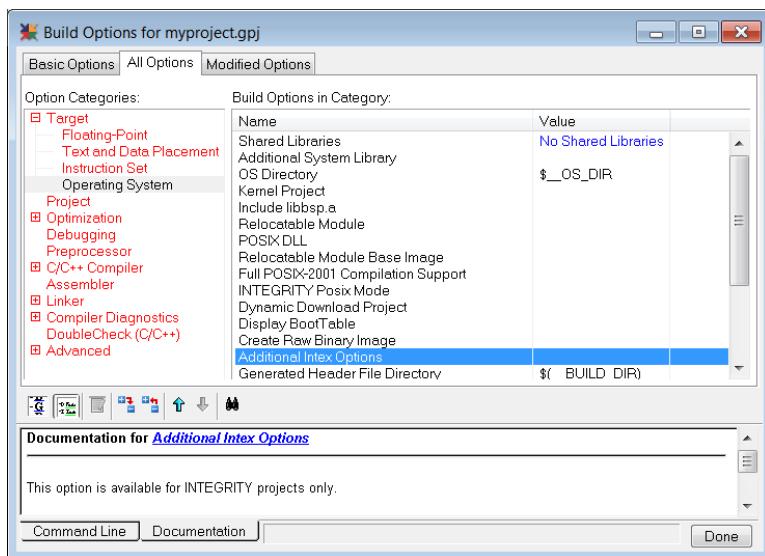
When building an INTEGRITY application, Integrate detects that the kernel has been linked with CRCs enabled and automatically generates CRCs to protect the virtual AddressSpace text and data sections within the integrated image. To avoid wasting additional space and an Integrate warning, enable **-checksum** on the kernel only, and not in virtual AddressSpaces. There is no need to link individual virtual AddressSpaces with the **-checksum** option.

For information about handling CRC failures, see the *INTEGRITY BSP User's Guide*.

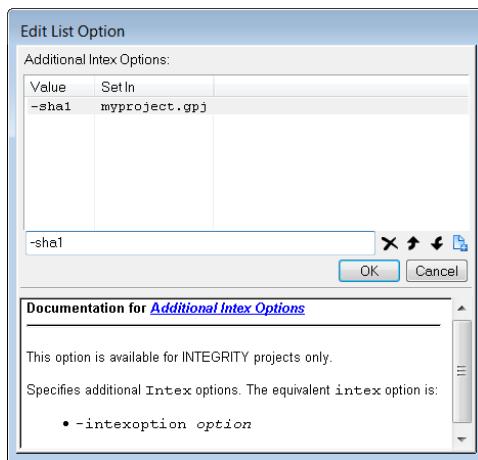
5.10.1.2 Invoking SHA-1 on an INTEGRITY Monolith

To enable SHA-1 on your INTEGRITY monolith project **myproject.gpj**, you must pass an intex option to the monolith and link the SHA-1 library into the KernelSpace part of the monolith.

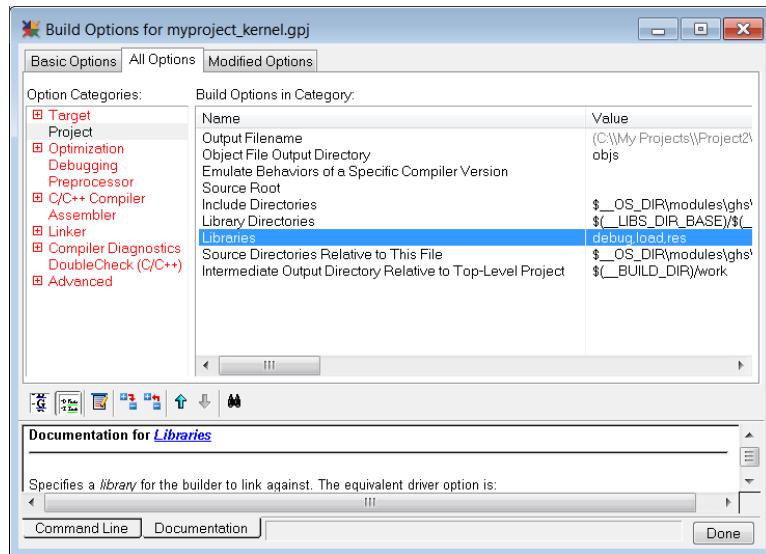
1. Enable SHA-1, in the INTEGRITY monolith project **myproject.gpj** as follows:
 - (a) Right-click **myproject.gpj** and select **Set Build Options**.
 - (b) On the **All Options** tab, select **Target⇒Operating System**.
 - (c) In the Build Options pane, double click **Additional intex options** to edit the option.



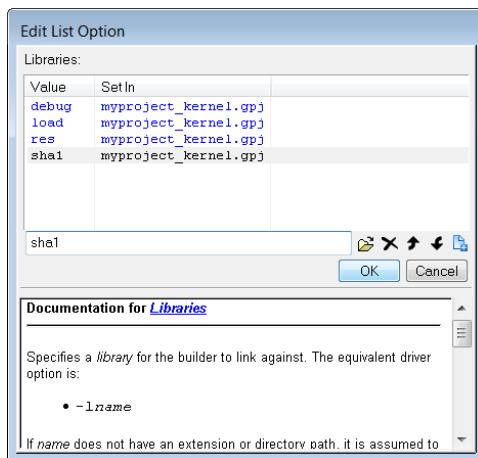
- (d) On the Edit List Option dialog, enter **-sha1** and click **OK**.



2. Add the SHA-1 library to the kernel in the INTEGRITY monolith, which provides the target side code to perform the run-time check:
 - (a) Right-click **myproject_kernel.gpj** and select **Set Build Options**.
 - (b) On the **All Options** tab, select **Project**.
 - (c) In the Build Options pane, double-click **Libraries**.



- (d) On the Edit List Option dialog, enter sha1, then click **OK**.



5.10.2 Using INTEGRITY Version Constants to Conditionalize Code

The include file <INTEGRITY_version.h> contains version constants. When this file is included, these can be used at compile time to conditionalize code for different versions of INTEGRITY.

The following constants are defined and can be used to specify which version of INTEGRITY a particular piece of code should be compiled for. Use the #ifdef preprocessor directive to write code that compiles differently depending on what version number is specified with these macros:

```
--INTEGRITY_MAJOR_VERSION  
--INTEGRITY_MINOR_VERSION  
--INTEGRITY_PATCH_VERSION  
--INTEGRITY_VERSION_STRING
```

You can only use the __INTEGRITY_PATCH_VERSION to differentiate between two INTEGRITY versions that both have the same __INTEGRITY_MAJOR_VERSION and __INTEGRITY_MINOR_VERSION. __INTEGRITY_VERSION_STRING cannot be used to conditionalize code.

5.10.3 Using makefiles

You should use the MULTI Project Manager and **.gpj** files to build INTEGRITY projects. However, you can use makefiles with INTEGRITY by using the command-line drivers described later in this section. To use makefiles:

1. Open the example makefile provided in the **examples\HelloWorld** directory located off the root directory. This makefile contains board-specific options you must set.
2. After these options are set, use this makefile to compile the **helloworld1** example. The resulting Dynamic Download image is called **helloworld1** and will be in the same directory as the makefile.

The **examples\SharedMemory** directory contains a set of makefiles that demonstrates how you might use makefiles to compile multiple virtual AddressSpace programs and integrate them using **intex**.

5.10.3.1 Using the Command-Line Driver

You should use the MULTI Project Manager to organize and build INTEGRITY applications because standard command-line compiler drivers are insufficient to handle the complexity involved with potentially managing many virtual AddressSpace programs and a KernelSpace program in a single boot image. However, some users will require some amount of command-line driver support. This section describes the command-line driver support available for INTEGRITY.

The INTEGRITY compiler driver is named **ccintarch**. Examples in this section use the compiler driver for x86, **ccint86**. You must invoke the compiler driver once per AddressSpace.

After using the compiler driver, you must call **intex**, which takes either an **.int** file or a list of the AddressSpaces to be packaged together into one INTEGRITY Application. For more information, see “Command Line Syntax” in the *Integrate User’s Guide*.

Note: Beginning with INTEGRITY 11, **intex** ships with INTEGRITY and is located in **rtos_install_dir\multi\bin\host**. However, **ccint*** ships with Green Hills tools and is located in **tools_install_dir**.

The following sections describe commonly used options for building INTEGRITY applications and provide example driver commands.

5.10.3.2 Command-Line Driver Options

The following command-line driver options are commonly used for building INTEGRITY applications.

-os_dir INTEGRITY_Install_Directory

A required option for building INTEGRITY applications using the driver. This option informs the driver of the location of the INTEGRITY installation being used for development. For example, if the INTEGRITY installation is in **C:\ghs\int11**, the appropriate option is **-os_dir C:\ghs\int11**.

-bsp *bspname*

This is a required option for building INTEGRITY applications using the driver. This option informs the driver of the target BSP that is being used for development. For example, if using the pcx86, the appropriate option is **-bsp pcx86**.

-alt_tools_path *tools_installation_directory*

A required option for running intex that specifies where to find the Green Hills Tools distribution, which contains necessary utilities. **Note:** This flag is new in INTEGRITY 11.0 and must be added to any command lines used with earlier releases.

5.10.3.3 Command-Line Driver Examples

The command:

```
ccint86 -os_dir C:\GHS\int11  
-bsp pcx86  
file1.c file2.cxx -o vas
```

causes the following to occur:

- The C source file, **file1.c**, and the C++ source file, **file2.cxx**, will be compiled into object code and linked into a virtual AddressSpace program named **vas**. Pass **-G** if you want debug symbols for your code.

There are many different ways to take this virtual AddressSpace and create an INTEGRITY application.

The command:

```
intex -os_dir C:\GHS\int11 -bsp pcx86  
-alt_tools_path C:\GHS\multi616  
vas -o monolith_image
```

causes the following to occur:

- The Integrate utility will be run in command-line mode, taking as input **vas**, and the **default.bsp** BSP description file for the pcx86 BSP.
- An INTEGRITY application image will be created that consists of the single virtual AddressSpace **vas**, and the KernelSpace program, **kernel**, located in the pcx86 BSP's build directory.
- The name of the INTEGRITY application image created will be **monolith_image**.

The command:

```
intex -os_dir C:\GHS\int11 -bsp pcx86  
-alt_tools_path C:\GHS\multi616  
-dynamic vas -o dynamic_download_image
```

causes the following to occur:

- The Integrate utility will be run in command-line mode, taking as input **vas** and the **default.bsp** BSP description file for the pcx86 BSP.
- An INTEGRITY application image named **dynamic_download_image** will be created which is a dynamic download that consists of the single virtual AddressSpace **vas**.

The command:

```
intex -os_dir C:\GHS\int11 -bsp pcx86  
-alt_tools_path C:\GHS\multi616  
-intfile myprog.int -o myprog
```

causes the following to occur:

- The Integrate utility will be run in Integrate Configuration File mode, taking as input the Integrate Configuration file **myprog.int** and the **default.bsp** BSP description file for the pcx86 BSP.
- An INTEGRITY application image will be created that is constructed according to the specifications of **myprog.int**, the final image will be named **myprog**

Chapter 6

Configuring Applications with the MULTI Project Manager

This chapter contains information about using the New Project Wizard (NPW) and MULTI Project Manager to configure INTEGRITY options and contains the following sections:

- Configuring Monolith Projects with the NPW
- Modifying Monolith Projects with the Project Manager
- Configuring Dynamic Download Projects with the NPW
- Modifying Dynamic Download Projects with the Project Manager
- Configuring KernelSpace Projects with the NPW
- Modifying KernelSpace Projects with the Project Manager
- Adding Items to INTEGRITY Projects
- Copying Examples with the MULTI Project Manager
- Converting and Upgrading Projects with the MULTI Project Manager

You can use the New Project Wizard to configure many options of your INTEGRITY application. In previous sections, most of these options were skipped by clicking **Finish** and accepting the defaults. This chapter provides information about how to use the NPW to configure these options.

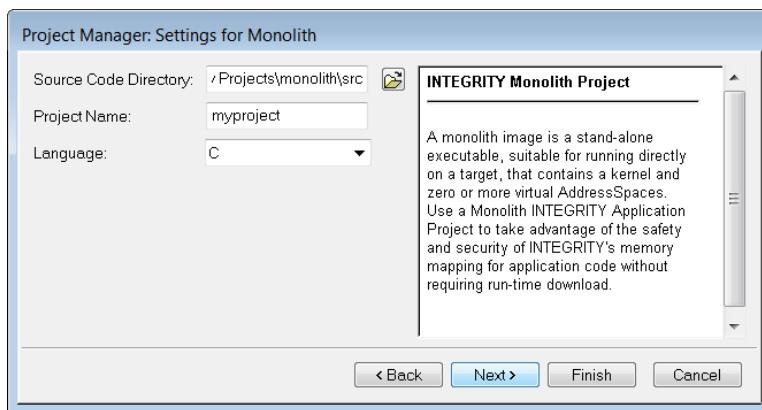
You can also use the MULTI Project Manager to modify the configuration of existing INTEGRITY applications. For existing projects, many options can be accessed by selecting the project and selecting **Configure** or **Modify Project**. This chapter provides information about how to use the dialog boxes to configure or modify existing projects.

For additional information about using the MULTI Project Manager to modify projects, see “The Project Manager GUI Reference” in *MULTI: Managing Projects and Configuring the IDE*.

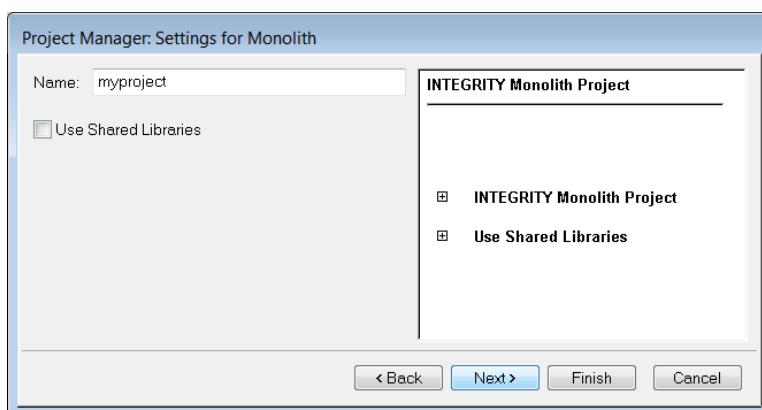
6.1 Configuring Monolith Projects with the NPW

When you are creating a new INTEGRITY project, you can specify Monolith project settings using the NPW:

1. Follow the “Creating a Monolith INTEGRITY Project with the New Project Wizard” instructions in the “Building INTEGRITY Applications” chapter, but instead of clicking **Finish** on the Select Item to Add Screen, click **Next**.
2. Set the following options on the first Settings for Monolith screen:

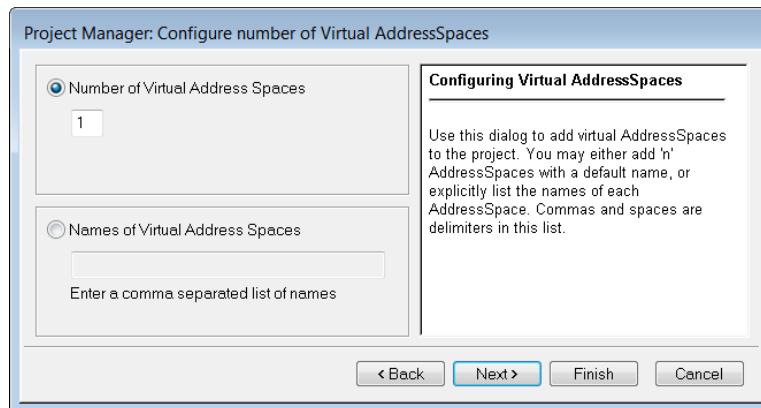


- Use the **Source Code Directory** field to choose the location of the new project.
 - Use the **Project Name** field to choose the basename of the new project.
 - In **Language**, select **C**, **C++**, or **Ada** (when applicable).
3. Click **Next**. The following Settings for Monolith screen opens.



- **Name** — Edit this field to change the name of the project.

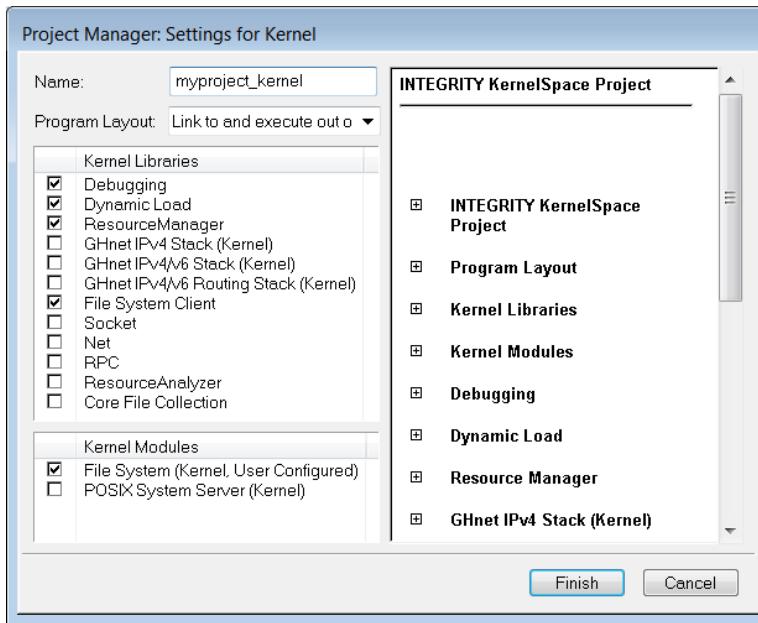
- **Use Shared Libraries** — If this box is checked, virtual AddressSpaces will use shared libraries. If the box is not checked, virtual AddressSpaces will be statically linked with their INTEGRITY and language specific-libraries.
4. Click **Next**. The Configure number of Virtual AddressSpaces screen opens. Use this screen to select whether AddressSpaces use default names, or use names you specify.



- **Number of Virtual AddressSpaces** — To create AddressSpaces with default names, enter the number of virtual AddressSpaces to be created in this field. This is the easiest method to specify AddressSpaces. The Project Manager will create names for the constituent virtual AddressSpace projects by appending numbers onto the base name chosen in the previous window.
- **Names of Virtual AddressSpaces** — To create AddressSpaces with specific names, enter AddressSpace names in a comma separated list. Spaces and commas are delimiting characters in this dialog and should not be used in AddressSpace names. Additionally, for Debugging purposes, AddressSpace names should comply with the restrictions documented in the “Task and AddressSpace Naming” section of the “Run-Mode Debugging” chapter.

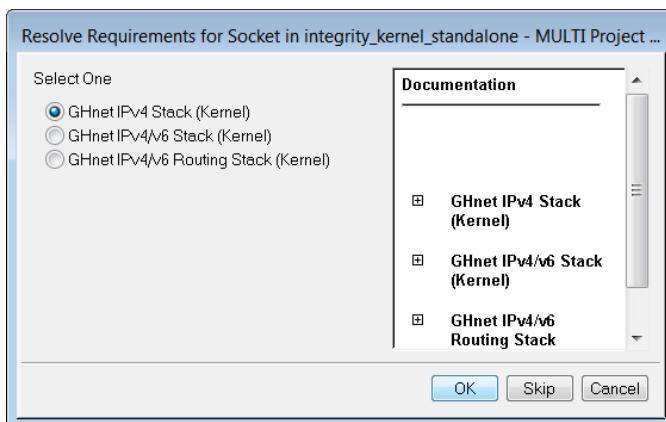
To add more virtual AddressSpaces after you create your project, see “Adding and Configuring Virtual AddressSpaces with the Project Manager” later in this chapter.

5. Click **Next**. The following Settings for Kernel screen opens. Use this screen to configure options for program layout, kernel libraries, and kernel modules.

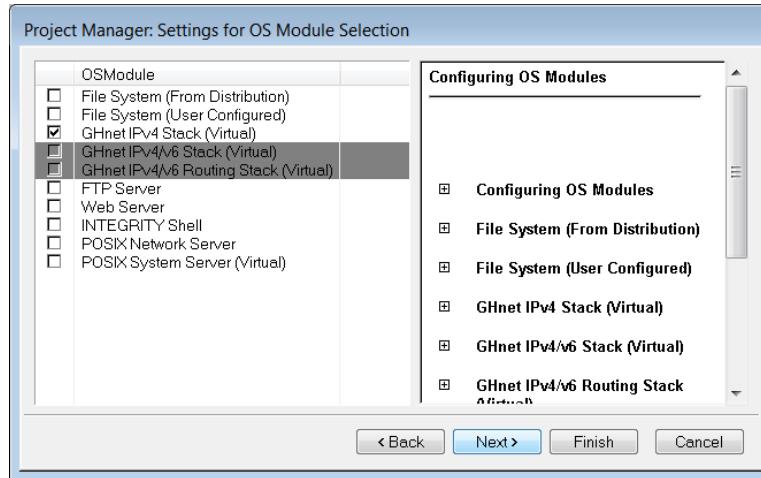


- **Name** — Edit this field to change the name.
- **Program Layout** — Select from **Link to and Execute out of RAM** or **Link to ROM and execute out of RAM**.
- **Kernel Libraries** — Select libraries to link them into the KernelSpace program. Some libraries automatically enable other libraries, for example, the Resource Analyzer requires a TCP/IP stack.
- **Kernel Modules** — Select modules to include them in the KernelSpace program. Some modules have dependencies that automatically cause other modules to be pulled in.

If there is a choice of libraries or modules that can be used, a Resolve Requirements dialog will open with available options to choose from, similar to the following:



6. Click **Next**. The OS Module Selection window opens.



7. Select the check box for any OS Modules you want to include in your project.

- Some items are mutually exclusive, so only one can be selected at any time. Selecting certain options will automatically clear the check box of mutually exclusive options. Clearing the check box for a mutually exclusive option will make the other options available.
- Some items have dependencies that automatically cause other modules to be pulled in. If there is a choice of options that can fulfill the requirement, a Resolve Requirements or other configuration dialog will open with available options to choose from.

8. Click **Finish**. Your Monolith project will open in the MULTI Project Manager configured with the specified options.

6.2 Modifying Monolith Projects with the Project Manager

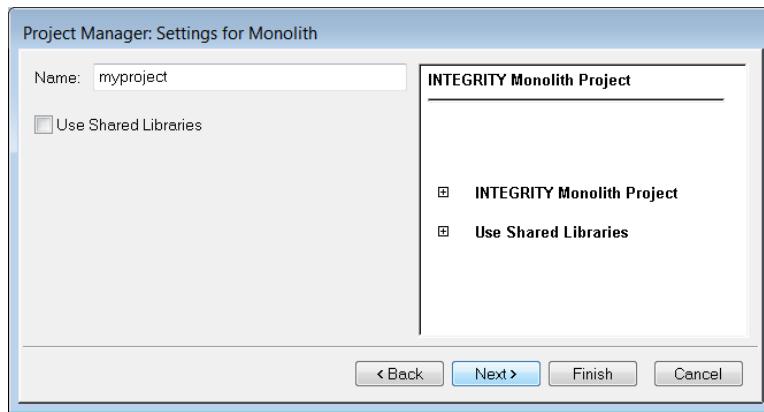
You can modify an existing project with the **Configure** and **Modify Project** menu selections:

1. Open an existing Monolith project in the Project Manager.
2. Right-click the Monolith project.
 - To change the project name or shared library settings, select **Configure** and see the “Settings for Monolith Dialog” section below.
 - To modify or add items to your project, select **Modify Project** and select the item you want to add or modify.
 - To add a virtual AddressSpace, select **Add INTEGRITY Virtual AddressSpace** and see “Adding and Configuring Virtual AddressSpaces with the Project Manager” later in this chapter.
 - To add or modify OS Modules, select **Add INTEGRITY OS Module** and see “Adding OS Modules with the Project Manager” later in this chapter.

To modify kernel settings in an existing Monolith project, see “Modifying KernelSpace Projects with the Project Manager” later in this chapter.

6.2.1 Settings for Monolith Dialog

When you right-click a Monolith project and select **Configure**, the Settings for Monolith dialog opens:



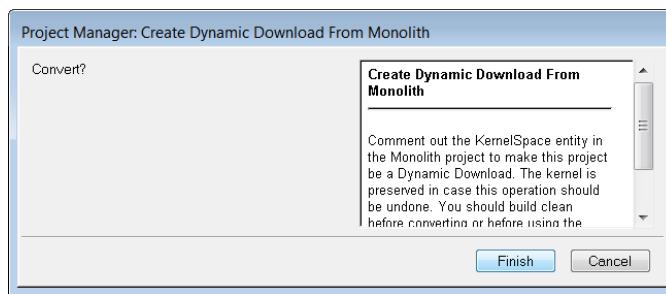
- **Name** — Edit this field to change the name of the project.
- **Use Shared Libraries** — If this box is checked, virtual AddressSpaces will use shared libraries. If the box is not checked, virtual AddressSpaces will be statically linked with their INTEGRITY and language specific-libraries.

6.2.2 Converting a Monolith to a Dynamic Download

If you have created a Monolith INTEGRITY Application, you can convert it to a Dynamic Download INTEGRITY Application.

Note: Depending on the contents of your project, it may not be appropriate to convert to a Dynamic Download. For example, if your Monolith contains a POSIX System Server, converting it would cause a conflict between the kernel loader agent used to load Dynamic Downloads and the POSIX loader.

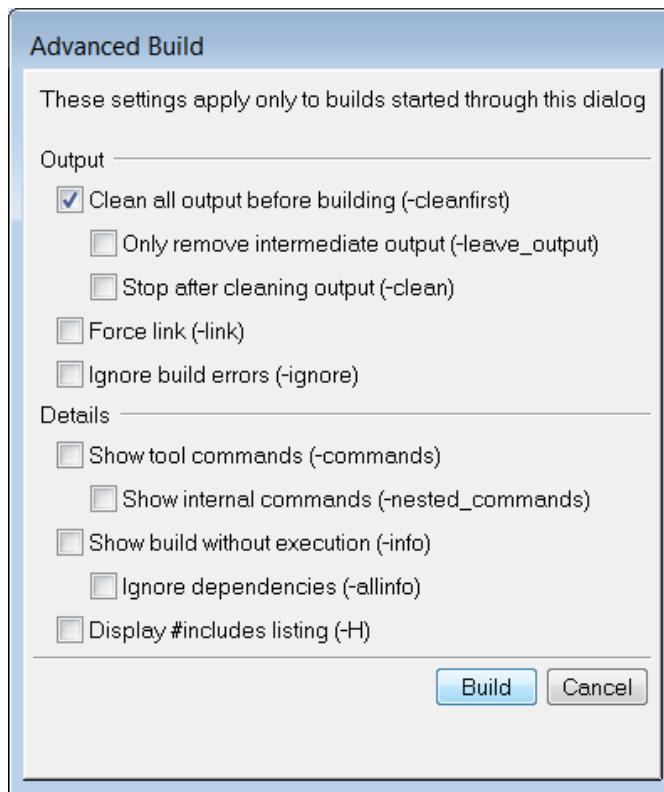
1. Right-click the Monolith Project and select **Modify Project⇒Create Dynamic Download From Monolith**.



2. Click **Finish** in the confirmation box.

The Monolith's kernel build files and **.int** file entries will be commented out. If you want to restore the project, follow the directions in “Converting a Dynamic Download to a Monolith”.

3. Right-click the Dynamic Download and select **Advanced Build**.

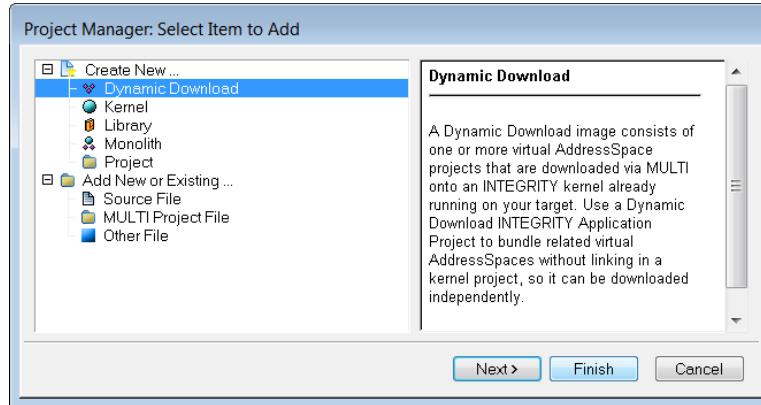


4. On the Advanced Build dialog, select **Clean all output before building (-cleanfirst)**, and click **Build**.

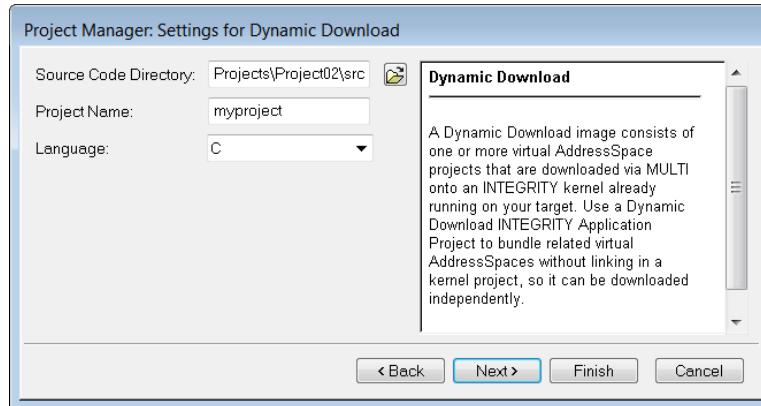
6.3 Configuring Dynamic Download Projects with the NPW

When you are creating a new INTEGRITY project, you can specify Dynamic Download project settings using the NPW:

1. Follow the “Creating a Dynamic Download Application with the New Project Wizard” instructions in the “Building INTEGRITY Applications” chapter, but instead of clicking **Finish** on the Select Item to Add Screen, click **Next**.

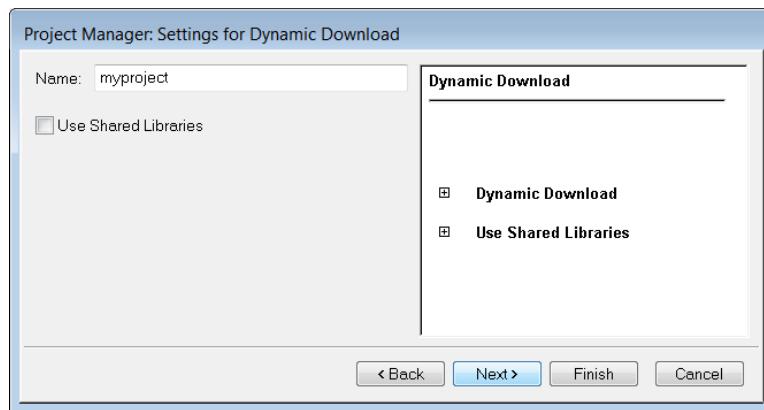


2. Set the following options on the first Settings for Dynamic Download screen:

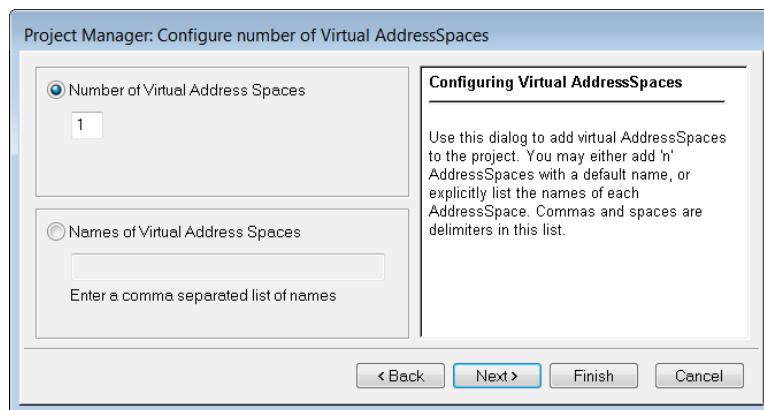


- Use the **Source Code Directory** field to choose the location of the new project.
- Use the **Project Name** field to choose the basename of the new project.
- In **Language**, select **C**, **C++**, or **Ada** (when applicable).

3. Click **Next**. The next Settings for Dynamic Download screen opens.



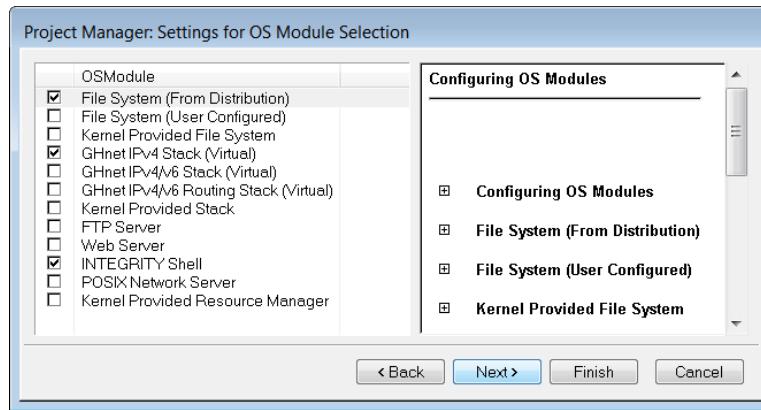
- **Name** — Edit this field to change the name of the project.
 - **Use Shared Libraries** — If this box is checked, virtual AddressSpaces will use shared libraries. If the box is not checked, virtual AddressSpaces will be statically linked with their INTEGRITY and language specific-libraries.
4. Click **Next**. The Configure number of Virtual AddressSpaces screen opens. Use this screen to select whether AddressSpaces use default names, or use names you specify.



- **Number of Virtual AddressSpaces** — To create AddressSpaces with default names, enter the number of virtual AddressSpaces to be created in this field. This is the easiest method to specify AddressSpaces. The Project Manager will create names for the constituent virtual AddressSpace projects by appending numbers onto the base name chosen in the previous window.
- **Names of Virtual AddressSpaces** — To create AddressSpaces with specific names, enter AddressSpace names in a comma separated list. Spaces and commas are delimiting characters in this dialog and should not be used in AddressSpace names. Additionally, for Debugging purposes, AddressSpace names should comply with the restrictions documented in the “Task and AddressSpace Naming” section of the “Run-Mode Debugging” chapter.

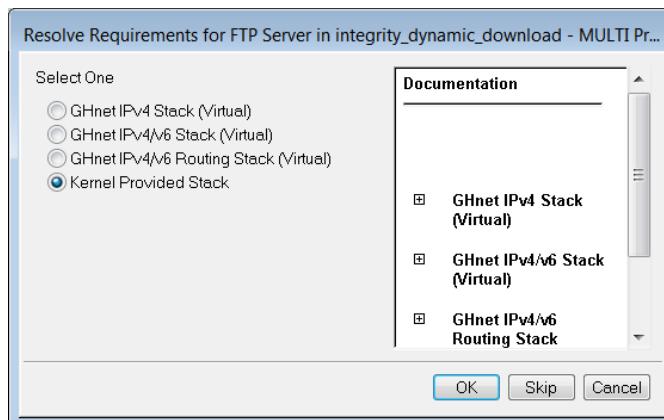
To add more virtual AddressSpaces after you create your project, see “Adding and Configuring Virtual AddressSpaces with the Project Manager” later in this chapter.

- Click **Next**. The OS Module Selection dialog opens.



- Select the check box for any OS Modules you want to include in your project.

- Some items are mutually exclusive, so only one can be selected at any time. Selecting certain options will automatically clear the check box of mutually exclusive options. Clearing the check box for a mutually exclusive option will make the other options available.
- Some items have dependencies that automatically cause other modules to be pulled in. If there is a choice of options that can fulfill the requirement, a Resolve Requirements or other configuration dialog will open with available options to choose from.



- Click **Next**. Your Dynamic Download project will open in the MULTI Project Manager configured with the specified options.

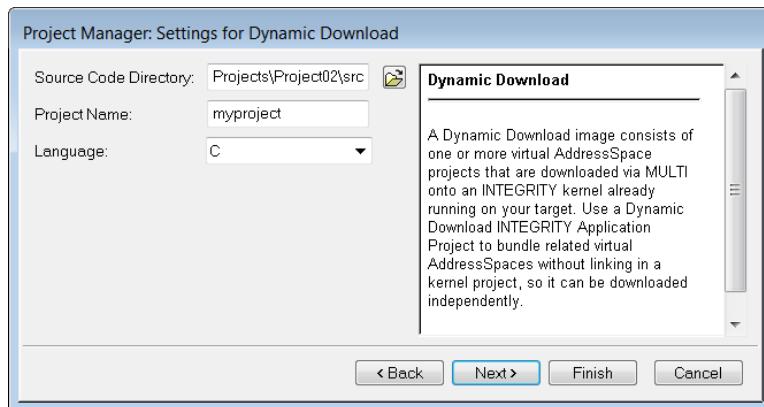
6.4 Modifying Dynamic Download Projects with the Project Manager

You can modify an existing project with the **Configure** and **Modify Project** menu selections:

1. Open an existing Dynamic Download project in the Project Manager.
2. Right-click the Dynamic Download project.
 - To change the project name or shared library settings, select **Configure** and see the “Settings for Dynamic Download Dialog” section below.
 - To modify or add items to your project, select **Modify Project** and select the item you want to add or modify.
 - To add a virtual AddressSpace, select **Add INTEGRITY Virtual AddressSpace** and see “Adding and Configuring Virtual AddressSpaces with the Project Manager” later in this chapter.
 - To add or modify OS Modules, select **Add INTEGRITY OS Module** and see “Adding OS Modules with the Project Manager” later in this chapter.

6.4.1 Settings for Dynamic Download Dialog

When you right-click a Dynamic Download project and select **Configure**, the Settings for Dynamic Download dialog opens:

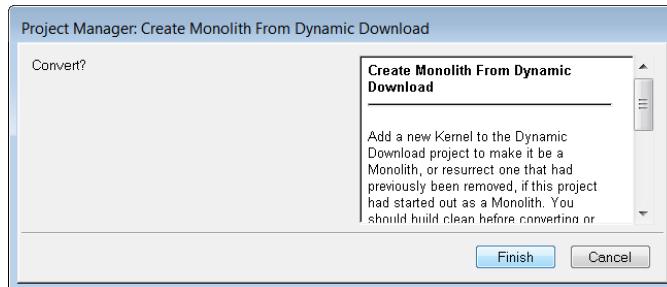


- **Name** — Edit this field to change the name of the project.
- **Use Shared Libraries** — If this box is checked, virtual AddressSpaces will use shared libraries. If the box is not checked, virtual AddressSpaces will be statically linked with their INTEGRITY and language specific-libraries.

6.4.2 Converting a Dynamic Download to a Monolith

If you have created a Dynamic Download INTEGRITY application, you can convert it to a Monolith INTEGRITY application.

1. Right-click the Dynamic Download project and select **Modify Project**⇒**Create Monolith From Dynamic Download**.



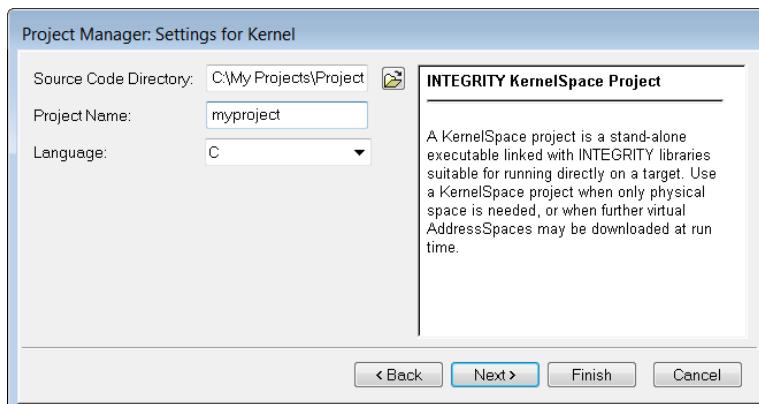
2. Click **Finish** in the confirmation window.

If this project has always been a Dynamic Download, a new kernel will be created and added to the project. If the project was converted from a Monolith originally, the original kernel will be restored. If you wish to restore the project, follow the directions in “Converting a Monolith to a Dynamic Download” section earlier in this chapter.

6.5 Configuring KernelSpace Projects with the NPW

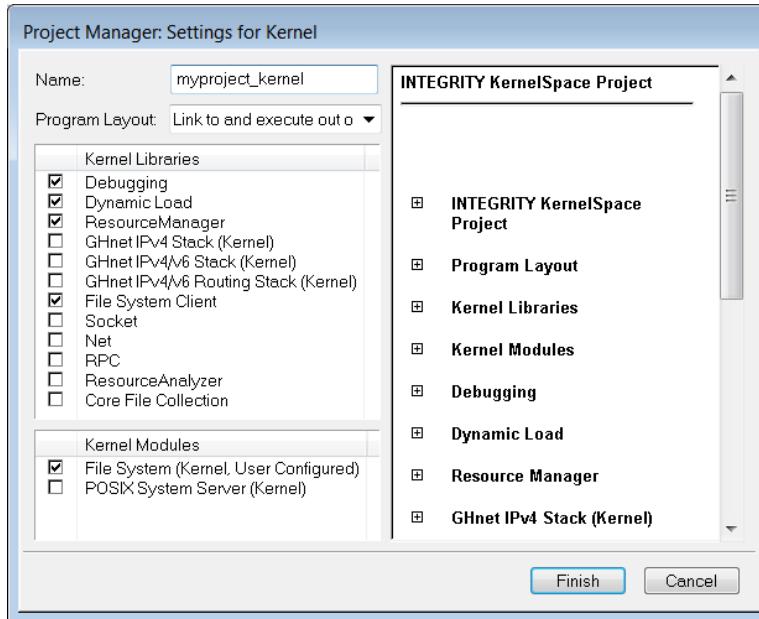
When you are creating a new INTEGRITY project, you can specify KernelSpace project settings using the NPW:

1. Follow the “Creating a KernelSpace Project with the New Project Wizard” instructions in the “Building INTEGRITY Applications” chapter, but instead of clicking **Finish** on the Select Item to Add screen, click **Next**.
2. Set the following options on the first Settings for Kernel screen:



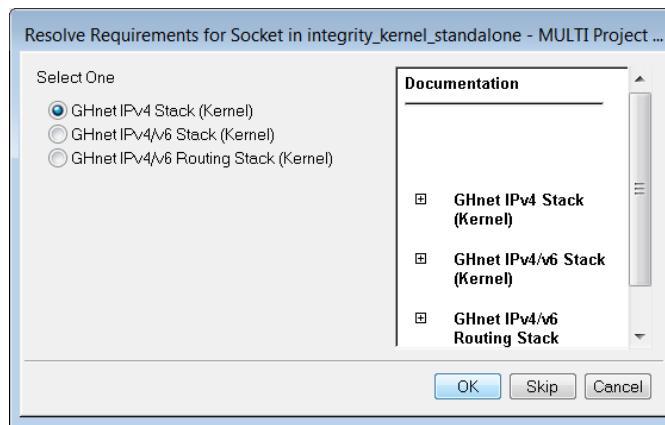
- Use the **Source Code Directory** field to choose the location of the new project.
- Use the **Project Name** field to choose the basename of the new project.
- In **Language**, select **C**, **C++**, or **Ada** (when applicable).

3. Click **Next**. The next Settings for Kernel screen is displayed.

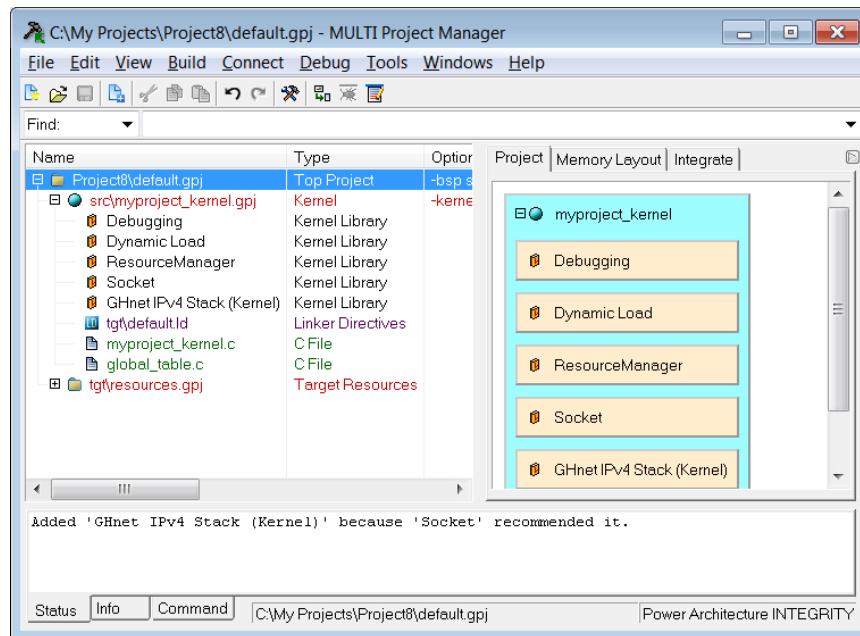


- **Name** — Edit this field to change the name.
- **Program Layout** — Select from **Link to and Execute out of RAM** or **Link to ROM and execute out of RAM**.
- **Kernel Libraries** — Select libraries to link them into the KernelSpace program. Some libraries automatically enable other libraries, for example, the Resource Analyzer requires a TCP/IP stack.
- **Kernel Modules** — Select modules to include them in the KernelSpace program. Some modules have dependencies that automatically cause other modules to be pulled in.

If there is a choice of libraries or modules that can be used, a Resolve Requirements dialog will open with available options to choose from, similar to the following:



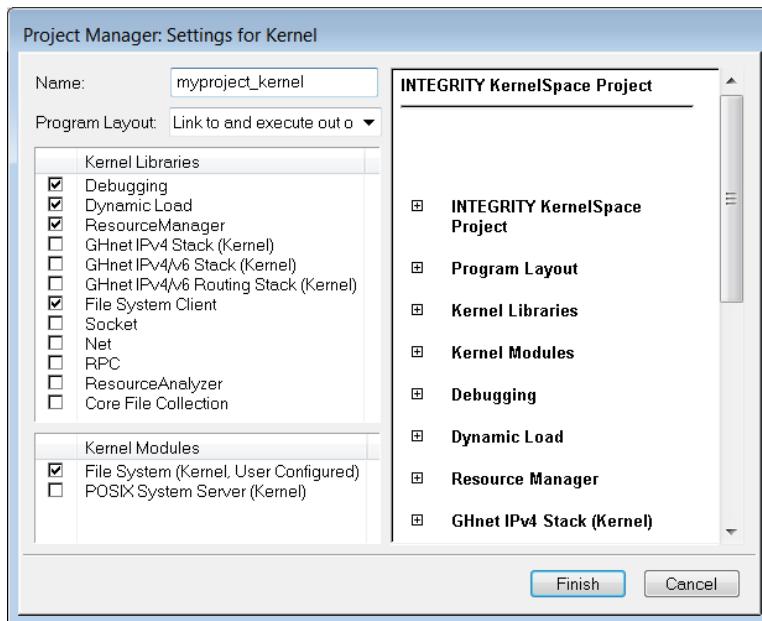
4. Click **Finish**. Your KernelSpace project will open in the MULTI Project Manager configured with the specified options.



6.6 Modifying KernelSpace Projects with the Project Manager

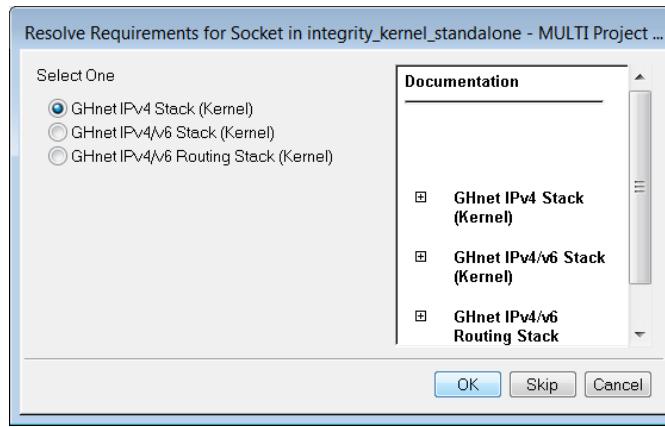
To modify program name, layout, kernel libraries, or kernel modules in an existing KernelSpace project:

1. Open an existing KernelSpace project in the Project Manager.
2. Right-click the Kernel project and select **Configure**.
3. Use the Settings for Kernel screen to configure the following options:

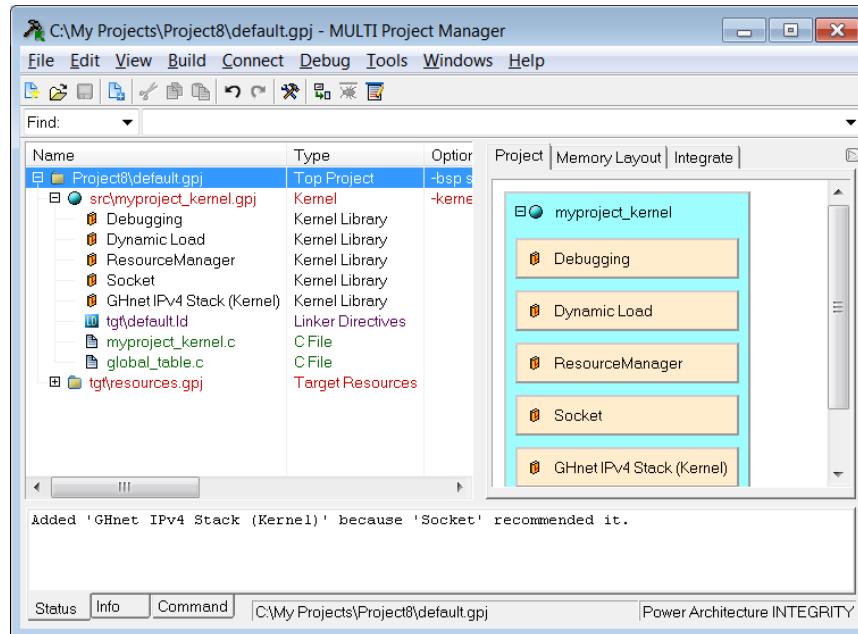


- **Name** — Edit this field to change the name.
- **Program Layout** — Select from **Link to and Execute out of RAM** or **Link to ROM and execute out of RAM**.
- **Kernel Libraries** — Select libraries to link them into the KernelSpace program. Some libraries automatically enable other libraries, for example, the Resource Analyzer requires a TCP/IP stack.
- **Kernel Modules** — Select modules to include them in the KernelSpace program. Some modules have dependencies that automatically cause other modules to be pulled in.

If there is a choice of libraries or modules that can be used, a Resolve Requirements dialog will open with available options to choose from, similar to the following:



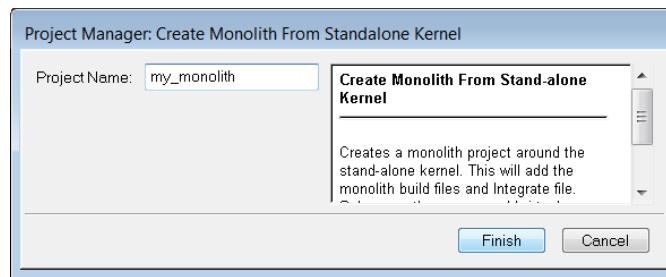
4. Click **Finish**. Your KernelSpace project will open in the MULTI Project Manager configured with the specified options.



6.6.1 Converting a KernelSpace Project to a Monolith

If you have created a standalone KernelSpace project, you can convert it to a Monolith INTEGRITY Application.

1. Right-click the Kernel project and select **Modify Project**⇒**Create Monolith from Standalone Kernel**.
2. Fill in a name for the new Monolith. It should be different than the name of the Kernel.



3. Click **Finish**. The Project Manager creates the build files and Integrate files necessary for the project to be a Monolith.
4. To revert the project, click the **Undo** button.

6.7 Adding Items to INTEGRITY Projects

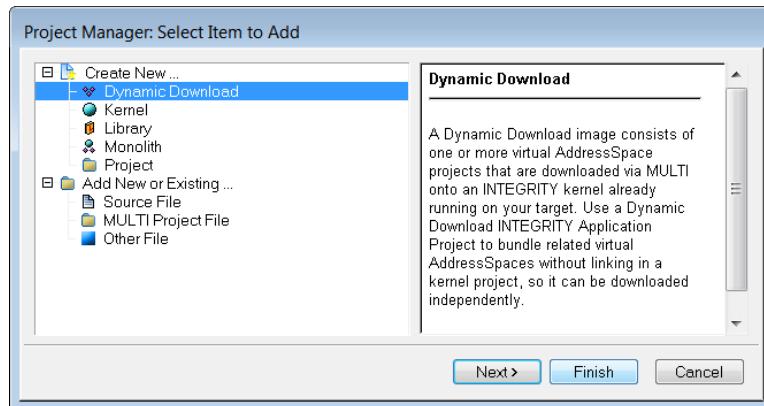
You can use the MULTI Project Manager to add items to an existing INTEGRITY project. Depending on the type of project selected, you can use the **Add Item** menu selection to add other projects, virtual AddressSpaces, OS Modules, libraries, or source files.

To add an item to an existing project:

- In the Project Manager, right-click the **.gpj** you want to add the item into and select **Modify Project**.
 - If the **.gpj** you selected is a Dynamic Download or Monolith project, you can select **Add INTEGRITY Virtual AddressSpace** or **Add INTEGRITY OS Module** from the sub-menu. See the “Adding and Configuring Virtual AddressSpaces with the Project Manager” and “Adding OS Modules with the Project Manager” sections below for more information.
 - If the **.gpj** you selected is another type of project, select **Add Item** to open the Select Item to Add dialog. The items available to add will vary depending on the type of project selected. See “Select Item to Add Dialog” below for more information.

6.7.1 Select Item to Add Dialog

The Select Item to Add dialog opens when you select **Add Item** on an existing INTEGRITY Project, or after you create an INTEGRITY Top Project with the NPW.

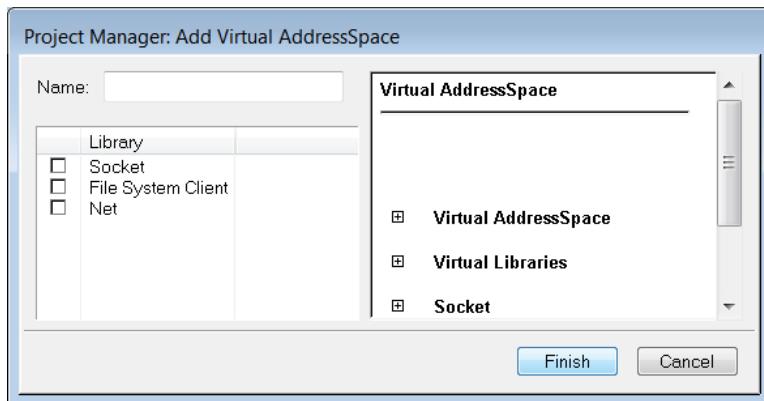


1. Select the item you want to add in this dialog. The items available to add will vary depending on what type of project this dialog was opened from. For information about the INTEGRITY Application types available under **Create New**, see the “Building INTEGRITY Applications” chapter.
2. Click **Finish** to use default settings, or **Next** to specify configuration options. For more information about configuration options, see the sections about configuring the different INTEGRITY Application types earlier in this chapter.

6.7.2 Adding and Configuring Virtual AddressSpaces with the Project Manager

To add a new virtual AddressSpace or configure an existing virtual AddressSpace in an existing Dynamic Download or Monolith INTEGRITY project:

1. Open the Dynamic Download or Monolith project in the Project Manager.
 - To add a new virtual AddressSpace, right-click the Dynamic Download or Monolith project and select **Modify Project**⇒**Add INTEGRITY Virtual AddressSpace**. The Add Virtual AddressSpace dialog opens.
 - To configure an existing virtual AddressSpace, right-click the virtual AddressSpace in the Project Manager and select **Configure**. The Settings for Virtual AddressSpace dialog opens.



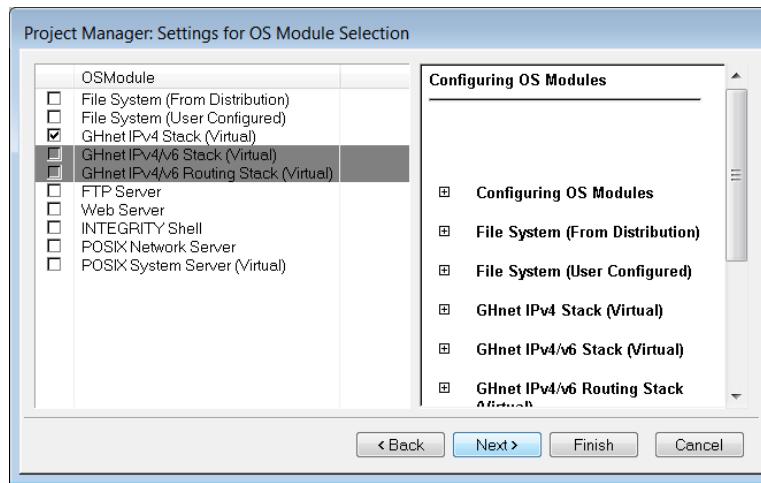
2. If you are creating a new virtual AddressSpace, enter a **Name**. Spaces and commas are delimiting characters in this dialog and should not be used in AddressSpace names. Additionally, for Debugging purposes, AddressSpace names should comply with the restrictions documented in the “Task and AddressSpace Naming” section of the “Run-Mode Debugging” chapter.
3. Select the libraries that you want to link into the virtual AddressSpace.
4. Click **Finish**. The virtual AddressSpace is added to your project or configured, and the project will open in the Project Manager.

6.7.3 Adding OS Modules with the Project Manager

To add or modify OS Modules in an existing Dynamic Download or Monolith project:

1. Open the Dynamic Download or Monolith project in the Project Manager.

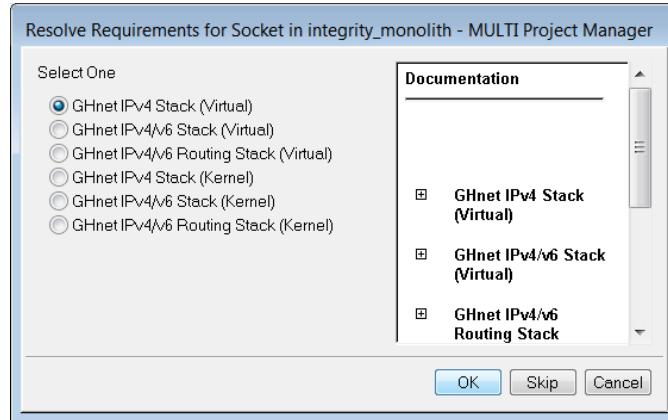
- Right-click the Dynamic Download or Monolith project and select **Modify Project⇒Add INTEGRITY OS Module**.



- Select the check box for any OS Modules you want to include in your project.

Some items are mutually exclusive, so only one can be selected at any time. If a mutually exclusive item has already been added, the invalid selections will be dimmed. Clear the check box for an item to make the mutually exclusive options available.

- Some items have dependencies that automatically cause other modules to be pulled in. If there is a choice of options that can fulfill the requirement, a Resolve Requirements or other configuration dialog will open with available options to choose from.

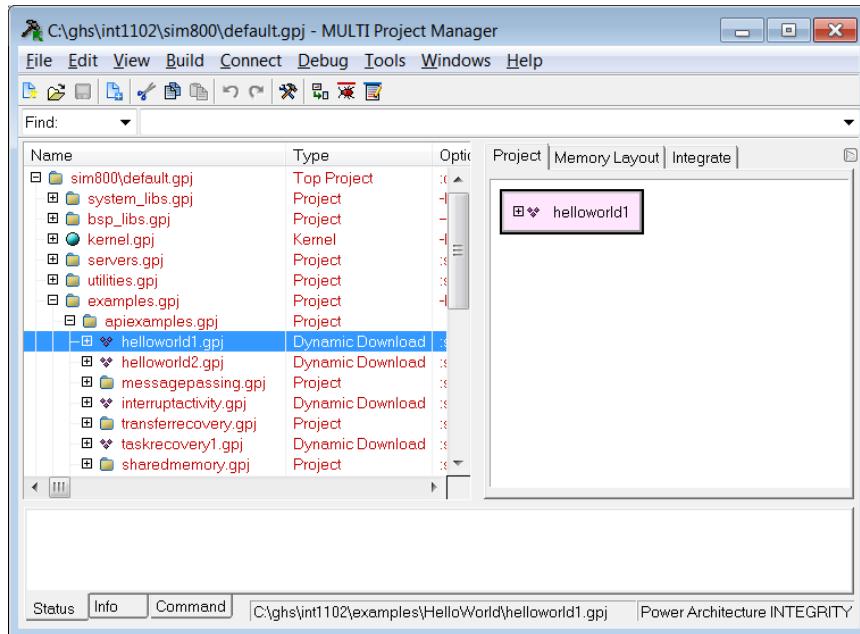


- Click **Finish**. Your Dynamic Download or Monolith will open in the MULTI Project Manager configured with the specified options.

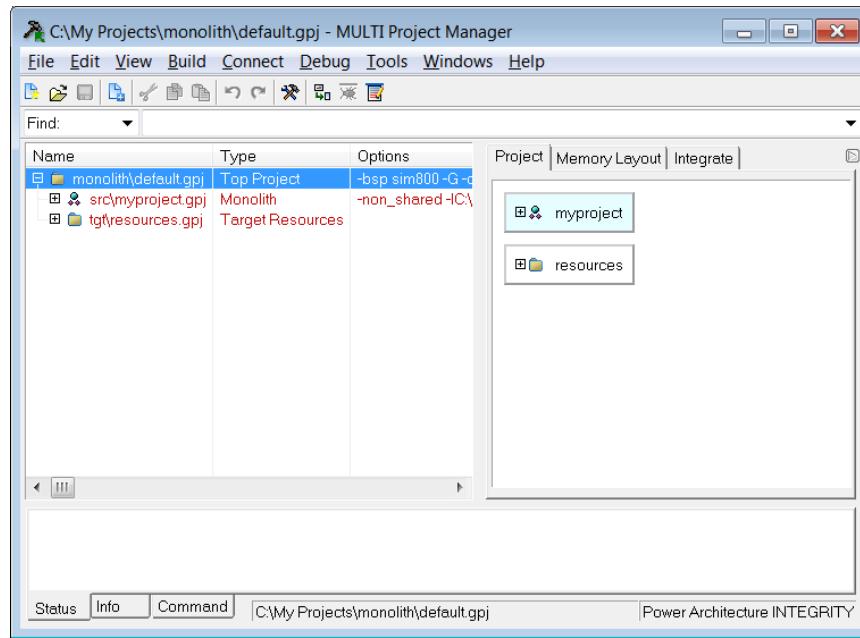
6.8 Copying Examples with the MULTI Project Manager

The standard INTEGRITY distribution contains various application examples in **examples.gpj**. If you want to modify the example without changing the installed version, you can use the MULTI Project Manager to make a local copy.

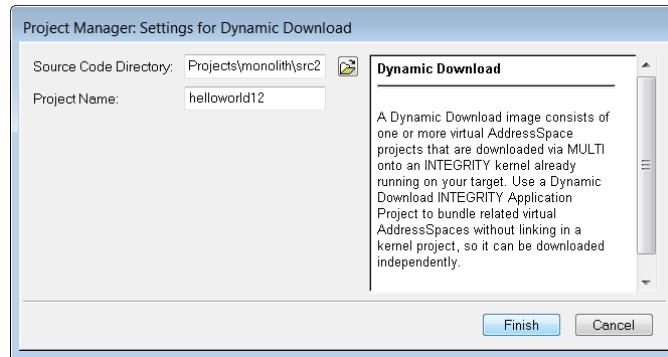
1. Using the MULTI Project Manager, create a New Top Project, or open a project you have already created.
2. Select **File⇒Open Reference BSP Project**. Select the appropriate BSP to open the example distribution in a new Project Manager window.
3. In the newly created Project Manager window, select the example you want to copy, for example, **helloworld1.gpj**.



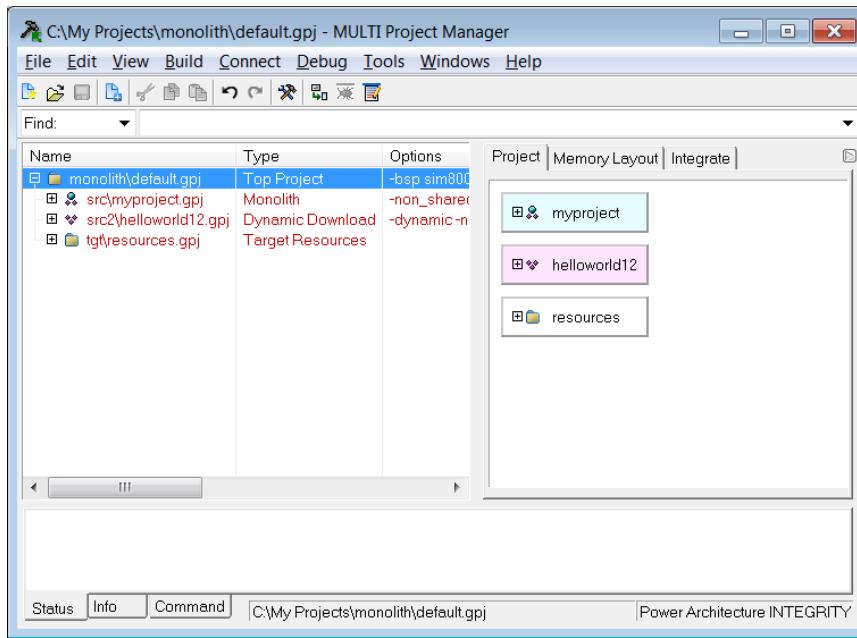
4. Select **Edit⇒Copy helloworld1.gpj Local**.
5. Go back to your original Project Manager window. Select your Top Project, then select **Edit⇒Paste Local**.



6. Use the dialog to specify the directory for the copy, and change the name of the project if desired.



7. Click **Finish**. A copy of the example is placed in our project. You can modify this version without affecting the installed version.



6.8.1 Troubleshooting Copy/Paste in the Project Manager

Generally any INTEGRITY Application can be copied and pasted into another project. The following are common problems that may be encountered when copying and pasting your own applications.

- Header files not referenced in the copied .gpj file will not be detected.
- Changing the basename of the project can cause problems with the Intex generated header file.
- If a project depends on options defined in a parent, you may have to define them by hand in the pasted version.

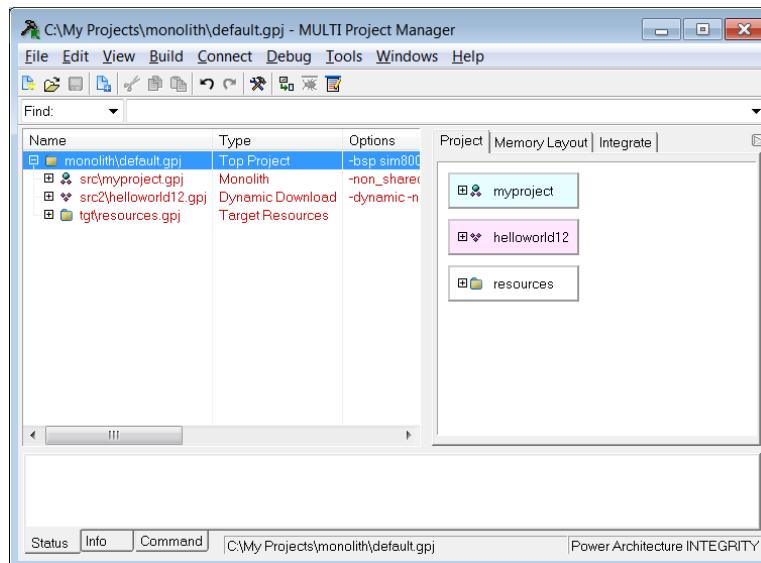
6.9 Converting and Upgrading Projects with the MULTI Project Manager

You can use the MULTI Project Manager to convert projects from one BSP to another, to convert velOSity projects to INTEGRITY, or to upgrade projects created with earlier INTEGRITY releases.

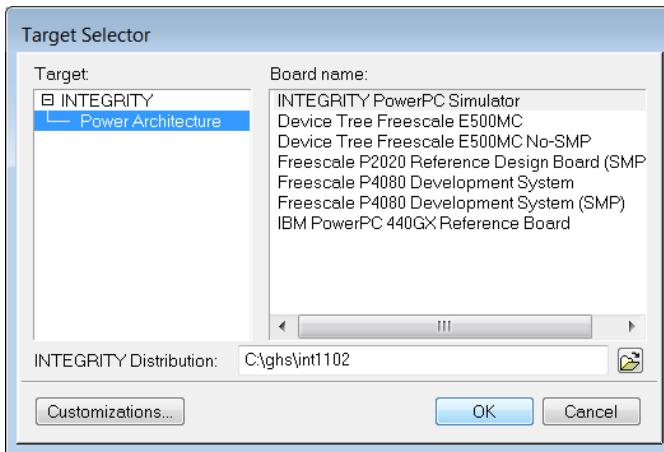
Note: If you upgrade a project to a newer INTEGRITY distribution, it will no longer work with the older distribution. If you want your project to continue working with the older version of INTEGRITY, we recommend that you copy your project directory and all its contents, and perform the upgrade on the copy.

To convert or upgrade a project:

1. In the MULTI Project Manager, open **default.gpj** for the Project you want to convert. Make sure you open **default.gpj** from the Project directory, not the version in the *bspname* directory.
2. Right-click the Top Project and select **Configure**.



3. The **Target Selector** dialog will open. Select the **Target** and **Board name** to which you want to convert.



4. Make sure you select the correct INTEGRITY installation directory in the **INTEGRITY distribution** field and click **OK**.

Your project will be converted to the selected target, board and INTEGRITY distribution.

5. If you modified or customized any of the files in the **tgt** directory of your original project, you will need to make the same changes to files in the **tgt** directory of the new, converted project.

When a project is converted, a **tgt.old** subdirectory is created in the Project directory which contains the files from the original Project's **tgt** directory. You can use the files in **tgt.old** as reference when making your customizations to the files in the new **tgt** directory.

6. If you use a customized **INTEGRITY.ld** file in your project, you will need to merge any changes between the older distribution version of **INTEGRITY.ld** and the newer distribution version of **INTEGRITY.ld** into your customized **INTEGRITY.ld** file. In particular, if you get an error message stating that the heap is missing, it is likely that the newer distribution version of **INTEGRITY.ld** has new sections or other changes that need to be merged to your custom **INTEGRITY.ld**.
7. When upgrading a project from a version prior to INTEGRITY 11 that includes a virtual TCP/IP stack, you may need to increase the **MemoryPoolSize** and **StackSize** for that **AddressSpace**. To determine the correct sizes, create a new Dynamic Download that contains the desired stack and use the fields in the **.int** file as guidelines.
8. When upgrading a project from a version prior to INTEGRITY 11 that includes a POSIX System Server and/or a User Configured File System, remove **-lshm_client** from the User Configured File System build file. This will prevent link warnings about unresolved symbols **shm_area_password** and **shm_area_name**.

Note: While the **Configure** dialog can be used to downgrade a project to an earlier INTEGRITY distribution, doing so is not recommended and additional modifications will be required.

Chapter 7

Common Application Development Issues

This chapter discusses some common issues that concern developers:

- Creating and Destroying INTEGRITY Tasks
- Application Memory Configuration
- Virtual to Physical Mapping for Code and Data
- Application Development Restrictions
- Clearing Overruns and Handling Multiple Events Simultaneously
- Weak Symbols

7.1 Creating and Destroying INTEGRITY Tasks

This section provides information about the following methods for creating and destroying INTEGRITY Tasks:

- Creating Tasks Dynamically via INTEGRITY API
- Creating Tasks Dynamically via POSIX API
- Creating Tasks Statically via Integrate
- Using Exit() vs. exit()
- Automatically Created Tasks

Different methods result in different types of Tasks. The method for destroying or exiting these Tasks depends on the method used for creation. In INTEGRITY, the functions Exit() and exit() have different behavior which varies depending upon the Task creation method.

Green Hills tools and debugging facilities carry restrictions on the naming of Tasks that are not enforced by INTEGRITY. These restrictions should be taken into consideration when naming Tasks to ensure they will be available for debugging with rtsserv2. For information, see “Task and AddressSpace Naming” in the “Connecting with INDRT2 (rtsserv2)” chapter of this manual.

7.1.1 Creating Tasks Dynamically via INTEGRITY API

Tasks can be created dynamically using the INTEGRITY API. (For detailed information about INTEGRITY Task functions, see the “Task Kernel Calls” chapter of the *INTEGRITY Kernel Reference Guide*:

- Tasks created by CreateProtectedTask() or CreateANSICTask() should be destroyed by their caller to ensure that extra per-Task memory used for the C library is recovered.

The caller uses CloseProtectedTask() or CloseANSICTask(), respectively, to close CreateProtectedTask() and CreateANSICTask().

- Tasks created using CreateProtectedTask() or CreateANSICTask() can return from their entry points. This causes them to execute the Exit() function which effectively halts the Task.

The caller can use GetTaskStatus() to detect a status of **StatExited**, which means the Task can be cleaned up with CloseProtectedTask() or CloseANSICTask().

Another way for a Task to communicate to its caller that it is ready to be reclaimed is with a Semaphore. The following is an example of the Initial Task creating three Tasks dynamically with CreateProtectedTask(), then destroying them with CloseProtectedTask(). A counting Semaphore is used to inform the Initial Task when all its children are ready to be closed:

```
#include <INTEGRITY.h>
#include <stdio.h>

static Semaphore sem;

void task(void)
{
    ReleaseSemaphore(sem);           /* I'm done! */
    Exit(0);
}

int main(void)
{
    Task children[3];
    int i;

    CreateSemaphore(0, &sem);
    /* Spawn the children */
    for (i=0; i<3; i++) {
        CreateProtectedTask(1, (Address)task, 0x2000,
                            "child", &children[i]);
        RunTask(children[i]);
    }

    /* Wait for the Tasks to finish */
```

```
    for (i=0; i<3; i++)
        WaitForSemaphore(sem);

    /* Tasks are finished and halted, clean them up */
    for (i=0; i<3; i++)
        CloseProtectedTask(children[i]);

    return 0;
}
```

7.1.2 Creating Tasks Dynamically via POSIX API

Tasks can be created dynamically using the POSIX API. (For more information, see the “Using the POSIX API and Utilities” chapter of the *INTEGRITY Libraries and Utilities User’s Guide*.)

- Tasks created with the POSIX API `pthread_create()` (also called “*threads*” or “*threads*”) should follow POSIX rules for terminating these threads. `pthread_exit()` is a common way for such threads to terminate themselves.
- A POSIX thread that simply returns from its entry point will automatically be terminated.
 - If the thread was created in a detached state, then the parent need not perform any cleanup for the child.
 - If the thread was created in a joinable state, then the parent must use `pthread_join()` to clean up after the child.

For detailed information about these calls, see the “POSIX Reference” chapter of the *INTEGRITY Libraries and Utilities Reference Guide*.

7.1.3 Creating Tasks Statically via Integrate

Tasks can be created statically using the Integrate utility. (For more information about Integrate, see the *Integrate User’s Guide*.)

By default, each virtual AddressSpace has an Initial Task. This Task is statically created and configured by Integrate. The Integrate utility provides a more advanced method of setting up an AddressSpace with other Tasks in addition to the Initial Task. These Tasks are specified in the Integrate configuration file for the INTEGRITY application project.

Integrate configuration files can also configure multiple Tasks executing at arbitrary entry points within a virtual AddressSpace. The INTEGRITY kernel ensures that the Initial Task runs completely through to `main()` before allowing any other Integrate-created Tasks in the AddressSpace to run. This prevents other Tasks from executing before the environment is fully initialized.

- If you want to be able to debug Tasks from their user-specified entry point, in the BSP description file, set `DefaultStartIt` to `false` (`default.bsp` is set up this way). When `DefaultStartIt` is `false`, Tasks are created, but halted before execution into user code. Tasks can be attached and debugged from their entry point.

- If you want to create a system that executes without intervention from the MULTI Debugger, set `DefaultStartIt` to true.

The Initial Task is created by Integrate, regardless of whether it is explicitly listed in the specification, or an Integrate configuration file is used for the application. The only reason to list the Initial Task in an Integrate configuration file is if you want to change its default configuration, such as its stack size (for more information, see “Stack Configuration” later in this chapter).

The Integrate configuration file must specify:

- Any Tasks (other than the Initial Task) that you want statically defined.
- An Object entry for each additional Task. These entries are required for Tasks to function properly, with complete access to the C/C++ run-time environment and INTEGRITY API.

The following example is a portion of an Integrate configuration file that specifies additional Tasks and corresponding Object entries. This example specifies a virtual AddressSpace (`myspace`), containing three Tasks in addition to the Initial Task. All three Tasks have the same user-code entry point (the function `taskentry`):

```
AddressSpace
  Filename          myspace
  Language          C++
  Task              task1
    EntryPoint      taskentry
  EndTask
  Task              task2
    EntryPoint      taskentry
  EndTask
  Task              task3
    EntryPoint      taskentry
  EndTask

  Object 10
    Task          task1
  EndObject
  Object 11
    Task          task2
  EndObject
  Object 12
    Task          task3
  EndObject
```

- Object numbers 10–12 are specified because in a virtual AddressSpace, Objects 1–9 are reserved by default.

After an Object number is specified, Tasks in `myspace` can pass an Object number as a Task argument in INTEGRITY API calls that take Task arguments. Using the above example, the

number 10 indicates task1. When task1 is running, calls to the INTEGRITY API function CurrentTask() will return the value 10.

When additional Tasks are specified in a virtual AddressSpace in this manner, the actual entry point for the Task is an initialization function that performs extra per-Task initialization of the C/C++ run-time environment before transferring control to the user entry point (taskentry in this example). When the system starts up, the KernelSpace Initial Task processes the Integrate BootTables and automatically creates these Tasks for the myspace AddressSpace, without any additional user code. If an Integrate configuration file is used for a Dynamic Download application, these Tasks are created when the application is loaded to the already running system and the LoaderTask processes the Integrate BootTables.

7.1.4 Using Exit() vs. exit()

Note: AddressSpaces linked with the POSIX library have different behavior for Exit() and exit(). For information, see the “Introduction to POSIX on INTEGRITY” chapter.

When a Task calls the INTEGRITY API function Exit(), the Task stops running and its status is set accordingly.

- Tasks created statically via Integrate can return from their entry points or explicitly call the INTEGRITY API function Exit().
- Tasks created by CreateProtectedTask() or CreateANSICTask() can return from their entry points or call Exit(). The calling Task terminates, but resources are not recovered as when CloseProtectedTask() or CloseANSICTask() are used. (CloseProtectedTask() and CloseANSICTask() cannot be used for statically created Integrate Tasks).

A virtual Initial Task should call exit() if, and only if, it is the last Task still running. Returning from main() is the same as calling exit(). When the Initial Task in an AddressSpace calls the C library function exit():

- All the atexit() registered functions are called.
- Access to all INTEGRITY libraries is terminated (for example, the C library, **libsocket.a**, etc.)

The Initial Task is the only Task that causes this behavior, when other Tasks call exit(), it has no effect on the AddressSpace libraries. As a result, if exit() is required, the Initial Task must be the last Task to exit() to prevent other Tasks from attempting to access libraries that have already been shut down.

Because exit() closes all file descriptors in the current AddressSpace, it is especially important that no other Tasks are running and accessing file descriptors, since the result of a file descriptor operation is boundedly undefined when the descriptor is concurrently closed.

7.1.5 Automatically Created Tasks

The following special Tasks are automatically created during initialization:

- Initial Task
- Idle Task

The following sections provide details about these Tasks that are automatically created by INTEGRITY.

7.1.5.1 Initial Task

By default, an Initial Task is automatically created in each INTEGRITY AddressSpace. The Initial Task begins execution at the main() function of each AddressSpace. KernelSpace and each virtual AddressSpace have an Initial Task, which is named **Initial** when viewed in the Task Manager.

The Initial Task in KernelSpace is created by the kernel itself and does the following:

- Performs some initialization, including processing the Integrate boot tables that are automatically composed when building an INTEGRITY application.
- Executes KernelSpace user code beginning at the function main(). If the KernelSpace program does not include a main() function, it means that no user code is executing in KernelSpace, and the KernelSpace Initial Task will terminate soon after bootup.
- Transfers control to main() immediately after bootup, and before the user has a chance to establish an INDRT (rtserv2) connection. If you want to debug code that is executing in main(), it is helpful to insert a forever loop or a HaltTask(CurrentTask()) call in the code so that the Initial Task can be attached and debugged.

The Initial Task in a virtual AddressSpace does the following:

- Performs some initialization, so that the C/C++ run-time environment and INTEGRITY API can be executed by Tasks in the AddressSpace.
- Transfers control to the function main(), where user code begins.

7.1.5.2 Idle Task

The Idle Task is automatically created during initialization. It is used to call BSP_Doze(), which places the processor into a low power mode. Idle Tasks run at priority 0 so they will not preempt any other Tasks in any AddressSpace.

In a Symmetric Multiprocessing (SMP) system, Idle Tasks are required. With SMP, there is one Idle Task per processor and the NO_IDLE_TASK option is not supported.

The Idle Task can also be used in conjunction with the **It** debug agent command to determine the fraction of time on each processor during which the system was idle. For more information, see the “Debug Agent Commands” chapter of this manual.

The Idle Task cannot call the INTEGRITY API or be used for other purposes, such as command-line procedure calls through the Debugger. Source customers can remove the Idle Task by defining NO_IDLE_TASK in the target build file and rebuilding **libkernel.a**.

7.2 Application Memory Configuration

Embedded applications often require memory in order to complete run-time operations such as task creation and stack use. This section provides information about the following memory concepts that are important because they frequently require configuration:

- MemoryPool Configuration
- ExtendedMemoryPool Configuration
- Heap Configuration
- Stack Configuration
- INTEGRITY Shared Memory Configuration

7.2.1 MemoryPool Configuration

Each AddressSpace has a MemoryPool that is managed by the INTEGRITY kernel. The MemoryPool is used for allocations of kernel Objects and other data structures that are managed by the INTEGRITY kernel on behalf of applications. For example, when the kernel receives a request to create a new Task, the kernel allocates a Task Object and a Task Control Block (TCB) to manage the Task. The memory to hold the Object and TCB comes from the AddressSpace's MemoryPool. By satisfying an AddressSpace's request for kernel allocations from the AddressSpace's MemoryPool, the INTEGRITY kernel guarantees that no AddressSpace can adversely affect the availability of these resources in other AddressSpaces.

Guaranteed resource availability is one of the key reliability and security features of INTEGRITY. This method contrasts with most other real-time operating system implementations, which use a central store for kernel allocations. Using a central store makes it possible for errant applications in those operating systems to exhaust memory and cause failures in unrelated applications. INTEGRITY prevents this from occurring by using a MemoryPool.

When an AddressSpace is created, the physical memory that makes up an AddressSpace's MemoryPool is divided into memory pages and placed on the AddressSpace's free list. Kernel service calls (such as calls to create a new Task) automatically use memory from the free list. In addition, an AddressSpace can obtain (and return) pages from its free list by using the `GetPageFromAddressSpaceFreeList()` and `PutPageOnAddressSpaceFreeList()` API calls (although most applications do not need to access this memory directly). For more information about these API calls, see the “AddressSpace and Memory Mapping Kernel Calls” chapter of the *INTEGRITY Kernel Reference Guide*.

7.2.1.1 MemoryPool Size Configuration

By default, virtual AddressSpaces have a MemoryPool size of 64KB. MemoryPools are statically configured by the Integrate utility at system build time. AddressSpaces requiring a different MemoryPool size can specify one in the Integrate Configuration File with the `MemoryPoolSize` keyword. For more information about using Integrate Configuration Files, see the “Building INTEGRITY Applications” of this manual and the *Integrate User’s Guide*.

Note: Executable code and data that make up an AddressSpace are pre-allocated by Integrate and do not take up part of the user-specified MemoryPool size.

7.2.2 ExtendedMemoryPool Configuration

Note: Using ExtendedMemoryPool is not recommended on targets with no extended memory, such as 64-bit targets. On such targets, increasing the size of the MemoryPool is more efficient than using an ExtendedMemoryPool.

Each AddressSpace has an ExtendedMemoryPool that is managed through a global MemoryRegionPool named `_ghs_ExtendedPhysicalMemoryRegionPool`. An AddressSpace's ExtendedMemoryPool differs from its MemoryPool in several ways.

First, the MemoryPool contains only kernel RAM, while the ExtendedMemoryPool contains both kernel RAM and extended RAM. Kernel RAM is general-purpose RAM that is suitable for use for kernel data structures such as Objects (which, among other things, requires that it be permanently mapped into KernelSpace). Extended RAM is general-purpose RAM that is not kernel RAM. On 32-bit processors that support physical address widths greater than 32 bits, it is possible for nearly all of the RAM to be extended RAM, in which case the ExtendedMemoryPool is capable of holding much more RAM than the MemoryPool.

Second, the MemoryPool is maintained by the kernel as part of the AddressSpace, while the ExtendedMemoryPool is maintained as a MemoryRegionPool under the user's control. The MemoryPool is accessed through kernel calls (such as `GetPageFromAddressSpaceFreeList()` and `PutPageOnAddressSpaceFreeList()`) on the AddressSpace. The ExtendedMemoryPool is accessed through MemoryRegionPool API calls (such as `AllocateAlignedMemoryRegion()` and `DeallocateAlignedMemoryRegion()`) on `_ghs_ExtendedPhysicalMemoryRegionPool`. Further, all RAM in the MemoryPool is carved up into pages, while RAM in the ExtendedMemoryPool can consist of blocks of arbitrary varying sizes.

Third, when a page of RAM is allocated from the MemoryPool, its contents are all zeros, and when a page of RAM is returned to the MemoryPool, its contents are cleared to zero. Conversely, when a block of RAM is allocated from the ExtendedMemoryPool, its contents are not necessarily all zeros, and when a block of RAM is returned to the ExtendedMemoryPool, its contents are not cleared to zero.

The uses of extended RAM from an AddressSpace's ExtendedMemoryPool are somewhat limited. Extended RAM cannot be used for kernel data structures of any kind. Further, many IODevices will reject extended RAM. However, the ExtendedMemoryPool is automatically used to satisfy certain allocations:

- In the process of extending its heap, a virtual AddressSpace always tries to allocate RAM from its ExtendedMemoryPool first. Thus, any virtual AddressSpace with a non-empty ExtendedMemoryPool can use extended RAM for its heap.
- The loader automatically uses RAM from the loader AddressSpace's ExtendedMemoryPool to attempt to satisfy ExtendedMemoryPool requirements of loaded AddressSpaces.

Physical memory is referred to as “kernel physical memory” if it is permanently mapped into KernelSpace, and is referred to as “extended physical memory” otherwise. A system can include physical memory that is not RAM, for example, ROM or memory-mapped registers for an I/O device. However, an AddressSpace's MemoryPool will never contain physical memory that is not kernel RAM. Similarly, an AddressSpace's ExtendedMemoryPool must never contain physical

memory (kernel or extended) that is not RAM. Consequently, the application must never add a MemoryRegion that represents non-RAM physical memory to `--ghs_ExtendedPhysicalMemoryRegionPool`.

7.2.2.1 ExtendedMemoryPool Size Configuration

Virtual AddressSpaces have an ExtendedMemoryPool size of zero, by default. ExtendedMemoryPools are statically configured by the Integrate utility at system build time. AddressSpaces requiring a non-zero ExtendedMemoryPool size can specify one in the Integrate Configuration File with the ExtendedMemoryPoolSize keyword. For more information about using Integrate Configuration Files, see the “Building INTEGRITY Applications” of this manual and the *Integrate User’s Guide*.

7.2.3 Heap Configuration

Heap memory is memory that is dynamically allocated and freed by the high-level language runtime library functions (such as, the C++ functions `malloc()`, `free()`, `new()`, and `delete()`). AddressSpaces do not need a heap if they do not contain any code using these runtime library functions.

Unlike the MemoryPool, the INTEGRITY kernel does not manage the AddressSpace’s runtime heap. Heap is wholly managed within the runtime libraries that run in user space with the application. The Green Hills toolchain that comes with INTEGRITY provides default implementations of these runtime library functions.

The Green Hills TCP/IP stack also uses heap memory for internal buffer allocation, and for sockets communication. For more information, see the “Using GHnet v2” chapter of the *INTEGRITY Networking Guide*.

7.2.3.1 Heap Size Configuration

The size of the pre-allocated segment memory used for the heap is specified by the Integrate utility at system build time. The default heap size is 64KB. To override this default for any individual AddressSpace, use the HeapSize keyword in the application’s Integrate Configuration File (for more information, see “Integrate Configuration File” in the “Building INTEGRITY Applications” chapter). For example:

```
AddressSpace
  Filename      myinspace
  Language      C
  HeapSize      0x3000
EndAddressSpace
```

The heap size for KernelSpace is configured differently. The linker directives file (typically, `ram_sections.ld`) for a KernelSpace project contains a `.heap` section, whose size is specified with a `pad` directive, as follows:

```
.heap      align(16)      pad(__INTEGRITY_HeapSize)      NOCLEAR      :
```

To change the size of the KernelSpace heap, change the value of the `__INTEGRITY_HeapSize` constant, which is generally defined in the `DEFAULTS` block of your `default.ld` file.

Alternatively, you can use a linker constant to adjust the size of the kernel heap on a per build file basis:

1. In the Project Manager, right-click the kernel build file and select **Edit⇒Set Build Options**.
2. Select the **All Options** tab. In the **Linker** category, double-click **Additional Linker Options (before start file)**.
3. In the Edit List Option dialog, enter:
`-C __INTEGRITY_HeapSize=0x80000.`
4. Click **OK** to close the dialog, and **Done** to close the Build Options window.

This adds the following line to your kernel build file:

```
-lnk=-C __INTEGRITY_HeapSize=0x80000
```

7.2.3.2 Heap Growth

When an AddressSpace exhausts its pre-allocated heap, and application code requests more dynamically allocated memory, the Green Hills runtime libraries are capable of growing the heap to handle the additional memory requirement.

The `sbrk` library routine grows the heap by obtaining physical pages from the AddressSpace's `MemoryPool` and `ExtendedMemoryPool` and mapping them to a chunk of unused virtual memory within the AddressSpace. The pages are mapped into a reserved region of virtual memory adjacent to the end of the pre-allocated heap section, allowing the heap to grow contiguously. The size of this region can be set using the `HeapExtensionReservedSize` keyword in the application's `Integrate` configuration file. If application code continues to request more memory, the `MemoryPool` and `ExtendedMemoryPool` will eventually become exhausted. Once the AddressSpace's `MemoryPools` are exhausted, no more heap can be allocated.

If an AddressSpace's reserved heap extension region is smaller than its `MemoryPool`, then it is possible for the heap extension region to be exhausted before the `MemoryPool`, so that it will no longer be possible to map physical pages from the `MemoryPool` contiguous to the existing heap. Unless the `HEAP_EXTEND_CONTIGUOUS()` macro is used, the physical pages are then mapped elsewhere. As a result, the new memory from the `MemoryPool` becomes the new heap, and unused memory at the end of the pre-allocated heap remains unused.

While this feature provides additional flexibility for application designers, the fact that `MemoryPool` can be used for heap as well as for kernel operations must be taken into consideration when deciding how large to make the `MemoryPool`. Automatic heap growth can be disabled for an AddressSpace by placing the `DISABLE_HEAP_EXTEND()` macro at file scope.

By defining the HEAP_EXTEND_CONTIGUOUS() macro at file scope, heap growth can be limited only to the heap extension region, which is contiguous with the initial heap. In addition to preventing unused areas of the heap from being lost by preventing the heap from moving, it also bounds the amount of physical memory resources that can be used for heap growth. This behavior will become the default in future releases. Namely, support for heap expansion into any available, but discontiguous virtual region will be removed.

While allowing the heap to be backed by extended physical MemoryRegions allows much larger amounts of dynamic memory to be used, the maximum size of a single dynamic allocation is still limited by the implementation of library functions, such as malloc() and calloc(), which are documented in the “Libraries and Header Files” chapter of the *MULTI: Building Applications* manual for your target architecture.

For systems supporting large pages, when the PreferredPageSize keyword is added to your BSP’s **default.bsp** file, heap extensions may be rounded up to the nearest preferred page size for more efficient mapping. For more information about the PreferredPageSize Integrate keyword, see the “Target Section” chapter in the *Integrate User’s Guide*.

Note: You cannot return memory to the MemoryPool by calling the free() runtime library function. Calls to free() will return memory to the malloc() library’s free list, not the MemoryPool.

7.2.4 Stack Configuration

Tasks require a run-time stack in order to execute. Determining task stack size is a common and sometimes troublesome configuration issue. If you specify a stack size that is too large, it results in memory waste. If you specify a stack that is too small, it results in stack overflow and failed execution.

Green Hills tools and INTEGRITY provide mechanisms that make it easier to determine optimal stack size, observe stack size usage during execution, and immediately detect overflows at run-time as they occur, rather than waiting for a subtle corruption to manifest itself at some later point. This functionality can save an enormous amount of time during development, and improve time to market.

The specification of stack sizes depends on how the Task was created. There are two types of tasks in INTEGRITY:

- Statically Defined Tasks — defined and configured using Integrate.
- Dynamically Created Tasks — created and configured using INTEGRITY Task or POSIX pthread APIs.

The following sections provide information about configuring stack size for both types of Tasks.

7.2.4.1 Statically Defined Tasks

In INTEGRITY, Tasks can be statically defined and configured using an Integrate Configuration File or the Integrate GUI. The Initial Task, which executes the main() function of an

AddressSpace, is a statically defined Task. The stack size for statically defined Tasks must also be specified statically. The default stack size is 4KB. You can override the default stack size for either the entire application, or for an individual task.

To override the default stack for the entire application (that is, all statically created Integrate Tasks):

- Use the DefaultStackLength keyword in a BSP description file (**.bsp**) used for the application. (If DefaultStackLength is not specified in the **.bsp** file, stack size defaults to PageSize.)

For example, to set the default stack size to 8 KB for all statically created Tasks in all the INTEGRITY applications, add the following line to the BSP description file used for the INTEGRITY application:

```
DefaultStackLength          0x2000
```

To override the default stack size for an individual Task that was statically created:

- Use the StackLength keyword in the Integrate configuration file (**.int**) used for the INTEGRITY application.

For example, to set the stack size to 8 KB for the Initial Task in the AddressSpace whose virtual AddressSpace program (a C code application) is named myprog:

```
AddressSpace
  Filename      myprog
  Language      C
  Task Initial
    StackLength   0x2000
  EndTask
EndAddressSpace
```

To set the stack size to 8 KB for Task foo:

```
Task      foo
  EntryPoint  foofunc
  StackLength 0x2000
EndTask
```

For more information about Task specification in an Integrate Configuration File, see the “Task Section” chapter of the *Integrate User’s Guide*.

Tasks can also be configured using the Integrate GUI. The Integrate Task configuration window contains a field to specify the stack size. The StackLength specification overrides any previous default set by Integrate, or by a BSP description file used for the application.

There are two special stacks that cannot be overridden with these methods:

- The stack location and size of the KernelSpace Initial Task is determined by the parameters of the .stack section of the KernelSpace program's linker directives file. This should not be modified because this is a system Task whose stack requirements have already been determined on a per-processor architecture basis.
- The stack used for the kernel itself, and for interrupt service routines, is specified by the .kstack section of the KernelSpace program's linker directives file. This section specifies the size and location of this stack and is used for the execution of the scheduler, interrupt service routines, and callback functions. Normally, the size of this section should not need modification.

7.2.4.2 Dynamically Created Tasks

Tasks can be created dynamically, using INTEGRITY Task creation APIs, such as CommonCreateTask(), or using POSIX pthread task creation APIs, such as pthread_create(). When Tasks are created this way, the stack size is specified as part of the API call.

When using INTEGRITY Task creation APIs, the stack size is specified as the third parameter the call, for example:

```
Error CommonCreateTask(Value Priority, Address EntryPoint,
    Address StackSize, char *Name, Task *NewTask);
Error CommonCreateTaskWithArgument(Value Priority,
    TASKENTRYPOINT EntryPoint, Address Argument, Address StackSize,
    char *Name, Task *NewTask);
```

For more information about these calls, see the “Task Kernel Calls” chapter of the *INTEGRITY Kernel Reference Guide*.

When using the POSIX API, tasks are created with pthread_create(). The default stack size for tasks created with pthread_create() is 8KB. This default stack size can be overridden with the pthread_attr_setstacksize() API call. For more information about POSIX APIs, see the POSIX chapters in the *INTEGRITY Libraries and Utilities User’s Guide* and the *Libraries and Utilities Reference Guide*.

When a Task is created in a virtual AddressSpace using CommonCreateTask(), CommonCreateTaskWithArgument(), or pthread_create(), the stack is allocated by grabbing free physical pages from the AddressSpace MemoryPool and mapping them to unused virtual memory. Thus, the user-specified StackSize is rounded up to the nearest page size (typically 4KB for CPUs with an MMU). System designers must consider stack size usage when configuring the MemoryPool size for the AddressSpace.

When CommonCreateTask(), CommonCreateTaskWithArgument(), or pthread_create() is executed from KernelSpace, the stack is allocated from the KernelSpace heap (specified by the .heap section in the KernelSpace project's linker directives file). Therefore, the stack sizes of KernelSpace Tasks must be considered when configuring the KernelSpace heap.

7.2.4.3 Using the gstack Utility to Determine Stack Size Requirement

Green Hills tools include **gstack**, a utility that displays the maximum possible stack size given any execution entry point. This utility helps developers calculate how large to make a Task's stack. This utility does not work if the program uses function pointers (common with C++) or recursion. For more information about **gstack**, see the “The gstack Utility Program” in the *MULTI: Building Applications* manual.

Sometimes the need for a larger stack becomes obvious when the application fails due to a stack overflow. Detecting stack overflow is important in field systems, where corrective measures may need to be taken to keep the system functional. Detecting stack overflow is also important during development as a productivity tool. Without good stack overflow detection facilities, an overflow can cause devious corruption that cost developers time to market. For more information regarding stack overflow considerations, see “Detecting Stack Overflow” in the “Run-Time Error Checking” chapter of this manual.

To determine the maximum possible stack size required, given a particular entry point in the code, use the **gstack** utility as follows:

```
gstack -s myentrypoint myprog

Task myentrypoint, 232 byte stack produced by the call chain of:

Framesize      Function
-----        -----
24            main
8             HangUp
0             CloseConnection
16            Close
```

In the above example, `myprog` is the name of the virtual AddressSpace program or KernelSpace program.

Note: `gstack` does not take into account recursive functions or calls through function pointers.

7.2.4.4 Observing Stack Usage During Execution

INTEGRITY includes two methods that help estimate the maximum stack usage of tasks at run-time.

One method samples each Task's stack pointer value. You can use this method to view high water mark information while debugging with `rtserv2` by using the `It` debug agent command. For more information, see the “Debug Agent Commands” chapter.

The other method stores a known pattern into each Task's stack area when the Task is created. The high water mark of the stack can then be tracked during the lifetime of the Task. To use this method, you must have stack tracking enabled.

When using stack tracking, the OSA Explorer displays information about each Task, including stack location, size, and high water mark while the target is in `SysHalt` mode. This visibility into the stack usage while the application is being developed helps designers choose appropriate stack

sizes for their production systems. For more information, see the “OSA Explorer Task Tab” in the “Object Structure Aware Debugging” chapter of this manual.

Note: The sampling method and the stack tracking method each use different sampling mechanisms and may produce different estimates.

You can retrieve stack usage statistics produced by either of these methods programmatically. For more information, see “Retrieving Stack Usage Statistics Programmatically” in the *INTEGRITY Libraries and Utilities Reference Guide*.

Stack tracking does incur some run-time overhead. Unless your performance requirements are generous, stack tracking is suggested for debugging purposes only. Stack tracking is disabled in kernel libraries built with NO_STACK_TRACKING. See “Choosing Your Kernel” in the *INTEGRITY Kernel User’s Guide* for more information.

7.2.4.5 Stack Overflow Detection

It is important to detect that a task has insufficient stack space prior to fielding a product. Yet, it is quite possible that a stack overflow could go unnoticed during development if the operating system does not provide a mechanism for detecting when this occurs.

On many operating systems, when a stack exhausts its allocated space, it simply starts to overwrite whatever memory happens to reside below the stack segment. Depending on what resides below the stack, the overflow may not be immediately fatal. It could cause subtle corruptions that manifest themselves in seemingly unrelated places. Most embedded developers have spent some very painful hours trying to track down these kinds of failures.

INTEGRITY solves this problem by using virtual memory to implement a guard page. The guard page is a page (or more) of unmapped virtual memory below each stack segment.

Statically-defined INTEGRITY tasks, and tasks created dynamically in a virtual AddressSpace using the CommonCreateTask() and CommonCreateTaskWithArgument() APIs are automatically provided with guard pages below their stacks. When a task overflows its stack, it writes into the guard page, causing a hardware MMU exception. The task is immediately suspended by the kernel. When debugging, the task’s status in the Task Manager will display SUSPENDED, which makes it easy to see that something went wrong.

In addition, if debugging the task, the Debugger prints a message indicating that it encountered an exception. The instruction causing the exception is typically in the prologue of a function (where stack space is allocated), and the instruction is usually a store through the stack pointer. This immediate feedback allows developers to find stack overflows and correct them, saving development time, and time to market.

7.2.5 INTEGRITY Shared Memory Configuration

The INTEGRITY model encourages system designers to modularize their system into independent virtual AddressSpaces. If the AddressSpaces must communicate with each other, the suggested method is to use INTEGRITY Connections. This method generally yields the most robust, maintainable, reliable, and secure system overall.

However, some applications may require the use of shared memory between AddressSpaces. Using shared memory typically represents a trade-off between performance and

security/reliability. With INTEGRITY, there are two ways to configure shared memory areas:

- Dynamically — using the POSIX mmap() API. For information, see “POSIX Shared Memory” in the “Using the POSIX API and Utilities” chapter of the *INTEGRITY Libraries and Utilities User’s Guide*.
- Statically — using Integrate. This method is described in the following section.

7.2.5.1 Using Integrate to Configure Statically Shared Memory

With INTEGRITY it is possible to allocate a fixed size physical memory region that is mapped into and shared by multiple virtual AddressSpaces. The following is an example Integrate configuration file that is appropriate for use in an INTEGRITY application project:

```

Kernel
  Filename kernel
  Object 25
    MemoryRegion
      Name          CommonName
      Length        0x1000
    EndObject
  EndKernel

  AddressSpace as1
    Filename as1
    Language C
    Object
      MemoryRegion  as1
      MapTo         CommonName
      Start         0x8000000
      Length        0x1000
    EndObject
  EndAddressSpace

  AddressSpace as2
    Filename as2
    Language C
    Object
      MemoryRegion  as2
      MapTo         CommonName
      Start         0x8000000
      Length        0x1000
    EndObject
  EndAddressSpace

```

In this example:

- A 4 KB size MemoryRegion is allocated in KernelSpace and tagged with the name CommonName
- The MemoryRegion Object is 25 (Object numbers specified for KernelSpace must be higher than the number specified by InitialKernelObjects in the BSP description file used for the application).

- The actual physical location of this MemoryRegion is unspecified; Integrate will choose an appropriate available chunk.
- Two virtual AddressSpaces, as1 and as2, both have CommonName mapped at virtual address 0x8000000. As a result, Tasks created in either AddressSpace can access addresses 0x8000000 to 0x8000fff, and such accesses will be to the same shared memory.

Shared access is not automatically synchronized, an INTEGRITY Binary Semaphore accessible to both AddressSpaces could be used for this. On some processors (such as ARM processors where the cache is tagged with virtual addresses) user code must insert an explicit cache flush in order to guarantee that a write in one AddressSpace can be read correctly in another AddressSpace.

7.3 Virtual to Physical Mapping for Code and Data

INTEGRITY provides complete memory protection and partitioning of applications at the AddressSpace level when run on a CPU with a memory management unit (MMU). Each virtual AddressSpace is completely isolated and protected from other AddressSpaces, and the special kernel AddressSpace (also known as KernelSpace) is protected from the virtual AddressSpaces. Designers can install shared memory areas and communication channels between protected AddressSpaces, if desired.

Unlike other operating systems, which employ a single flat physical memory space for the kernel code, data segments, and the segments of all applications, INTEGRITY employs true virtual memory for its virtual AddressSpaces. For example, the text segment of all virtual AddressSpaces typically starts at the same virtual address.

The following sections describe the two types of INTEGRITY applications that include virtual AddressSpaces: INTEGRITY Monolith applications, and Dynamic Download applications.

7.3.1 Virtual to Physical Mapping for Monolith Applications

A monolith consists of an INTEGRITY kernel and a set of virtual AddressSpaces built into a single image by the Integrate utility. Running Integrate is part of the build process.

In a monolith application, Integrate assigns physical memory to each virtual memory segment required by the constituent virtual AddressSpaces. Each virtual segment (for example a text segment or data segment) is mapped to a contiguous chunk of physical memory. You can view this kernel to virtual mapping by running the **gdump** utility on the monolith image, as follows:

```
gdump -virtual_mapping myimage

Virtual          Physical          Size Section
=====  ======  ======  =====
0x00010000  0x00361000  0x00013000 .philosopher.text
0x00024000  0x00374000  0x00003000 .philosopher.data
0x00028000  0x00377000  0x00001000 .Initial.stack
0x00029000  0x00378000  0x00010000 .philosopher.heap
0x00010000  0x00388000  0x00013000 .producer.text
0x00024000  0x0039b000  0x00003000 .producer.data
0x00028000  0x0039e000  0x00001000 .Initial.stack
0x00029000  0x0039f000  0x00010000 .producer.heap
```

In the above display:

- The philosopher virtual AddressSpace's text segment starts at virtual address 0x10000 and is mapped to kernel address 0x361000.
- Both the philosopher and producer AddressSpaces predefine a single task, the InitialTask, which is always created for any AddressSpace.
- The stack segments of these tasks are denoted by the .Initial.stack sections.

Integrate includes the mapping information as part of a special BootTable section, which is part of the final load image. On system startup, INTEGRITY reads this BootTable and creates the

mappings as each AddressSpace is booted. In general, users do not need to be concerned with the actual physical memory locations of AddressSpaces.

For more information about the Integrate utility and its role in the application build process, consult the *Integrate User’s Guide*. For more information about Monoliths, see “Monolith INTEGRITY Application Project” in the “Building INTEGRITY Applications” chapter.

7.3.2 Virtual to Physical Mapping for Dynamic Download Application

Integrate is used to build the image that makes up a dynamic download application. A dynamic download application does not include a kernel. Instead of specifying virtual to physical mappings for each virtual segment, Integrate specifies the virtual address of each segment, and its corresponding size.

When a dynamic download application is loaded by the INTEGRITY dynamic loader, the loader processes the BootTable information and creates the virtual to physical mappings for each new AddressSpace. The loader grabs free physical pages for each page of virtual memory needed. Therefore, the virtual to physical mappings are not contiguous as they are with a monolith.

For more information about dynamic download applications, see “Dynamic Download INTEGRITY Application Project” in the “Building INTEGRITY Applications” chapter.

7.4 Application Development Restrictions

This section provides information about restrictions affecting INTEGRITY application development.

7.4.1 Restrictions on Hand-Written Assembly Code

Except on x86 and x64 processors, INTEGRITY uses a single global register variable. This implies that hand-written assembly code that exists within a C Library Enabled Task must not use or modify the first global register (as defined by the documentation for the **-globalreg** compiler option).

7.4.2 Restrictions on MULTI Builder and Driver Options

Changing MULTI Optimization options from the defaults shipped in INTEGRITY **default.gpj** files can alter the code behavior in ways that are untested by Green Hills Software. Making changes to Optimization options is not supported, and is performed at the user's own risk.

Changing the Debugging options, for example changing the debugging level from **-G** to **-gs**, can also alter code behavior in ways that are not supported. For more information, see the documentation for the **-consistentcode** option in the *MULTI: Building Applications* book.

The Special Data Area build options available in MULTI are not supported by INTEGRITY. These options include Small Data Area (**-sda**), Small Data Area with Threshold (**-sda=size**), and Zero Data Area with Threshold (**-zda=size**). Do not use or modify these options when building INTEGRITY applications.

7.4.3 MULTI Builder and INTEGRITY Documentation Restrictions

Some of the information in *MULTI: Building Applications* is not pertinent to INTEGRITY users, or superseded by information in INTEGRITY manuals:

- The command-line driver descriptions are geared towards building applications for stand-alone use. Although many of the processor-specific driver options are supported on INTEGRITY systems, this information is superseded by the information in the “Using the Command-line Driver” section of this manual.
- Information about reentrancy of runtime libraries in the “Libraries and Header Files” chapter is augmented by the “Thread-Safety of Run-time Libraries” section of the *INTEGRITY Libraries and Utilities User’s Guide*.
- The “Customizing the Green Hills Run-Time Environment” section of the “Libraries and Header Files” chapter is not applicable to INTEGRITY users because INTEGRITY provides a full-featured run-time environment out of the box for developers.
- Some of the special embedded systems features described in *MULTI: Building Applications*, such as position independent code, are not pertinent or supported on INTEGRITY systems.

7.4.4 Restrictions on KernelSpace Application Code

Except where stated otherwise in the documentation, all privileged instructions, registers, and facilities are for the exclusive use of the INTEGRITY kernel and ASPs. It is illegal for an IODeviceVector member function, a TimerDriver member function, a synchronous callback, or KernelSpace Task to use privileged instructions, registers, or facilities unless their use is either explicitly authorized by the documentation, or performed in such a manner that the INTEGRITY kernel and ASPs are unable to observe their use.

Note: Privileged instructions, registers, and facilities are those instructions, registers, and facilities that cannot be used by a Task in a virtual AddressSpace, whether because they cause exceptions or because they have no effect when executed in a virtual AddressSpace.

For example, it is illegal to invoke an INTEGRITY API call from a KernelSpace Task with interrupts disabled, and it is illegal to invoke a documented kernel entry point from a synchronous callback, a TimerDriver member function, or an IODeviceVector member function with interrupts disabled.

7.5 Clearing Overruns and Handling Multiple Events Simultaneously

It is a good programming practice for a Task's event loop to clear overruns and handle multiple events simultaneously when receiving on Objects that can accumulate overruns. This practice helps ensure that if a Task is not scheduled for a long time (for example, because the Task is halted by a debugger or prevented from running by some higher-priority Task), when the Task does get to run again, it will not spin for a long period of time catching up on all the messages it missed.

For example, if a Task calls SetClockAlarm() to set a repeating alarm on a Clock and then repeatedly calls AsynchronousReceive() to wait for alarms on the Clock, each time WaitForActivityAndReturnStatus() (or an analogous function) notifies the Task that an alarm has expired on the Clock, the Task should call ClearClockAlarmOverruns() to clear all remaining alarms on the Task and handle all the alarms together. The following code snippet illustrates this practice:

```
void HandleTimerEvents(Value NumberOfTimerEvents);

void main(void)
{
    ...

    Clock TimerClock;
    Activity TimerActivity;
    Time TimerInterval;

    ...

    CheckSuccess(CreateVirtualClock(HighestResStandardClock,
                                    CLOCK_ALARM, &TimerClock));
    CheckSuccess(CreateActivity(CurrentTask(),
                               TIMER_ACTIVITY_PRIORITY, false, TIMER_ACTIVITY_ID,
                               &TimerActivity));

    CheckSuccess(SetClockAlarm(TimerClock,
                           true, NULLTime, &TimerInterval));
    CheckSuccess(AsynchronousReceive(TimerActivity,
                                    (Object)TimerClock, NULL));

    ...

    while (1) {
        Value Id;
        Error Status;

        WaitForActivityAndReturnStatus(&Id, &Status);

        if (...) {
            ...
        } else if (Id == TIMER_ACTIVITY_ID) {
            Value Overruns;
            CheckSuccess(ClearClockAlarmOverruns(TimerClock, &Overruns));
            HandleTimerEvents(1 + Overruns);
            CheckSuccess(AsynchronousReceive(TimerActivity,
```

```
        (Object)TimerClock, NULL));
    } else if (...) {
        ...
    }
}
```

Note: The intention here is handling multiple events simultaneously, not just receiving multiple events from the Object simultaneously. Clearing the overruns in the event loop is not necessarily enough. In particular, if the handler for a block of overruns on a single Object takes a long time to complete, other Activities in the Task that have equal or greater priority may be wrongly starved of handling.

The following implementation of the HandleTimerEvents() function is an example of handling multiple events simultaneously:

```
void DoHighFrequencyPolling(void);
void DoLowFrequencyPolling(void);

void HandleTimerEvents(Value NumberOfTimerEvents)
{
    static LowFreqCount = 0;

    DoHighFrequencyPolling();

    LowFreqCount += NumberOfTimerEvents;
    if (LowFreqCount >= 100) {
        LowFreqCount = 0;
        DoLowFrequencyPolling();
    }
}
```

7.6 Weak Symbols

The INTEGRITY RTOS makes extensive use of weak externals to improve modularity and simplify configuration. A weak external allows a program to reference a function or variable whether or not it is included in the final linked executable without causing a link error. Symbols which are weakly defined and which are linked into the executable behave exactly the same as normal external symbols; their addresses are resolved by the linker. Weak externals which do not exist in the final executable have their addresses forced to null (zero), which can be checked at run time.

To declare a symbol as a weak external, the following pragma is used:

```
#pragma weak FunctionName
```

The most common use of weak externals in INTEGRITY is to allow certain functionality to be included in the final executable at link time rather than at compile time. It is important to wrap each call to a weak external with a test to determine if that function exists. The following example demonstrates how weak externals are used:

```
if (FunctionName)
    /* FunctionName exists, so call it */
    FunctionName(arg);
```


Chapter 8

ISIM - INTEGRITY Simulator

ISIM is a simulation environment for the INTEGRITY Real-Time Operating System. This chapter provides information on the following topics:

- Introduction to ISIM
- ISIM Features
- ISIM Target BSPs
- Running ISIM
- ISIM Socket Emulation
- ISIM Caveats

8.1 Introduction to ISIM

With ISIM, programmers can develop and test embedded INTEGRITY-based applications when target hardware is not yet manufactured or supply is too limited to accommodate the entire programming team. Unlike conventional RTOS simulators that run as native UNIX or PC applications, ISIM simulates the same code that runs on the target. Thus, application size characteristics are known during simulation. In addition, the exact same compilers and development tools are used for simulation and for actual hardware development. This increases the value of the development tools investment and the productivity of the programming team.

ISIM provides complete INTEGRITY simulation, including virtual memory and memory protection. ISIM provides a robust system debugging environment. For example, breakpoints can be placed anywhere in the system, including interrupt service routines (ISRs).

The complete MULTI Development Environment is supported for ISIM, including:

- Multitask debugging via rtserve2
- Fast Dynamic Download
- MULTI EventAnalyzer for post-mortem system analysis

Note: The computer used for multitask debugging via rtserv2 can be different from the computer running ISIM. The two computers can communicate via Ethernet.

Simulation speed is the one disadvantage of providing RTOS simulation via an instruction set simulator instead of running as a native workstation application. However, unlike conventional instruction set simulators, ISIM is highly tuned for RTOS simulation performance.

8.2 ISIM Features

ISIM has built-in Ethernet and networking support, which makes it possible to use rtserv2 to debug applications. In addition, sockets applications can be run on top of ISIM.

ISIM simulates a fixed amount of RAM. This RAM limit is specified via the special `.ramlimit` section in the **default.Id** used for the ISIM BSP. To change the amount of simulated RAM:

1. In the MULTI Project Manager, open the Build Options window for your kernel.
2. On the **All Options** tab, expand the **Linker** category and select **Symbols**.
3. In the Build Options pane, select **Define Linker Constant** and set the linker constant `--INTEGRITY_RamLimitSize` to the desired value.

isimppc and **isimarm** can also simulate ROM/flash. To link the kernel into ROM:

- Link the **flash.Id** in the **sim800** or **simarm** directory.

See the comments in **flash.Id** for information about how to link the kernel into ROM and how to run it out of ROM, if desired.

Any attempts to access outside the range of RAM and ROM will cause undefined results, and ISIM will display error messages.

Newer simulators that run in fast mode are available as separately licensed MULTI components for PowerPC and ARM. The newer simulators are significantly faster than older simulators, but they do not offer identical feature sets. For more information about fast mode simulators and backwards compatibility, see the “Simulator for PowerPC (simppc) Connections” or “Simulator for ARM (simarm) Connections” chapter of the respective *MULTI: Configuring Connections* manual.

8.3 ISIM Target BSPs

ISIM refers to the following host-based simulation programs:

- **isimppc** — for PowerPC
- **isimarm** — for ARM (simulates the ARM7 family)

These programs simulate INTEGRITY systems. Throughout INTEGRITY documentation, these programs are referred to generically as ISIM or **isim***.

These systems are built from the following special BSPs, which are intended for use on ISIM. These are basic BSPs that developers can use to simulate the appropriate processor family.

- **sim800** — for PowerPC systems.
- **simarm** — for ARM systems.

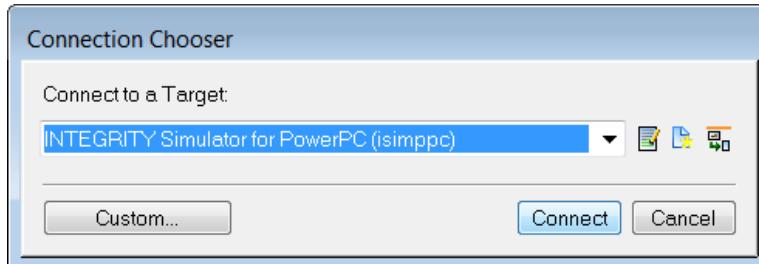
8.4 Running ISIM

ISIM can be run from the MULTI GUI or from the command-line.

Note: The following instructions use the INTEGRITY Simulator for PowerPC, but the process is the same for all targets. Selections available in the Connection Chooser will reflect your target.

To run ISIM from MULTI:

1. Open the INTEGRITY executable in the MULTI Debugger.
2. In the Debugger, click the **Connect** button. The Connection Chooser will open.



3. In the Connection Chooser, select **INTEGRITY Simulator for PPC (isimppc)** and click **Connect**.

Depending on what MULTI window you open Connection Chooser from, the **INTEGRITY Simulator for your target** selection may not appear. In this case, you must create a new connection, specifying the INTEGRITY Simulator for your target as the server.

4. After a connection to ISIM has been established, click **Go**.

You can toggle the Target pane and I/O pane from the Debugger's command pane. INTEGRITY console messages are displayed in the Target pane.

To run ISIM from the command-line:

- Run **isimppc** and **simarm** with a single argument, the name of the INTEGRITY system executable. For example:

```
c:\ghs\multi616\isimppc kernel  
c:\ghs\multi616\isimarm kernel
```

Console messages (such as the bootup banner) will display to stdout.

8.5 ISIM Socket Emulation

Applications running on ISIM can send and receive data across a network using standard sockets communication. Applications that use sockets must link with **libsocket.a**. For the ISIM BSPs, **libsocket.a** is actually a build of **libsocketemulation.gpj**, not the standard **libsocket.gpj**. This socket emulation library bypasses the TCP/IP stack and sends networking requests directly to the host-based ISIM program for fast execution. This results in faster networking simulation than actually running a full TCP/IP stack on top of ISIM, however there is one limitation: sockets ports used by INTEGRITY applications must not conflict with other applications running natively on the host computer system.

Socket emulation provides most, but not all, sockets library functionality. It does not support the following:

- multicast receive
- raw sockets
- outgoing ping
- fcntl() calls
- some setsockopt() and ioctl() requests
- sendmsg() and recvmsg() ancillary data and return flags
- select() with other file descriptor types
- spawning AddressSpaces with inherited socket file descriptors

Socket emulation uses the host's TCP/IP stack, and may behave slightly differently (in terms of timing, etc.) than an INTEGRITY-native TCP/IP stack. If the host's time is changed during a select call, the call may time out earlier or later depending on the host time change.

Note: Socket emulation calls should not be made using data in memory that is dynamically unmapped or mapped with restricted permissions because socket emulation bypasses the normal INTEGRITY memory access permission and fault handling mechanisms. If a page fault occurs during a socket emulation call (for example, the buffer passed to a send call is unmapped), nothing in the system will be given a chance to service the fault. If a socket emulation operation is attempted on a page that is mapped with insufficient permissions or becomes unmapped, the call may either fail, or succeed by accessing memory despite its mapping and access permissions.

8.5.1 ISIM Socket Port Remapping

Sometimes a target system will use a socket port that is reserved on the host system, for example to implement a web server. On a UNIX system with reserved ports, or on a host system with a server already running on that port, the target will be unable to open its desired socket port.

ISIM allows emulated socket ports to be remapped with the **-port_translate** and **-remote_port_translate** options. For example, **-port_translate 80 8080** remaps a web server port from the standard port 80 to the nonstandard 8080, which is less likely to conflict with the host. If the target were to connect to its own web server on port 80, the **-remote_port_translate 80 8080** option would similarly remap this outgoing connection. Any number of port translation options can be specified to remap target services that may conflict with the host.

When running multiple ISIM instances on the same host, the run-mode debug socket port must also be remapped in order to allow rtserv2 to connect to all ISIM instances. For more information, see the “Connecting to Multiple ISIM Targets” section in the “Connecting with INDRT2 (rtserv2)” chapter.

8.6 ISIM Caveats

Currently, serial port simulation is provided to display diagnostics and console messages only. Input is not read from the simulated serial port. When ISIM is connected to MULTI, the Target pane can be used to communicate INDRT2 commands such as **nc** (network configuration), **lm** (list dynamically loaded modules), and **lt** (list tasks).

When using Dynamic Download, wait until **Download Succeeded** appears in the ISIM console window before attaching to any tasks. Otherwise, you will interfere with ISIM’s memory initialization for the virtual task.

When ISIM is run through MULTI, it handles all Host I/O calls by default. This behavior can be disabled if desired. To run ISIM with Host I/O call handling disabled, do one of the following:

- Pass in **syscalls_off** as a command-line parameter.
- In the Connection Chooser, click **Custom** and add **-syscalls_off** to the custom command line.

Time on INTEGRITY simulators is variable and depends on other factors, including the speed of the host system, the speed of the simulator, and the load on the simulated target. As a result, time-based events on a simulator may not be the same as “wall clock” time.

ISIM is a CPU-intensive application, but consumes very little CPU when the simulated system is idle (no application tasks to run). At the expense of simulation and debugging speed, you may manually lower the priority of the ISIM process running on the host computer.

INTEGRITY Simulators do not support using the Serial Shell because Simulators do not provide serial input.

INTEGRITY Simulators do not support using the POSIX Network Server interactive shell commands (**/bin/sh**).

Chapter 9

Freeze-Mode Debugging

This chapter describes how to set up and use freeze-mode connections, which are used for system-level kernel debugging on hardware targets. The chapter provides information on the following topics:

- Freeze-Mode Debugging Using Probes
- Debugging out of RAM
- Debugging out of ROM/Flash
- Freeze-Mode Debugging Using Host I/O

Note: Throughout this manual, the terms “freeze-mode debugging” and “system debugging” are synonymous.

INTEGRITY Supports both freeze-mode and run-mode debugging. Freeze-mode debug connections are made using either **mpserv** (for a real hardware target) or ISIM (for a simulated target). With freeze-mode debugging, hitting a breakpoint halts all execution of code on the affected target processor core, which includes execution of an INTEGRITY kernel. Run-mode debug connections are made using **rtserv2**. With run-mode debugging, hitting a breakpoint only halts the affected Task, and the INTEGRITY kernel, kernel debug agent, and all other Tasks continue to run. For information about run-mode debugging, see the “Run-Mode Debugging” chapter.

9.1 Freeze-Mode Debugging Using Probes

A probe is the actual hardware that attaches to the debug port of the target board. The host PC or UNIX workstation connects to the probe via an Ethernet or USB connection. For more information, see the *Green Hills Debug Probes User’s Guide* or the *MULTI: Configuring Connections* manual for the processor in use.

This chapter contains information regarding all types of freeze-mode debug solutions, such as connecting with **mpserv** and a Green Hills Debug Probe. The **bspname.notes** file also contains information about which debug solutions support your board.

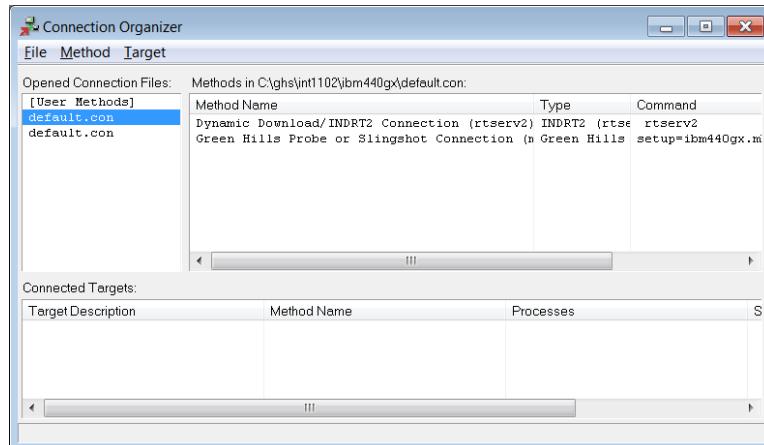
INTEGRITY also supports *Probe Run Mode*, which allows targets with live memory access (such as PowerPC and ARM Cortex) to be debugged with a run-mode connection via the JTAG interface without the use of a dedicated Serial or Ethernet port on the target. Probe Run Mode allows you to make both a freeze-mode and run-mode connection to your target using only the probe's debug connection. For more information, see “Using Probe Run Mode” in the “Connecting with INDRT2 (rtserver2)” chapter of this manual.

Note: Some boards do not have a debug/JTAG port, while others have a port, but it lacks the ability to connect to the probe hardware. Consult the board manufacturer and/or Green Hills technical support for other possible methods of performing freeze-mode debugging on this type of system.

9.2 Debugging Out of RAM

To use a freeze-mode debug connection such as mpserv for downloading and/or freeze-mode debugging:

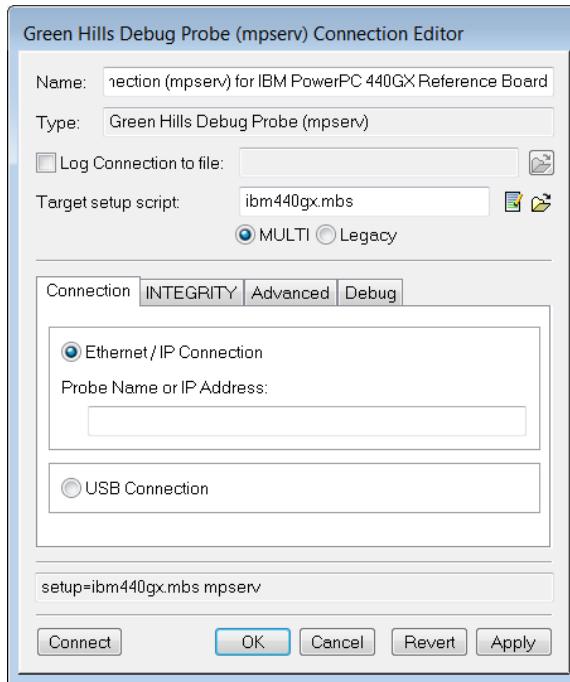
1. In the MULTI Project Manager, build an INTEGRITY KernelSpace or Monolith Application project to be debugged.
2. Select the built .gpj and click the **Debug** button to open it in the Debugger.
3. In the Debugger, select **Target⇒Show Connection Organizer**.



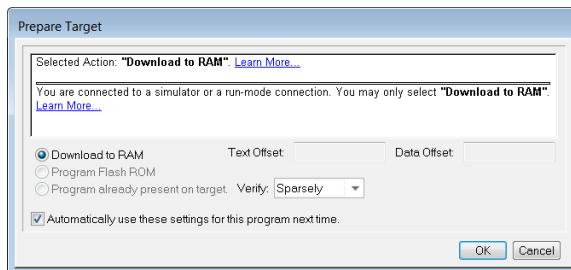
The MULTI Connection Organizer provides a graphical interface for selecting connection options. For more information, see “Connecting to Your Target” in the *MULTI: Debugging* manual. For information about specific connectivity issues for each type of debug server, see the *MULTI: Configuring Connections* manual for the processor in use.

4. Using the Connection Organizer, establish a freeze-mode debug connection to the target using either mpserv (for a real hardware target) or ISIM (for a simulated target).

- Specify the setup script in the Connection Editor. In many cases, a debug server script (or MULTI script) must be run to ensure that the target is properly initialized before an INTEGRITY kernel or application can be downloaded into RAM. See the ***bspname.notes*** file for information about which script to run. When done editing the Connection settings, click **Connect**.



5. In the Debugger, select **Debug⇒Prepare Target**. The Prepare Target dialog will open:



When debugging from RAM, the setup script runs and the project is downloaded to RAM. The Debugger will stop at the program's entry point, ready to debug. For information about preparing the target, see the “Preparing Your Target” chapter in *MULTI: Debugging*.

9.3 Debugging Out of ROM/Flash

Debugging out of ROM (or flash) is similar to debugging out of RAM, however there are some differences. The most important difference is that breakpoints cannot be used because ROM cannot be written to arbitrarily like RAM can.

Most BSPs that support booting from flash are configured for ROM-copy, in which the kernel boots from flash, but then copies itself to RAM and executes from RAM. This yields better performance because accessing RAM is often faster than accessing ROM. When using the ROM-copy configuration, you can debug the program out of ROM up until the ROM-to-RAM copy. After the jump to RAM, you can debug the program like a normal RAM program, and use software breakpoints.

Note: This section describes the most common method of flash debugging. However, all targets are different, and flash configuration varies widely. It is critical that you read your BSP's **.notes** carefully for board-specific requirements of flash programming and debugging.

To debug from ROM:

1. In the MULTI Project Manager, build an INTEGRITY KernelSpace or Monolith Application project to be debugged. The project should be linked to flash, normally by including the **flash.ld** file in the kernel. See the “KernelSpace Project” section of the “Building INTEGRITY Applications” chapter for more information.

Note: Some versions of MULTI prevent debugging a Monolith Application from ROM. The monolith can be flashed, but for debugging, the kernel should be used instead of the monolith. If you flash a monolith, close MULTI, then reopen MULTI and select the kernel executable for debugging.

2. Select the built **.gpj** and click the **Debug** button to open it in the Debugger.
3. In the Debugger, select the **.gpj** and select **Target⇒Show Connection Organizer**.
4. Using the Connection Organizer, establish a remote connection using the appropriate connection options for the particular debug hardware and connectivity method. For additional information, see the “Debugging Out of RAM” instructions earlier in this chapter.

Specify the setup script in the connection. In many cases, a debug server script (or MULTI script) must be run to ensure that the target is properly initialized. Some BSPs use a different setup script file to flash the target. See the **bspname.notes** file for information about which script to run.

5. In the Debugger, select **Debug⇒Prepare Target** and choose to flash or verify the executable as appropriate. See the “Preparing Your Target” and “Programming Flash Memory” chapters of the *MULTI: Debugging* manual for more information.
6. After you flash the executable, the MULTI Debugger can be single-stepped from the program’s ROM/flash entry point to begin execution. Remember that software breakpoints cannot be used when debugging out of ROM.

To enter hardware breakpoints to enable stopping in ROM/flash code:

- (a) In the MULTI Debugger, select **View**⇒**Breakpoints**.
- (b) Select the **Hardware** tab and click **New HW Breakpoint**.
- (c) Configure as desired.

Alternately, you can use the **hardbrk** command. For more information, see the “hardbrk” section of *MULTI: Debugging Command Reference*.

7. If the **flash.Id** file is configured for ROM-run (and the BSP supports it), the program will execute entirely from ROM, using RAM only for stack and heap. However, most BSPs that support booting from flash are configured for ROM-copy, in which the kernel boots from flash, but then copies itself to RAM and executes from RAM.

Note: Some Power PC processors based on the e500v1 core suffer from chip errata that affect debugging from reset. For information, see the **reset_fixup** and **no_boot_rom** options in the *Green Hills Debug Probes User’s Guide*.

9.4 Freeze-Mode Debugging Using Host I/O

Debug servers can handle Host I/O calls as a way to facilitate communication between the target and the host debugger. This can be useful for freeze-mode debugging. For example, with this capability the target can use Host I/O calls to access the host file system, TCP/IP stack, or clock.

In INTEGRITY, there is a distinction between freeze-mode Host I/O calls and run-mode Host I/O calls. For example, by default, Host I/O file access is only performed through the run-mode INDRT2 connection (rtserve2). (For information about using Host I/O calls with the run-mode INDRT2 connection (rtserve2), see the “Host I/O” section of the “Run-Mode Debugging” chapter.)

Freeze-mode debug servers will handle Host I/O calls by default. To instruct the debug server not to trap on the Host I/O call instruction (and therefore ignore Host I/O calls from the kernel), use the following command:

```
target syscalls off
```

To re-enable Host I/O call handling, use the following command:

```
target syscalls on
```

When Host I/O call handling is enabled in the freeze-mode debug server, the following occurs:

- KernelSpace Host I/O calls are routed to the freeze-mode debug server.
- The INTEGRITY kernel allows tasks to make Host I/O calls in order to print to standard output/error, and uses Host I/O to set the time on startup when a real-time clock is not present.

Because using the host for time and being able to route standard output/error to the host may not always be desired, each feature also depends on the definition of a weak symbol. For example in **global_table.c**, you could define the following variables as necessary:

```
char __ForceStdOutputsToHostIO;  
char __ForceTimeToHostIO;
```

Alternatively, the weak symbol can be defined by setting the following linker option in the kernel build project:

```
-lnk=-D__ForceStdOutputsToHostIO=1  
-lnk=-D__ForceTimeToHostIO=1
```

9.4.1 Redirecting the Console over Host I/O

If there is no serial port available for the INTEGRITY console, or the console serial port is in use for an rtserver connection, it is possible to redirect the console over freeze-mode Host I/O. To do this, define a weak symbol, for example in **global_table.c**:

```
char ForceConsoleToHostIO;
```

Alternatively, the weak symbol can be defined by setting the following linker option in the kernel build project:

```
-lnk=-DForceConsoleToHostIO=1
```

Note: On CPU architectures (such as Blackfin) where the ABI defines C symbols to be prepended with an underscore, the linker option would be

```
-lnk=-D_ForceConsoleToHostIO=1
```

Chapter 10

Connecting with INDRT2 (rtser2)

This chapter describes how to establish a debug connection between MULTI and a running target using INDRT2 and rtser2. This type of debug connection provides run-mode debugging with individual task control.

This chapter covers the following topics:

- Introduction to rtser2 and INDRT2
- Building in Run-Mode Debugging Support
- Connecting to rtser2

Note: Throughout this manual, the terms “run-mode debugging” and “application-level debugging” are synonymous.

10.1 Introduction to rtser2 and INDRT2

The MULTI debug server, **rtser2**, is used to connect to a running INTEGRITY system to perform run-mode debugging. When connected to rtser2, the MULTI Debugger Target list displays all the Tasks in the system, and any user Task in the Target list can be selected and individually debugged.

rtser2 enables advanced features such as Dynamic Downloading of virtual AddressSpaces, and use of tools such as the MULTI Profile window and the EventAnalyzer.

INDRT2 is the name of the KernelSpace debug agent used to support communication with the Run-Mode Debug Server, rtser2. Because it is not a Task, INDRT2 debugging is supported even when all the Tasks in the system are compromised, as long as communications with the target are still functioning properly, and interrupts are still firing.

Note: INDRT2 is not a secure module. Use of this debug agent is valuable during the debugging phase, but it should almost always be removed from your product before shipping. For more information, see “Security and the INDRT2 Debug Agent” in the “Security Issues” chapter.

10.1.1 Communication Media

An INDRT2 (rtserv2) connection can be established using Ethernet, serial or Probe Run Mode. A BSP typically provides drivers to support an Ethernet or serial device. For more information about setting up network configuration, see the “Network Configuration” chapter of the *INTEGRITY Networking Guide*.

The following are general recommendations for INDRT2 (rtserv2) connections:

- Use Ethernet whenever possible due to better performance.
- Probe Run Mode allows targets with live memory access to be debugged with a run-mode connection via the JTAG interface without the use of a dedicated Ethernet or serial port on the target. For more information, see “Using Probe Run Mode” later in this chapter.
- The baud rate at which a particular BSP communicates over a serial port varies. See the ***bspname.notes*** file for details.
- On BSPs that support only one serial port, the port cannot be used for debugging when it is being used for diagnostics. If the serial port is already in use, close the terminal program used to view the serial port output before establishing your run-mode connection.

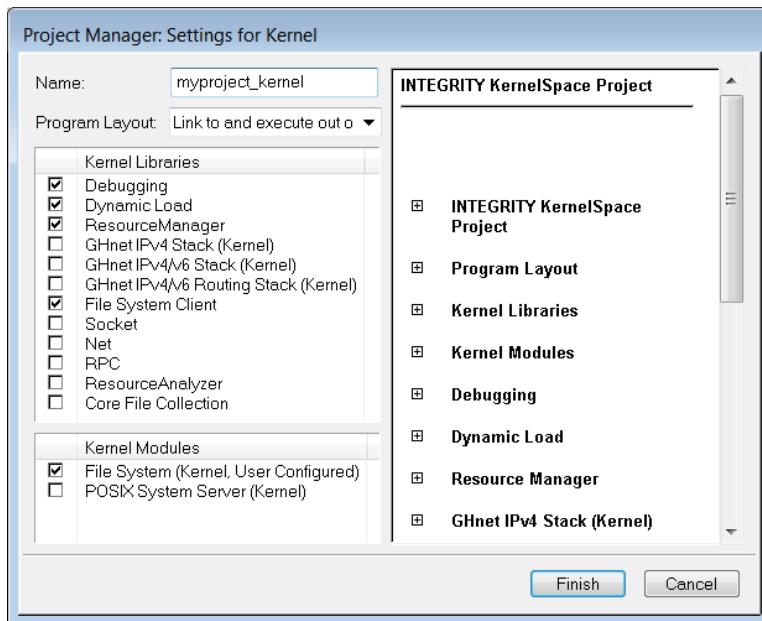
10.2 Building in Run-Mode Debugging Support

To enable run-mode debugging support, an INTEGRITY KernelSpace program must be linked with the library **libdebug.a**. **libdebug.a** is automatically included in **kernel.gpj** for KernelSpace projects created with the MULTI Project Manager and in the **kernel.gpj** that ships with INTEGRITY reference BSPs.

To include **libdebug.a** in an existing project:

1. Right-click **myproject_kernel.gpj** and select **Configure**.

The Settings for Kernel dialog will open.



2. Select the **Debugging** check box.

10.2.1 libdebug.a Configuration

In most cases, the default configuration of **libdebug.a** is appropriate. However, users can modify the configuration if memory usage needs to be reduced, or disabling certain transport mechanisms is desired.

For information about how to modify the configuration of **libdebug.a**, see “Configuring **libdebug.a**” in the *INTEGRITY Implementation-Dependent Behavior Guide*.

10.3 Connecting to rtserv2

rtserv2 is the debug server used to handle debugging requests for INTEGRITY application debugging. Before debugging application code, the MULTI Debugger must determine with which target it will communicate. MULTI communicates with rtserv2, a host-resident debug server, and sends requests such as Read Register, Write Memory, etc. during a debugging session. In turn, rtserv2 handles the debugging requests with the target and establishes communication from the host to the target (for example, over Ethernet).

You can use the MULTI Connection Editor to create and save connection methods that correspond to your particular host and target systems, and desired debugging options. After you have created a connection method, you can use the MULTI Connection Chooser to connect to your target quickly and easily. For general instructions about creating and using Connection Methods, see “Connecting to Your Target” in the *MULTI: Debugging book*. The information in this chapter is specific to INTEGRITY and is meant as a supplement to those instructions.

You can establish a debug server connection with any of the following methods, which are discussed in more detail in the sections that follow:

- Use the MULTI Run-Mode Partner to automatically set up a connection — see “Using the Run-Mode Partner” below.
- Use Probe Run Mode to make both a freeze-mode and run-mode connection to your target using just the probe’s debug connection — see “Using Probe Run Mode” later in this chapter.
- Use the Connection Organizer and Connection Editor to graphically set up or modify a connection method — see “Connecting to rtserv2 Using the MULTI Connection Organizer” later in this chapter.
- Type in a custom connection command-line — see “Connecting to rtserv2 Using a Custom Connection Command-Line” later in this chapter.

10.3.1 Using the Run-Mode Partner

MULTI supports automatically creating a run-mode connection to the target system via rtserv2. This connection is known as the *Run-Mode Partner*. If you set the run-mode partner, the Debugger automatically establishes the run-mode connections after an INTEGRITY kernel has booted.

If you use “Creating MULTI Workspaces for Installed BSPs” instructions in the “Getting Started with INTEGRITY” chapter, the run-mode partner will automatically be established when you run your application. Otherwise, you can use the **Set Run-Mode Partner** dialog box in MULTI to specify your run-mode connection. For information, see “The Set Run-Mode Partner Dialog Box” in *MULTI: Debugging*.

10.3.2 Using Probe Run Mode

When connecting to Green Hills Probes, INTEGRITY run-mode connections are traditionally made by connecting the host machine to the same network as the target, with INTEGRITY sending information over Ethernet to the host machine. However, this technique requires that your hardware has Ethernet or serial drivers. Probe Run Mode allows you make both a freeze-mode and run-mode connection to your target using just the probe's debug connection. For information about system requirements for Probe Run Mode, see the “Probe Run Mode” section of the *Green Hills Debug Probes User's Guide*.

To connect to Probe Run Mode, see the instructions in “Connecting to rtserver2 Using a Custom Connection Command-Line” and use your probe's IP address or hostname and port 2222 for the rtserver2 connection. For example:

```
rtserver2 192.168.1.101:2222
```

By default, the Probe Run Mode connection will not be considered for run-mode partnering. This can be overridden by defining a global character named `GipcTarget_EnableAutoPartner` in your KernelSpace program. Doing so will prevent MULTI from considering the Ethernet connection for run-mode partnering.

For information about implementing the GIPC Target interface in the BSP, see “GIPC Target Interface” in the *BSP User's Guide*.

For additional information about Probe commands, including how to attach to Probe Run Mode in a running kernel, see the *Green Hills Debug Probes User's Guide*.

10.3.3 Connecting to rtserver2 Using the MULTI Connection Organizer

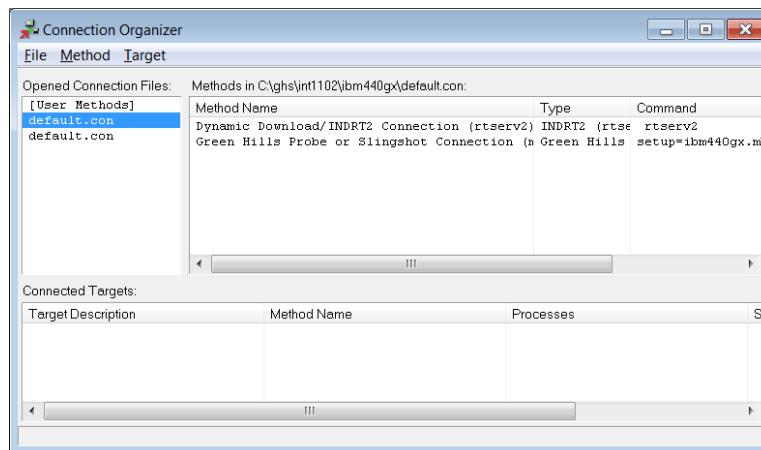
You can use the MULTI Connection Organizer and Connection Editor to modify an existing connection, or to create a new connection. The easiest way to connect to rtserver2 through the Connection Organizer is to modify an existing connection.

Note: An INTEGRITY kernel must be running before you connect to the target.

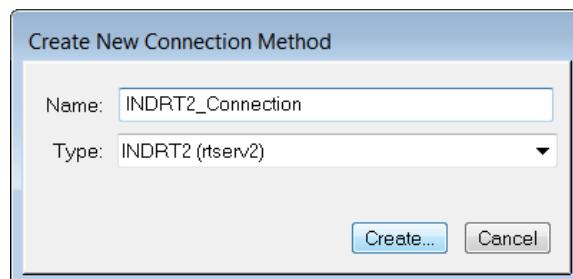
1. Open the `bspname\default.gpj` file in the MULTI Project Manager.

Each BSP contains example connections that you can customize for your own use. These same connections are used by the New Project Wizard for every project created for that BSP.

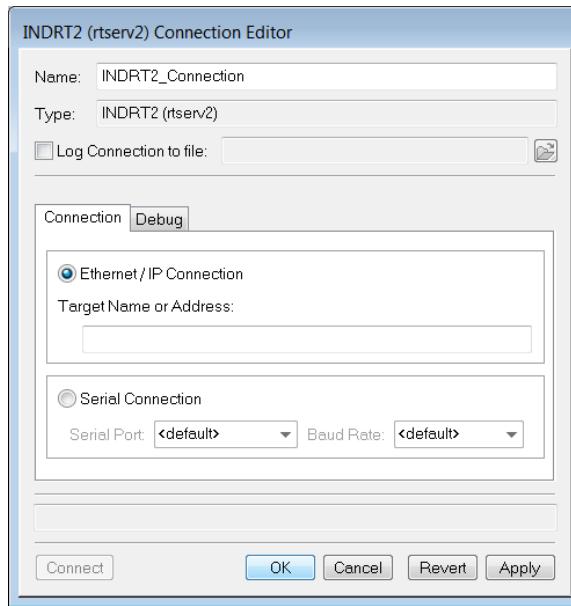
2. In the MULTI Project Manager, select **Connect⇒Connection Organizer**.
3. In the **Opened Connection Files** pane, select `default.con`.



- To modify an existing connection, double-click the **Dynamic Download/INDRT2 Connection** to open the Connection Editor.
- To create a new connection:
 - (a) Select **Method**⇒**New** to open the Create New Connection Method dialog box.
 - (b) In the Create New Connection Method dialog, specify a **Name** and a **Type**. For an rtserve2 connection, select **INDRT2 (rtserve2)** as the **Type**.



- (c) Click **Create** to open the Connection Editor.



The INDRT2 (rtserver2) Connection Editor includes **Connection** and **Debug** tabs to set options specific to your target and host operating systems. When you first open the Connection Editor, the options are set to default values. Some of the fields may require user input.

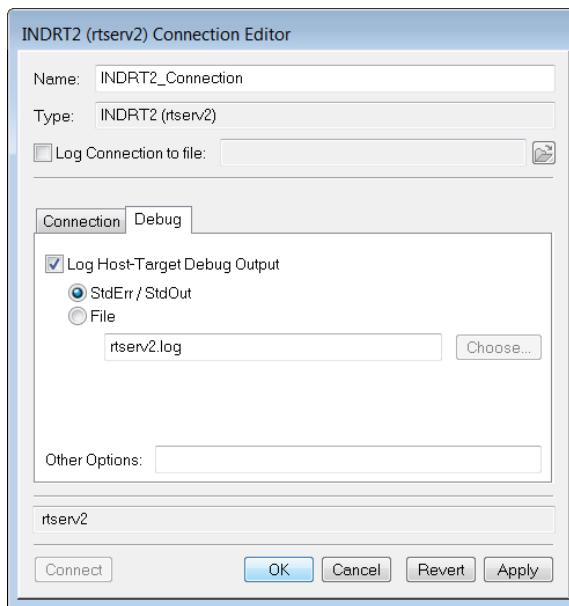
4. On the **Connection** tab, select **Ethernet/IP Connection** or **Serial Connection**.

- **Ethernet/IP Connection** — Select to use an Ethernet connection.
 - **Target Name or Address** — Specify your target name or IP Address.
- **Serial Connection** — Select to use a serial connection.

Note: Not supported on Solaris.

- **Serial Port** — Specify which host serial port to use for your serial INDRT connection. The default serial port selections are:
 - * Windows — COM1
 - * Non-Windows — ttyS0
- **Baud Rate** — Specify the serial port communication speed. The default baud rate is 9600.

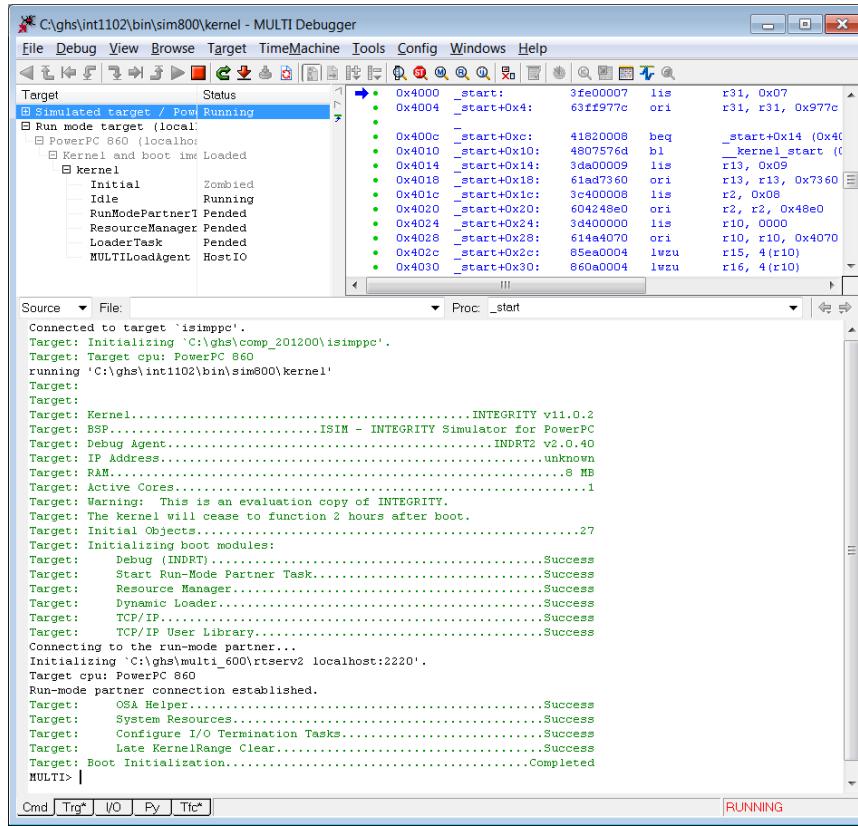
5. Use the **Debug** tab for the following settings:



- **Log Host-Target Debug Output** — Select to enable logging of all communications between rtserver2 and your target. Logging is disabled by default. Enter or select the debug output filename.
- **Other Options** — Allows you to add other, optional arguments directly to the command line. Only use this field if directed to do so by Green Hills Technical Support.

6. When done customizing the connection, click **OK** to return to the Connection Organizer .
7. Right-click the **Dynamic Download/INDRT2 Connection** and select **Connect to Target**.

After you connect to rtserver2, the MULTI Debugger Target list displays all the Tasks in the system and is updated regularly. (For more information, see “Using the MULTI Debugger with rtserver2” in the “Run-Mode Debugging” chapter.)

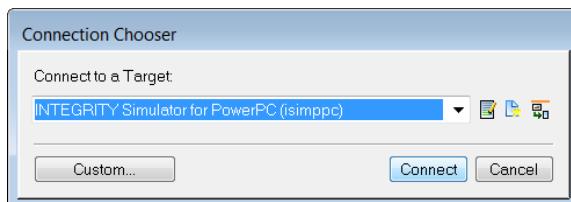


- To view the Target pane or I/O pane, click on the **Trg** or **I/O** tabs at the bottom of the MULTI Debugger.

10.3.4 Connecting to rtserver2 Using a Custom Connection Command-Line

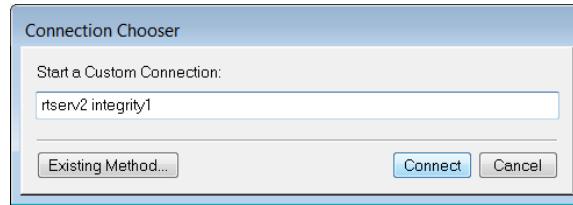
In earlier MULTI releases, the only way to create a new connection from MULTI was to type in an explicit command-line. This manual method can be still be used:

- In the MULTI Project Manager, click the **Connect** button to open the Connection Chooser.
- Click **Custom** in Connection Chooser, then enter an explicit connection command in the **Start a Custom Connection** field, for example:



- To establish an Ethernet debugging connection to a board running INTEGRITY with a BSP that supports Ethernet using integrity1 as the hostname that the system administrator has set up for the board, enter:

```
rtserve2 integrity1
```



- To enable logging of communications between rtserve2 and your target, use the **-log** option in your custom connection command. For example, to enable logging on the example Ethernet connection above, enter:

```
rtserve2 -log=filename integrity1
```

- To establish a serial debugging connection, enter one of the following commands (port selection may vary):
 - For Windows: rtserve2 -serial com1 9600
 - For Linux: rtserve2 -serial /dev/ttyS0 9600

Note: Serial rtserve2 connections are not supported on Solaris.

10.3.5 Connecting to Multiple ISIM Targets

By default, ISIM accepts rtserve2 connections on the default UDP socket port of 2220. Only one ISIM instance on a given host can listen on a particular port. To connect to multiple ISIM instances on the same host, some of the ISIM instances must listen on different ports.

For example, to create an ISIM connection that listens on port 3330 instead of 2220:

1. In the Connection Chooser, click the **Create a New Connection Method** button.
2. For the **Connection Type**, select **INTEGRITY Simulator** and click **Create**. The Connection Editor will open.
3. Select the **Debug** tab. In the **Other Options** field enter: `-host_indrt_port 3330`
4. Create a custom rtserve2 connection that specifies the target name (or dotted-decimal IP address) and the port number in the following format:

```
rtserve2 target_name[:port_number]
```

- (a) Using the Connection Chooser, select your rtserve2 connection and click the **Edit** button.

- (b) Select the **Connection** tab and enter `localhost:3330` in the **Target Name or Address** field.

When using socket emulation for TCP/IP communications on an ISIM target, those emulated socket ports may also conflict with other ISIM instances or with the host's reserved ports or running services. These ports can also be remapped. See the "ISIM Socket Port Remapping" section for details.

10.3.6 Disconnecting from rtserver2

To terminate a debugging session with rtserver2:

- Select **Target⇒Disconnect from Target** in the Task Manager.

If you are running the kernel with a debug server (for example, **mpserv**), it is very important to disconnect from rtserver2 before disconnecting from the kernel debug server. When rtserver2 loses its connection to the kernel (such as when the kernel stops running), rtserver2 will timeout eventually, but the MULTI session used to connect to rtserver2 will be unresponsive until the timeout is completed. To avoid this, always disconnect from rtserver2 first.

Chapter 11

Run-Mode Debugging

This chapter provides instructions for run-mode debugging with **rtserv2**, and covers the following topics:

- Run-Mode Debugging Limitations
- Using the MULTI Debugger with rtserv2
- Using the Target Pane
- Using the I/O Pane
- Using Breakpoints with Run-Mode Debugging

Note: Throughout this manual, the terms “run-mode debugging” and “application-level debugging” are synonymous.

INTEGRITY Supports both freeze-mode and run-mode debugging. Run-mode debug connections are made using **rtserv2**. With run-mode debugging, hitting a breakpoint only halts the affected Task, while the INTEGRITY kernel, kernel debug agent, and all other Tasks continue to run. Freeze-mode debug connections are made using either **mpserv** (for a real hardware target) or ISIM (for a simulated target). With freeze-mode debugging, hitting a breakpoint halts all execution of code on the affected target processor core, which includes execution of the INTEGRITY kernel. For information about freeze-mode debugging, see the “Freeze-Mode Debugging” chapter.

11.1 Run-Mode Debugging Limitations

Note: Some operations performed by the Run-mode Debug Agent may silently consume one or more resources. Because the Debug Agent is not allowed to dynamically allocate new resources on demand, users should avoid using the Debug Agent at or near its resource limits.

Run-mode debugging with rtserv2 is limited to debugging Task-context code only. Setting breakpoints in KernelSpace via run-mode is inherently dangerous. If you set a run-mode breakpoint on a function that is used or shared by the kernel, exception handlers, the BSP, or any other code that executes outside of the context of a Task, you can cause the kernel to hang or crash. INDRT2 helps prevent this situation by automatically disabling such hazardous breakpoints

when it detects the situation, but not all such situations can be detected automatically. Users must avoid setting breakpoints in code shared between Tasks and non-Task contexts. This restriction applies to both explicit breakpoints, and implicit breakpoints that are set as a side-effect of single stepping a Task.

As a general rule, a user debugging an INTEGRITY system should never halt or otherwise modify the state of a Task (e.g., perform a command-line procedure call in the Task) unless the user fully understands the behavior of that Task. When debugging system Tasks, you must be careful because the act of debugging them may have unintended consequences such as disrupting your connection to the target. In particular, you should avoid halting or otherwise manipulating the Loader and its related Tasks, the OSA Agent Tasks, system Idle Task(s), Tasks corresponding to virtual device drivers that you depend on to maintain your run-mode connection, or any other Task that your system critically depends on (e.g., a Task that must run to prevent the system from catching on fire).

In some cases, the system automatically prevents potentially dangerous attempts to debug Tasks that should not be debugged through a run-mode debug connection. Examples of Tasks for which the system might automatically block such attempts include (but are not limited to) the following:

- Any Task critical to the operation of the kernel itself, such as Idle Tasks and IOTermination Tasks.
- Tasks used to implement the communication layer used by the run-mode debug connection, possibly including Tasks that are part of Ethernet drivers and the associated infrastructure (such as DriverDebugServices Tasks) and Tasks that are part of IDB or LBP.
- If the run-mode debug connection is passing through sockets (e.g., it is built on top of IDB), then any Task (such as the InetServer Task) that directly executes within the GHnet v2 TCP/IP stack, including any user Tasks that call directly into the stack through the “direct” APIs. (However, user Tasks that interact with the stack through sockets are not affected.)
- Any Task critical to the behavior of the system as a whole, such as Loader helper Tasks.

In order to debug a Task with rtserv2, MULTI must have access to the executable of the AddressSpace containing the Task. MULTI will attempt to locate the file and will prompt you to specify the file if it is unable to find it on its own. If the filename is unspecified, or it cannot be accessed by MULTI, no debugging operations will be available.

Run-mode command-line procedure calls are unsafe in KernelSpace Tasks because MULTI is not able to determine whether a KernelSpace Task is in the middle of a kernel call. The user is responsible for ensuring that a KernelSpace Task is not in the middle of a kernel call before performing a run-mode command-line procedure call using that Task. In general, you should halt the Task, confirm that the Task is not in the middle of a kernel call (usually by verifying that the Task is in code for which debug information is available), perform the command-line procedure call if safe, then resume the Task.

When a kernel Object is being Created, Closed or Moved, executing a Debug Agent command that operates on that class of Object may result in information output that is incomplete.

Some commands, such as **r argv**, **string**, and **restart** are not supported.

A maximum of 63 software breakpoints can be set for any given Task at a time. Additionally, rtserver does not support overlapping execution breakpoints or overlapping of execution and data breakpoints within a single AddressSpace. For more information, see “Using Breakpoints with Run-Mode Debugging” later in this chapter.

By default, most kernels support debugging a maximum of 126 combined AddressSpaces and Tasks, collectively known as “entities”. This maximum can be modified by specifying `--INTEGRITY_DebugEntities` in the linker directives file, as described in “Number of Debug Entities” below. If the total number of entities in your system exceeds the configured maximum, debugging information may be misleading and unreliable.

Machine instructions that branch to themselves can interfere with normal run-mode debugging operations in certain circumstances. These types of instructions are typically associated with optimized infinite loops, and are very uncommon. In many cases, the INTEGRITY Debug Agent is able to handle instructions that branch to themselves, however not all cases have practical solutions.

INDRT2 is not a secure module. Use of this debug agent is valuable during the debugging phase, but it should almost always be removed from your product before shipping. For more information, see “Security and the INDRT2 Debug Agent” in the “Security Issues” chapter.

11.1.1 Number of Debug Entities

By default, most kernels support debugging a combined maximum of 126 Tasks and AddressSpaces, which are collectively called “entities”. If the number of entities in your system exceeds the configured maximum, you may encounter problems connecting to the target with the Debugger, and debugging information may be misleading and unreliable.

You can verify the number of debug entities supported by your BSP by examining your linker directives file. The DEFAULTS section of `default.ld` specifies `--INTEGRITY_DebugEntities`. This setting can be modified to accommodate a larger or smaller number of entities as needed.

To specify the correct number of entities, add 2 to the number of entities you want to debug. In an example `default.ld` file that supports debugging 126 entities, the value will be specified as follows:

```
--INTEGRITY_DebugEntities = 128
```

Note: The debug agent will use at most 4093 entities. Allocating space for entities above this limit will have no benefit.

If you dynamically create and destroy Tasks and AddressSpaces while connected to your target with MULTI, it takes a small amount of time for these resources to be freed before becoming available for reuse. As a result, the number of entities can be exceeded unexpectedly. If Tasks and AddressSpaces are being created and destroyed quickly, in order to ensure that they will all be available for debugging, limit the number of entities to no greater than 63 (if using the default in `libdebug.a`), or half the value set in `--INTEGRITY_DebugEntities`.

11.1.2 Task and AddressSpace Naming

Task and AddressSpace names are used in debugging to help the user identify which Task or AddressSpace is being referenced. Green Hills tools and debugging facilities carry restrictions on names that are not enforced by INTEGRITY. These restrictions should be considered when naming INTEGRITY AddressSpaces and Tasks. The APIs used for naming INTEGRITY AddressSpaces and Tasks do not enforce these restrictions, but if the restrictions are not adhered to, these entities may not be available for debugging through rtserv2.

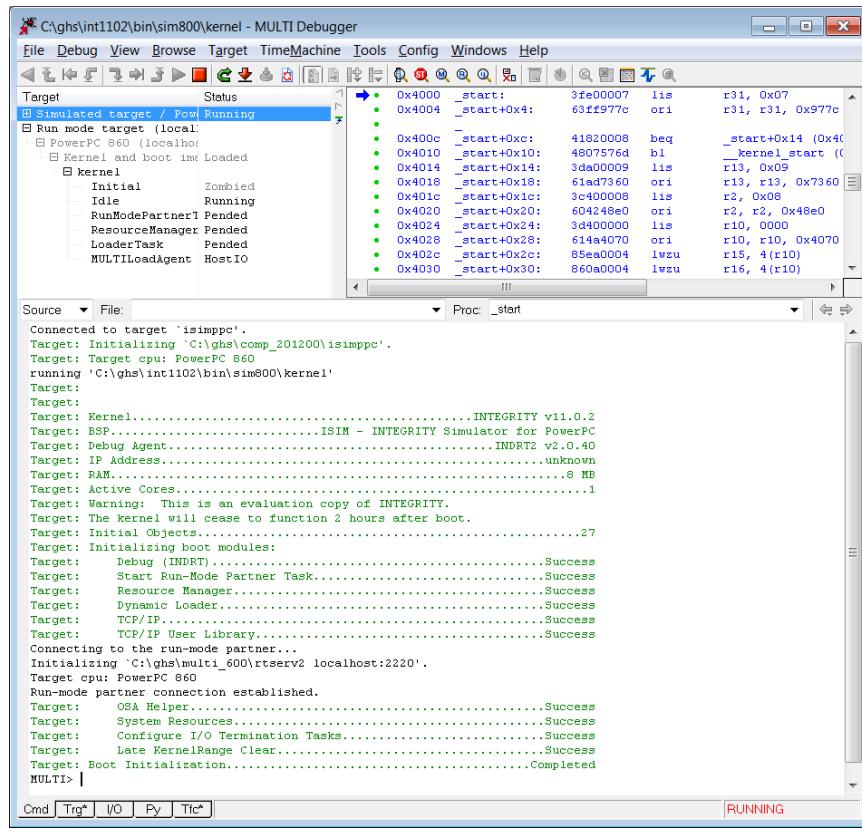
- Task and AddressSpace names must be encoded one ASCII character per octet (byte).
- Task and AddressSpace names must be between 1 and 31 ASCII characters in length, followed by a null byte (i.e., null-terminated).
- Task and AddressSpace names must contain only ASCII characters 32 (space “ ”) through 127 (tilde “~”).
- Task and AddressSpace names must not contain backslash (“\”), forward slash (“/”), double quote (“””), asterisk (“*”), or question mark (“?”). This restriction will be enforced in a future version of INTEGRITY.
- Task names should be unique.

If a Task or AddressSpace name is displayed that contains the string `name_too_long` or `invalid_name`, it indicates that the name violates one or more of these restrictions.

11.2 Using the MULTI Debugger with rtserv2

After successful connection to INDRT2 (rtserv2), the MULTI Debugger Target list displays all of the active tasks in the INTEGRITY system by name. It will take a couple of seconds for the window to fill completely after the connection, and it is updated regularly afterwards.

The following graphic demonstrates the basic look and feel of the MULTI Debugger Target list. (The tasks you see upon connecting to an INTEGRITY target system may not be the same.)



The first time you open the MULTI Debugger, the Target list is located at the top of the window, with the source pane below. INTEGRITY users should move the source pane to the right side of the Debugger window (as shown in the picture above). To do this, click the **Move target list to left** button (), located to the right of the column headings. MULTI remembers the location of the Target list across sessions.

The MULTI Debugger Target list displays the following information:

- The **Target** column displays the name specified by the user. Some special tasks, such as the LoaderTask used for Dynamic Downloading, are created by the INTEGRITY kernel.
- The **Status** column indicates the last known status of the task. Be aware that because a task's status may change often and quickly, not all task status changes are displayed in the window. Possible values include:
 - **Incipient** — The Task has just been created in a halted state and may not be fully initialized.
 - **Created** — The Task has been created and is halted at its user entry point.
 - **Pended** — Task is sleeping for some reason.
 - **Running** — Task is executing.
 - **Halted** — Task is halted.
 - **DebugBrk** — Task is stopped at a breakpoint.
 - **SUSPENDED** — Task has been halted due to a memory violation.

- **SysHalt** — System has been placed into halt mode, no tasks are running.
- The **CPU** column displays the approximate percentage of the CPU that each task is using.

11.2.1 Controlling Target Settings

To control target settings in the MULTI Debugger:

- Select **Debug⇒Target Settings** to see target settings that can be enabled or disabled.

Most of the items in the target settings menu are self-explanatory. Some of the important selections are described here.

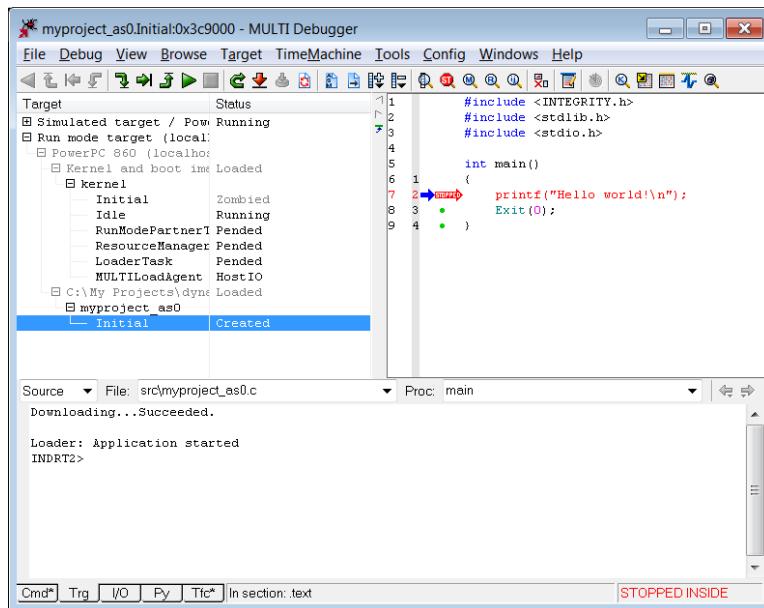
- When **Stop on Task Creation** is enabled, newly spawned tasks are artificially kept from running except by control of the Debugger. This allows you to catch and debug Tasks that would normally run by before you can react.
- When **Stop on Task Creation** is disabled, newly spawned tasks run as normal, but they are still displayed in the list of INTEGRITY tasks in the MULTI Debugger Target list. To manually attach and debug a task, click the appropriate entry in the MULTI Debugger Target list. The code for that task will appear in the Source Pane.
- When **Halt On Attach** is enabled, the Debugger halts a task when you click it to attach to it in the MULTI Debugger Target list. By default, **Halt on Attach** is disabled when first connected to an INTEGRITY target.

11.3 Using the Target Pane

After you connect to INDRT2 (rtserver2), you can use the Target pane for entering user commands to the debug server and to display debug server messages. To use the Target pane:

- Select the **Trg** tab at the bottom of the MULTI Debugger.

When a Dynamic Download is performed, the MULTI Target pane displays the Dynamic Download status as shown in the following:



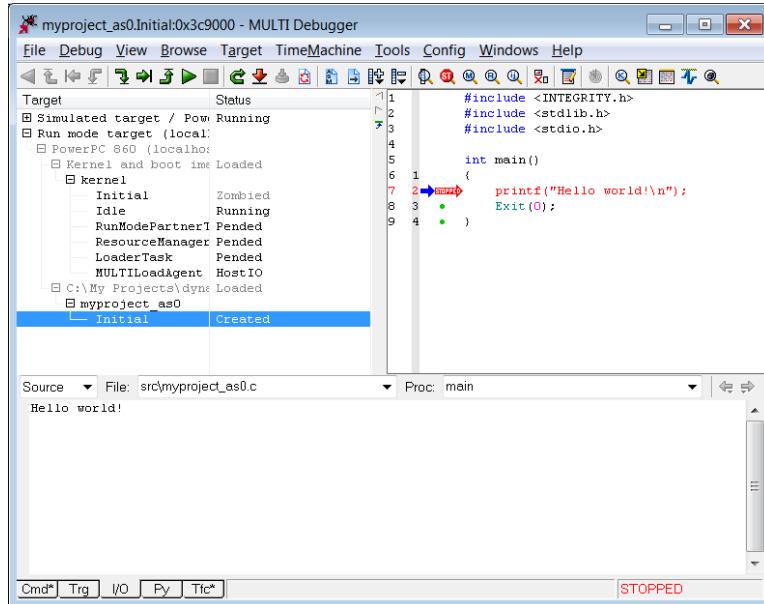
You can use the Target pane to issue debug agent commands. These commands allow for task control and inspection of operating system objects in a text mode command interface and are described in the “Debug Agent Commands” chapter.

11.4 Using the I/O Pane

You can use the **I/O** pane at the bottom of the MULTI Debugger as a virtual terminal. It displays output directed to standard output (stdout). Input to this window goes to standard input (stdin). This is a useful debugging tool. For example, to note when a task executes a piece of code, you can insert a message as follows:

```
printf("Hello world!\n");
```

The string will be displayed in the I/O pane when the code is executed:



11.4.1 Host I/O

Host I/O allows an INTEGRITY system to perform file input and output, using a Debugger to perform the operations on the host platform. INTEGRITY Tasks can open, read, and write files (including stdin, stdout, and stderr which are emulated in the MULTI I/O pane).

The following are examples of what an INTEGRITY Task can execute:

```
printf("Hello, world\n");
cout << "Hello, world" << endl;
hostio_fopen("c:\\\\ghs\\\\int40\\\\infile", "r");
```

Tasks must be attached and be debugged with the MULTI Debugger while Host I/O is in use. Only the task that opened the Host I/O file has access to it. Attempts to use the file descriptor by other tasks will fail.

For unattached tasks, only writes to the standard output are supported. If an unattached task executes another type of Host I/O operation, such as opening a file, the task may hang (pending) attempting to execute the operation. A hung task can be attached to and continued, but the results of the operation will be undefined.

Host I/O only supports the following file calls:

- open()
- access()
- creat()
- rename()
- truncate()
- unlink()
- stat() (partial support)
- read()
- write()
- fstat() (partial support)
- lseek()

Note: C FILE routines and C++ iostream routines also work over Host I/O because they use the above primitives.

You can use Host I/O alone, or you can use it with a file system. To use Host I/O only, link in **libhostio.a**. Do not link in any other file system such as VFS.

When you use Host I/O with a file system, any call that takes a file descriptor will go through Host I/O if the file descriptor refers to a Host I/O file.

To use Host I/O for calls that take a path, include the header file **<unistd.h>** and use the following calls. These calls are exactly the same as the corresponding POSIX calls, but they force the operation to go through Host I/O:

```
int    hostio_open(const char* filename, int flags);
int    hostio_open2(const char* filename, int flags, int mode);
FILE* hostio_fopen(const char* filename, const char* mode);
int    hostio_access (const char *path, int mode);
int    hostio_creat(const char *filename, int prot);
int    hostio_rename(const char *oldName, const char *newName);
int    hostio_stat(const char *name, struct stat *buf);
int    hostio_truncate(const char *name, off_t length);
int    hostio_unlink(const char *filename);
```

For information about using Host I/O for stop-mode debugging, see the “Using Host I/O” section of the “Freeze-Mode Debugging” chapter.

11.5 Using Breakpoints with Run-Mode Debugging

rtserv2 supports both hardware and software breakpoints. Hardware breakpoints are set using processor-specific special purpose registers. Software breakpoints are implemented by replacing the normal application instructions with a special trap/breakpoint instruction.

Execution breakpoints include all software breakpoints and any hardware (or virtual memory) breakpoints that trigger on execution. Execution breakpoints are hit when they are reached, before the corresponding application instruction is executed. If an execution breakpoint is set on the current PC of a halted Task, and the Task is then resumed, the breakpoint will not be hit immediately. Execution must leave and then return to that address before the breakpoint will be hit.

Data breakpoints are hardware breakpoints that trigger on reading or writing to memory. Data breakpoints may be hit before the read/write occurs, or after the read/write has completed, depending on the architecture being used.

rtserv2 does not support overlapping execution breakpoints or overlapping of execution and data breakpoints within a single AddressSpace, including breakpoints set on different Tasks within the same AddressSpace. Developers are responsible for ensuring that breakpoints within an AddressSpace do not overlap because MULTI does not generally detect overlapping breakpoints and will allow you to set them. The resulting behavior is not defined and no guarantees are made as to how many execution breakpoints will be hit, or the order in which these breakpoints may be hit.

11.5.1 Using Software Breakpoints with rtserve2

Software breakpoints are implemented by replacing the target instruction with a special trap/breakpoint instruction. This causes a trap and enables the Debugger to gain control. The program text must be writable for software breakpoints.

In the MULTI Debugger, software breakpoints are set with Task breakpoints. Common methods to set Task breakpoints are:

- Click a green dot in the MULTI Debugger for an attached Task (a red stop sign will display).
- Use the **b** Debugger command.

The breakpoint is installed into target memory when the Task is resumed. INTEGRITY installs and removes software breakpoints as appropriate at context switch time so a software breakpoint set for a particular Task cannot affect any other Tasks in the system. When a Task is detached, the MULTI Debugger relinquishes control and removes all software breakpoints before resuming the Task.

Setting breakpoints on and single-stepping of instructions that branch to themselves may not work as expected on some architectures. While the MULTI compiler rarely generates code with instructions that branch to themselves, developers should be aware of this limitation.

A maximum of 63 software breakpoints can be set at a time. Customers with kernel source installation can modify this maximum in **libdebug.a**. For most kernels, a maximum of 126

combined AddressSpaces and Tasks can be debugged with rtserver2 by default. This maximum can be modified by specifying `__INTEGRITY_DebugEntities` in the linker directives file, as described in “Number of Debug Entities” earlier in this chapter.

11.5.2 Using Hardware Breakpoints with rtserver2

The target debug agent supports two memory access breakpoint mechanisms, namely, hardware breakpoints and virtual memory breakpoints. From the point of view of the MULTI Debugger, these mechanisms are indistinguishable and are referred to collectively as “hardware breakpoints”. For the purpose of discussing how to use breakpoints with rtserver2, we make the distinction between these two terms.

Hardware breakpoints use processor specific special purpose registers to instruct the processor which address ranges and access types (read, write and/or execute) should trigger the breakpoint trap. Hardware breakpoints are a very limited resource and not all processors provide hardware breakpoints. Hardware breakpoints are useful because they allow you to detect when data is being accessed. Hardware breakpoints can also allow you to detect when program execution reaches a particular address, and these hardware breakpoints are permissible even when your program is executing out of ROM.

Virtual memory breakpoints are implemented using the processor’s memory management unit (MMU) to protect pages of memory. Unlike hardware breakpoints, virtual memory breakpoints are limited only by the number of breakpoints your debug agent supports. However, virtual memory breakpoints impose a significant performance reduction on tasks running with virtual memory breakpoints set. In addition, virtual memory breakpoints cannot be used with KernelSpace Tasks.

rtserver2 will first attempt to set a hardware breakpoint. If that fails, it will try the target again, requesting a virtual memory breakpoint.

For virtual tasks, if the hardware does not support setting hardware breakpoints, rtserver2 sets hardware breakpoints by unmapping the page of the desired address. On the subsequent address fault the kernel checks if a hardware breakpoint has been hit. Using hardware breakpoints can cause performance degradation if many instructions access memory on the same page that is unmapped due to the hardware breakpoint.

Chapter 12

Dynamic Downloading

This chapter contains:

- Introduction to Dynamic Downloading
- LoaderTask
- Initiating a Dynamic Download
- Dynamic Download Status
- Unloading or Reloading and Application

12.1 Introduction to Dynamic Downloading

Dynamic Downloading is the mechanism used to run a set of virtual AddressSpaces on a currently running kernel.

The basic steps for Dynamic Downloading with MULTI are:

1. Adding the LoaderTask to an AddressSpace — see “LoaderTask” in this chapter.
2. Connecting with rtserver2 — see “Connecting to rtserver2” in the “Connecting with INDRT2 (rtserver2)” chapter.
3. Selecting an image for download — see “Initiating a Dynamic Download” in this chapter.

There is also an API for Dynamic Downloading modules. For information, see the “Dynamic Downloading Library” chapter in the *INTEGRITY Libraries and Utilities User’s Guide* for more information.

For information about creating a project suitable for Dynamic Downloading, see “Dynamic Download INTEGRITY Application Project” in the “Building INTEGRITY Applications” chapter.

12.2 LoaderTask

Note: Linking the Dynamic Load library (**libload.a**) into KernelSpace has been deprecated and will not be supported in a future release. For future compatibility, add the LoaderTask as a virtual module.

The LoaderTask should be added as a virtual module to an INTEGRITY monolith. The benefit of using the virtual LoaderTask is that it allocates a fixed set of resources for the purpose of loading, most importantly a fixed set of physical memory. (When you use the KernelSpace LoaderTask, the physical memory for loaded images comes from the common kernel free page list, which can affect other kernel Tasks that may want free pages.) With either the virtual or physical method, the LoaderTask entry must be visible in the Task Manager before attempting a download with the MULTI Debugger. The LoaderTask runs at high priority when a download is in progress and shares the CPU with any other Tasks at the same priority. When there is no download in progress, the LoaderTask is pended and has no effect on other Tasks in the system.

To add the LoaderTask as a virtual module in a monolith INTEGRITY Application:

1. Link the KernelSpace program with the library **libvload_kernel_helper.a**.
2. The **ram_sections.id** file included by the **default.id** link map for the KernelSpace program contains the necessary page-aligned section **.mr_rwe__download**, which allows the virtual loader to work with rtserv2. If you are using a custom **.id** file, make sure this section is included or copied from the **ram_sections.id** file.
3. Link the desired virtual AddressSpace with **libvload.a**, unless you are already using a library that includes support for virtual loading (for example, **libposix_sys_server.a**).
4. Edit the **.int** file for the application to specify non-default attributes. See **posix_system.int** in *install_dir\modules\ghs\posix_sys\posix_system.gpj* for an example. Copy the attributes of the **posix_system_server** AddressSpace and adjust the MemoryPoolSize to accommodate the memory requirements of the programs you plan to download. Make sure to also copy the Initial Task attributes, such as **TimeSlice MAXIMUM**.

To add the LoaderTask to KernelSpace:

- Link the KernelSpace program with the library **libload.a**.

The KernelSpace Initial Task will automatically create and start the LoaderTask.

12.2.1 Download Area Requirements

The LoaderTask uses the download area for temporary storage of data during the loading process. The memory for the area comes from different places, depending on which loader variety is used.

- For the KernelSpace loader, the **.download** section defined in the **ram_sections.id** file included by the **default.id** link map is used.

- For the virtual loader, the `.mr_rwe__download` memory reservation is used.

Regardless of which loader is used, the same set of rules and requirements apply.

In previous versions of INTEGRITY, the download area was used to hold the entire dynamic download image while it was being loaded, which meant it had to be large enough to hold the largest dynamic download that could be attempted on the system. Beginning with INTEGRITY 10, the loader can download images that are much larger than the available download area. This is accomplished using a technique called “Sparse Downloading”, which only uses the download area to store certain important parts of the dynamic download image and the bulk of the data is transferred directly into target memory. This method does incur some performance penalty, but substantially reduces the size requirement for the download area.

The loader now uses the download area in the following way:

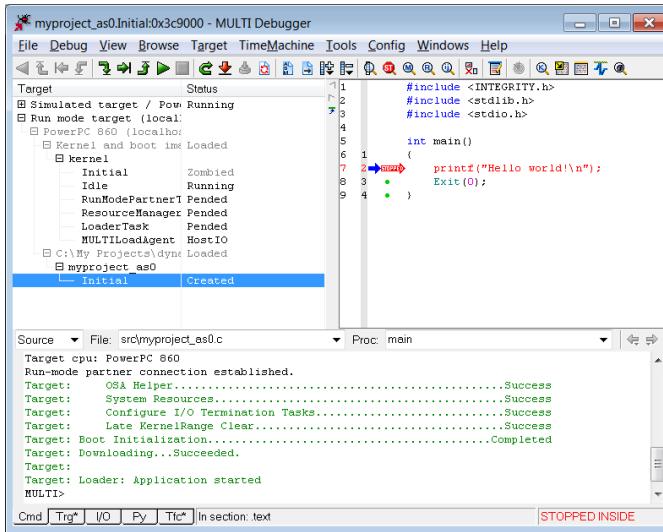
- If the download area is large enough to hold the entire image, sparse downloading is not used. Instead the entire image is transferred.
- If the download area is not large enough to hold the entire image, sparse downloading is used. In this case, the download area must be large enough to fit parts of the dynamic download image such as the section headers, relocation information, the `.boottable` section, the symbol table, etc. This typically takes up a few hundred kilobytes of space, depending on the image. The actual section contents are transferred directly into the target locations in memory.
- If the download area is not large enough to hold the information described above, the download will fail.

12.3 Initiating a Dynamic Download

To initiate a Dynamic Download do one of the following:

- Right click the Dynamic Download project in the Project Manager and select **Load Module executable**.
- In the MULTI Debugger select **Target⇒Load Module** and select one of the modules from the most-recently used list, or select **Load Module** and specify the name of INTEGRITY application (`.ael` file) in the Load Which Object dialog.

The download will proceed after the module name is entered or selected, provided that the LoaderTask and the MULTILoadAgent are running.



The download image must be created from a Dynamic Download project. A Dynamic Download project has the type **INTEGRITY Application** and either has **-dynamic** set or includes a **.int** file that indicates the project is a Dynamic Download.

If your Dynamic Download project only contains one virtual AddressSpace, make sure you download the whole image and not the executable for the virtual AddressSpace. For example, in **rtos_dir\bspname\default.gpj**⇒**examples.gpj**⇒**apiexamples.gpj**⇒**helloworld1.gpj**:

- **helloworld1** is the Dynamic Download project. This is the correct file to download with rtserver2.
- **hello1** is the virtual AddressSpace executable.

For more information about Dynamic Download projects, see “Dynamic Download INTEGRITY Application Project” in the “Building INTEGRITY Applications” chapter.

Any debugging sessions involving Tasks executing in the downloaded virtual AddressSpace should be terminated before attempting a download that will cause the same image to be reloaded. Tasks and any other memory associated with the original download are reclaimed before the next download is processed by the LoaderTask.

12.4 Dynamic Download Status

There are three possible results of a download. The MULTI Target Pane will display the status.

- If the Target pane displays **Downloading....Succeeded**, the virtual AddressSpaces were successfully loaded and initialized by the LoaderTask. The Initial Task for each new AddressSpace will be displayed in the Task Manager. Any new task can be attached and debugged by clicking the appropriate entry in the Task Manager.

- If the Target pane displays **Downloading...Failed**, it is most likely the result of loading a bad executable. The program might not have been built from a valid Dynamic Download project, or memory might be exhausted by attempting to load too many images. The amount of physical memory provided for Dynamic Downloads is dependent on the size of the KernelSpace .download section, and the total amount of RAM available. If a download fails, it could mean that the .download section size needs to be increased or that there is simply not enough RAM on the board for the number of applications to run simultaneously. If console messages are enabled, information regarding the download may help you find the source of the problem.
- If no result is displayed, it indicates that the LoaderTask was not able to complete the loading process for this program. It could mean that for some reason the LoaderTask was not able to run, or that an internal error occurred. If console messages are enabled, information regarding the download may help you find the source of the problem.

12.5 Unloading or Reloading an Application

After an INTEGRITY application has been downloaded, it can be either reloaded or unloaded.

1. Select the application in the MULTI Debugger Target list.
2. Right-click and select either **Unload Module** or **Reload Module**.

Images with different names can be Dynamically Downloaded and live on the target simultaneously. Because information is stored for each downloaded image, the LoaderTask uses the dedicated .download section to store the downloaded image.

To unload a specific downloaded images, do one of the following:

- Select **Target⇒Unload Module** from the Task Manager.
- Enter the following MULTI Target pane command:

```
unload filename
```

- Type the following command into a MULTI Debugger window:

```
target unload filename
```

Note: For the purposes of unloading an image, the path is ignored by the LoaderTask. Unloading **/tmp/filename** or **filename** will have the same effect.

Any debugging sessions involving tasks executing in the downloaded virtual AddressSpace should be terminated before unloading or attempting another download that will cause the same image to be reloaded. The tasks and any other memory associated with the previous download are reclaimed before the next download is processed by the LoaderTask.

Chapter 13

Profiling with rtserver2

This chapter provides information about using MULTI profiling and contains the following sections:

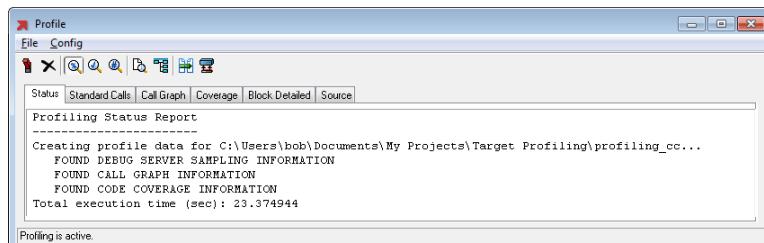
- Using the MULTI Profile Window
- Generating Profile Data Programmatically
- Profiling Overhead and Limitations

13.1 Using the MULTI Profile Window

Profiling helps you locate performance bottlenecks in your code. You can use the MULTI Profile window for an INTEGRITY target after a connection with the run-mode debug server, **rtserver2**, has been established. (For more information, see the “Connecting with INDRT2 (rtserver2)” chapter.) The following types of profiling information can be gathered and analyzed in the MULTI Profile window:

- PC Samples
- Block Coverage Profiling
- Call Count Data

After MULTI has read in the profile data, you can use the MULTI Profile window to analyze the data. For example, you can select **Standard Calls** tab to bring up a list of all the functions the Task executed, sorted by execution time.



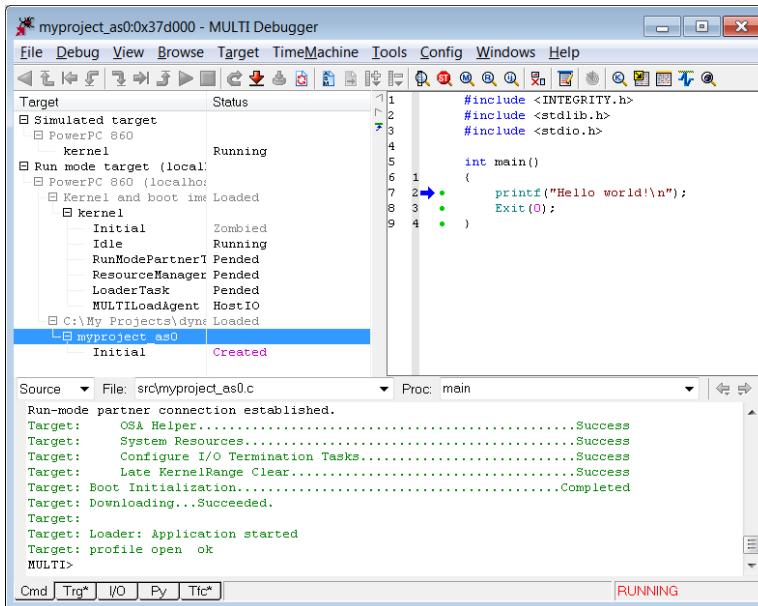
The following sections provide information about using the MULTI Profile window for each of these information types. For more information about using the MULTI Profile window, see the “Collecting and Viewing Profiling Data” chapter of the *MULTI: Debugging* manual.

13.1.1 Program Counter (PC) Samples

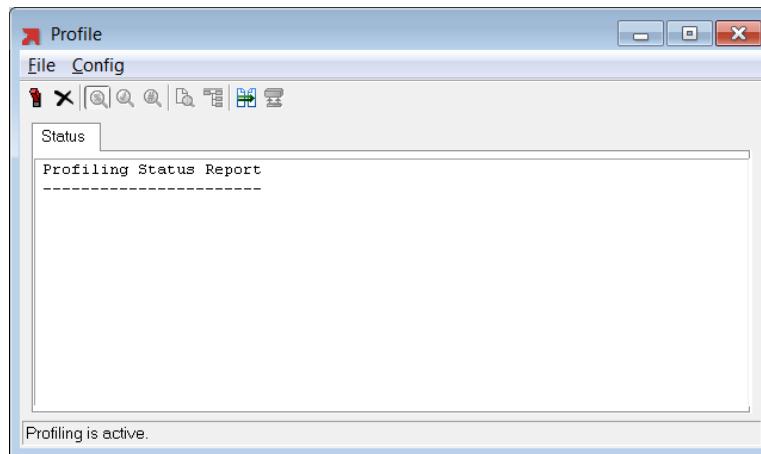
Program counter (PC) samples contain data about where the Task spends its time. You can view how much time is spent in each function, each basic block, each source line, each assembly instruction, and the entire program. You may be able to make execution faster by using this information to improve code that accounts for an unnecessarily large percentage of the total execution time.

MULTI profiling relies on a sampling mechanism in the target to collect program counter samples over a long period of execution. No recompilation or re-linking is needed to generate execution time profiling data for a Task, virtual AddressSpace, or the kernel AddressSpace. The following instructions apply to profiling all types of PC samples:

1. The first step is to determine the scope of execution you wish to sample. You control the scope of execution by selecting either a Task, virtual AddressSpace, or the kernel AddressSpace entry from the MULTI Debugger Target list.
 - If you select a Task, only the execution of that Task will be sampled. This is known as “Task Profiling”.
 - If you select a virtual AddressSpace, all Tasks executing in that AddressSpace will be sampled. This is known as “AddressSpace Profiling”.
 - If you select the kernel AddressSpace, you enable a special case known as “System Profiling”. System Profiling includes execution that takes place within the kernel itself, in addition to any Task that executes in the kernel AddressSpace.



2. Select **View⇒Profile** in the Debugger to open the MULTI Profile window. This window should remain open for the duration of the example.



3. If Tasks are halted, run them by clicking the MULTI Debugger's **Go** button. Allow Tasks to run for at least several seconds.

Profile data is only gathered while Tasks are executing. For Task profile data, the Task you are profiling must be running; for AddressSpace profile data, all Tasks you want to profile in the AddressSpace must be running.

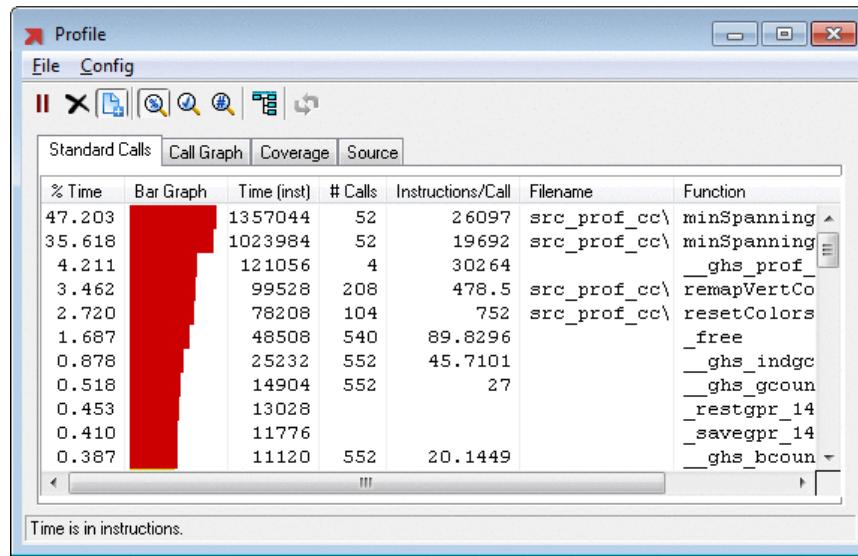
4. When satisfied with the sample, click the **Process Data** button in the Profile window. MULTI will read in the gathered data and the Profile window should open with a message similar to the following:

Creating profile data for ...

FOUND DEBUG SERVER SAMPLING INFORMATION

- If the execution time displayed is 0, it most likely means that your program has not run long enough to return valid profile data.
- If information was not gathered correctly, the message `Need fresh data from a program run to process` will display and you should restart the process.

5. After profile data has been processed, the analysis features of MULTI profiling can be used. Select the **Standard Calls** tab to see a summary of processing time per function.



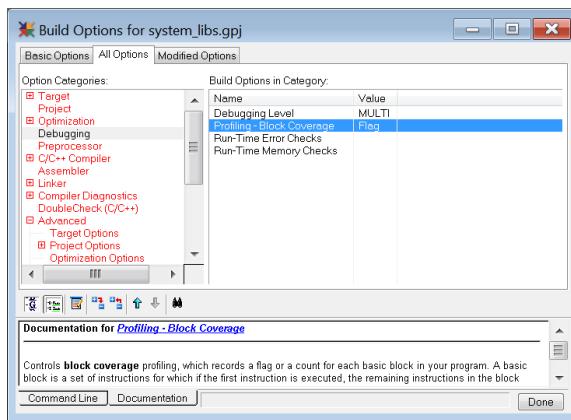
6. Only one Task or AddressSpace may be profiled at a time. To profile a different Task or AddressSpace, close the MULTI Profile window and start again by selecting another Task or AddressSpace in the Debugger Target list.

13.1.2 Block Coverage Profiling

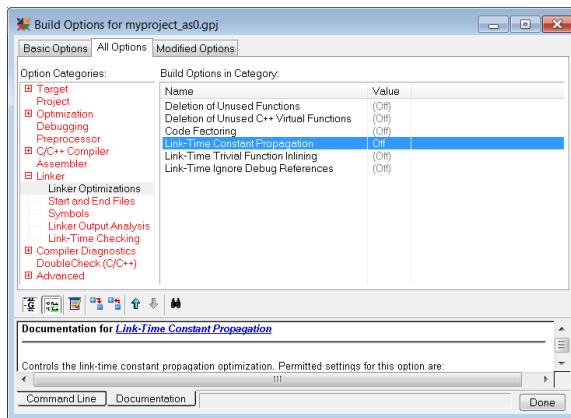
Block Coverage profiling provides a profile of basic block executions. This data can be used to determine which parts of a program are never executed during a profiling session. If a given basic block is never executed, the group of instructions making up the block is considered dead code. Because the application never reaches the dead code, you can either remove the code from the application, or try to discover if any functionality is missing from the application as a result.

Both KernelSpace and virtual AddressSpaces can be profiled for Block Coverage. To enable Block Coverage profiling, application code must be recompiled and re-linked as follows:

1. Right-click the **.gpj** file for the AddressSpace in the MULTI Project Manager and select **Set Build Options**.
2. On the **All Options** tab, select **Debugging** in the Categories pane. In the Build Options pane, right-click **Profiling - Block Coverage** and select **Off, Flag, or Count (32-bit counters)**.



- In the Categories pane, expand **Linker** and select **Linker Optimizations**. If **Link-Time Constant Propagation** is listed in the Build Options pane, right-click it and select **Off**.



- Close the Build Options window and rebuild the .gpj file for the INTEGRITY Application containing the AddressSpace.
- Establish an rtserver connection and run the re-built application.
- Select a user-created Task in the AddressSpace you wish to profile.
- In the Debugger, select **View⇒Profile** to open the Profile window.

Target-resident buffers, defined by the compiler, are used to hold the Block Coverage data. The Initial Task must call the C library exit() function to automatically flush the profile data to the host disk via Host I/O.

- If the Initial Task does not call exit() on its own, halt the selected Task, then click the **Dump Profiling Info** button in the MULTI Profile window to flush the profile data.

Block Coverage data will be available after the Initial Task executes the exit() function or

profiling data has been dumped. The profile data is written to **bmon.out** in the current working directory on the host.

- Click **Process Data**. If coverage analysis data was successfully read in, a message such as the following will display in the MULTI Profile window:

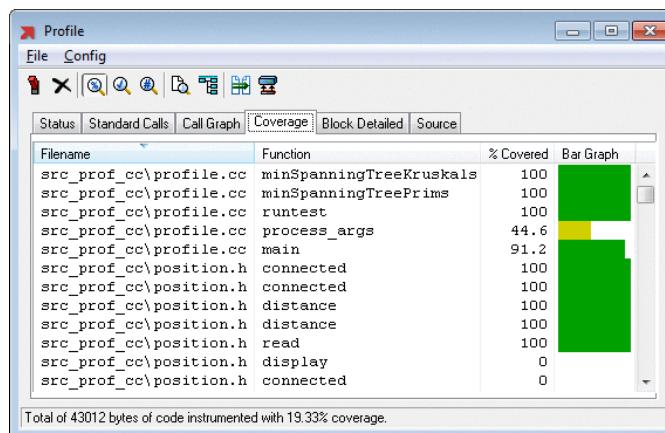
Creating profile data for prog...

FOUND DEBUG SERVER SAMPLING INFORMATION

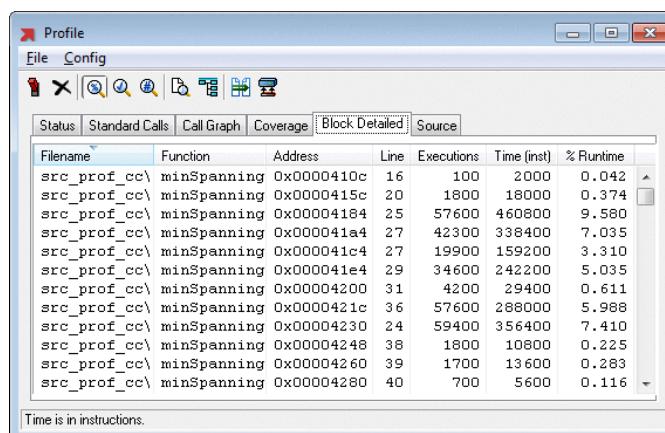
FOUND CODE COVERAGE INFORMATION

- Now the analysis features of MULTI Profile window can be used.

- Select the **Coverage** tab to see the basic Block Coverage analysis report.



- Select the **Block Detailed** tab to see detailed Block Coverage analysis report.



11. After you collect profiling data, you can use the buttons in the Profile window to view profiling information directly in the source pane of the Debugger:
 - Click the **Count View** button in the Profile window to view the total number of times each line (or instruction) was executed. The number of executions is shown to the left of each source line or instruction in the Debugger.
 - Click the **Coverage View** button in the Profile window to view unexecuted source lines or instructions. Any unexecuted source lines or instructions will be highlighted in the Debugger.

For more information, see “Viewing Profiling Information in the Debugger” in the *MULTI: Debugging* manual.

13.1.3 Call Count Data

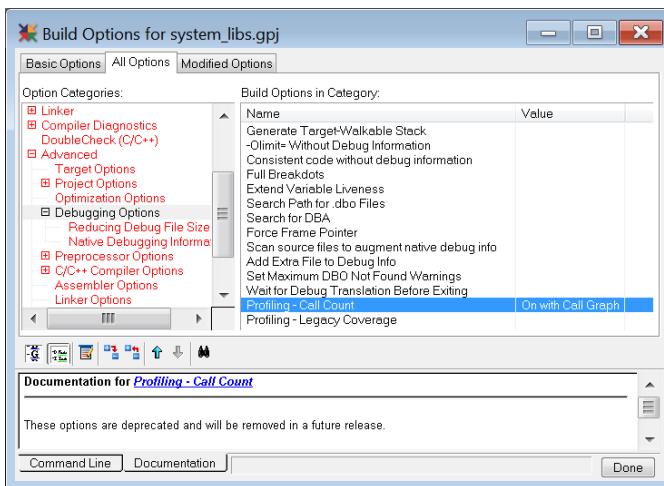
Call count profiling entails determining the number of times each function is executed within the selected AddressSpace. This type of profiling can be performed with or without call graph support.

- Call Count data without graph support records how many times each function is called.
- Call Count data with call graph support records how many times each function is called, and also which child functions are called by each parent function and how many times each child is called. **Note:** Call graph support has been deprecated and will be removed in a future INTEGRITY release.

Note: The Call Count logging library is thread-tolerant, which means that although all the Tasks within an AddressSpace will be profiled for their calls, there is no guarantee that every call will be logged. However, this is not a significant limitation for locating bottlenecks in application performance.

Both KernelSpace and virtual AddressSpaces can be profiled for Call Counts. However, virtual code linked with shared libraries cannot support call graphs; your application must be linked with static libraries. To profile Call Count data, application code must be recompiled and re-linked with Call Counts enabled as follows:

1. Right-click the **.gpj** file for the AddressSpace in the MULTI Project Manager and select **Set Build Options**.
2. On the **All Options** tab, select **Advanced⇒Debugging Options**.
3. In the Build Options pane, right-click **Profiling - Call Count**, and set to **On** or **On with Call Graph**.



4. Rebuild the **.gpj** file for the INTEGRITY application containing the AddressSpace.
5. Establish an rtserver2 connection and run the re-built application.
6. Select a Task in the AddressSpace you wish to profile.
7. In the Debugger, select **View⇒Profile** to open the Profile window.
8. If the Initial Task does not call exit() on its own, halt the selected Task, then click the **Dump Profiling Info** button in the MULTI Profile window to flush the profile data. Call Count data will be available after the Initial Task executes the exit() function or profiling data has been dumped.

If you get an error similar to the following:

```
mcount: could not allocate memory
```

it indicates the size of the `.heap` needs to be increased until buffers can be allocated. The AddressSpace's Initial Task creates target-resident buffers to hold the Call Count data, and the run-time heap is used to allocate these buffers. Depending on the size of the application, the default linker directives files, **default.ld** (for KernelSpace) and **INTEGRITY.ld** for (virtual AddressSpaces), may not have a large enough `.heap` section. For information about how to increase the `.heap` section, see the “Heap Size Configuration” section of this manual or the “`HeapSize [AddressSpace]`” keyword in the *Integrate User’s Guide*.

9. Click **Process Data**. If Call Counts were successfully read in, a message similar to the following is displayed in the MULTI Profile window:

```
Creating profile data for prog...
```

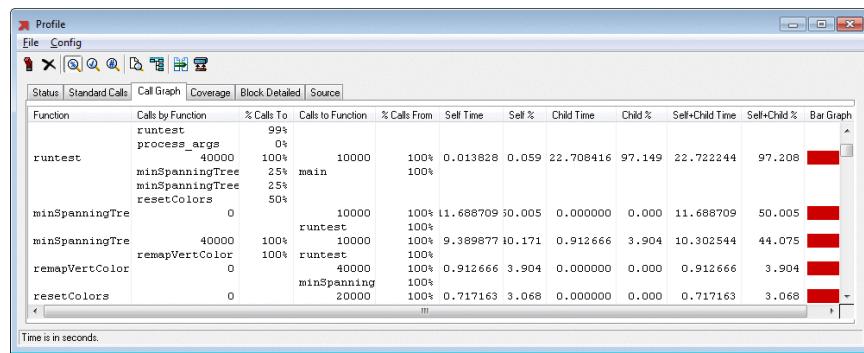
```
FOUND DEBUG SERVER SAMPLING INFORMATION
```

FOUND CALL COUNT INFORMATION

The profile data is written to **mon.out** for Call Counts, or **gmon.out** for Call Counts with call graph support in the current working directory on the host.

- Now the analysis features of MULTI profiling can be used. Depending on whether you selected Call Count profiling with or without call graph support, do one of the following:

- Select the **Call Graph** tab to see Call Count data with call graph support in a tabular format.



- Click the **Dynamic Call Graph** button to see Call Count data with call graph support in a graphical format.

- After you collect profiling data, you can use the buttons in the Profile window to view profiling information directly in the source pane of the Debugger.

- Click the **Count View** button in the Profile window to see the number of times each function was called. The Call Count for each function is displayed to the left of the beginning of each function in the Debugger.

For more information, see “Viewing Profiling Information in the Debugger” in the *MULTI: Debugging* manual.

13.2 Generating Profile Data Programmatically

If profiling is enabled for a project, calling `exit()` causes Call Count or coverage analysis data to be written to disk for MULTI to process. For programs that do not call `exit()`, you can obtain data from the target with either of the following:

- Click the **Dump Profiling Info** button in the Profile window.
- Use a programmatic method. Any Task within the AddressSpace can cause the profile data within that AddressSpace to be written to disk.

Note: Do not Dump Profiling Info when the Task being profiled is Pended.

To use the programmatic method:

1. Use any Task within an AddressSpace to write the AddressSpace's profile data to disk with either of the following:
 - Insert the following call into the code where appropriate:

```
DumpProfileBuffers();
```

- Or, type the following C expression into the MULTI Debugger's command pane:

```
DumpProfileBuffers()
```

When the function call completes, the profile data files should be present in the current working directory on the host (**mon.out** for call counts, **gmon.out** for call counts with call graph support enabled, and **bmon.out** for coverage analysis data).

This function call does not generate PC samples data (see the “PC Samples” section for information about accessing PC sample data.)

2. The associated profile data buffers are not flushed after a call to `DumpProfileBuffers()`. Although the routine can be called more than once, each call causes the entire data set generated since the start of the application's execution to be dumped to disk. To prevent newly read data from being added to data previously read within the same profiling session, select **Config⇒New Data⇒Replaces Old** in the MULTI Profile window.
3. Click **Process Data** in the MULTI Profile window to force MULTI to read in the available data.

13.3 Profiling Overhead and Limitations

The overhead associated with profiling varies based on the type of profiling used:

- PC sample profiling adds no additional memory overhead.
- A small amount of kernel run-time is added in order to process the PC samples. These system calls will cause a noticeable increase in the wall clock time elapsed for the running system.
- Call Count profiling uses the run-time heap to hold profile data. The size of the heap may need to be modified. For more information, see the “Heap Configuration” section in the “Common Application Development Issues” chapter of this manual.
- Each function call that is compiled for profiling has a constant time overhead that is needed to save the profile information into these buffers.
 - Block Coverage analysis uses static data added by the compiler to store the coverage information. Architectures with 32 bit addresses require 8 bytes per basic block of code; architectures with 64 bit addresses require 12 bytes per basic block.
 - The overhead of a load/increment/store operation is added for each basic block as well, which may significantly increase run-time.
- Due to the run-time penalty and increased footprint associated with instrumenting the application, Call Count and Block Coverage analysis profiling are meant for use as debugging tools and not meant for use in final production code.
- Only code executed from a Task Context should be built for profiling. For example, you should not turn on profiling for BSP code.
- PC sample profiling can be enabled with optimized production code by building debugging support into the KernelSpace program. This is accomplished by linking **libdebug.a**.

MULTI profiling relies on a sampling mechanism in the target to collect program counter samples over a long period of execution. In INTEGRITY, the system timer is used as the sampling mechanism and it shows overall execution characteristics. The system timer is not meant to be a benchmark timer; a fine-granularity timer should be used for that purpose. For example, on a PowerPC CPU, the time base register can be read at the beginning and at the end of a selected piece of code.

Chapter 14

Object Structure Aware Debugging

This chapter provides information about Object Structure Aware (OSA) Debugging and contains the following sections:

- Introduction to OSA Debugging
- Using the OSA Explorer
- Using the OSA Object Viewer

14.1 Introduction to OSA Debugging

With INTEGRITY, two Object Structure Aware (OSA) tools provide GUIs for browsing Object information, and for performing kernel aware debugging:

- The **OSA Explorer** is useful for browsing information about an entire operating system. It can be used while debugging in run-mode or freeze-mode.
- The **OSA Object Viewer** is useful for browsing information about an individual INTEGRITY Object. It can only be used while debugging in run-mode.

You can view kernel data structures in GUI windows anytime the processor is halted when employing freeze-mode debugging, such as via a BDM/JTAG port (using **mpserv**), or through a simulator (using **isimppc** or **isimarm**). You can also use the OSA debugging features from a run-mode (rtserv2) connection with a Debugger attached to a KernelSpace Task and the system in a quiescent state (for example, after an emulated system halt).

Note: Even during system halt, the Tasks responsible for system debugging can still be running and modifying the Object listings. As a result, there is a slight possibility of getting inconsistent data while using OSA, which can result in an OSA Explorer error message.

By examining the state of kernel Objects, you may be able to detect programming errors, such as deadlock, that result in incorrect operation. The information in this chapter is meant as an overview of the capabilities. For a tutorial on using these features, see the **KernelAwareDebug** example in the **examples** directory.

14.2 Using the OSA Explorer

The OSA Explorer displays operating system Tasks on the target. The OSA Explorer displays a snapshot in time of kernel state. A system halt operation is used to ensure that the data being read is consistent. If you are using INTEGRITY with a run-mode connection, you can launch the OSA Explorer when the target is not frozen by a System halt and MULTI will automatically perform the system halt. When you close the OSA Explorer, system execution will resume.

When using the OSA Explorer with an rtserver2 connection, your kernel image should be located in the current working directory because MULTI searches the working directory first for debugging information. If your kernel is not located in the working directory, MULTI might attempt to use debugging information from a different kernel, which can result in an error message similar to the following:

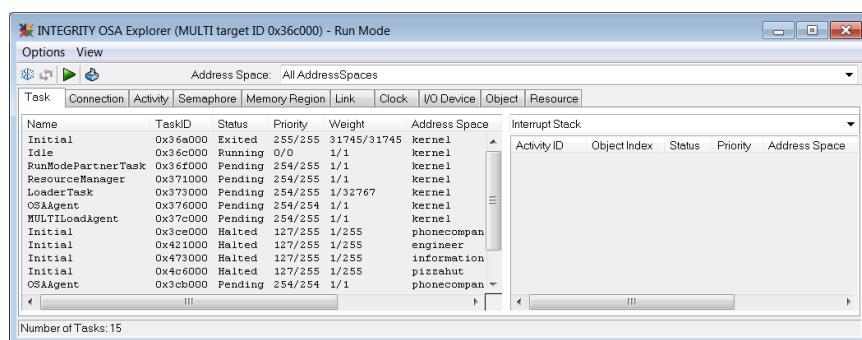
```
Failed to get basic kernel information for RTOS (Integrity).
The kernel may not yet be fully initialized or the symbol
information (below) may not correspond to the code running on the target.
```

To open the OSA Explorer:

1. When debugging in freeze mode, click **Halt**.
2. Select **View⇒OSA Explorer** or click the **OSA Explorer** button.

When debugging with rtserver2, the system will automatically go into SysHalt mode so the data structures can be read from the kernel.

When the OSA Explorer opens, the display will be similar to the following:



The top of the OSA Explorer window contains a drop-down menu to select which AddressSpace to display. You can select All AddressSpaces, but this can cause a performance hit if you have many of them. The bottom portion of the window contains the following tabs which represent the INTEGRITY Objects and resources:

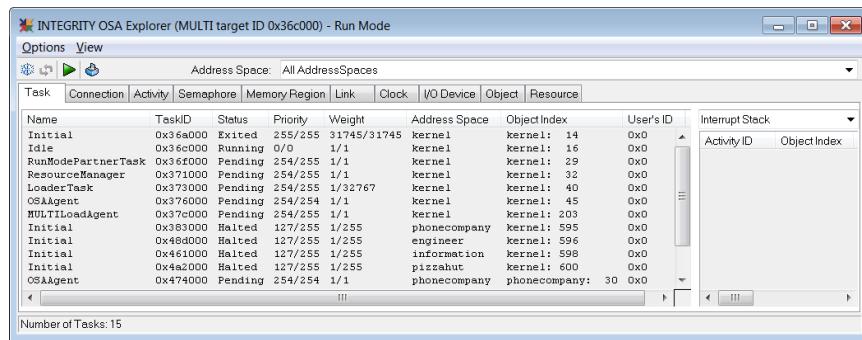
Task | Connection | Activity | Semaphore | Memory Region | Link | Clock | I/O Device | Object | Resource

Select any of these tabs to display all Objects of that type in the selected AddressSpace. The **Object** tab displays a list of all of the Objects in the selected AddressSpace, regardless of type. All of the tabs are described in the more detail in following sections.

When viewing any type of Object, various attributes about those Objects are displayed in the OSA Explorer. You can turn the display of an attribute on or off by right-clicking in the OSA Explorer window and selecting or deselecting the attribute.

14.2.1 OSA Explorer Task Tab

When the **Task** tab is selected, the left pane displays a list of all the Tasks in the selected AddressSpace. The list contains the following attributes:



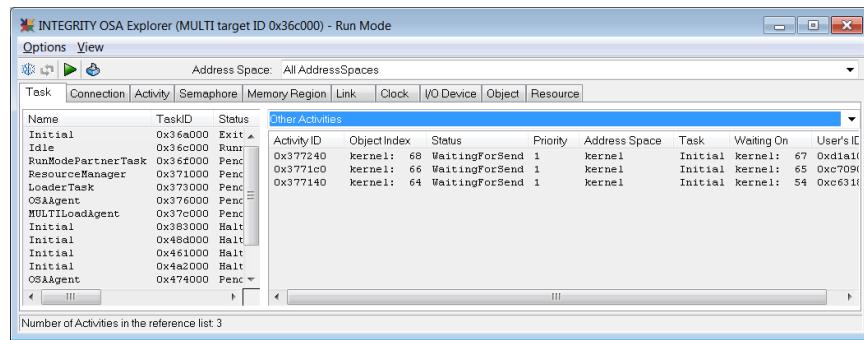
- **Name** — Name of the Task
- **TaskID** — Kernel address of the Task control block
- **Status** — Task status
- **Priority** — Task priority and maximum priority
- **Weight** — Task weight and maximum weight
- **AddressSpace** — AddressSpace in which the Task resides
- **Object Index** — AddressSpace Object index of the Task Object
- **User's Id** — Task Identification as defined by the User at Task creation time
- **Stack Start** — Address of the bottom of the stack
- **Stack End** — Address of the top of the stack
- **Stack HWM/Size** — Task stack high water mark and stack size
- **Executable** — Executable filename for the Task's AddressSpace

Note: To see any non-visible options, right-click in the window, and select them from the shortcut menu.

The right pane of the **Tasks** tab has a drop-down menu that provide additional Task information:

1. Select a Task.

2. Select one of the following from the drop-down menu:
 - **Interrupt Stack** — displays Activities on the Task’s Interrupt Stack
 - **Other Activities** — displays other Activities owned by the Task
 - **Owned Binary Semaphores** — displays Binary Semaphores owned by the Task
 - **Owned HL Semaphores** — displays Highest Locker Semaphores owned by the Task



14.2.1.1 Debugging Tasks with the OSA Explorer

You can debug Tasks with the OSA Explorer. Both KernelSpace and virtual Tasks can be debugged through kernel aware debugging. Tasks can be debugged by selecting them in the OSA Explorer Task list, or in the main Debugger window Target list. To debug Tasks with the OSA Explorer:

1. In the Task list, double-click a Task or right-click a Task and select **Debug Task**. This changes the selected Task in the main Debugger window.
2. Set a breakpoint in the Task.
3. Run the kernel again.
4. When the breakpoint is hit, it will be the active Task, and all standard debugging features will be available.

Note: When breakpoints are hit in virtual Tasks, the Debugger window for the kernel will jump to an incorrect memory address.

The OSA Explorer does not support Task debugging in run-mode. For run-mode Task debugging, use INDRT2 (rtserve2) instead. See the “Using the MULTI Debugger with rtserve2” in the “Run-Mode Debugging” chapter for information about debugging Tasks with rtserve2.

14.2.2 OSA Explorer Connection Tab

When the **Connection** tab is selected, a list of all the Connections in the selected AddressSpace is displayed. The list contains the following attributes:

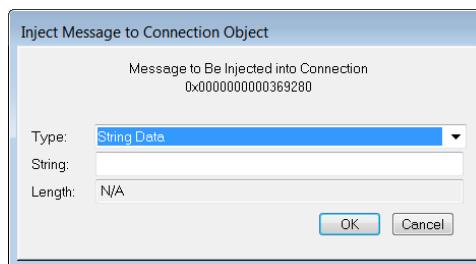
Connection ID	Address Space	Other End	AS of Other End	Object Index	ObjIndex of Other End
0x369280	kernel	0x369800	kernel	kernel: 5	kernel: 27
0x369800	kernel	0x369280	kernel	kernel: 27	kernel: 5
0x369d00	kernel	0x369d80	kernel	kernel: 47	kernel: 49
0x369d80	kernel	0x369d00	kernel	kernel: 49	kernel: 47
0x369e00	kernel	0x369e40	kernel	kernel: 51	kernel: 52
0x369e40	kernel	0x369e00	kernel	kernel: 52	kernel: 51
0x3772c0	kernel	0x377300	kernel	kernel: 70	kernel: 71
0x377300	kernel	0x3772c0	kernel	kernel: 71	kernel: 70
0x37a480	kernel	0x37a4c0	kernel	kernel: 205	kernel: 206
0x37a4c0	kernel	0x37a480	kernel	kernel: 206	kernel: 205
0x47a3c0	phonecompany	0x48e3c0	information	phonecompany: 10	information: 10
0x47a400	phonecompany	0x4623c0	pizzahut	phonecompany: 11	pizzahut: 10
0x47a440	phonecompany	0x3843c0	engineer	phonecompany: 12	engineer: 10

Number of Connections: 16

- **Connection ID** — Kernel address of the Connection Object
- **Address Space** — AddressSpace in which the Connection resides
- **Other End** — Kernel address of the other end Connection Object
- **AS of Other End** — AddressSpace in which the other end Connection resides
- **Object Index** — AddressSpace Object index of the Connection Object
- **ObjIndex of Other End** — AddressSpace Object index of the other end Connection Object

To inject a message on a Connection end:

1. Right-click a **Connection** entry and select **Inject Message**.
2. In the pop-up window, select the type of message to send:



- **String Data** corresponds to Buffer type DataBuffer. If you select String Data, enter a string in the available field and click **OK**.

- **Data Immediate** corresponds to the Buffer type DataImmediate. If you select DataImmediate, enter one integer in each available field (total of 2) and click **OK**.

3. To enable the send, click **Go** in the OSA Explorer.

The send will not complete until the other end has properly received it, or returned an Error. (This will not be useful unless the other end of the Connection is expecting to receive the correct kind of Buffer.)

This feature can be used to control run-time behavior. A Task can be created to wait for a string or integer command on a Connection, then take appropriate action depending on the string or integer received.

14.2.3 OSA Explorer Activity Tab

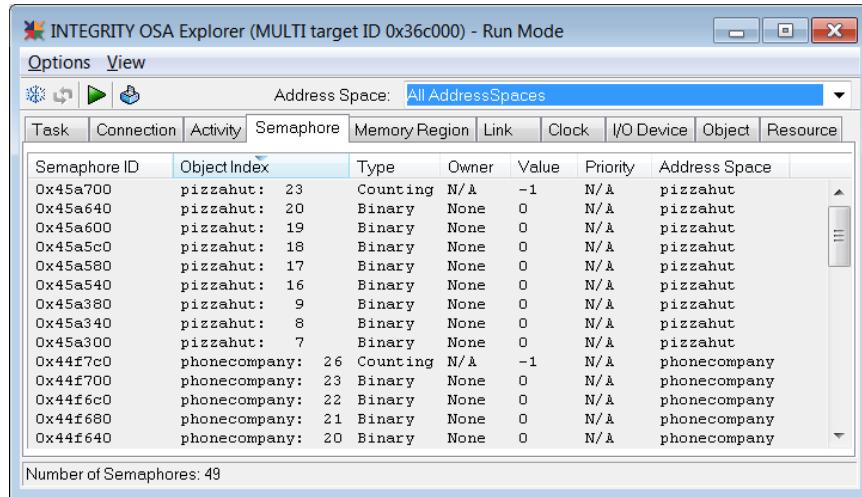
When the **Activity** tab is selected, a list of all the Activities in the selected AddressSpace is displayed. The list contains the following attributes:

ActivityID	Object Index	Status	Priority	Address Space	Task	Waiting On	User's ID
0x369500	kernel: 15	NotRestartable	1	Kernel	Initial	FunList	0x0
0x369500	kernel: 17	RunningTheTask	1	Kernel	Idle	FunList	0x0
0x3698c0	kernel: 30	WaitingForSend	1	Kernel	RunModePartnerTask	kernel: 26	0x0
0x369980	kernel: 33	Waiting	1	Kernel	ResourceManager	N/A	0x0
0x3699c0	kernel: 34	WaitingForSend	1	Kernel	ResourceManager	kernel: 27	0x0
0x369a00	kernel: 35	WaitingForSend	1	Kernel	ResourceManager	kernel: 36	0x1
0x369b00	kernel: 41	Waiting	1	Kernel	LoaderTask	N/A	0x0
0x369cc0	kernel: 46	WaitingForSend	1	Kernel	OSAagent	kernel: 42	0x0
0x369e00	kernel: 48	WaitingForSend	2	Kernel	LoaderTask	kernel: 47	0xcdeadc
0x369e90	kernel: 53	WaitingForSend	1	Kernel	LoaderTask	kernel: 51	0xcdeadc
0x377140	kernel: 64	WaitingForSend	1	Kernel	Initial	kernel: 54	0xc631f
0x3771c0	kernel: 66	WaitingForSend	1	Kernel	Initial	kernel: 65	0xc709c
0x377240	kernel: 68	WaitingForSend	1	Kernel	Initial	kernel: 67	0xdialc

- **Activity ID** — Kernel address of the Activity Object
- **Object Index** — AddressSpace Object index of the Activity Object
- **Status** — Status of the Activity
- **Priority** — Priority of the Activity
- **Address Space** — AddressSpace in which the Activity resides
- **Task** — Task the Activity belongs to
- **Waiting On** — Object the Activity is waiting on
- **User's ID** — Activity Identification as set up by the User at Activity creation time

14.2.4 OSA Explorer Semaphore Tab

When the **Semaphore** tab is selected, a list of all the Semaphores in the selected AddressSpace is displayed. The list contains the following attributes:



- **Semaphore ID** — Kernel address of the Semaphore Object
- **Object Index** — AddressSpace Object index of the Semaphore Object
- **Type** — Type of Semaphore (Counting, Binary, or Highest Locker)
- **Owner** — Task that owns the Semaphore (for Binary and Highest Locker only)
- **Value** — Semaphore Value
- **Priority** — Priority of the Semaphore
- **Address Space** — AddressSpace in which the Semaphore resides

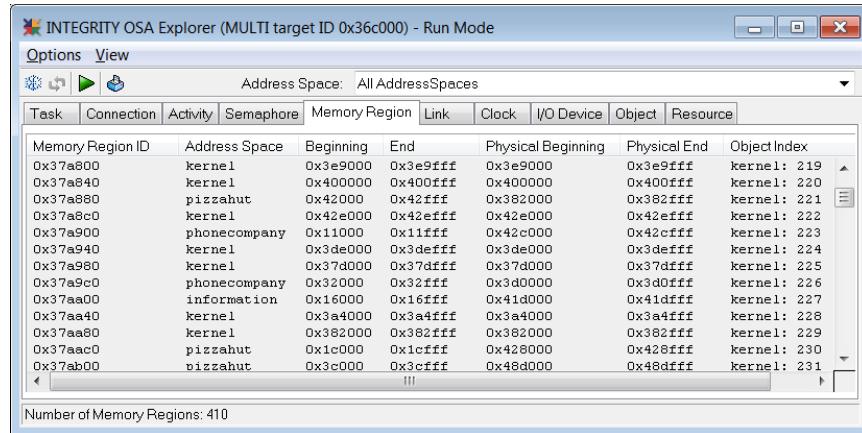
To release or take a Semaphore:

1. Right-click a Semaphore entry and select **Release Semaphore** or **Take Semaphore**.
2. If **Take Semaphore** is selected, a pop-up window will ask which Task context should own the semaphore. Select the desired Task.
3. To enable the semaphore action, click the **Go** button in the OSA window.

This can be used to break a deadlock in the system in order to allow debugging to continue.

14.2.5 OSA Explorer Memory Region Tab

When the **Memory Region** tab is selected, a list of all the MemoryRegions in the selected AddressSpace is displayed. The list contains the following attributes:



- **Memory Region ID** — Kernel address of the MemoryRegion Object
- **Address Space** — AddressSpace in which the MemoryRegion resides
- **Beginning** — Lowest address of the MemoryRegion
- **End** — Highest address of the MemoryRegion
- **Physical Beginning** — Lowest Physical address of the MemoryRegion
- **Physical End** — Highest Physical address of the MemoryRegion
- **Object Index** — AddressSpace Object Index of the MemoryRegion Object

14.2.6 OSA Explorer Link Tab

When the **Link** tab is selected, a list of all the Links in the selected AddressSpace is displayed. The list contains the following attributes:

Link ID	Address Space	Target Type	Target ID	Object Index	Target Index
0x3770c0	kernel	MR: [0x348000:0x367fff]	0x369240	kernel: 62	kernel: 4
0x377100	kernel	MR: [0x7f3000:0x7ffff]	0x3692c0	kernel: 63	kernel: 6
0x377280	kernel	Connection	0x369e40	kernel: 69	kernel: 52
0x44f180	phonecompany	Address Space	0x39c780	phonecompany: 1	kernel: 601
0x44f240	phonecompany	Task	0x31f500	phonecompany: 4	kernel: 439
0x44f280	phonecompany	Connection	0x369280	phonecompany: 5	kernel: 5
0x44f480	phonecompany	Task	0x39c5c0	phonecompany: 13	kernel: 594
0x44f4c0	phonecompany	MR: [0x4500:0x64fff]	0x31f5c0	phonecompany: 14	kernel: 394
0x44f500	phonecompany	MR: [0x4000:0xffff]	0x393a00	phonecompany: 15	kernel: 291
0x44f540	phonecompany	MR: [0xa000:0x2ffff]	0x3f5000	phonecompany: 16	kernel: 379
0x44f580	phonecompany	MR: [0x33000:0x33fff]	0x37adc0	phonecompany: 17	kernel: 242
0x44f5c0	phonecompany	MR: [0x65000:0xffffffff]	0x4380c0	phonecompany: 18	kernel: 446
0x49b180	engineer	Address Space	0x438b80	engineer: 1	kernel: 489
0x49b240	engineer	Task	0x39c5c0	engineer: 4	kernel: 594

Number of Links: 44

- **Link ID** — Kernel address of the Link Object
- **Address Space** — AddressSpace in which the Link resides
- **Target Type** — Type of Object the Link points to
- **Target ID** — Address of Object the Link points to
- **Object Index** — AddressSpace Object Index of the Link Object
- **Target Index** — AddressSpace Object Index of target Object

14.2.7 OSA Explorer Clock Tab

When the **Clock** tab is selected, a list of all the Clocks in the selected AddressSpace is displayed. The list contains the following attributes:

Clock ID	Name	Address Space	Object Index
0x369300	HighResTimer	kernel	kernel: 7
0x369340	StandardTick	kernel	kernel: 8
0x369ec0	HighResTimer	kernel	kernel: 54
0x377180	HighResTimer	kernel	kernel: 65
0x377200	HighResTimer	kernel	kernel: 67
0x44f200	StandardTick	phonecompany	phonecompany: 3
0x44fc00	HighResTimer	phonecompany	phonecompany: 6
0x49b200	StandardTick	engineer	engineer: 3
0x49b2c0	HighResTimer	engineer	engineer: 6
0x414200	StandardTick	information	information: 3
0x4142c0	HighResTimer	information	information: 6
0x45a200	StandardTick	pizzahut	pizzahut: 3
0x45a2c0	HighResTimer	pizzahut	pizzahut: 6

Number of Clocks: 13

- **Clock ID** — Kernel address of the Clock Object
- **Name** — Name of the Clock
- **Address Space** — AddressSpace in which the Clock resides
- **Object Index** — AddressSpace Object Index of the Clock Object

14.2.8 OSA Explorer IODevice Tab

When the **I/O Device** tab is selected, a list of all the IODevices in the selected AddressSpace is displayed. The list contains the following attributes:

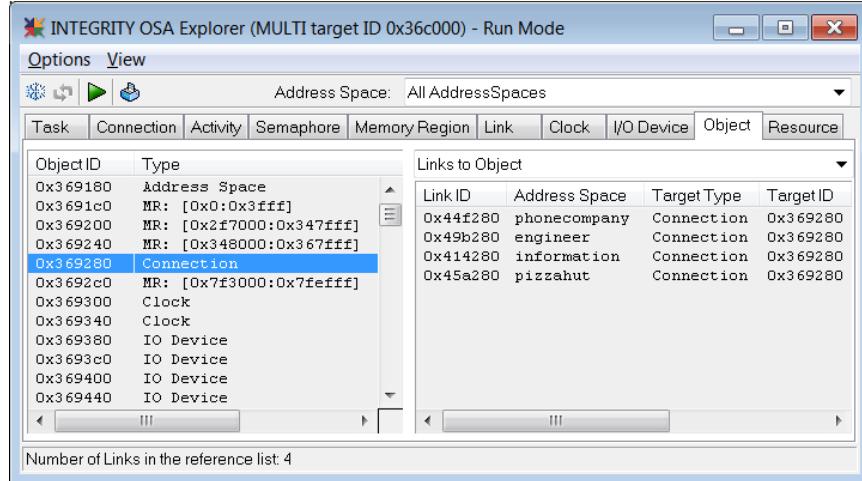
Device ID	Name	Object Index
0x369380	DebugDev[0]	kernel: 9
0x3693c0	DebugDev[1]	kernel: 10
0x369400	DebugNetDevs	kernel: 11
0x369440	ConfigurationIODevice	kernel: 12
0x369480	EtherDev	kernel: 13

Number of IODevices: 6

- **Device ID** — Kernel address of the IODevice Object
- **Name** — Name of the IODevice
- **Object Index** — AddressSpace Object Index of the IODevice Object

14.2.9 OSA Explorer Object Tab

When the **Object** tab is selected, the left pane displays a list of all the Objects in the selected AddressSpace. The list contains the following attributes:

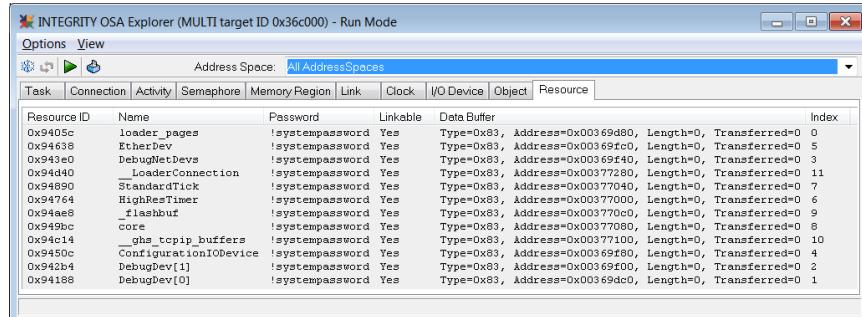


- **Object ID** — Kernel address of the Object
- **Type** — Type of the Object
- **Address** — The Object's resident AddressSpace. This column does not display by default. To view it, select **Address** in the shortcut menu.
- **Object Index** — AddressSpace in which the Object resides and Index within that AddressSpace.

You can select an Object in the list to display all Links to that Object in the right pane of the **Object** tab.

14.2.10 OSA Explorer Resource Tab

When the **Resource** tab is selected, a list of all of the entries in the Resource Manager is displayed. The list contains the following attributes:



- **Resource ID** — Kernel address of the data structure

- **Name** — Name of the resource
- **Password** — Password under which the resource was registered
- **Linkable** — Whether or not the resource was registered as linkable
- **Data Buffer** — More detailed view of the internal kernel data structure
- **Index** — Index in the Resource Manager table

14.3 Using the OSA Object Viewer

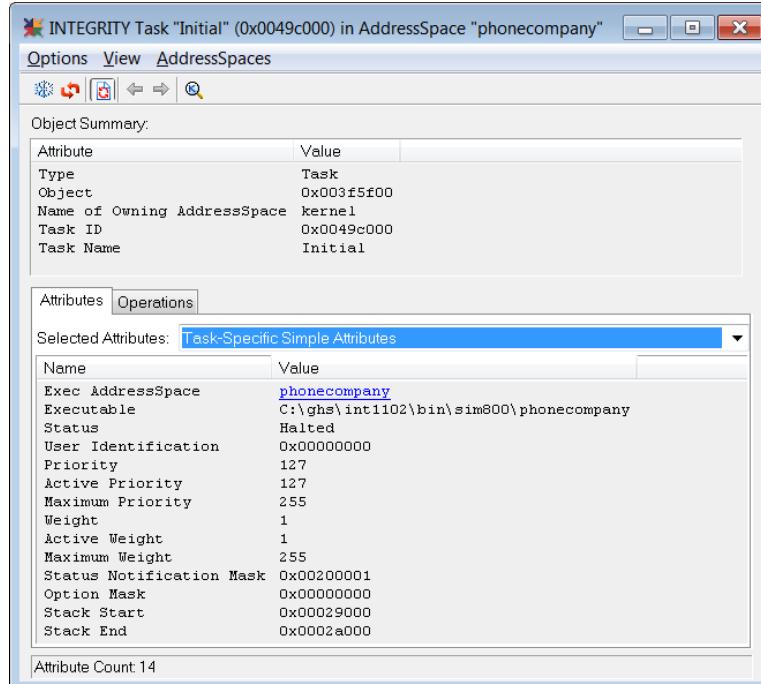
With the OSA Object Viewer, you can view INTEGRITY object attributes, inject messages into the kernel, and navigate between Objects. The OSA Object Viewer also keeps a history of all the Objects you have viewed which you can easily navigate. Because the OSA Object Viewer is typically used to display small chunks of information from the kernel, it is faster than the OSA Explorer.

To use the OSA Object Viewer, do one of the following:

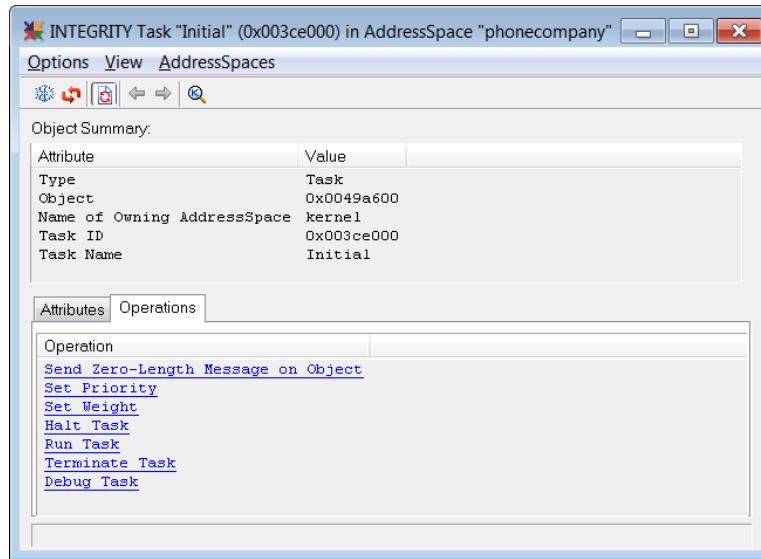
- Double-click an INTEGRITY Object variable in the Debugger source pane while the Task is halted.
- Select a Task, AddressSpace, or target in the Debugger Target list and click the **OSA Object Viewer** button ().
- Enter the **osaview** command in the Debugger command pane.

The OSA Object Viewer displays with an **Object summary** on top. The lower portion of the Object Viewer contains two tabs:

- **Attributes** tab — contains a list of attributes. You can click an underlined Object to view detailed information about the Object.



- **Operations tab** — contains a list of operations you can perform. You can click an underlined Operation to perform the operation.

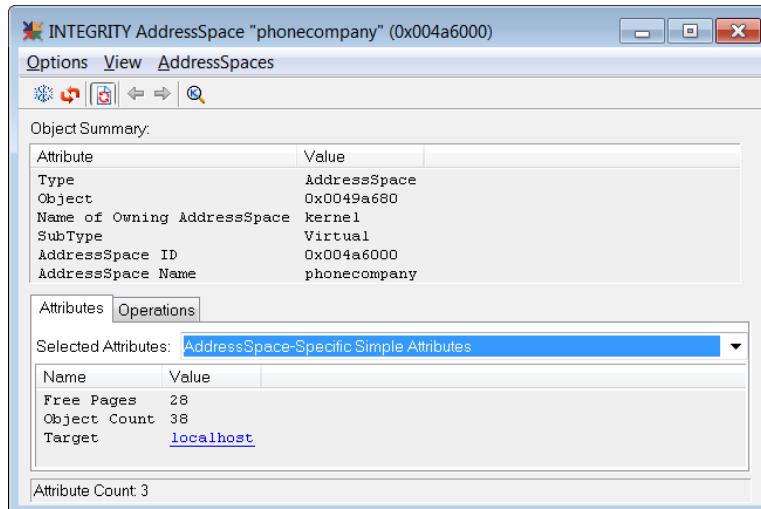


When you view an Object in the OSA Object Viewer, it may be displayed in either an existing window or in a new window, depending on whether the OSA Object Viewer is reusable. By default, the OSA Object Viewer window is reused, however, if you freeze the window, it will not be reused.

The contents of the Object summary, the attributes shown, and the list of operations available varies by Object type. The following sections provide more detailed information.

14.3.1 Common Features of Object View Windows

This section presents features of Object view windows that are common to all types of Objects. The following is an initial view of an AddressSpace Object:



The title bar indicates that the window is displaying an INTEGRITY Object. It also indicates the type of the Object (in this case, an AddressSpace).

Use the **AddressSpaces** menu to navigate to any AddressSpace Object in the system.

The Object window contains two panes. The upper pane (the **Object Summary** pane) displays basic information about the Object and is constant for any given Object at any given time. The lower pane (the Attributes/Operations pane) presents either additional attributes of the Object or the operations available to be performed on the Object, depending on whether the **Attributes** or the **Operations** tab is selected.

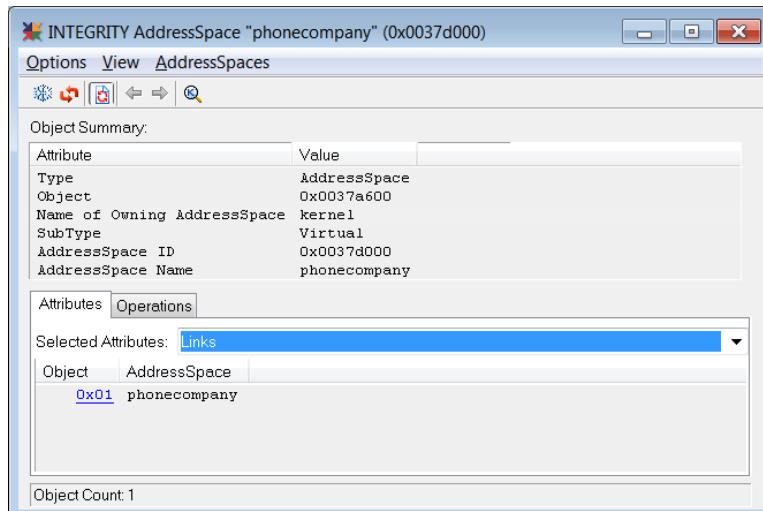
The upper pane always provides the following two pieces of information about the Object:

- The type of the Object (in this case, AddressSpace).
- The Object value (in this case 0x0049a680) and the name of the AddressSpace that owns the Object (in this case kernel).

Using the above example, code in AddressSpace kernel could refer to the Object using the C expression (AddressSpace)0x0049a680.

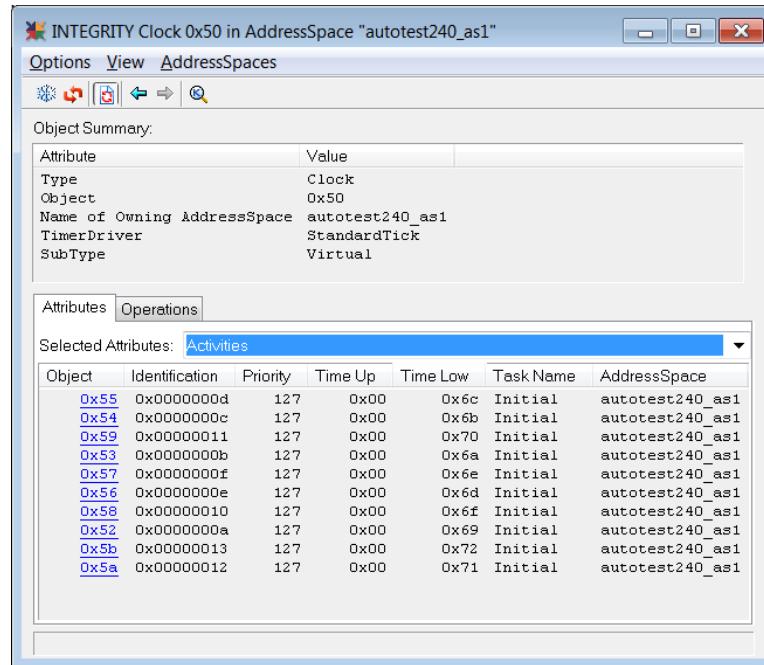
The **Attributes** tab in the lower pane contains a **Selected Attributes** drop-down menu, which controls what is displayed in the pane. You can select between simple attributes (with some basic information about the Object, which varies by Object type), a list of Links to the Object, a list of Activities waiting to send or receive messages on the Object (if any), or other Object type-specific attributes.

The following is an example of the **Links** drop-down menu for an AddressSpace. In this example, there is one Link to the AddressSpace. This Link is Object 0x01 in AddressSpace phonecompany (in other words, the AddressSpace's first Object is a Link to itself). You can follow a Link by clicking the underlined Object value.

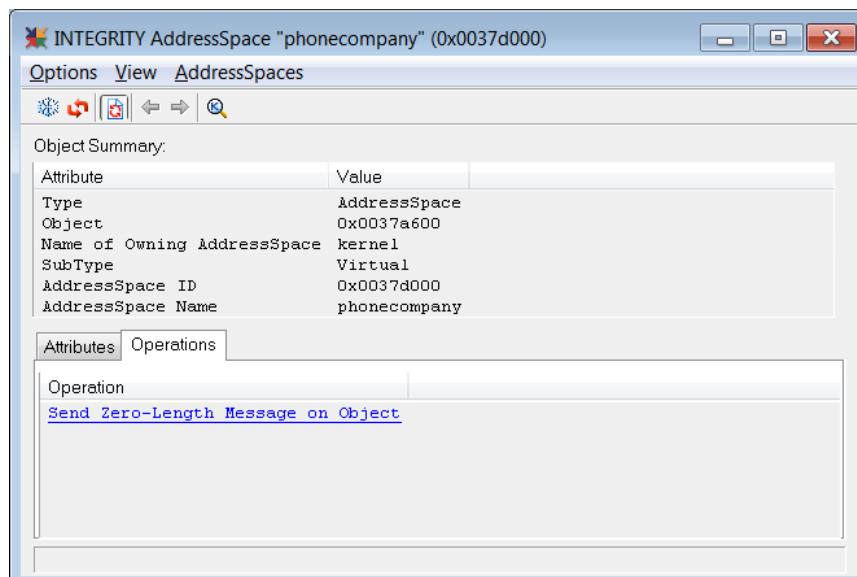


The following is an example of the **Activities** drop-down menu for a Clock. In this example, there are 10 Activities blocked waiting for zero-length messages to be sent on the Clock. Each row identifies an Activity, the Task of which the Activity is a part, the Activity's interrupt priority

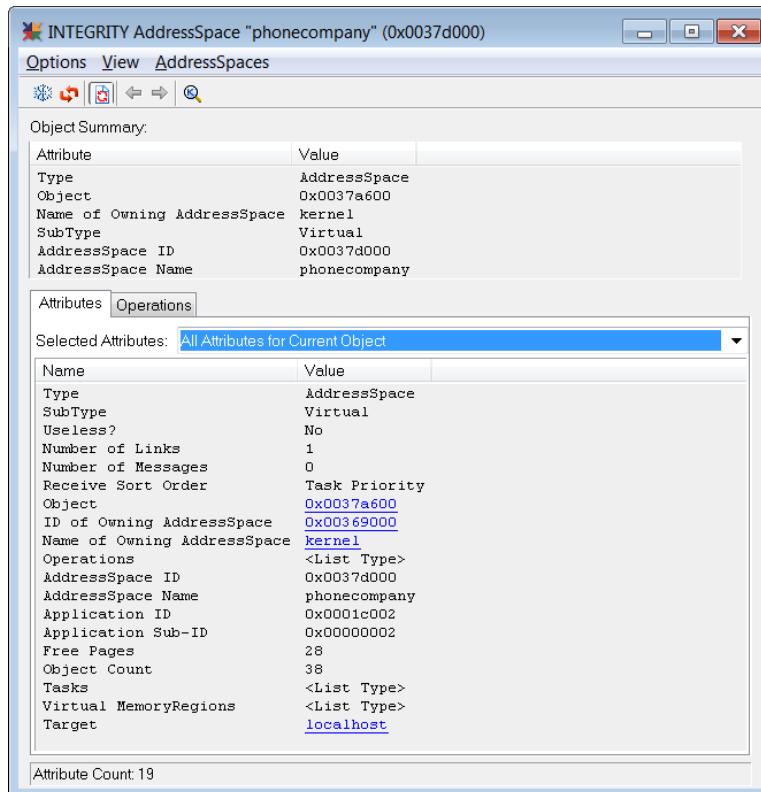
level, and a 64-bit integer (composed of the **Time Up** and **Time Low** fields) indicating the relative time at which the Activity was last inserted into the queue. When a zero-length message is sent on the Object, the Activity that will be notified first is the Activity of highest priority that has been waiting the longest (i.e., that has the smallest 64-bit insertion time).



The **Operations** tab in the lower pane allows some basic manipulation of the Object. Many Object types support a **Send Zero-Length Message on Object** operation that will either notify one Activity that is blocked on the Object or increment the overrun count on the Object. This is the only operation supported for some Object types, including AddressSpaces. In the following screenshot, the message is sent by clicking on the underlined text. If it is successful, the status bar at the bottom of the window will display the message **Operation performed successfully.**:



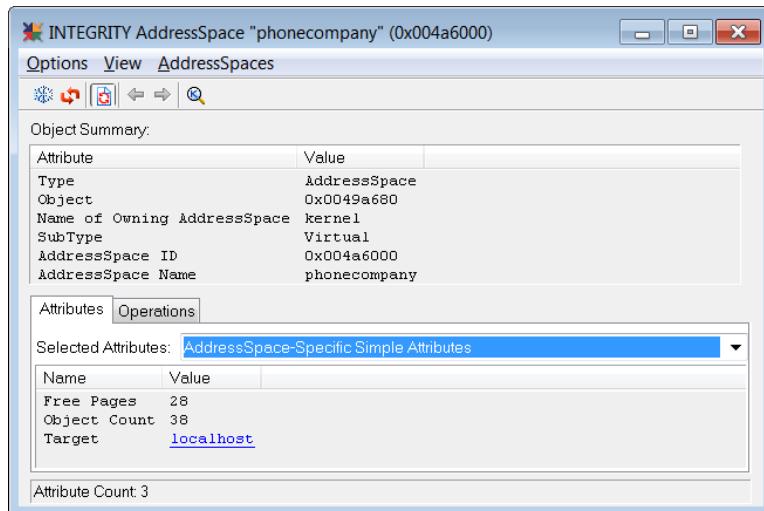
For more detailed attributes of an Object, select the **Attributes** tab and select the **All Attributes for Current Object** in the drop-down menu. The following additional information is common to many or all Object types:



- **Useless?** — The value of this field is normally **No**, indicating that the Object is still valid. If the Object is being closed or another Object upon which it critically depends has been closed, this field will be **Yes** (and any kernel call on the Object will fail with error `ObjectIsUseless`).
- **Number of Links** — The value of this field is the number of Links to the Object. This is the number of Link Objects displayed in the lower pane if the Links drop-down menu item is selected.
- **Number of Messages** — The value of this field is the number of overruns (zero-length messages) queued up on this Object. This applies to most Objects, but not Connections, Activities, or Links.

14.3.2 Viewing AddressSpace Objects

The following shows an AddressSpace Object in the OSA Object Viewer:



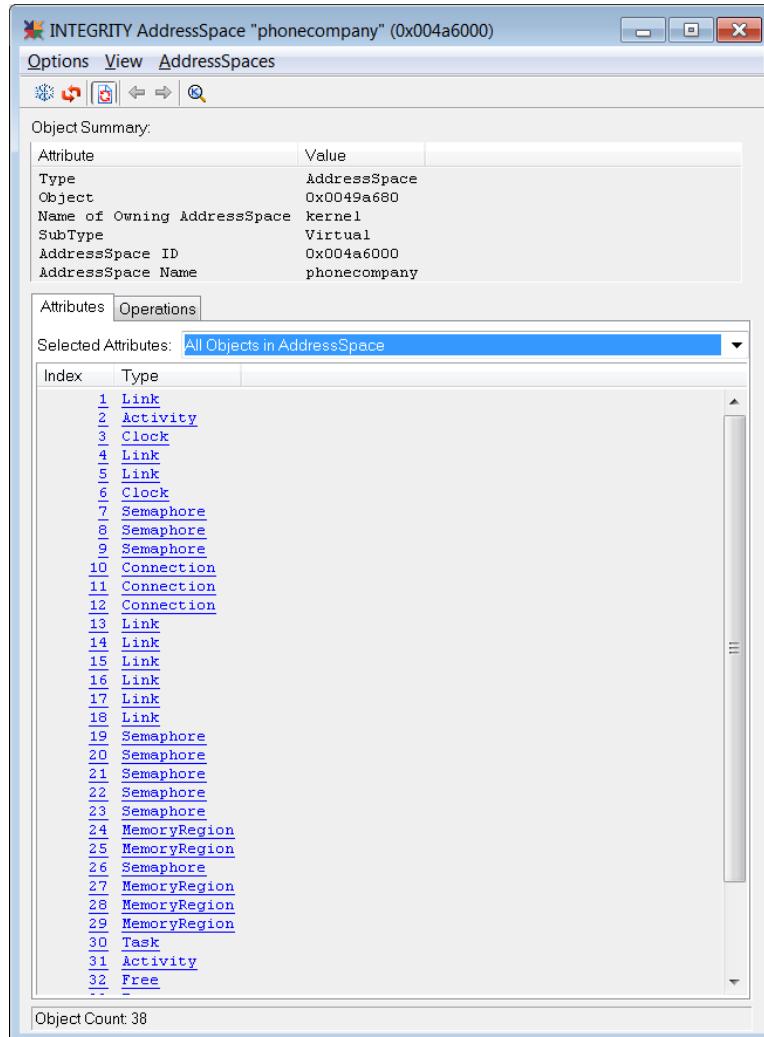
The **Object Summary** pane includes the following information specific to AddressSpace Objects:

- **SubType** — **Virtual** for a virtual AddressSpace, or **Kernel** for KernelSpace.
- **AddressSpace ID** — The unique ID of the AddressSpace (as returned by the GetAddressSpaceUniqueId() kernel call).
- **AddressSpace Name** — The name of the selected AddressSpace (if the AddressSpace has a name).

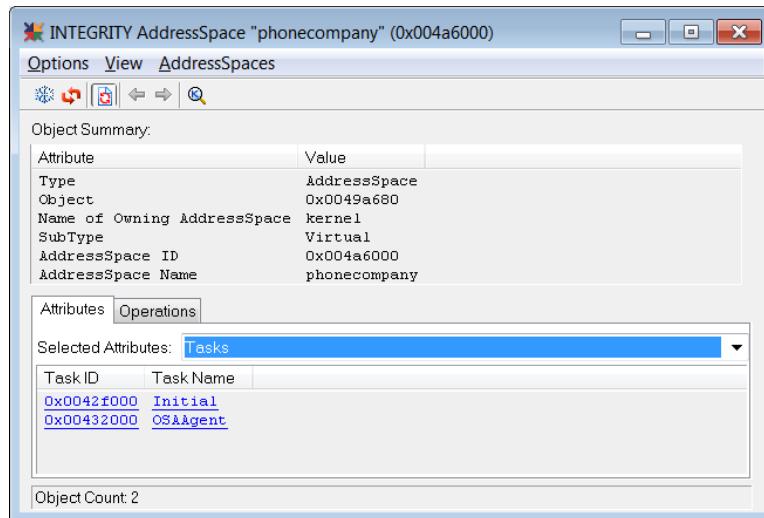
Select **AddressSpace-Specific Simple Attributes** on the **Attributes** tab for the following information:

- **Free Pages** — The number of free pages on the AddressSpace's page free list, as returned by the GetAddressSpaceFreeListCount() kernel call.
- **Object Count** — The size of the AddressSpace's Object Table (including both allocated Objects and free Objects), as returned by the GetObjectCount() kernel call.
- **Target** — The name of the target on which the AddressSpace exists. You can navigate to a list of AddressSpaces on the target by clicking the target name.

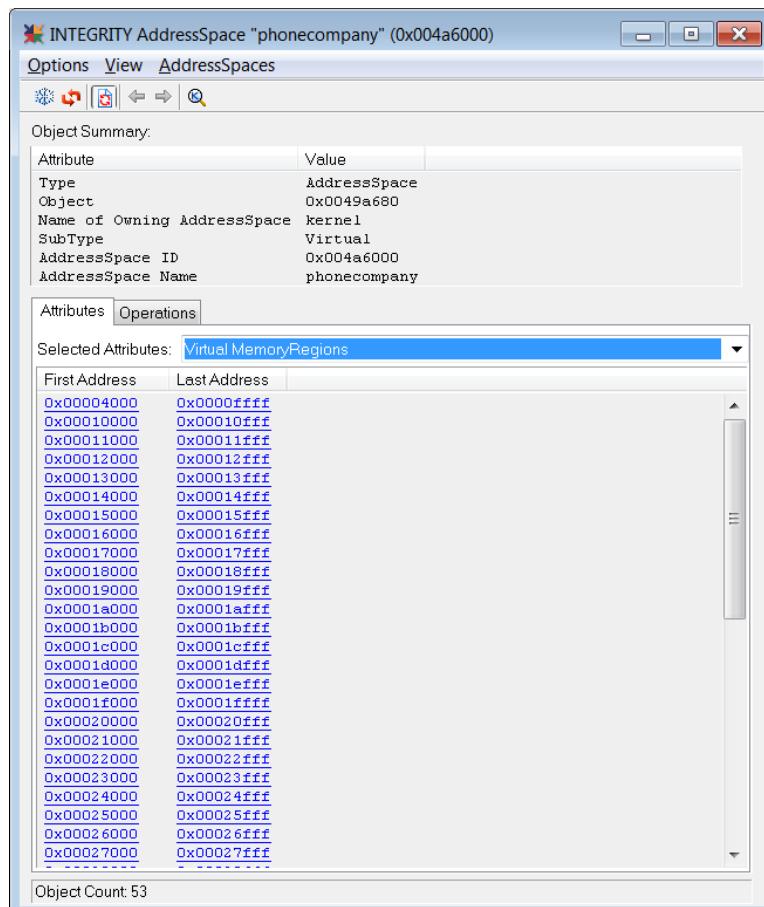
Select **All Objects in AddressSpace** from the drop-down to display all the Objects in the AddressSpace, including each Object's ObjectIndex and type. Click an entry to navigate to the corresponding Object. The status bar displays the AddressSpace's Object Count.



Select the **Tasks** drop-down to display all the Tasks that execute in the AddressSpace, including each Task's unique ID and name (if it has a name). Click an entry to navigate to the corresponding Task. Note that a Task that executes in one AddressSpace may be owned by another AddressSpace.

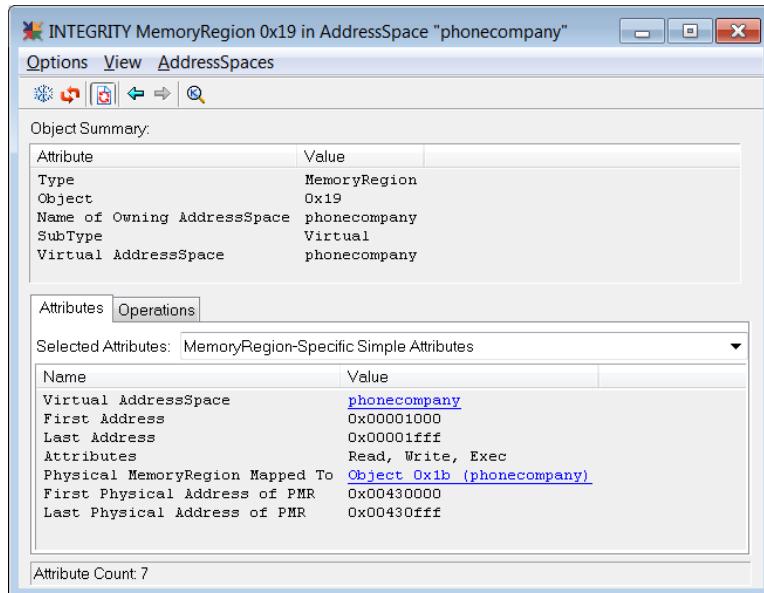


For a virtual AddressSpace, select **Virtual MemoryRegions** in the drop-down to display all the virtual MemoryRegions corresponding to virtual memory in the AddressSpace, including each virtual MemoryRegion's **First** and **Last** values (as returned by the GetMemoryRegionExtendedAddresses() kernel calls). Click an entry to navigate to the corresponding virtual MemoryRegion. Note that a virtual MemoryRegion corresponding to virtual memory in one AddressSpace may be owned by another AddressSpace.



14.3.3 Viewing MemoryRegion Objects

The following shows a MemoryRegion Object in the OSA Object Viewer:



The title bar indicates that the window is displaying a MemoryRegion Object. It also displays the name of the AddressSpace that owns the MemoryRegion and the Object value for the MemoryRegion.

The **Object Summary** pane includes the following information specific to MemoryRegion Objects:

- **SubType** — **Virtual** for a virtual MemoryRegion, **Physical** for a physical MemoryRegion, or **Lending Physical** for a physical MemoryRegion that is in use by a borrower.
- **Virtual AddressSpace** — The name of the virtual AddressSpace containing the virtual memory represented by the virtual MemoryRegion (for virtual MemoryRegions only).

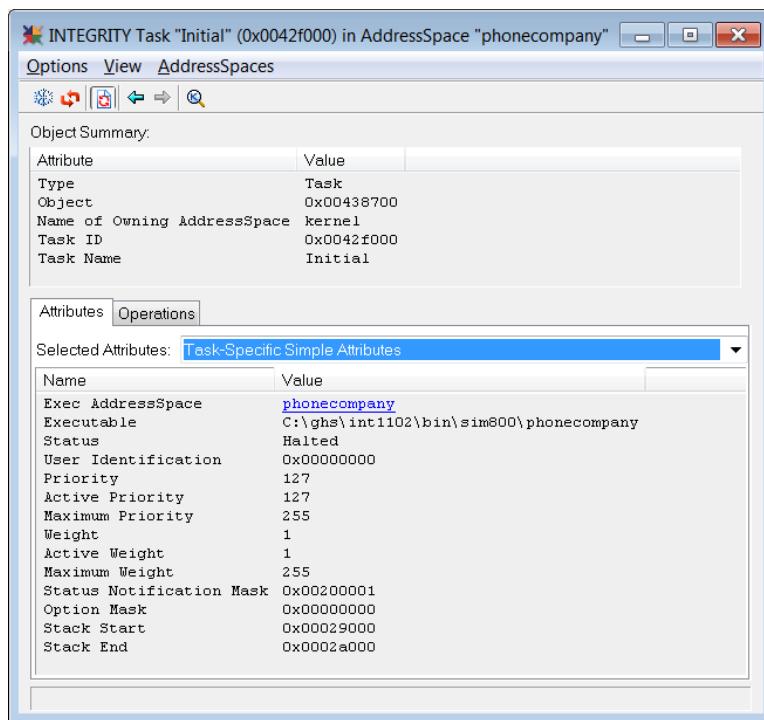
Select **MemoryRegion-Specific Simple Attributes** on the **Attributes** tab for the following information:

- **Virtual AddressSpace** — The virtual AddressSpace containing the virtual memory represented by the virtual MemoryRegion (applicable to virtual MemoryRegions only).
- **First Address, Last Address** — The address range represented by the MemoryRegion, as returned by the GetMemoryRegionExtendedAddresses() kernel call.
- **Attributes** — The attributes of the MemoryRegion, as returned by the GetMemoryRegionAttributes() kernel call, represented as a comma-separated list of attributes. Each attribute corresponds to one of the MEMORY attribute constants (**Read** corresponds to MEMORY_READ, etc.).

- **Physical MemoryRegion Mapped To** — If the MemoryRegion is a mapped virtual MemoryRegion, this field contains the physical MemoryRegion to which the virtual MemoryRegion is mapped.
- **First/Last Physical Address of PMR** - If the MemoryRegion is a mapped virtual MemoryRegion, this field contains the address range in KernelSpace represented by the corresponding physical MemoryRegion.

14.3.4 Viewing Task Objects

The following shows a Task Object in the OSA Object Viewer:



The title bar of a Task Object window includes the name of the Task, the unique ID of the Task (as can be obtained through the GetTaskUniqueId() kernel call), and the name of the AddressSpace in which the Task executes.

Note: Sometimes a Task Object is owned by one AddressSpace and executes in another AddressSpace. For example, Tasks defined in virtual AddressSpaces via Integrate are typically owned by KernelSpace, while the virtual AddressSpaces are given Links to the Tasks.

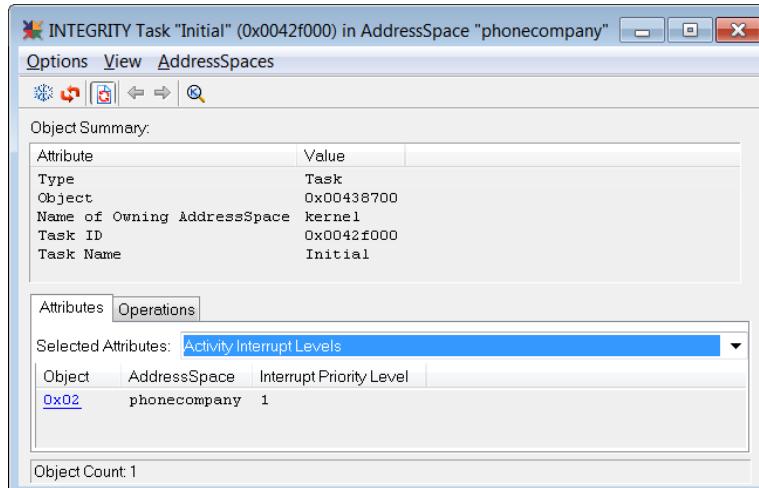
The **Object Summary** pane includes the following information specific to Task Objects:

- **Task ID** — The unique ID of the Task, as returned by the GetTaskUniqueId() kernel call.
- **Task Name** — The Task's name, if it has one.

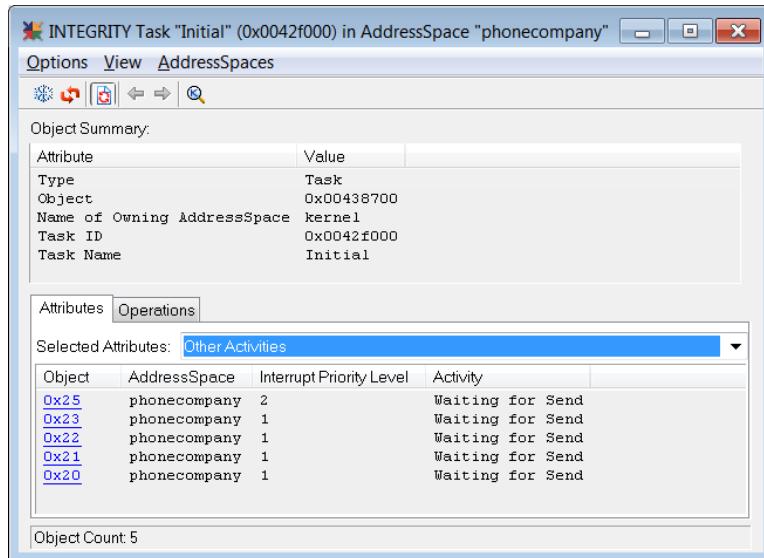
Select **Task-Specific Simple Attributes** on the **Attributes** tab for the following information:

- **Exec AddressSpace** — The name of the AddressSpace in which the Task executes. The name can be clicked to navigate to the AddressSpace.
- **Executable** — The path of the file containing the executable image used by the Task (as determined when the executable image was built). This path is as returned by the GetTaskExecutableFile() kernel call.
- **Status** — The status of the Task, which is one of: **Halted**, **Running**, **Pended**, **SUSPENDED**, or **Zombied**.
- **User Identification** — The user-specified ID of the Task, as returned by the GetTaskIdentification() kernel call.
- **Priority, Weight** — The priority and weight of the Task, as returned by the GetPriorityAndWeight() kernel call.
- **Active Priority** — The active priority of the Task, as returned by the GetActivePriority() kernel call.
- **Active Weight** — The active weight of the Task, which is either the Task's weight (if the Task's active priority is less than its maximum priority) or the maximum of the Task's weight and its maximum weight (if the Task's active priority is equal to its maximum priority).
- **Maximum Priority, Maximum Weight** — The maximum priority and weight of the Task, as returned by the GetMaximumPriorityAndWeight() kernel call.
- **Status Notification Mask** — The status notification mask of the Task, as returned by the GetTaskStatusNotificationMask() kernel call.
- **Stack Start, Stack End** - The Task's stack limits, as returned by the GetTaskStackLimits() kernel call.

Select **Activity Interrupt Levels** on the **Attributes** tab to see all the Activities in the Task that represent active interrupt levels (including the Task's Primary Activity). Each row displays an Activity Object and its interrupt priority level, as returned by the GetActivityInterruptPriorityLevel() kernel call:



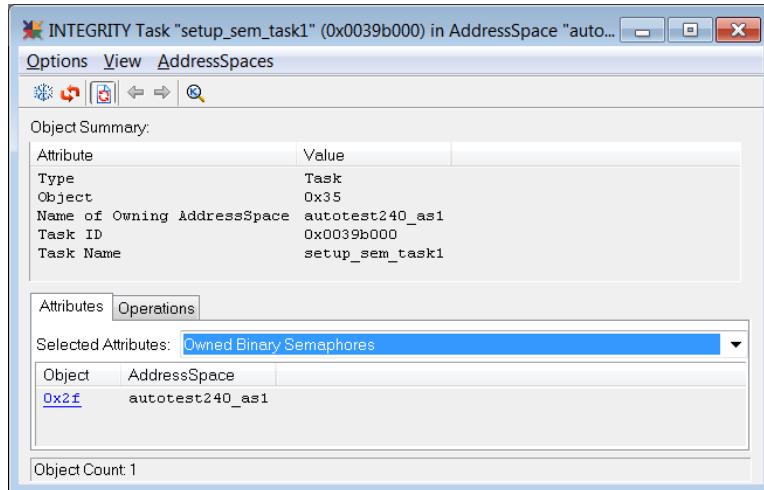
Select **Other Activities** on the **Attributes** tab to see all the other Activities in the Task (the ones that do not represent active interrupt levels).



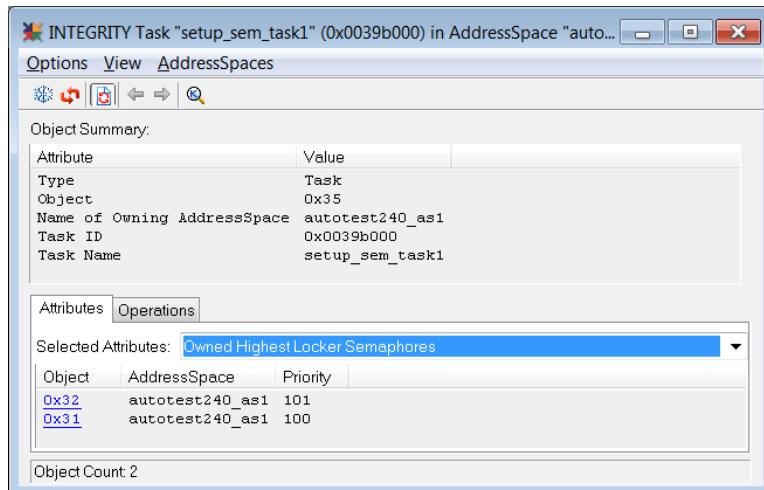
Each row displays an Activity Object, its interrupt priority level, and what the Activity is doing:, which is one of the following:

- **Idle** — The Activity is not engaged in a transfer.
- **Waiting for Send** — The Activity is receiving on an Object waiting for a sender or a zero-length message.
- **Waiting for Receive** — The Activity is sending on a Connection waiting for a receiver.
- **Send** — The Activity is engaged in an asynchronous message transfer, and is the sender.
- **Receive** — The Activity is engaged in an asynchronous message transfer, and is the receiver.
- **Request Task Interrupt** — The Activity is an Interrupt Activity that has been notified but is being masked (either because the Task is not running or because the Activity's interrupt priority level is not greater than the Task's current interrupt priority level).
- **WaitList** — The Activity is a WaitList Activity that has been notified and is on its Task's WaitList.
- **Recoverable Error** — The Activity is a WaitList Activity that has been notified with a recoverable error and removed from the WaitList by the Task, but the transfer has not yet been resumed with ContinueActivity().
- **Terminated Transfer** — The Activity is a WaitList Activity that represents a completed message transfer.

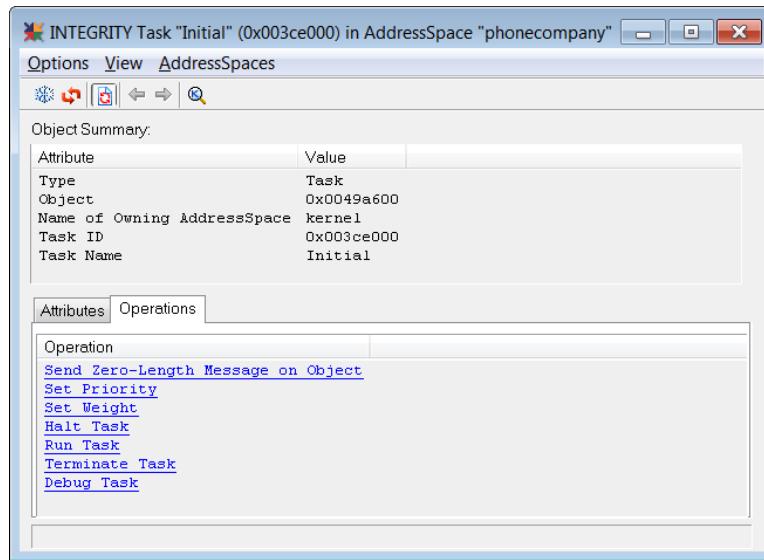
Select **Owned Binary Semaphores** on the **Attributes** tab to see all the Binary Semaphores owned by the Task:



Select **Owned Highest Locker Semaphores** on the **Attributes** tab to see all the Highest Locker Semaphores owned by the Task, including the priority of each Highest Locker Semaphore:



Select the **Operations** tab for the following Task-specific operations:



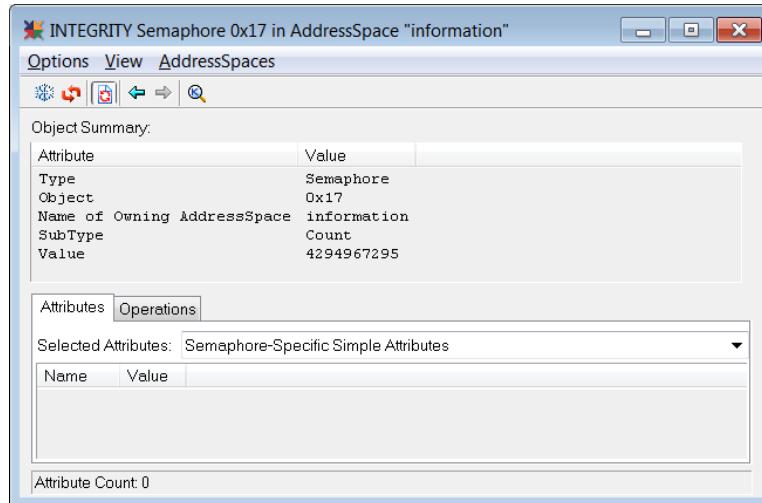
- **Send Zero-Length Message on Object** notifies one Activity that is blocked on the Object, or increments the overrun count on the Object.
- **Set Priority** and **Set Weight** are used to change the priority and weight of the Task, through a simulated SetPriorityAndWeight() call. Each operation pops up a window that prompts you for the new priority or weight value.
- **Halt Task** and **Run Task** perform basic run control on the Task.
- **Terminate Task** terminates the Task, preventing it from running further.
- **Debug Task** causes the Debugger to select that Task from the Task list.

14.3.5 Viewing Semaphore Objects

There are three kinds of Semaphores: Counting Semaphores, Binary Semaphores, and Highest Locker Semaphores. The Object Viewer windows for each are described in the following subsections.

14.3.5.1 Viewing Counting Semaphore Objects

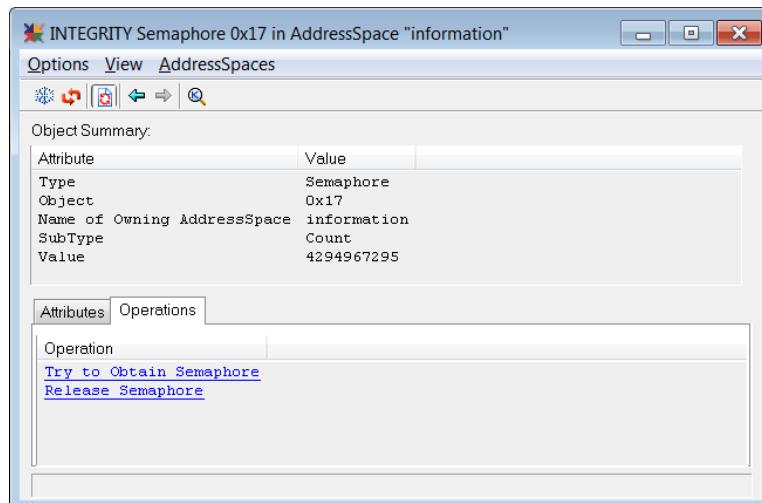
The following shows a Counting Semaphore Object in the OSA Object Viewer:



The **Object Summary** pane includes the following information specific to Semaphore Objects:

- **SubType — Count** identifies this as a Counting Semaphore.
- **Value** — The value of the Semaphore (as returned by the GetSemaphoreValue() kernel call). If the value is negative (as it is here), select **Activities** on the **Attributes** tab to see the Activities that are waiting for the Semaphore.

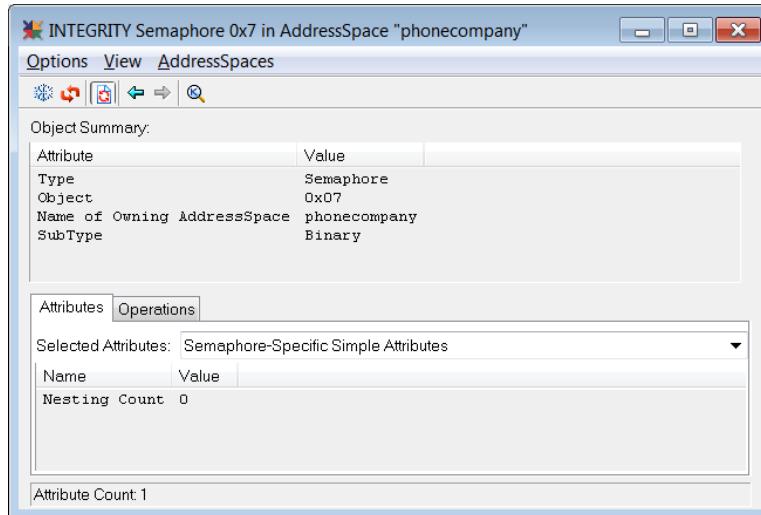
Select the **Operations** tab for the following Counting Semaphore operations:



- **Try to Obtain Semaphore** simulates a TryToObtainSemaphore() kernel call on the Semaphore. It will succeed only if the Semaphore's value is greater than zero.
- **Release Semaphore** simulates a ReleaseSemaphore() kernel call on the Semaphore. It will succeed provided that the Semaphore's value is not the maximum integer of type SignedValue.

14.3.5.2 Viewing Binary Semaphore Objects

The following shows a Binary Semaphore Object in the OSA Object Viewer:



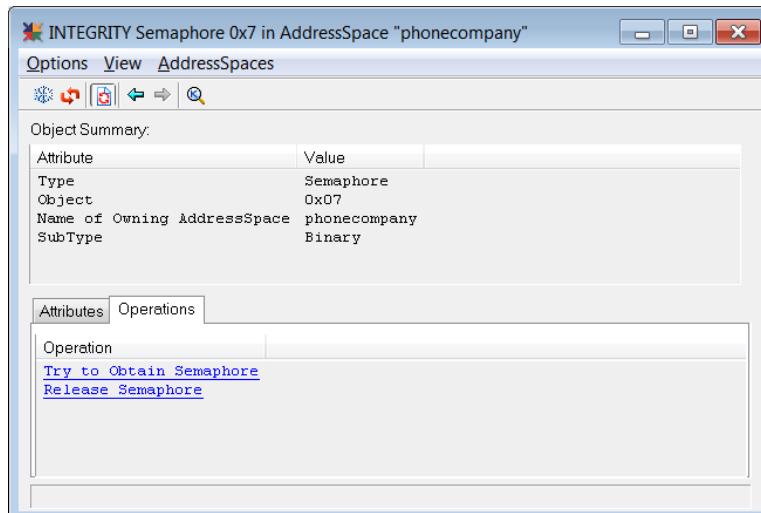
The **Object Summary** pane includes the following information specific to Semaphore Objects:

- **SubType — Binary** identifies this as a Binary Semaphore.
- **Owner** — The Task that owns the Binary Semaphore, if applicable.

Select **Semaphore-Specific Simple Attributes** on the **Attributes** tab to see the following:

- **Nesting Count** — If the Semaphore is owned, its nesting count is the number of times that its owner has acquired the Semaphore (which is greater than zero). If the Semaphore is unowned, its nesting count is zero.
- **Owner** — If the Semaphore is owned, this field indicates the Task that owns the Semaphore.

Select the **Operations** tab for the following Binary Semaphore operations:

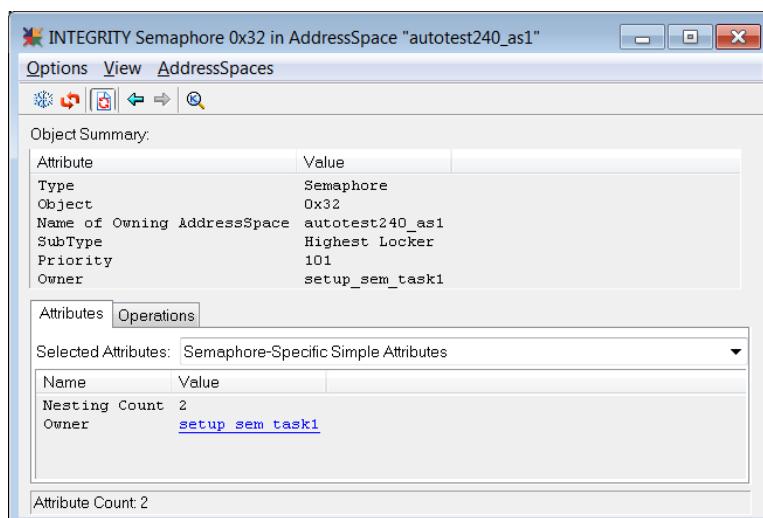


- **Try to Obtain Semaphore** simulates a TryToObtainSemaphore() kernel call on the Semaphore. After clicking on this operation, a window will prompt you to select a Task to attempt to acquire the Semaphore. The operation will succeed only if the Semaphore is unowned at the moment the operation is executed.
- **Release Semaphore** simulates a ReleaseSemaphore() kernel call on the Semaphore. It will succeed provided that the Semaphore is owned.

Notes: Releasing a Binary Semaphore while the Task that owns it is in the middle of a critical section may cause incorrect behavior. Releasing a Binary Semaphore associated with a LocalMutex may cause incorrect behavior.

14.3.5.3 Viewing Highest Locker Semaphore Objects

The following shows a Highest Locker Semaphore Object in the OSA Object Viewer:



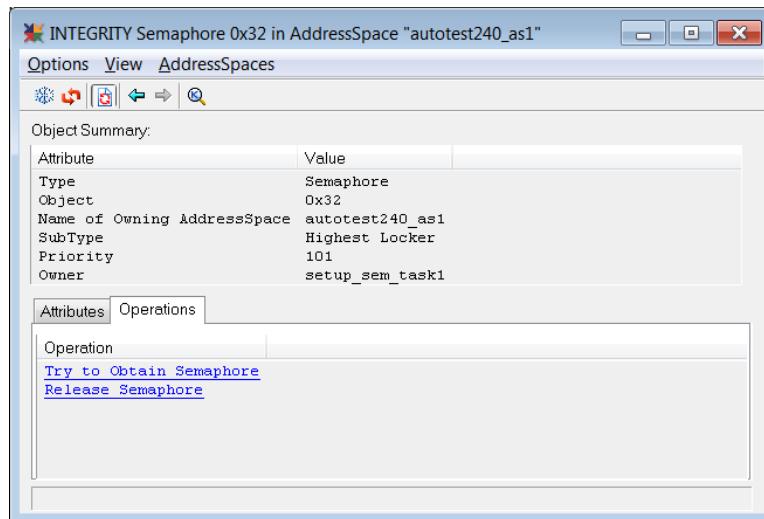
The **Object Summary** pane includes the following information specific to Semaphore Objects:

- **SubType — Highest Locker** identifies the Semaphore as a Highest Locker Semaphore.
- **Priority** — The priority of the Highest Locker Semaphore.
- **Owner** — The Task that owns the Highest Locker Semaphore, if applicable.

Select **Semaphore-Specific Simple Attributes** on the **Attributes** tab to see the following:

- **Nesting Count** — If the Semaphore is owned, its nesting count is the number of times that its owner has acquired the Semaphore (which is greater than zero). If the Semaphore is unowned, its nesting count is zero.
- **Owner** — If the Semaphore is owned, this field indicates the Task that owns the Semaphore.

Select the **Operations** tab for the following Highest Locker Semaphore operations:

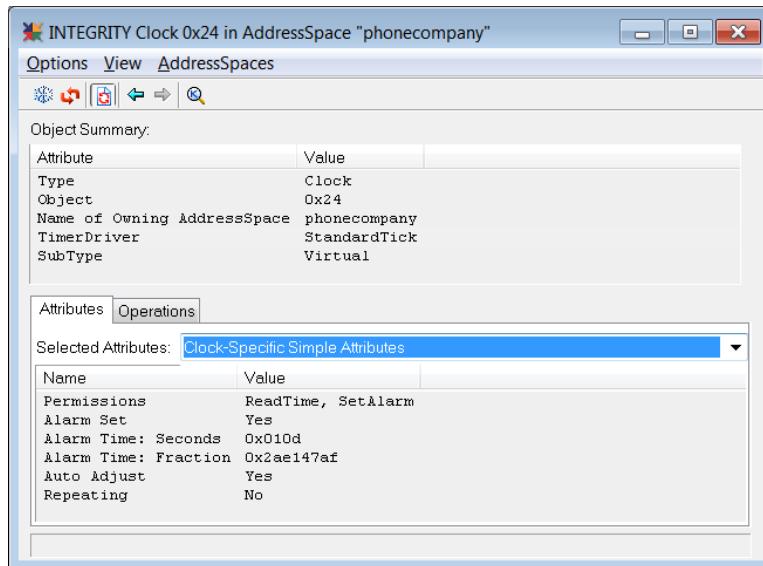


- **Try to Obtain Semaphore** simulates a TryToObtainSemaphore() kernel call on the Semaphore. After clicking on this operation, a pop-up window will prompt you to select a Task to attempt to acquire the Semaphore. The operation will succeed only if the Semaphore is unowned at the moment the operation is executed and the Semaphore's priority is not less than the priority of the selected Task. See the documentation for the TryToObtainSemaphore() kernel call for more information.
- **Release Semaphore** simulates a ReleaseSemaphore() kernel call on the Semaphore. It will succeed provided that the Semaphore is owned.

Note: Releasing a Highest Locker Semaphore while the Task that owns it is in the middle of a critical section may cause incorrect behavior.

14.3.6 Viewing Clock Objects

The following shows a Clock Object in the OSA Object Viewer:



The **Object Summary** pane includes the following information specific to Clock Objects:

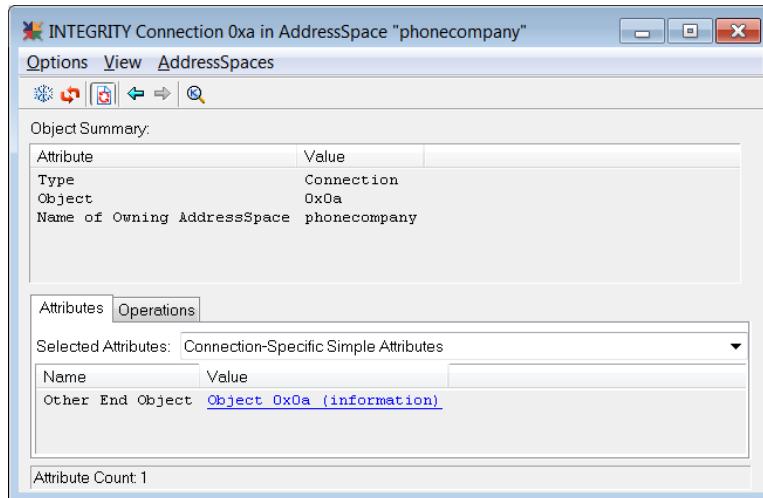
- **TimerDriver** — The name of the TimerDriver that implements the Clock (normally **StandardTick** for PrimaryClock, **HighResTimer** for HighResClock, and **RealTimeClock** for RealTimeClock).
- **Subtype** — **Physical** identifies the Clock as a Physical Clock (a Clock created by the kernel during kernel initialization). **Virtual** identifies the Clock as a Virtual Clock (a Clock created by a CreateVirtualClock() kernel call).

Select **Clock-Specific Simple Attributes** on the **Attributes** tab for the following:

- **Permissions** — A comma-separated list of permissions possessed by this Clock (**ReadTime**, **SetTime**, and **SetAlarm**, corresponding to the CLOCK_READTIME, CLOCK_SETTIME, and CLOCK_ALARM permission bits).
- **AlarmSet** — **Yes** if an alarm is set; **No** otherwise.
- **Alarm Time: Seconds**, **Alarm Time: Fraction** — The Time of the next alarm on this Clock. This field is only present if an alarm is set.
- **Auto Adjust** — **Yes** if the Time of the next alarm is adjusted when AdjustClockTime() adjusts the Time on the TimerDriver that implements this Clock; **No** if the Time of next alarm is not adjusted. This field is only present if an alarm is set.
- **Repeating** — **Yes** if the alarm is repeating; **No** otherwise. This field is only present if an alarm is set.
- **Repeat Interval: Seconds**, **Repeat Interval: Fraction** — The repeat interval for a repeating alarm. This field is only present if a repeating alarm is set.

14.3.7 Viewing Connection Objects

The following shows a Connection Object in the OSA Object Viewer:



Select **Connection-Specific Simple Attributes** on the **Attributes** tab for the following:

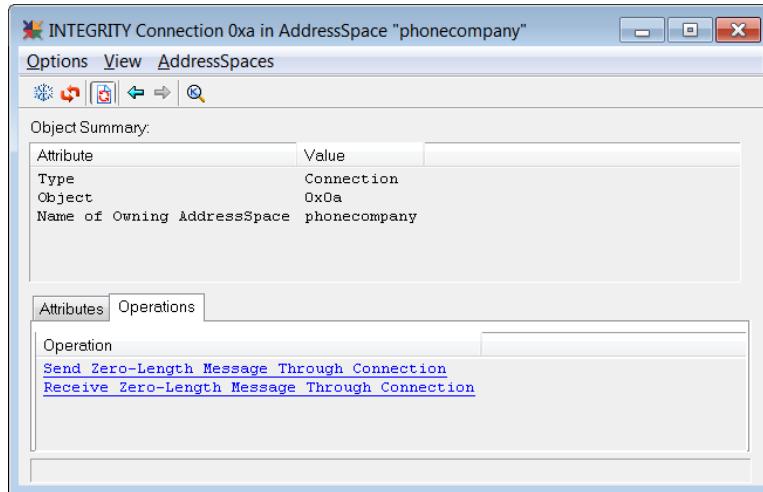
- **Other End Object** — The Connection Object at the other end of this Connection Object.

Use the **Selected Attributes** drop-down on the **Attributes** tab to view other information about the Connection:

- Select **All Attributes for Current Object** for additional information about the Connection.
- Select **Activities Queued Waiting for Send** to see the receive Activities blocked on the Connection waiting for a send on the other end.
- Select **Activities Queued Waiting for Receive** to see the send Activities blocked on the Connection waiting for a receive on the other end.

Note: Activities are displayed as they would be displayed while blocked on any other kind of Object. However, for a FIFO Connection, the interrupt priority levels of the blocked Activities do not affect the order that Activities will be dequeued.

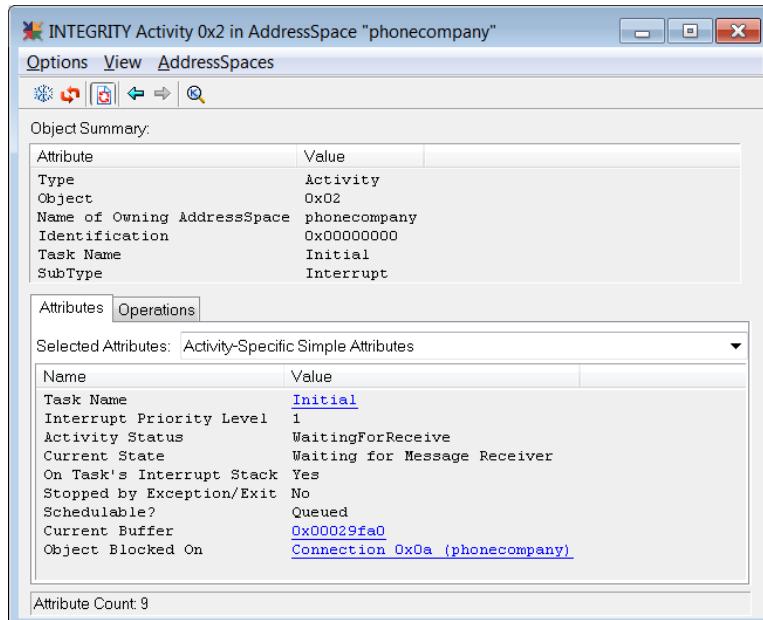
Select the **Operations** tab for the following Connection operations:



- **Send Zero-Length Message Through Connection** attempts to send a zero-length message through the Connection. It will succeed only if there is an Activity waiting to receive a message at the other end of the Connection. The Activity at the other need not be expecting a zero-length message.
- **Receive Zero-Length Message Through Connection** attempts to receive a zero-length message through the Connection. It will succeed only if there is an Activity waiting to send a zero-length message at the other end of the Connection.

14.3.8 Viewing Activity Objects

The following shows an Activity Object in the OSA Object Viewer:



The **Object Summary** pane includes the following information specific to Activity Objects:

- **Identification** — The Activity's identification (as established by the CreateActivity() kernel call).
- **Task Name** — The Task to which the Activity belongs.
- **SubType** — Primary, Interrupt, or WaitList.

Select **Activity-Specific Simple Attributes** on the **Attributes** tab for the following:

- **Task Name** — The name of the Task to which the Activity belongs.
- **Interrupt Priority Level** — The interrupt priority level of the Activity, as returned by GetActivityInterruptPriorityLevel().
- **Activity Status** — The status of the Activity, as returned by GetActivityStatus().
- **Current State** — Describes at a high level what the Activity is currently doing. See the list that follows for descriptions of the various states.
- **Direction** — Indicates whether the Activity is currently sending or receiving. This field only exists for Activities that are connected.
- **Other End of Transfer** — The Activity representing the other end of the transfer. This field only exists for Activities that are connected.
- **On Task's Interrupt Stack** — Indicates whether the Activity represents an interrupt level of its Task. This field only exists for Interrupt Activities.
- **Stopped by Exception/Exit** — Indicates whether the interrupt level represented by this Activity has been suspended by an exception.
- **Schedulable?** — Indicates whether the Activity corresponds to a schedulable interrupt level. See the list that follows for descriptions of the possible values.
- **Current Buffer** — The address of the next Buffer to transfer. This field only exists for Activities that are connected or waiting to transfer.
- **Object Blocked On** — The Object on which this Activity is blocked waiting for a send or receive. This field only exists for an Activity whose Activity Status is **WaitingForSend** or **WaitingForReceive**.

The **Current State** field describes at a high level what the Activity is currently doing and can have any of the following values:

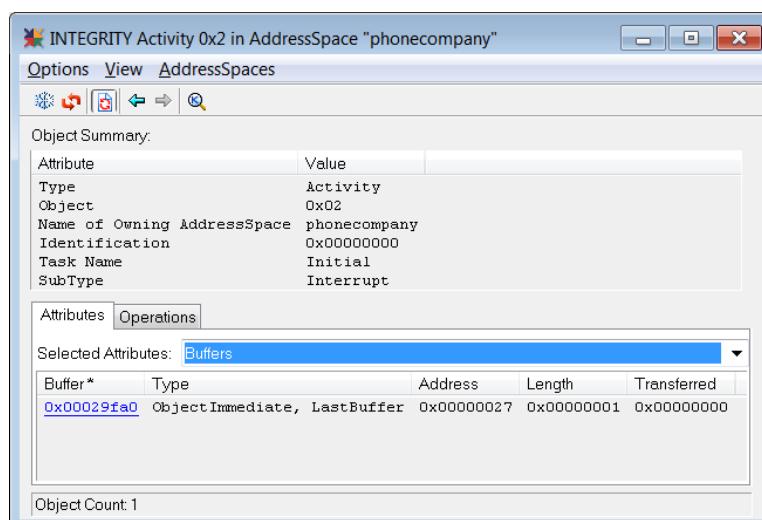
- **Idle** — The Activity is idle.
- **Waiting for Message to be Sent** — The Activity is engaged in a message receive and is blocked on an Object waiting for a message to be sent.
- **Waiting for Message Receiver** — The Activity is engaged in a message send and is blocked on an Object waiting for someone to receive the message.
- **Send** — The Activity is engaged in a message send with a peer.
- **Receive** — The Activity is engaged in a message receive with a peer.

- **Waiting for Activity** — The Activity is blocked in a WaitForActivity() call (or an analogous call).
- **Delivering Interrupt** — The Activity's interrupt is being delivered to the Task.
- **Dismissing Interrupt** — The Activity's interrupt level is being dismissed.
- **Executing Code** — The interrupt level is executing.
- **Requesting Task Interrupt** — The Activity is requesting a Task interrupt.
- **WaitList** — The Activity is on the Task's WaitList.
- **Recoverable Error** — The Activity is engaged in a message transfer that is suspended because of a recoverable transfer error.
- **Terminated Transfer** — The Activity's message transfer has terminated.

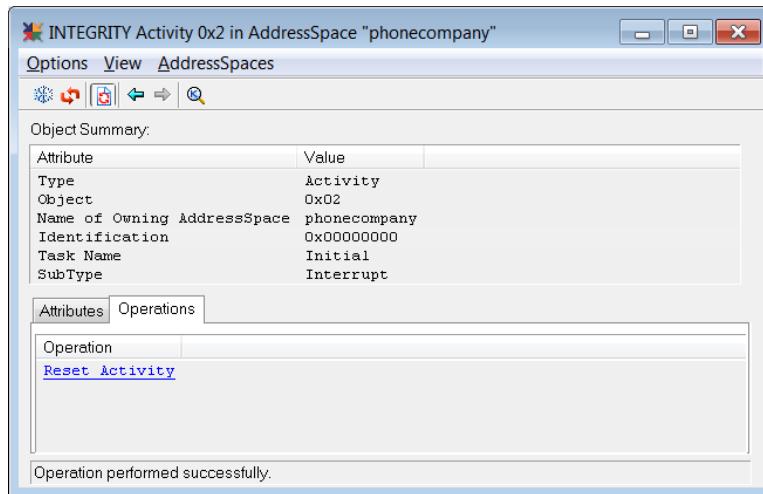
The **Schedulable?** field indicates whether the Activity corresponds to a schedulable interrupt level or transfer and can have any of the following values:

- **Runnable** — The Activity is schedulable.
- **Queued** — The Activity is queued on an Object, a WaitList, or a pending interrupt list.
- **Blocked Forever** — The Activity is blocked indefinitely.
- **Priority is Zero** — The Activity cannot run because its priority is zero.
- **Suspended** — The Activity is not schedulable.

When an Activity is waiting to transfer or is connected, you can select **Buffers** from the drop-down to see the current Buffer being transferred, and subsequent Buffers. For Buffers of type **ObjectImmediate**, **LinkImmediate**, or **ZeroCopyDataBuffer**, you can on underlined fields to navigate to the corresponding Objects:



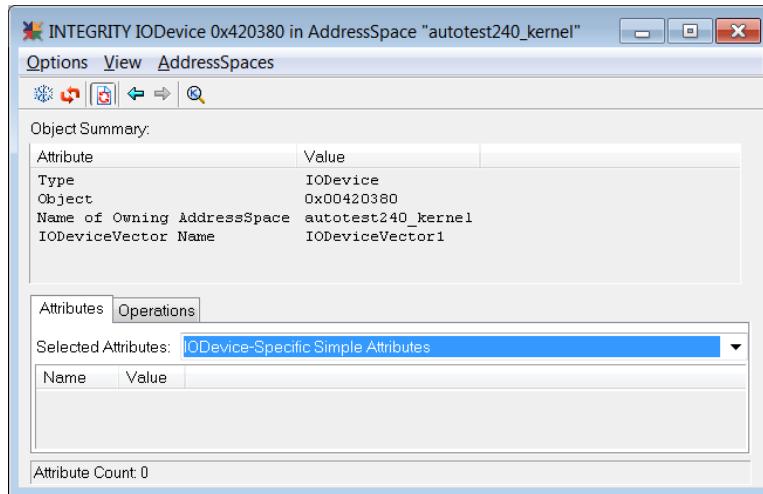
Select the **Operations** tab for the following Activity operation:



- **Reset Activity** simulates a ResetActivity() kernel call on the Activity.

14.3.9 Viewing IODevice Objects

The following shows an IODevice Object in the OSA Object Viewer:

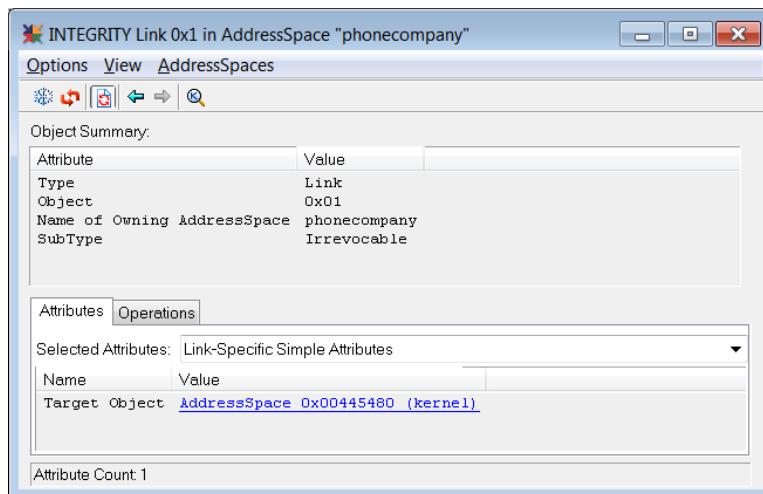


The **Object Summary** pane includes the following information specific to IODevices:

- **IODeviceVector Name** — The IODeviceVector that implements the IODevice.

14.3.10 Viewing Link Objects

The following shows a Link Object in the OSA Object Viewer:



Select **Link-Specific Simple Attributes** on the **Attributes** tab for the following:

- **Target Object** — The Object to which the Link points.

Chapter 15

Run-time Error Checking

This chapter provides information about run-time error checks, run-time memory checks and detecting stack overflow. It contains the following sections:

- Using Run-Time Error Checks
- Using Run-Time Memory Checks
- Detecting Stack Overflow
- Overhead of Run-Time Error Checking

Run-time error checking is the term used to describe a variety of debugging features aimed at detecting common run-time errors such as memory leaks and exceeding array bounds.

INTEGRITY supports a superset of the standard MULTI run-time error checking facilities. For additional information about MULTI’s general run-time error capabilities, see the “Run-Time Error Checks” section of the *MULTI: Building* book.

In addition to the standard set of errors described in the *MULTI: Building* book, INTEGRITY also supports per-task stack overflow checking.

INTEGRITY also supports run-time memory checks that are used to collect memory allocation information and debug memory allocation errors. For more information about run-time Memory Checking, see “Using the Memory Allocations Window” in the *MULTI: Debugging* book.

15.1 Using Run-Time Error Checks

Run-time error checking is enabled with a variety of compile options which are described in this section. For some types of errors, run-time error checking is achieved by compiler instrumentation of user code. For other types of errors, run-time error checking is achieved by special library code. In either case, the running code detects that an error condition has occurred and calls a built-in library routine that emits the error.

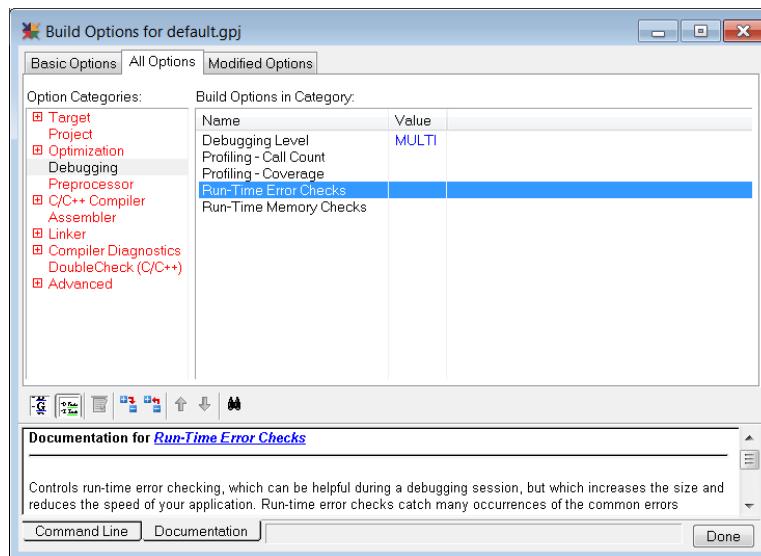
- When run-mode debugging is in progress, the error message is displayed in the MULTI Debugger I/O pane.

- If run-mode debugging is not in progress when a run-time error is detected, the error message is displayed on the serial console (if one exists for the BSP in use).
- When an attached task encounters a run-time error, it halts at the point the error occurs.
- When an unattached task encounters an error, it continues (for a non-fatal error) or halts itself (for a fatal error). If 100 non-fatal errors are encountered in an AddressSpace, any additional errors in that AddressSpace will be fatal and the tasks encountering the errors will halt. No errors will cause a task to exit.

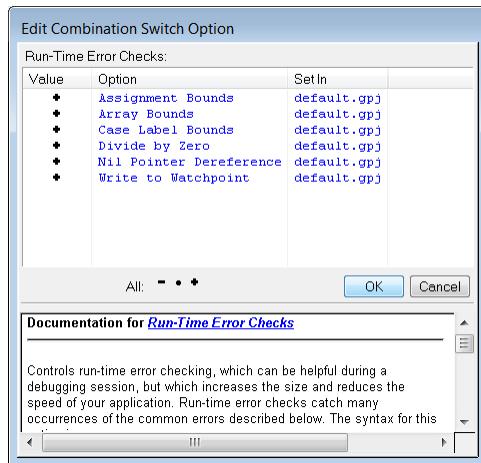
Although run-time error checking is meant to aid in fixing application-level code in protected virtual AddressSpaces, run-time error checking can also be enabled for KernelSpace tasks. However, for standard source or kernel source installations, it is recommended that system/kernel code (such as **libbsp.a** code) be compiled with all run-time error checking options disabled.

To enable MULTI run-time error checking options:

1. Right-click **myapp.gpj** and select **Set Build Options**.
2. On the **All Options** tab, select **Debugging**.
3. In the Build Options pane, double-click **Run-Time Error Checks**.



4. In the **Run-Time Error Checks** window, turn on the desired options. Only set the options for the project for which run-time error checking is desired.



The following common run-time errors can be detected:

- Assignment Bounds
- Array Bounds
- Case Label Bounds
- Divide by Zero
- Nil Pointer Dereference
- Write to Watchpoint

These run-time errors are covered in the sections that follow. For more information about detecting these errors, see the “Run-Time Error Checks” section of the *MULTI: Building* book.

15.1.1 Assignment Bounds

When you select **Assignment Bounds** in the Run-Time Error Checks window for a project, any attempt to assign a value too large for a given type will flag an error. For example:

```
short p;
foo(int n)
{
    p = n;          /* a run-time error would be flagged here */
}
bar(void)
{
    foo(100000);    /* passing value too large for "short" type */
}
```

The diagnostic message corresponding to this error is as follows:

Assignment out of bounds

15.1.2 Array Bounds

When you select **Array Bounds** on the Run-Time Error Checks window for a project, any attempt to index past the end of an array will flag an error. For example:

```
static char arr[100];
foo(int n)
{
    arr[n] = 0;           /* a run-time error would be flagged here */
}
bar(void)
{
    foo(100);           /* 100 is too large an index for arr */
}
```

The diagnostic message corresponding to this error is as follows:

```
Array index out of bounds
```

15.1.3 Case Label Bounds

When you select **Case Label Bounds** in the Run-Time Error Checks window for a project, any attempt to pass a value to a switch statement that does not match any of the switch statement's case labels will flag an error. For example:

```
foo(int n)
{
    /* a run-time error would be flagged here */
    switch (n) {
        case 0:
            printf("Should not have gotten here!\n");
            break;
        case 1:
            printf("Should not have gotten here!\n");
            break;
        case 2:
            printf("Should not have gotten here!\n");
            break;
    }
}
bar(void)
{
    /* pass a value not matched by any switch case */
    foo(3);
}
```

The diagnostic message corresponding to this error is as follows:

```
Case/switch index out of bounds
```

15.1.4 Divide by Zero

When you select **Divide by Zero** in the Run-Time Error Checks window for a project, any attempt to divide by zero will flag an error. For example:

```
foo(int n)
{
    /* a run-time error would be flagged here */
    return 100/n;
}
bar(void)
{
    foo(0);           /* passing a 0 divisor */
}
```

The diagnostic message corresponding to this error is as follows:

Divide by 0

15.1.5 Nil Pointer Dereference

When you select **Nil Pointer Dereference** in the Run-Time Error Checks window for a project, any attempt to dereference address 0 will flag an error. For example:

```
foo(int *p)
{
    *p = 0;           /* a run-time error would be flagged here */
}
bar(void)
{
    foo(0);           /* passing a NULL pointer */
}
```

The diagnostic message corresponding to this error is as follows:

Nil pointer dereference

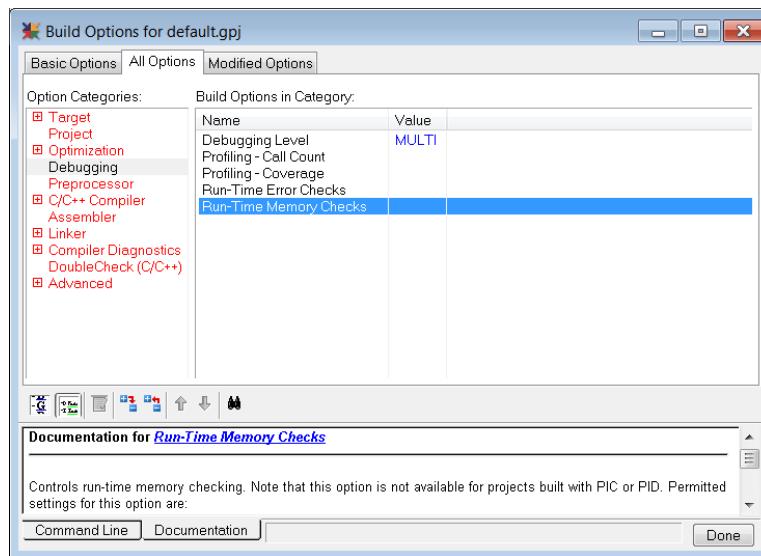
15.1.6 Write to Watchpoint

When you select **Write to Watchpoint** in the Run-Time Error Checks window for a project, it allows the MULTI Debugger to set up one fast watchpoint.

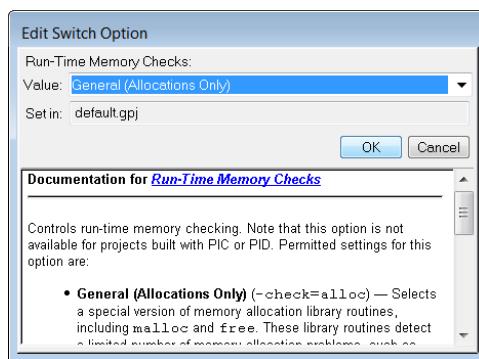
15.2 Using Run-time Memory Checks

Run-time memory checks are used to collect memory allocation information and debug memory allocation errors. To enable MULTI run-time memory checking options:

1. Right-click **myapp.gpj** and select **Set Build Options**.
2. On the **All Options** tab, select **Debugging**.
3. In the Build Options pane, double-click **Run-Time Memory Checks**.



4. In the **Value** field, select the type of Memory Allocation run-time check:



- **General (Allocations Only)** — User code is not instrumented, but alternative malloc-related functions are used. Although these routines have significant memory overhead for tracking internal information, the detection of these errors when they occur can save programmers a lot of time.

- **Intensive** — Generates errors for a wider range of memory allocation problems. This setting requires a significant amount of additional code to be downloaded to the target, but also automatically enables Nil Pointer Dereference checking at little additional performance cost. Because of the way the INTEGRITY libraries are built, selecting **Intensive** for ARM targets is not guaranteed to work.

Run-time memory checks are not supported when using shared libraries. You should disable the use of shared libraries when enabling any of the run-time memory check options (for information about disabling shared libraries, see “Shared Libraries” in the “Building INTEGRITY Applications” chapter of this manual). If you have disabled this option and still receive compile-time errors when building with run-time memory checking enabled, the option may be enabled in another project file. Make sure that no project file is using shared libraries.

For more information about run-time Memory Checking, see “Using the Memory Allocations Window” in *MULTI: Debugging* and “Run-Time Memory Checking” in the *MULTI: Building* book.

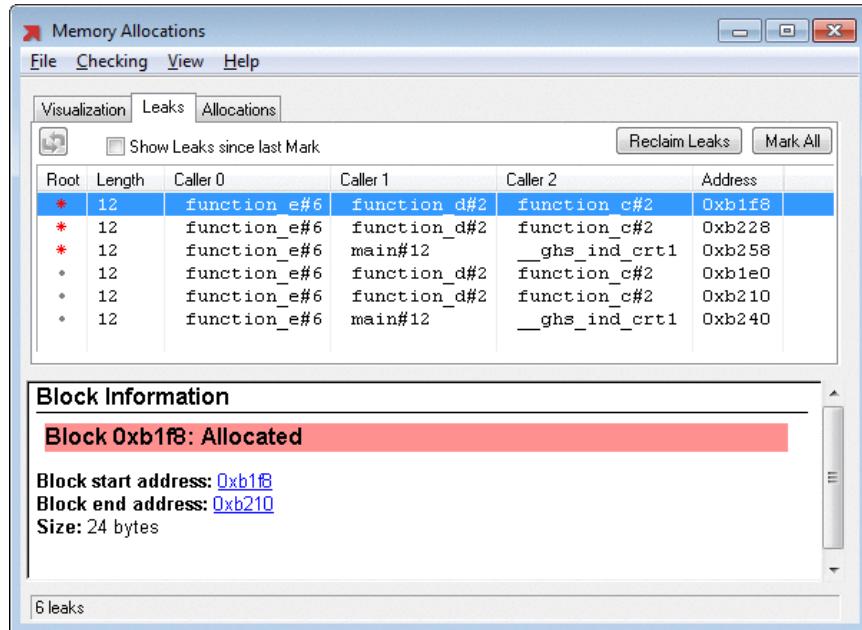
The following are some common memory allocation errors you can detect:

- Memory Leak
- Access Beyond Allocated Area
- Bad Free

15.2.1 Memory Leak

In order to detect memory leaks do one of the following:

- Type the Debugger command `findLeaks` during the debugging of a task. All memory leaks up to that point will be detected. To find all leaks, enter the command after debugging of the task has completed.
- Select **View⇒Memory Allocations** in the MULTI Debugger, then select the **Leaks** tab. All memory leaks up to that point will be detected. To find all leaks, wait until debugging of the task has completed. If leaks are found, they will be displayed in the window, similar to the following:



```
static char *p;
foo(void)
{
    p = malloc(100);
    p = 0;
/*
 * several leaks are now generated since the
 * allocated areas are no longer referenced
 */
}
```

False positives may occur in INTEGRITY applications where thread local storage is allocated from the MemoryPool, such as for Tasks declared in the .int file or created with CreateProtectedTask() or CommonCreateTask(). Multi-threaded INTEGRITY applications may also generate false positives if the thread stack comes from the MemoryPool.

15.2.2 Access Beyond Allocated Area

This level of checking requires that you select **Memory Allocation (Intensive)**. This option causes the compiler to instrument each memory access to check its validity, therefore the size and speed overhead is high. An example of such an error follows:

```
foo(void)
{
    int i;
    char *p = malloc(100);
    for (i = 0; i <= 100; i++)
        *p++ = '@';
```

```
    /* last store in the loop is BAD! */
}
```

The diagnostic message corresponding to this error is as follows:

```
Attempt to read/write memory not yet allocated
```

If the malloc internals get corrupted (for example, by writing past the end of an array), subsequent calls to malloc-related functions can cause the library to detect this corruption and display an appropriate diagnostic:

```
Malloc internals (free list?) corrupted
```

15.2.3 Bad Free

```
static char *p;
foo(void)
{
    int a;
    p = malloc(100);
    free(p);
    free(p);      /* error!  can't free twice */
    free(&a);      /* error!  never allocated */
}
```

Diagnostic messages corresponding to bad free are as follows:

```
Attempt to free something already free
Attempt to free something not allocated
```

15.3 Detecting Stack Overflow

Stack overflow detection capability is almost essential for debugging multitasking applications. Green Hills provides a utility, **gstack**, which can statically determine maximum stack usage from a given entry point for programs built with debugging information. However, run-time detection of stack overflows may still be desirable. For example, a program may have cycles/recursion, or may not be built with debugging information. More importantly, run-time stack flow detection can determine the problem regardless of whether the user remembers to run gstack after making modifications to the code, or in cases where running gstack is not practical.

15.3.1 Automatic Stack Overflow Detection

Automatic stack overflow detection is a mechanism that works for Tasks specified in the Integrate BootTables (such as the Initial Task in a virtual AddressSpace), or Tasks created via the CreateProtectedTask() API (consult the *INTEGRITY Kernel Reference Guide* for details regarding the use of this API). To automatically detect stack overflow INTEGRITY uses a “guard” page, which is a page of virtual memory just below a Task’s stack that is left unmapped. Because it is unmapped, when the Task attempts to access below its allocated stack, it generates an exception (an INTEGRITY violation) which is easily detected and corrected. This mechanism only works for Tasks running in a virtual AddressSpace because the guard page utilizes virtual memory.

When Tasks are created via Integrate BootTables or CreateProtectedTask(), one or more physical pages are used to hold the Task’s page-aligned stack area, so the guard page can be implemented below. No recompilation or special reconfiguration is required to get this stack overflow detection feature, and other than the space used for the allocated stack itself, there is no overhead associated with this feature.

15.3.2 Instrumented Stack Checking

Tasks created dynamically via the CreateANSICTask() API (or from higher level middleware APIs that make use of CreateANSICTask()), have their stacks allocated from their AddressSpace’s run-time heap and do not benefit from automatic stack overflow detection. For Tasks created via CreateANSICTask(), compiler instrumentation can be used to help detect stack overflow.

- When a run-time stack overflow is detected via this instrumented run-time error check, a message is typically displayed to the INDRT2 (rtserv2) I/O pane.
- In some cases where other run-time error checking options have been enabled and the affected Task is attached by the Debugger, this diagnostic message will be displayed in the Debugger’s command pane.

Instrumented stack overflow checking is performed by enabling the **-stack_check** compiler option to the project or individual files. Instrumented stack checking only supports checking routines called from a Task context. Routines called from non-Task contexts (e.g., ISRs, BSP initialization, etc.) must not be compiled with this option.

When the **-stack_check** is enabled, calls to `__stkchk()` are inserted by the compiler at the beginning of every function compiled with this option. The library code checks that the current

stack pointer has not exceeded the Task's stack limit. If the stack pointer has exceeded the stack limit, the Task halts. Additionally, if the AddressSpace was built with Run-time Memory Checks enabled (see "Using Run-time Memory Checks" for more information), the diagnostic message `Maximum stack use exceeded` is displayed.

- If you are attached to a non-Protected Task that overflows its stack, the Task will be halted in the stack checking code.
- If you are not attached to the Task that overflows its stack, it will attempt to halt itself.
- In some cases, stack corruption can cause the halt to fail. If this occurs, the Task goes into an infinite busy loop.
- If a Protected Task exceeds its stack by less than a page, it will get an INTEGRITY violation (the same behavior as when stack checking is not compiled into the system).

Typically, a limited amount of memory corruption will occur before the overflow can be detected. This may cause other Tasks in the same AddressSpace to behave unreliably. It is recommended that the overflow condition be redressed and the program restarted before attempting further debugging.

15.4 Overhead of Run-time Error Checking

It is expected that run-time error checking be used as a debugging tool, not for production code. The overhead for some types of error checking is substantial:

- For stack overflow detection, the overhead is a constant time call per function.
- For memory allocation (for example, bad free) errors, the library routines execute somewhat slower due to information gathering, but the user code is unchanged.
- For case/switch errors, user code has an extra compare and branch for each switch statement.
- If full memory checking is enabled, all memory accesses cause an extra compare, branch, and constant time run-time routine call.

The relative performance overhead depends upon the type of run-time error checking and the kind of application in use (if no calls to malloc() or free() are made in the application, bad free checking will not be used).

If necessary, you can take advantage of the MULTI Profile window to measure time degradation and the **gsize** or **gfunsize** utilities to measure code size degradation due to run-time error checking overhead. Dynamic memory allocation checking incurs overhead in the use of heap that is not easily measured, although running out of heap causes NULL return values from malloc(), etc., that can be easily detected.

Chapter 16

Using the MULTI ResourceAnalyzer

This chapter provides information about using the ResourceAnalyzer and contains the following sections:

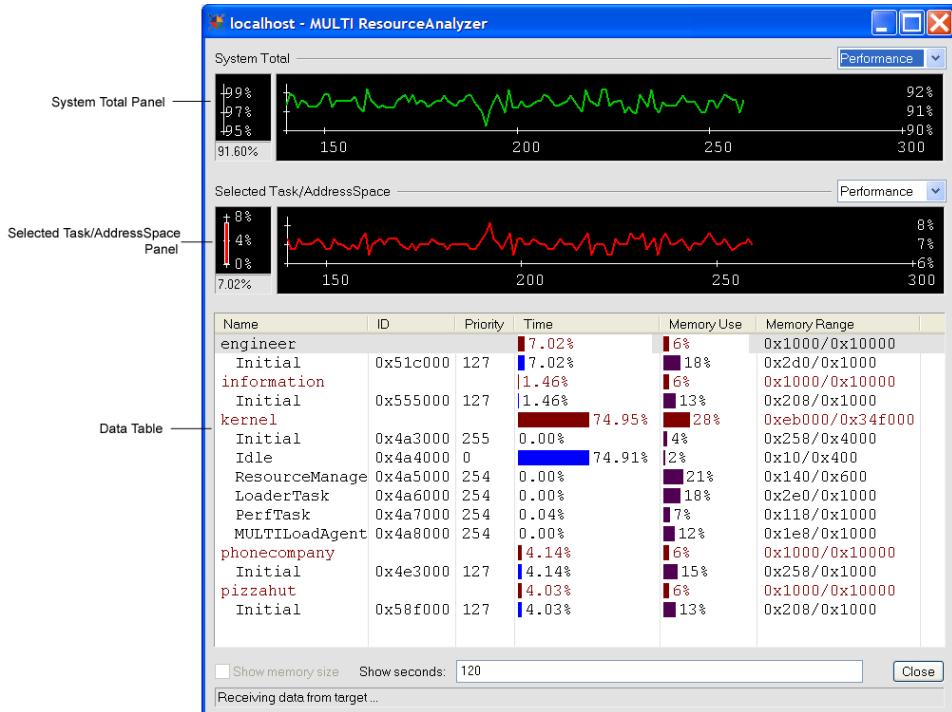
- Introduction to the MULTI ResourceAnalyzer
- Working with the ResourceAnalyzer

16.1 Introduction to the MULTI ResourceAnalyzer

The MULTI ResourceAnalyzer is a host-based graphical tool that allows system designers to monitor the CPU and memory usage of a real-time embedded system while it is running on the target.

In addition to assigning stack sizes to Tasks, designers of INTEGRITY systems must also assign memory to the AddressSpaces in which those Tasks run. The ResourceAnalyzer allows system designers to track the patterns of CPU and memory usage in order to maximize the efficiency of their memory assignments for both AddressSpaces and Tasks. The ability to track this information in a running system is invaluable even to the most experienced system designer.

The ResourceAnalyzer communicates with an INTEGRITY agent on the target system via a socket connection. The INTEGRITY agent periodically sends a snapshot of its system's memory and CPU usage statistics to the host, and the ResourceAnalyzer displays the information in an intuitive, configurable interface. The ResourceAnalyzer monitors the consumption of CPU resources for every Task and AddressSpace, as well as the overall system and monitors the consumption of memory resources for every Task and AddressSpace, including the kernel AddressSpace.



The ResourceAnalyzer user interface is divided into three main components:

- System Total panel — displays statistics for the entire embedded system.
- Selected Task/AddressSpace panel — displays statistics for the item selected in the data table. If you select a different Task or AddressSpace in the data table, the display changes to show statistics for the newly selected item.
- Data Table — lists all statistics sent from the target.

The panels graphically display the history and current values of performance (CPU) or memory statistics. Each panel contains a drop-down menu where you can select **Performance** to display CPU information, or **Memory** to display memory statistics. For more information, see “Working with the ResourceAnalyzer” and “System Total and Selected Task/AddressSpace Graphical Panels”.

16.2 Working with the ResourceAnalyzer

The ResourceAnalyzer communicates with the target system from the host over a socket connection, which relies on a network connection between the host and the target system. Before trying to run the ResourceAnalyzer, ensure that your development environment has a network connection.

Linking with ResourceAnalyzer library (**libperf.a**) causes the ResourceAnalyzer target agent, PerfTask, to be included in the system. When the ResourceAnalyzer establishes a connection to the target, the GUI communicates over a socket to the PerfTask. After the connection is established, PerfTask will periodically send up a data sample that describes the Task, AddressSpace, and system performance and memory use statistics that are displayed in the GUI.

To use the ResourceAnalyzer:

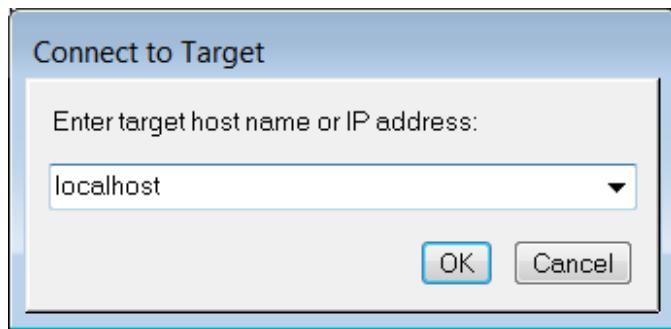
1. Configure your kernel with the following:

- On the Project Manager: Settings for Kernel dialog link the **ResourceAnalyzer** (**libperf.a**) and **Socket** (**libsocket.a**) libraries into the INTEGRITY kernel. These libraries are minimally intrusive to the run-time system's behavior; the INTEGRITY agent does not send data to the host unless the ResourceAnalyzer is connected.
- The INTEGRITY kernel must be configured with TCP/IP networking support. Use the Settings for Kernel dialog to link the appropriate **GHnet Stack** library into the kernel. For more information, see the “Building with the GHnet v2 TCP/IP Stack” section of the *INTEGRITY Networking Guide*.

For INTEGRITY simulators, socket emulation is used instead of a GHnet stack. Socket emulation is provided by linking in **libsocket.a** and no additional stack libraries are needed.

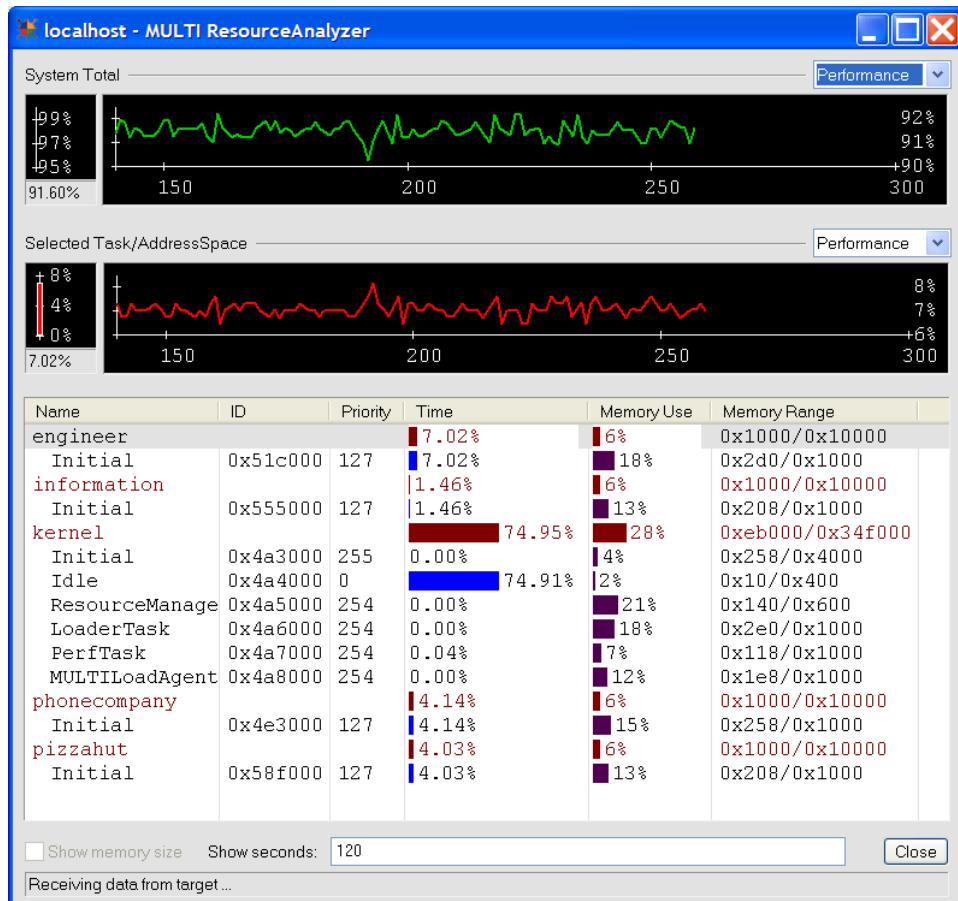
- Make sure that the **Debugging** (**libdebug.a**) library is linked into the INTEGRITY kernel. The **kernel.gpj** that ships with the INTEGRITY reference BSPs is linked with **libdebug.a** by default.
- **NO_STACK_TRACKING** must not be defined.

2. Establish a run-mode debugging connection to your target and download the application.
3. Select **Tools**→**ResourceAnalyzer** in the Debugger window. The Connect to Target dialog will open.



4. Enter the fully qualified hostname or IP address of the target to which you wish to connect. For INTEGRITY simulators, use the name or address of the host running the simulator.

The ResourceAnalyzer will open and graphically display the history and current values of performance and memory statistics.



The CPU resources of the system are tracked at the Task level, the AddressSpace level, and the kernel level (which is sometimes referred to as the “system” in UNIX resource analysis utilities such as **top**, **ps**, and **time**). In a multi-core system, the CPU usage may be greater than 100% because the ResourceAnalyzer reports the sum total of CPU usage for all cores. In a multi-core system, the maximum CPU usage is the number of cores multiplied by 100%:

- For each Task, the ResourceAnalyzer tracks the percentage of total execution time that the system spent in the Task.
- For each AddressSpace, the ResourceAnalyzer tracks the percentage of total execution time that the system spent on all Tasks within that AddressSpace.
- The ResourceAnalyzer tracks how much time was spent in the kernel relative to the total execution time. Thus percentages may not sum to 100% times the number of cores. The time not attributed to any AddressSpace was used by the system.
- For kernels that have the Idle Task(s) built in, the ResourceAnalyzer conveniently displays how much idle time is in the system for each core as the percentage of time attributed to the core-specific Idle Task. For example, Idle0 shows the idle time associated with Core 0 in an SMP system. For non-SMP systems, the one Idle Task shows system-wide idle time.

The memory resources of the system are tracked at the Task level, the AddressSpace level, and the entire run-time system level:

- For each Task, the ResourceAnalyzer tracks the most stack space that the Task used at any one time (high water mark), and the Task's allocated stack size.
- For each AddressSpace, the ResourceAnalyzer tracks the current memory usage and allocated memory quota.
- For the kernel AddressSpace, the memory quota reported corresponds to the amount of free memory pages in the system when the kernel booted.

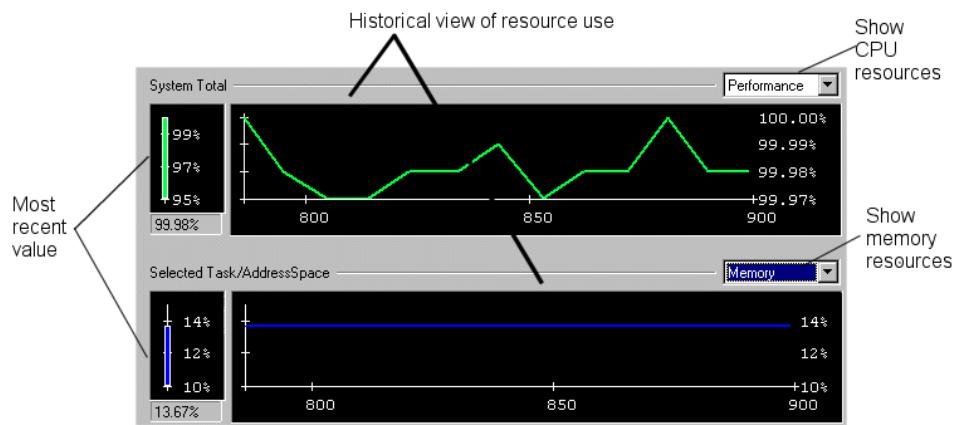
16.2.1 System Total and Selected Task/AddressSpace Graphical Panels

The ResourceAnalyzer's graphical panels make it easy to track the system's CPU and memory statistics.

- The System Total panel displays statistics for the entire system.
- The Selected Task/AddressSpace Panel displays statistics for a specific Task or AddressSpace.

To view statistics for a specific Task or AddressSpace:

1. Select that item in the data table.



2. Use the drop-down list to select whether CPU usage statistics (**Performance**) or memory usage statistics (**Memory**) are displayed in the respective graphical panel.

If you select **Memory**, the **Show memory size** field becomes available at the bottom of the ResourceAnalyzer. If you select **Show memory size**, the graphical display changes from the percentage listed in the **Memory Use** column of the data table to the amount of memory used as listed in the **Memory Range** column of the data table.

3. As the system runs, the Resource Analyzer charts the CPU and memory usage over time, creating a line graph. To adjust both the horizontal scale (number of seconds shown) and vertical scale (range of percentages) of the line graph, enter a new value in the **Show Seconds** field or select zoom options from the shortcut menu.

To the left of the line graph, the ResourceAnalyzer graphically displays the most current CPU or memory usage statistic:

- For **Performance** statistics, the graphed value corresponds to the current value of the data table's **Time** column.
- For **Memory** statistics, the graphed value corresponds to the current value of the **Memory Use** column (unless **Show memory size** is selected, in which case the first value in the **Memory Range** column is shown). This graph changes every time the ResourceAnalyzer receives new data from the target system.

16.2.2 Data Table

The data table is the bottom panel of the ResourceAnalyzer. It contains the statistics most recently gathered from the running target. These statistics change every time the target sends updated information to the ResourceAnalyzer.

	Name	ID	Priority	Time	Memory Use	Memory Range
AddressSpace	engineer			7.02%	6%	0x1000/0x10000
	Task	Initial	127	7.02%	18%	0x2d0/0x1000
	information			1.46%	6%	0x1000/0x10000
KernelSpace	Initial	0x51c000	127	1.46%	13%	0x208/0x1000
	kernel			74.95%	28%	0xeb000/0x34f000
	Initial	0x4a3000	255	0.00%	4%	0x258/0x4000
	Idle	0x4a4000	0	74.91%	2%	0x10/0x400
	ResourceManage	0x4a5000	254	0.00%	21%	0x140/0x600
	LoaderTask	0x4a6000	254	0.00%	18%	0x2e0/0x1000
	PerfTask	0x4a7000	254	0.04%	7%	0x118/0x1000
	MULTILoadAgent	0x4a8000	254	0.00%	12%	0x1e8/0x1000
	phonecompany			4.14%	6%	0x1000/0x10000
	Initial	0x4e3000	127	4.14%	15%	0x258/0x1000
pizza hut	pizzahut			4.03%	6%	0x1000/0x10000
	Initial	0x58f000	127	4.03%	13%	0x208/0x1000

The data table contains the following columns:

- **Name** — The name of the Tasks and AddressSpace on the target system. The Tasks are grouped under the AddressSpace in which they run.
- **ID** — The task ID of each Task.
- **Priority** — The priority level of each Task.
- **Time** — The percentage of total execution time that the system spent in a particular Task or AddressSpace. For example, if the **Time** column contains 15% then the system spent 15% of its total execution time running that particular Task. The time spent in an AddressSpace equals the percentage spent running each of its Tasks.
- **Memory Use**:
 - For each Task, **Memory Use** displays the highest percentage of the allotted stack space that the Task used at one time (its high water mark). For example, if the **Memory Use** column contains 10%, then the Task's high water mark was 10% of the total memory allocated to that Task.
 - For each AddressSpace, **Memory Use** displays the percentage of the allocated memory space that the AddressSpace is currently using. Memory use for an AddressSpace refers to MemoryPool usage and the allocation of memory from the free page list. The ResourceAnalyzer does not display statistics for heap usage. However, because the heap can be automatically expanded using pages from the free page list, MemoryPool usage may include pages that were used to expand the heap. For information about the MemoryPool and the free page list, see “MemoryPool Configuration” in the “Common Application Development Issues” chapter.
- **Memory Range**:
 - For each Task, **Memory Range** displays the Task's high water mark and the Task's total allocated stack space.
 - For each AddressSpace, **Memory Range** displays the AddressSpace's current memory usage and the AddressSpace's currently allocated memory space. The allocated memory space may be different than the AddressSpace's original MemoryPool size if additional pages have been added to the AddressSpace's free page list after it started running.

Chapter 17

Core File Debugging

This chapter contains:

- System Core Dumps
- Application Core Dumps
- Multiple Core Dumps
- Debugging Core Dumps
- Advanced Core Dumping

Note: The current core file debugging behavior and the core dumping library (**libcore.a**) have been deprecated. Core file debugging will continue to be supported, however the implementation may be modified in a future INTEGRITY release at which point the current behavior and library may be removed from INTEGRITY.

INTEGRITY uses **libcore.a** to support core dumping to RAM, flash memory, or the file system. INTEGRITY supports dumping both application cores and system cores.

For more information about Core Dump API calls discussed in this chapter, see the “Core Dump API” chapter of the *INTEGRITY Libraries and Utilities Reference Guide*.

17.1 System Core Dumps

When the kernel is linked with **libcore.a** and the core memory region is set up correctly, system core dumps are automatically generated whenever the kernel encounters a fatal error.

By default, system core dumps are stored in the `.mr__core` section and can be retrieved by dumping that section of memory to a file using a System Debug connection (`memdump raw filename .mr__core`). This `.mr__core` section is defined in **ram_sections.id** file and included in each BSP’s **default.id**. Its size is limited by the linker constant `__INTEGRITY_MaxCoreSize`, which can be changed in the file or overridden by passing a `-C` option to the linker.

If the function `BSP_FlashCoreLimits()` is defined in the kernel program, system core dumps are stored in flash memory. Storing system core dumps in flash memory can be useful for debugging a kernel/BSP problem when the BSP does not support a System Debug solution.

```
void BSP_FlashCoreLimits(Address* offset, Address* limit) {
    *offset = 0xF1000000;
    *limit  = 0xF11FFFFF;
}
```

The core dump is stored between the offset and limit addresses in flash memory.

To retrieve the core dump:

- Use a ROM monitor, or
- Reboot, connect with INDRT (rtserv), and dump the flash memory.

17.2 Application Core Dumps

You can configure INTEGRITY to write a core dump out to a file system whenever a Task encounters an INTEGRITY violation. Two things must be configured: the system side and the application side. For an example, see the **FieldUpgrade1** example under **featureexamples.gpj**.

To configure the system side:

1. An AddressSpace linked with **libcore.a** must have a Task which calls RegisterCoreFileErrorHandler() (declared in **<sys\core.h>**).
2. A file system should be present with sufficient space to hold any possible image.

To configure the application side:

- The following line must be a global declaration in every program:

```
DEFINE_ERROR_OPTIONS(IERROR_SYSTEM,NULL,0);
```

An application core dump contains:

- All sections for the AddressSpace.
- The registers and stack for the Task that encountered the INTEGRITY violation.

17.3 Multiple Core Dumps

It is possible to write multiple application core dumps to the file system. However, only one core file can be stored in RAM or flash. An application level core dump will overwrite any core file in RAM because the same storage is used to assemble the core file before it is written to the file system.

17.4 Debugging Core Dumps

Both system and application core dumps are debugged by executing `multi` from the command line with the core file as its argument. A file chooser dialog will appear requesting the INTEGRITY Application or kernel corresponding to the core dump.

- For a system core dump, a MULTI Debugger opens, with the program counter at the function `DumpKernelCore()`.
- For an application core dump, a Task List window opens, which you can use to select the Task to debug. The Task you debug stops wherever it encountered the INTEGRITY violation.

When debugging a core file, you can move up and down the stack, use the browse commands, and view memory and registers. You cannot run or step the program when debugging a core file. If memory is unreadable, or it is all zeros, it may not have been included in the dump. For an application level dump, this should not occur. For a system level dump, this occurs for all data that was allocated off of the free page list.

17.5 Advanced Core Dumping

There is a lower-level API for the core dump functionality. With this API, you can dump an application-level core to flash, or dump multiple Tasks into the same application core dump.

For more information, see the “Core Dump API” chapter of the *INTEGRITY Libraries and Utilities Reference Guide*.

Chapter 18

Debug Agent Commands

The debug agent supports several target commands that can be used to inspect and manipulate Tasks and other aspects of the running system. These commands can be entered either in the Target pane when connected via rtserver2, or on the INDRT2 serial port when the INTEGRITY BSP supports INDRT2 over a serial line. For more information, see the “Connecting with INDRT2 (rtserver2)” chapter of this manual.

Note: These commands require that the Initial Task is runnable and not halted for debugging.

18.1 Supported Debug Agent Commands

The debug agent supports the following commands:

```
lt  - list tasks
ct  - clear multitask profile timers
crt - crt ASName TaskName EntryPoint [StackSize [Priority [Weight]]]
rt  - Run a task: rt AddressSpace TaskName
ht  - Halt a running task: ht AddressSpace TaskName
kt  - Terminate a task: kt AddressSpace TaskName
is  - Enable / disable INDRT over serial: is [on | off]
clt - close task: clt AddressSpace Taskname
sm  - Inject message: sm <AddressSpace> <Object index> "<message>"
lm  - list loaded images
rset - reboots board (not always supported)
bl  - get or set board location (not always meaningful)
map - display BSP memory table
nc [d<dev>]
    [ [e EthernetAddress] | [i IPAddress] | [g Default Gateway] |
      [n Netmask] | [m Namesrvr] | [h Hostname] | [p<ON|OFF> (DHCP ON|OFF)] |
      [b<0|1> (TFTP Boot OFF|ON)] | [l Boot-Delay-seconds] |
      [t TFTP-Server-IPAddress] | [f filename-to-boot] | [r INDRT-Interface] |
      [v SNTP-Server-IPAddress] | [z HH:MM Timezone Offset] |
      [o SNTP Polling Interval (s)] |
      [s Save to non-volatile memory] ] - network configuration
```

Note: **crt**, **rt**, **ht**, **kt** and **clt** have been deprecated.

- **It** — Lists the Tasks in the system. The format of the output is similar to the following:

Task Id	Status	Memory: Used/Size or Pri Stack: HiWater/Size			Time	Task Name
pizza-kernel		0x0001d000/0x00596000				
0x0025c000	exited	255	0x00000180/0x00004000	0.11%	Initial	
*0x0025d000	running	0	0x00000018/0x00000400	98.92%	Idle	
mpizzahut		0x00001000/0x00010000				
0x001b8000	halted	127	0x00000120/0x00001000	0.00%	Initial	
minformation		0x00001000/0x00010000				
0x001a8000	halted	127	0x00000120/0x00001000	0.00%	Initial	
mengineer		0x00001000/0x00010000				
0x00198000	halted	127	0x00000120/0x00001000	0.00%	Initial	
mphonecompany		0x00001000/0x00010000				
0x00188000	halted	127	0x00000120/0x00001000	0.00%	Initial	

- Tasks are displayed under each AddressSpace in which they reside:
 - The asterisk to the left of the Task Id indicates the Task that is currently executing at the time of the **It** command (or if none are executing, the last Task executed).
 - Status is similar to the status displayed in the INDRT2 (rtser2) Task window and is generally self-explanatory.
 - Pri indicates the priority of the Task.
 - Memory: Used/Size or Stack: HiWater/Size:
 - * Memory: Used/Size is displayed for AddressSpaces. This column displays the amount of MemoryPool memory used in relation to the AddressSpace's total MemoryPool (its physical memory quota). This field does not display heap usage.
 - * Stack: HiWater/Size is displayed for Tasks. This column displays the current maximum stack use and the stack size for each Task. The stack use values will display 0 if the system is built with stack tracking disabled.
 - Time displays the relative execution time taken by each Task in the system.
 - TaskName displays the name of the Task. The Idle Task in KernelSpace runs a spin loop at minimum priority so that users analyzing system performance can determine the amount of CPU utilization. The Time displayed for the Idle Task is essentially the amount of slack time currently in the system. Note that the Idle Task runs at priority 0; therefore, in order to get a meaningful measurement of CPU utilization, no other Tasks should be run at this minimum priority level.

Note: With SMP, there is one Idle Task per processor and the NO_IDLE_TASK option is not supported.

- **ct** — Clears or resets the time counters used for the purpose of displaying the relative execution percentages.
- **crt** — Creates a Task in an AddressSpace. This command is meant to execute a simple C function in a Task context. Note that the newly created Task is C Library enabled.

This command uses the AddressSpace's Initial Task to create the Task. For this operation to succeed, the Initial Task must not be halted (directly, via a breakpoint, or programmatically by calling `HaltTask()` on it). It may, however, have exited.

After the Task is created successfully, it is put into Halted state. You can use the **rt** command to run the Task at a later time. The **crt** command (discussed later) does not terminate the Task. You can terminate a Task previously created with the **rt** or the **crt** command by using the **kt** command while the Task is still in the system.

The **crt** command uses the following syntax:

```
crt AddressSpaceName TaskName EntryPoint [StackSize] [Priority]  
[Weight]
```

- `AddressSpaceName` must be the name of a valid AddressSpace. Enclose the AddressSpace name in double quotes if the name contains white space. The **lt** command shows the names of the AddressSpaces running in the system.
- `TaskName` must be unique in the given AddressSpace. `TaskName` must be enclosed in double quotes if it contains white space.
- `EntryPoint` is the logical address in the AddressSpace where the new Task should begin execution. In KernelSpace, `EntryPoint` is the kernel address. In a virtual AddressSpace, `EntryPoint` is the virtual address in that AddressSpace.
- `StackSize` is optional. It sets the size of the stack for the Task. The default stack size is the page size defined by the ASP. For Tasks created in KernelSpace, a free kernel page is used for the stack. Therefore the stack size for Tasks created in KernelSpace is the page size. If the `StackSize` is specified as zero, the stack size defaults to page size.
- `Priority` is optional. It sets the priority for the Task. Default is 127.
- `Weight` is optional. It sets the weight for the Task. Default is 1.

The following command creates a Task named `mydebug` in the `mcsender` AddressSpace at an entry point of address `0x1012c` with the default stack size, priority 255, and the default weight:

```
crt mcsender mydebug 0x1012c 0 255
```

- **rt** — Runs a Task in an AddressSpace. If the Task does not exist, and the `EntryPoint` is specified, the **rt** command creates the Task and runs it. When creating a new Task, the priority of the Task is 127, the weight is 1, and the stack size is the page size defined by the ASP.

The **rt** command runs a Task created by the **rt** or the **crt** command. It can also run a Task that has been in the system and is in Halted state.

When using the **rt** command to create a new Task, all the notes mentioned in the **crt** command apply. Refer to the **crt** command for details.

The **rt** command uses the following syntax:

```
rt AddressSpaceName TaskName [EntryPoint]
```

- **AddressSpaceName** must be the name of a valid AddressSpace. Enclose the AddressSpace name in double quotes if the name contains white space. The **It** command shows the names of the AddressSpaces running in the system.
- **TaskName** must be unique in a given AddressSpace. **TaskName** must be enclosed in double quotes if it contains white space.
- **EntryPoint** is optional. It sets the logical address in the AddressSpace where the new Task should begin execution. If **EntryPoint** is specified, the **rt** command creates a new Task in the AddressSpace and runs the Task. In KernelSpace, **EntryPoint** is the kernel address. In a virtual AddressSpace, **EntryPoint** is the virtual address in that AddressSpace.

If **EntryPoint** is not presented, the **rt** command attempts to run an existing Task named **TaskName** in the AddressSpace. The command does nothing if it fails to find the named Task in the AddressSpace.

- **ht** — Halts a Task running in the system. If the Task is not in running state, the Task does nothing.

The **ht** command uses the following syntax:

```
ht AddressSpaceName TaskName
```

- **AddressSpaceName** must be the name of a valid AddressSpace. Enclose the AddressSpace name in double quotes if the name contains white space. The **It** command displays the names of the AddressSpaces running in the system.
 - **TaskName** must be the unique name of a Task running in the AddressSpace.
-
- **kt** — Terminates a Task. Only Tasks created by **crt** or **rt** can be terminated by **kt**. The syntax of this command is the same as the **ht** command.
 - **is** — Enable / disable INDRT2 control over the serial port. Uses the following syntax:

```
is [on|off]
```

- **clt** — Cleans up kernel resources allocated to a Task created by **rt** and **crt**. Uses the same syntax as **ht**.

The **crt** and **rt** commands take kernel resources (such as memory) when creating a new Task. Even if the Task terminates by calling Exit(), its kernel resources are not freed until **clt** is executed. Failure to call **clt** after a Task has terminated causes a memory leak. Only Tasks created by the **rt** or **crt** commands can be closed by the **clt** command. Before such a Task can be closed, it must have been terminated by either calling Exit(), or via an explicit **kt** command.

This command uses the AddressSpace's Initial Task. For this operation to succeed, the Initial Task must not be halted (directly, via a breakpoint, or programmatically by calling HaltTask() on it).

- **sm** — Injects a message over a Connection. Uses the following syntax:

```
sm AddressSpace ObjectIndex "messagedata"
```

- **AddressSpace** must be the name of a valid AddressSpace. Enclose the AddressSpace name in double quotes if the name contains white space. **lt** shows the names of the AddressSpaces running in the system.
- **ObjectIndex** describes the index of the Connection Object to send the message on.
- "messagedata" is a string (enclosed in quotes) to send on the Connection.

- **lm** — Lists the currently dynamically loaded images. If no modules are currently loaded, (None) is displayed.

Note: When the Loader is in a virtual AddressSpace, the **lm** command cannot access the Loader state and will always display (None).

- **rset** — Reboots board (not supported on all targets).
- **bl** — Sets board location (not supported on all targets).
- **map** — Displays the BSP memory table.
- **nc** — Actually a family of commands used to set and display the network configuration. Consult the “Network Configuration” chapter of the *INTEGRITY Networking Guide* for more information.

The following commands have been deprecated and are supported for backward compatibility. They will be replaced in a future release:

- **mr** — Displays the contents of memory in a given AddressSpace. This command takes the following arguments:

```
mr AddressSpace Address NumberOfBytes
```

- **AddressSpace** is the name of the AddressSpace. command.
- **Address** is the memory address of interest and is interpreted as a virtual address when a virtual AddressSpace is specified.
- **NumberOfBytes** is optional. To specify a hexadecimal value, prefix the number with **0x**. If no **NumberOfBytes** value is specified, the command reads one byte from the target address.

For example:

```
mr kernel 0x1234 0x5  
00001234: 9083 0028 80  
  
mr kernel 0x1234  
00001234: 90
```

- **mw** — Writes a 4-byte value to a memory location in a given AddressSpace. This command takes the following arguments:

```
mw AddressSpace Address Value
```

- **Address** must be 4-byte aligned and is interpreted as a virtual address when a virtual AddressSpace is specified. To specify a hexadecimal value, prefix the number with **0x**.

Chapter 19

Security Issues

This chapter contains the following sections:

- Security and INTEGRITY Kernel
- Security and the INDRT2 Debug Agent
- Security and Networking Software
- Security and ResourceManagers
- Security and the EventAnalyzer
- Multi-Level Security

The INTEGRITY kernel is designed from the ground up to be a perfect fit for security-critical environments. INTEGRITY provides the ability to prove resource availability. INTEGRITY also provides a number of methods to prevent malicious or erroneous Tasks from harming the kernel or code running in AddressSpaces other than their own, and deterministic response for kernel services.

Some INTEGRITY applications have more stringent security requirements than others. Some of INTEGRITY's layered software modules may not be appropriate for the most security-demanding applications or may require more complex methods to ensure that they are used in a secure manner. The security issues for each module are described in the following sections.

19.1 Security and INTEGRITY Kernel

The security of any INTEGRITY application depends on the security of the INTEGRITY kernel and the BSP. Developers must take precautions to protect the kernel and BSP from security holes. Many security holes that can affect the entire system can be prevented by putting libraries and code in virtual AddressSpaces instead of KernelSpace.

Linking libraries into the kernel can cause security holes. For example, **libload.a**, **libdebug.a**, and **libres.a** allow remote access to the kernel. To protect security, libraries that allow such access should be removed from the kernel before shipping. The kernel libraries that ship with INTEGRITY are secure and can be used in shipping products.

Any bug in code running in KernelSpace can create a security hole. For example, `ReadIODeviceStatus()` and `WriteIODeviceStatus()` can be abused through a buffer overrun bug in some `IODevice`'s `ReadStatus()` or `WriteStatus()` member function. Developers must make sure that code running in the kernel does not contain bugs that will compromise security.

Incorrect BSP design can create security holes. Failure to comply with all of the requirements in *INTEGRITY BSP Guide* and your BSP's `.notes` file may result in an insecure system.

INTEGRITY implements an optional Partition Scheduler that can help protect security by providing guaranteed CPU resource availability at the partition level. For more information, see the “Enhanced Partition Scheduler” chapter of the *INTEGRITY Libraries and Utilities User’s Guide*.

19.2 Security and the INDRT2 Debug Agent

INDRT2, which is included by linking `libdebug.a` into the KernelSpace program, is not a secure module. Use of this debug agent is valuable during the debugging phase, but it should almost always be removed from your product before shipping. INDRT2 allows your system to be controlled remotely, which is an important capability when debugging problems on your target, however it is most likely not appropriate for your final product. While use of this debug agent in a production system is possible, system designers must understand and be comfortable with its impact upon security.

INDRT2 is designed to help you during the product development phase of your project and may even be used during some of your testing. However, you should test your product in the same configuration that you ship it. Because Green Hills Software does not recommend unconditionally including the INDRT2 debug agent in your shipping product, you should perform exhaustive testing of your product in its final configuration, which is usually without the INDRT2 debug agent.

For more information about INDRT2, see the “Connecting with INDRT2 (`rtserv2`)” and “Run-Mode Debugging” chapters of this manual.

19.3 Security and Networking Software

Standard TCP/IP and sockets are known to be insecure. INTEGRITY does not currently attempt to address the issue of secure communication across an insecure network. This is left for third party vendors.

For more information about TCP/IP and sockets, see the “Using GHnet v2” chapter of the *INTEGRITY Networking Guide*.

19.4 Security and ResourceManagers

ResourceManager registrations and requests are accomplished by sending messages across a Connection, one end of which is accessed by the ResourceManager Task. When a successful registration of an Object occurs, the Object is transferred into the ResourceManager's AddressSpace, taking up memory until it is successfully requested and transferred to another

AddressSpace. In addition, registering buffer resources causes the ResourceManager to use heap memory to store the buffer. Although the amount of heap memory used by a ResourceManager can be bounded, a malicious Task could register so many Objects that the ResourceManager's AddressSpace runs out of memory.

Another security issue pertains to the naming of Objects. The namespace of Objects is flat, meaning that after a Task successfully registers an Object with a particular name, that name cannot be used for a subsequent registration as long as the first Object remains registered. A malicious Task could register Objects with certain names that might prevent other Tasks in different AddressSpaces from using the same name, thus precluding proper operation.

These security issues can be mitigated by allowing only authorized AddressSpaces to access a given ResourceManager. Using multiple ResourceManagers, it is possible to create isolated resource domains in which membership is statically defined by the system integrator. For more information, see the “Using ResourceManagers to Securely Partition System Resources” section in the *INTEGRITY Libraries and Utilities User’s Guide*.

Some INTEGRITY middleware packages, such as the File System and TCP/IP stack, utilize the ResourceManager to connect clients and servers. For an example of how to provide controlled access to these services without exposing their clients to unrelated resources, see the **ResourceProtect2** and **IsolatedDownload** examples shipped with your INTEGRITY distribution.

For more information about the ResourceManager, see the “ResourceManagers” chapter of the *INTEGRITY Libraries and Utilities User’s Guide*.

19.5 Security and the EventAnalyzer

When using Live-mode, event data is transferred to the host computer via sockets and is therefore not secure. The KernelSpace live-mode agents (MEVLogTask and MEVCmdTask) cannot be used and are not a threat to security if the KernelSpace program does not include TCP/IP and sockets.

In Post-Mortem mode, logging is secure because all events are logged in target memory. Transportation of this data to a location other than the associated processor's target memory is the user's responsibility.

For more information about the EventAnalyzer, see the *EventAnalyzer User’s Guide*.

19.6 Multi-Level Security

On a particular CPU, application code can be designed to run in separate AddressSpaces, providing protection between these AddressSpaces. A Task in one AddressSpace cannot access the memory or Objects in another AddressSpace unless it has explicitly been given the right to do so. You may think of each AddressSpace as a security domain. In fact, software at one security level may run simultaneously on the same processor as software at another security level, provided that it resides in a distinct AddressSpace.

However, because code running in one AddressSpace can transfer data, Objects, etc. to another AddressSpace (and vice-versa), when INTEGRITY Connections are placed between

AddressSpaces, it effectively brings both AddressSpaces to the same security level. Sharing of other kernel Objects between AddressSpaces, such as Semaphores, could also allow information to be communicated between AddressSpaces.

Typically, INTEGRITY users who want to place software of varying security levels onto the same processor do not allow any Object sharing between AddressSpaces at different security levels. Instead, any communication between AddressSpaces at different security levels is handled via a proprietary interprocess communication mechanism, such as an IODevice that implements a communication channel with encryption, labeled messaging, verification, etc.

Chapter 20

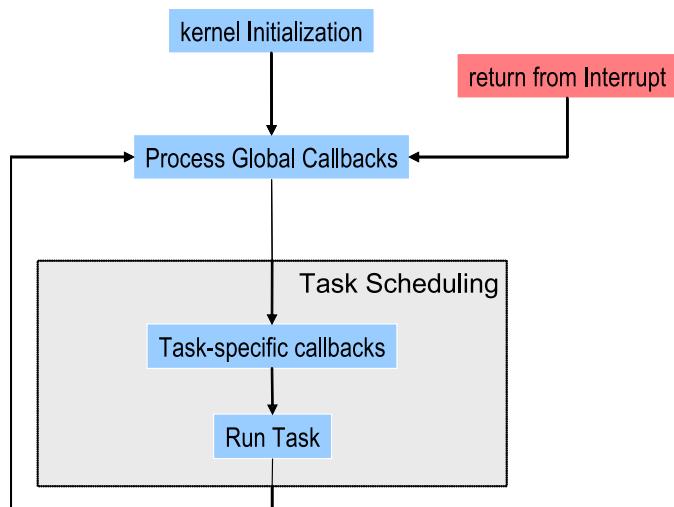
Synchronous Callbacks

This chapter contains:

- Using Callbacks with INTEGRITY
- Initializing a Callback
- Callback Handlers
- Placing a Callback
- Callbacks and KernelSpace Device Drivers
- Callbacks and IODevices
- Callbacks and Timers
- Callbacks and Message Queues
- Callback Restrictions

20.1 Using Callbacks with INTEGRITY

A Synchronous Callback (Callback) is a function that is called as a result of an exception but whose execution is delayed until the kernel is in a consistent state.



Callbacks run in “Synchronous callback context”, which is a distinct context with more constraints on which functions may be called than a Task context.

- INTEGRITY kernel calls support Callbacks only if they are non-blocking kernel calls that explicitly state that they support Callbacks.
- INTEGRITY library calls support Callbacks only if they explicitly state that they support Callbacks.
- In general, any function that starts with CALLBACK_* or INTERRUPT_* can be called from a callback. (Some functions beginning with CALLBACK_* can be used legally in other contexts as well. For example, CALLBACK_ReleaseSemaphore() can also be called from an IODeviceVector member function.)
- MULTI standard library functions support Callbacks only if they are safe to call from all INTEGRITY contexts. These functions are indicated with a “K” in the “Green Hills Standard C Library Functions” section in the “Libraries and Header Files” chapter of *MULTI: Building Applications*.

Because the INTEGRITY kernel typically does not block or mask interrupts to achieve mutual exclusion or perform atomic operations, an Exception Handler may be executing when critical kernel data structures are in an inconsistent state. By registering a Callback, the Exception Handler notifies the kernel to conclude its current processing and execute the Callback as soon as possible.

Callbacks are executed in supervisor mode and in KernelSpace, but not in a Task, which is why INTEGRITY API calls are not generally legal.

A Callback cannot predict what Task is executing at any particular time. As a result, Callbacks cannot access Task-specific data. Similarly, Callbacks cannot rely on a particular Task to be the currently running Task.

The execution time for a Callback is taken from the currently scheduled Task. Callbacks cannot be scheduled, and continue to execute until they are complete, although Callbacks can be interrupted by an exception. For this reason, Callbacks must not block. Callbacks can also be scheduled on the next tick of the Periodic Tick Timer, or they can be deferred until some other Callback is called. For more information, see the “Functions Available to the BSP” chapter in the *BSP User’s Guide*.

20.2 Initializing a Callback

Before a Callback can be used, it must be initialized. When the Callback is initialized, a callback function is associated with the CALL structure. This is done with the INTERRUPT_InitCall() function, which is usually called from the initialization routine of the driver or module using the callback. In the example below, the function ExampleCallbackFunction() is associated with the CALL structure DriverCallback. For simplicity, it is shown as part of BSP_InitializeDevices().

```
CALL DriverCallback;

BSP_InitializeDevices(void)
{
    ...
    INTERRUPT_InitCall(&DriverCallback, ExampleCallbackFunction,
                        "ExampleName");
    ...
}
```

20.3 Callback Handlers

The Callback contains a function that is executed whenever the Callback is triggered. The function is defined as follows:

```
typedef void (*CALLBACK) (Address TheParameter);
```

The function is called when the system runs the Callbacks that have been placed. Because a Callback is executed when the kernel has reached a consistent state, many operations are legal in synchronous callback context that are not legal in an interrupt context. However, for a variety of reasons, callbacks may not use most of the INTEGRITY API. Certain INTEGRITY API operations have analogous functions that are safe to be used from callbacks. For example, CALLBACK_ReleaseSemaphore() can be used to release a Counting Semaphore from a callback, and CALLBACK_RunTask() can be used to run a Task from a callback.

```
void ExampleCallbackFunction(Address arg)
{
    Error TheError;
    TheError = CALLBACK_ReleaseSemaphore(DeviceSemaphore);
    if (TheError != Success) {
        /* Set a flag and report the error later */
        DeviceError = TheError;
    }
}
```

The argument to the callback is set when the Callback is placed.

Because releasing a Counting Semaphore from an exception handler is so common, the functions INTERRUPT_InitReleaseSemaphoreCall() and INTERRUPT_ReleaseSemaphore() provide functionality similar to this example.

20.4 Placing a Callback

After a callback has been initialized, it can be triggered. This is typically done from an exception handler. The exception handler will place the callback in the kernel's list. A callback can be placed so that it is run as soon as the system has reached a consistent state, or a callback can be associated with a certain task, and executed when that task would be run.

The following functions are commonly used to place a callback. For a complete list of functions that may be used to place a callback, see the “Functions Available to the BSP” chapter in the *BSP User’s Guide*.

- INTERRUPT_PlaceSynchronousCall() — Causes the callback to occur.

The following example shows an exception handler placing a callback.

```
static EVENT MyExceptionHandler(XFRAME *unused, Address dummy)
{
    ... /* Exception Processing */

    INTERRUPT_PlaceSynchronousCall(&DriverCallback, (Address)0);

    return EVENT_HANDLED;
}
```

Note: If an exception handler is executed in rapid succession, the Callback will only be placed once. This means that there is not a one-to-one correspondence between calls to INTERRUPT_PlaceSynchronousCall() and execution of the Callback function.

20.5 Callbacks and KernelSpace Device Drivers

The following KernelSpace functions are used to create KernelSpace Device Drivers:

- INTERRUPT_InitCall() — Initializes a CALL structure to be used for callbacks.
- INTERRUPT_PlaceSynchronousCall() — Causes the callback to occur.
- INTERRUPT_InitReleaseSemaphoreCall() — Initializes a CALL structure to be used for a callback to release a semaphore.
- INTERRUPT_ReleaseSemaphore() — Causes the named semaphore to be released. It can only be used with a CALL structure that has been initialized via INTERRUPT_InitReleaseSemaphoreCall().

For detailed descriptions of these functions, see the “Functions Available to the BSP” chapter of the *INTEGRITY BSP User’s Guide*.

For example usage, see the “KernelSpace Device Driver Examples” and “Modifications to the KernelSpace Device Driver Example” sections in the “Device Drivers” chapter of the *INTEGRITY BSP Guide*.

20.6 Callbacks and IODevices

When INTERRUPT_IODeviceNotify() is called from a Task or from a callback, the message is sent on the IODevice immediately. When it is called from an IODevice member function or from an interrupt handler, INTERRUPT_IODeviceNotify() only attempts to place a synchronous call to a callback that can later send the message on the IODevice.

INTERRUPT_IODeviceNotifyWithTask() always attempts to place a synchronous call.

Because these functions sometimes use a callback internally, the number of INTERRUPT_IODeviceNotify() calls on an IODevice may be greater than the number of messages actually sent on that IODevice and notifications can be dropped. For example, if a

device causes two interrupts in quick succession, the interrupt handler may be called twice, but there may not have been time to run callbacks in between the two interrupts. If the interrupt handler called INTERRUPT_IODeviceNotify() twice, once in response to each interrupt, it may still be that only one additional message was sent on the IODevice. Similarly, if an IODevice member function calls INTERRUPT_IODeviceNotify() twice on the same IODevice before returning, only one message is sent. There is only one callback per IODevice for notifications.

20.7 Callbacks and Timers

Some timer and clock functions must be called in callback context. For more information, see the “Timers and Clocks” chapter of the *BSP User’s Guide*.

20.8 Callbacks and Message Queues

When using kernel callbacks with message queues, use “Interrupt-send message queues”. Interrupt-send message queues are used within KernelSpace only and are designed to allow sending from interrupt level or kernel callbacks, in addition to the normal usage from a task context. Interrupt-send message queues allow device drivers to use message queues to communicate with the tasks in an application, as an alternative to implementing an IODevice.

The functions INTERRUPT_AllocateMessageQueueBuffer() and INTERRUPT_ReserveMessageBuffer() functions require that the caller be in an interrupt handler or kernel callback.

In order for message queues to send interrupt level or kernel callbacks, they must be created with MESSAGE_QUEUE_INTERRUPT_SEND flag set. This flag enables sending from interrupt level or kernel callbacks, in addition to sending normally from a task context. However, it is incompatible with MESSAGE_QUEUE_SHARED and MESSAGE_QUEUE_DISTRIBUTED. (For more information about these flags, see “Creating Message Queues” in the *INTEGRITY Libraries and Utilities User’s Guide*.

20.9 Callback Restrictions

Callbacks can only be used with non-blocking kernel calls.

Callbacks can only be used with library functions if the function description explicitly states that they are supported. For more information, see the *Libraries and Utilities Reference Guide*.

20.9.1 ResourceManager and Callbacks

Unless otherwise specified, calls to ResourceManager functions cannot be called from kernel interrupt handlers or kernel Callbacks. They can only be made from a Task context.

For more information, see the “ResourceManager Reference” chapter of the *INTEGRITY Libraries and Utilities Reference Guide*.

20.9.2 Callbacks and Stack Size Configuration

The “Stack Configuration” section in the “Common Application Development Issues” chapter of this manual provides information about methods to modify the stack. However, the `.kstack` section of the KernelSpace program’s linker directives file, which specifies the size and location of the stack that is used for callback functions cannot be changed with these methods. Normally, the size of this section should not need modification.

20.9.3 Restrictions on BSPs and Drivers

Writing code for a BSP and device drivers differs from application code. There are some actions that a Task can take that driver and BSP code cannot. It is important to understand these restrictions to avoid problems with the BSP or driver. These restrictions apply to code running in a Synchronous Callback.

Note: Many of these restrictions can be avoided by moving the processing into a Task, but some synchronization must then occur.

20.9.3.1 Time Efficiency

BSP and driver functions invoked within kernel context are non-preemptible. All other kernel activity except for interrupts is inhibited while these functions are running. These functions must therefore take very little time to execute or the responsiveness of the entire system can suffer.

The INTEGRITY kernel is designed to be highly responsive to real-time needs. It has many preemption points to guarantee that it can run critical Tasks as soon as possible after they become ready to run. However, a single poorly-written BSP function invoked within kernel context can render the system unresponsive.

20.9.3.2 Dynamic Memory Allocation

BSPs and drivers may not perform dynamic memory allocation, such as by calling `malloc()`. The `malloc()` function protects its critical sections with a lock because multiple tasks may call `malloc()` concurrently. During initialization, this lock has not been created. In the other restricted execution contexts used by BSPs and drivers, acquiring the lock could lead to blocking. Furthermore, the lock code is designed to execute only in the context of a Task.

Dynamically allocating memory in restricted contexts is also contrary to INTEGRITY’s guarantee of resource availability. If the BSP or driver relies on dynamic memory allocation, nothing prevents an earlier driver or action from exhausting the heap and causing a failure in another critical driver.

20.9.3.3 Calling API Functions

Code executing as part of a kernel call context should not make calls to other kernel functions. As mentioned before, the INTEGRITY kernel is not designed to be re-entrant. If similar functionality is required, a callback can be registered to perform non-blocking kernel calls.

Drivers are usually designed so they do not attempt to call API functions from code in a kernel call context. Using an IODevice implementation is one way to avoid using such calls.

BSPs and drivers may not access `errno` using the default `errno` macro because that macro retrieves a pointer to a per-task `errno`.

BSPs and drivers may only access run-time library functions that are safe to call from all INTEGRITY contexts. These functions are indicated with a “K” in the “Green Hills Standard C Library Functions” section in the “Libraries and Header Files” chapter of *MULTI: Building Applications*.

20.9.3.4 Floating Point Registers and Other Extended Register Sets

Floating point registers and registers in other extended register sets (e.g., vector registers) may only be accessed in the context of a Task. Because of latency issues, use of these registers is not allowed in Interrupt Service Routines (ISRs), kernel calls, IODeviceVector member functions, TimerDriver member functions, callbacks, context switch hooks, or exception handlers.

By implication, functions defined with the `__interrupt` C/C++ language keyword (or defined using `#pragma ghs interrupt`) cannot be called in these contexts either because the compiler automatically emits instructions to save/restore the volatile registers within extended register sets for such functions.

Chapter 21

Developing Symmetric Multiprocessing (SMP) Applications

The Symmetric Multiprocessing (SMP) environment provides unique benefits and poses unique challenges not present in a uniprocessor environment. This chapter will help you successfully develop applications which work well in an SMP environment. The discussion is broken down into the following topics:

- Introduction to SMP
- Understanding Task Scheduling
- Managing Shared Data
- Debugging SMP Systems

This chapter provides information about developing SMP applications with INTEGRITY. For information about writing a BSP that supports SMP, see the “Memory Access Ordering” and “Shared-Memory Multiprocessor BSP Details” chapters in the *INTEGRITY BSP User’s Guide*.

21.1 Introduction to SMP

A multiprocessor or multicore system is a computer system with more than one logical processor. A symmetric multiprocessor (SMP) is a multiprocessor system with important properties: every processor in the system is identical, and every processor in the system has access to all the system resources. This symmetry allows the operating system to automatically and dynamically schedule tasks to any processor in the system, and move them from one processor to another in a fairly transparent fashion.

Migrating from a uniprocessor system to a multicore system provides a relatively straightforward method of increasing the processing power of a system without having to completely re-architect the application software for a multicore environment. In recent years SMP-capable multicore processors have proliferated in desktop and server computing. Multicore processors such as the Freescale QorIQ P2020 and the Intel Core 2 Duo are now becoming more common in higher-end embedded computer systems.

21.1.1 SMP and Shared Libraries

Shared libraries are not supported on SMP systems. For backward compatibility, integrate will recognize shared libraries and put a separate copy of any shared libraries in each AddressSpace, but the libraries will not actually be shared. At your earliest convenience, you should convert any projects running on SMP systems to use static libraries instead.

21.2 Understanding Task Scheduling

Note: This section describes the behavior of the current INTEGRITY scheduler for SMP. Future releases may provide new scheduling policies which may behave differently.

By default, all Tasks are eligible for execution on all processors. If there are P processors, the scheduler ensures that the highest-priority P Tasks are assigned to processors. For example, if there are four processors available, and 6 runnable Tasks with priorities 255, 255, 200, 150, 127 and 126, the scheduler will run the two priority 255 Tasks, the priority 200 Task and the priority 150 Task. This means that in an SMP system, priority by itself cannot be used to provide mutual exclusion. For more information, see the “Task Priority Does Not Provide Mutual Exclusion” section later in this chapter.

The scheduler also provides processor-affinity when it does not conflict with real-time strict priority scheduling requirements. When a Task becomes runnable it will be assigned to the last processor on which it ran, if its priority is higher than the priority of the Task currently running on the CPU. Otherwise it will find a lower-priority Task to replace on a different CPU. The goal of processor-affinity is to improve the cache locality of a Task by attempting to run it on a processor which likely has its working set in cache. Migrating to another processor will likely mean that the system caches are forced over time to migrate the Task’s working set, which can be an expensive prospect.

21.2.1 Kernel Lock Contention

At present, only one processor at a time in the system may be executing the INTEGRITY kernel. If processor A is currently running the kernel, a Task on processor B must wait for the Kernel Lock. If the Task running on processor B is of a higher priority than the Task that is causing processor A to run the kernel, as soon as A reaches a preemption point it will hand off the Kernel Lock to processor B. If B’s Task is not higher priority, then the Task on B will wait until A completes its in-kernel work.

To perform well in an SMP system an application should separate its Tasks into kernel-bound and compute-bound sets, if possible. This will prevent the compute-bound portions of the application from contending for the Kernel Lock. As a result, they will be able to execute faster, and the kernel-bound Tasks in the system will compete for access to the kernel with fewer Tasks.

21.2.2 Processor Binding

The default scheduler supports unbound Tasks and single processor bound Tasks. In some cases, Tasks may be desired to be bound to a subset of processors. This includes cases where subsets of processors may have some special affinity such as one subset being high performance processors

and another subset being high efficiency. The CPU subset scheduler supports binding Tasks to subsets that can include multiple processors. To enable the CPU subset scheduler, link **libsubsetsched.a** in to the kernel. When using the CPU subset schedulers, subsets must be defined by the BSP by implementing the optional `BSP_GetNumProcessorSubsetGroups()` and `BSP_GetProcessorSubsetGroup()` functions. See the *INTEGRITY BSP User's Guide* for information on implementing these functions.

The `SetTaskProcessorBinding()` and `SetTaskProcessorSubsetBinding()` API can be used to override the default any-processor scheduling behavior for a Task. Once bound to one or more processors, a Task will execute whenever it is the highest-priority runnable Task for that CPU. Non-bound Tasks are still eligible to run on a CPU which has Tasks bound to it. When a bound Task and a globally schedulable Task of the same priority are being considered to run on a CPU, the bound Task takes precedence. When a bound Task of a smaller subset and a bound Task of a larger subset of the same priority are being considered to run on a CPU, the Task with a smaller subset takes precedence. If their priorities differ, the selection is made strictly by priority.

This mechanism may be useful for inter-communicating Tasks which spend the majority of their time doing interprocess communication (IPC) rather than computation. Binding them all to a single CPU ensures that they will have cache locality with respect to one another and will further reduce kernel lock contention.

21.3 Managing Shared Data

Management of shared data in threaded software in an SMP system presents many of the same challenges as in a uniprocessor environment. But there are some unique issues that arise in a truly concurrent environment.

Note: We strongly recommend using system-provided locking facilities (LocalMutex, Semaphore, pthread_mutex_t, etc.) to provide mutual exclusion to shared data that resides in normal memory. These have been designed and implemented to provide atomic, coherent operation.

System-provided locking facilities, such as LocalMutexes, provide the proper semantics to ensure a consistent global view of any data structure residing in normal memory that is always read or modified while holding a lock on its contents. When using home-grown locking primitives, it is important to ensure that they provide the correct memory barriers to ensure a consistent global view of the data. If you do not, or cannot, use system-provided locking mechanisms, you must read the “Memory Access Ordering” chapter of the *INTEGRITY BSP User’s Guide* for information about how to prevent memory access reordering problems.

21.3.1 Task Priority Does Not Provide Mutual Exclusion

For SMP, Task execution is concurrent. With more than one CPU in the system, Tasks of different priority levels may execute concurrently. Design patterns that rely on using Task priority to maintain mutually-exclusive access to shared data, with the assumption that a low-priority Task cannot execute while a high-priority Task is running, will not work under SMP without modification. There are several strategies to overcome this when migrating an existing design using this pattern to SMP:

- Redesign so that locks are used instead of priority to provide exclusion.
- If using Highest Locker Semaphores, use WaitForHighestLockerSemaphore() (which can block) instead of using TryToObtainSemaphore().
- Use SetTaskProcessorBinding() to limit execution of cooperating Tasks to a single CPU.

21.3.2 Shared Memory Consistency

Note: A complete consideration of the theory of memory ordering in a multiprocessor system and the need for and proper use of memory barriers is beyond the scope of this document. If the reader is not completely familiar with these concepts, we strongly recommend the use of system-provided locking facilities such as LocalMutexes and that all shared data reside in normal memory and be read and written only while holding the lock on that data.

When sharing access to data structures, it is critical that Tasks observe a consistent state of those data structures, regardless of which processor in the system they are running on. While SMP hardware systems can generally be expected to provide cache coherency, it is much less common for them to provide the appearance of global, strongly-ordered memory across processors.

Modern processor architectures such as the Power architecture and the Intel IA-32/Intel 64 architecture generally guarantee that a given processor observes the effects of its own memory

accesses in the order they appear in a program. But the ordering seen by a different processor in the system may be drastically different. The memory ordering model provided by the Intel architecture is much stronger than that provided by the Power architecture, but neither provides total consistency.

Thus, it is important that the application make use of some sort of memory barrier operations to ensure that stale data is not used. The easiest way to do that is to use the system-provided locking facilities, such as LocalMutexes, for mutual exclusion. These locks provide the proper acquire and release memory barrier semantics to ensure that as long as a given data structure is always read or modified while holding a lock on that data, the view of the data will be coherent and consistent.

A common half-way pattern that code might employ in a uniprocessor is to hold a lock while modifying the data, but not while reading the same data structure. On an SMP system, this pattern generally requires using memory barriers in both the reader and the writer to ensure that the reader observes changes to the state of the validity and other properties of the data before observing the data it qualifies.

For a concrete example, consider the following uniprocessor code fragment, which is not sufficient for SMP:

```
#include <INTEGRITY.h>
#include <INTEGRITY_arch.h>

#define MAX_ALLOC      512
typedef struct AllocationSlotStruct {
    Boolean      Valid;
    Value       Data;
} AllocationSlot;

typedef struct AllocatorStruct {
    LocalMutex          Lock;
    volatile AllocationSlot   Slots[MAX_ALLOC];
} Allocator;

/* InitAllocator() must be called before any other use is made of      */
/* *TheAllocator.                                                       */
Error InitAllocator(Allocator *TheAllocator)
{
    Error E;
    int i;

    E = CreateLocalMutex(&TheAllocator->Lock);
    if (E != Success)
        return E;

    for (i = 0; i < MAX_ALLOC; i++) {
        TheAllocator->Slots[i].Valid = false;
    }

    return Success;
}

/* Writer. Use a lock to allocate a slot.                                */

```

```

Value AllocateSlot(Allocator *TheAllocator, Value Data)
{
    int i;
    Value ReturnIdx = ~(Value)0;

    CheckSuccess(WaitForLocalMutex(TheAllocator->Lock));
    for (i = 0; i < MAX_ALLOC; i++) {
        if (!TheAllocator->Slots[i].Valid) {
            ReturnIdx = i;
            /* Write the data before setting the valid indicator, to */
            /* ensure that the lockless reader sees the correct data */
            /* when it finds a Valid slot. */
            TheAllocator->Slots[i].Data = Data;
            TheAllocator->Slots[i].Valid = true;
            break;
        }
    }
    CheckSuccess(ReleaseLocalMutex(TheAllocator->Lock));

    return ReturnIdx;
}

/* Reader. Locklessly index a slot. */
Boolean LookupSlot(Allocator *TheAllocator, Value IIdx, Value *Data)
{
    if (TheAllocator->Slots[Idx].Valid) {
        *Data = TheAllocator->Slots[Idx].Data;
        return true;
    } else {
        return false;
    }
}

```

This is a simple writer-locked, reader-lock-free allocator one could use for allocating indexed objects such as file descriptors. As written, it has two key implicit ordering constraints which an SMP system may violate without explicit memory barriers:

- In `AllocateSlot()`, it is critical that the `Data` of the slot be written before `Valid` is set to `true`. On a uniprocessor, the `volatile` keyword in the declaration of `Slots[]` is sufficient to ensure this will occur. On a multiprocessor, the stores must be explicitly ordered.
- In `LookupSlot()`, it is critical that the `Data` of the slot be read only after `Valid` is seen to be `true`. Otherwise, a race condition could result in reading stale `Data` from a `Valid` slot.

The race condition in `LookupSlot()` can occur because the processor may perform the loads in an order other than the program order. As a result, the following execution — with Processor A executing `AllocateSlot()` and Processor B executing `LookupSlot()` — is possible:

Processor A	Processor B
-	Load Data
Store Data	-
Store Valid	-
-	Load Valid

In this case, even though Processor A performed its stores in program order, the load of `Data` in `LookupSlot()` was reordered by the hardware such that it preceded the load of `Valid`.

The solution here is to add explicit memory barriers to the code to ensure that the memory accesses occur in explicit order. (However, simply acquiring the Allocator Lock in `LookupSlot()` would avoid the need for explicit barriers at the cost of increased contention.) You can accomplish this in a straightforward and portable manner using the `ASP_SMPBarrier()` API, which is documented in the “ASP Function Calls” chapter of the *INTEGRITY Kernel Reference Guide*.

To ensure that `AllocateSlot()` performs its stores in the proper order, add a STORE-STORE barrier:

```
/* Write the data before setting the valid indicator, to */
/* ensure that the lockless reader sees the correct data */
/* when it finds a Valid slot. */
TheAllocator->Slots[i].Data = Data;
ASP_SMPBarrier(ASP_BARRIER_STORE, ASP_BARRIER_STORE);
TheAllocator->Slots[i].Valid = true;
```

Likewise, to ensure that `LookupSlot()` performs its loads in the required order, add a LOAD-LOAD barrier:

```
if (TheAllocator->Slots[Idx].Valid) {
    ASP_SMPBarrier(ASP_BARRIER_LOAD, ASP_BARRIER_LOAD);
    *Data = TheAllocator->Slots[Idx].Data;
    return true;
} else {
    return false;
}
```

These two changes guarantee that if `LookupSlot()` sees a `Valid` slot, it will return the correct data.

21.3.3 SMP Atomic Operations

In a threaded uniprocessor environment, when modifying shared data without holding a lock, it is important that those updates are atomic with respect to other threads. This is especially important in an SMP environment because the real-world likelihood of another thread observing the intermediate results of a non-atomic update increases dramatically. For INTEGRITY SMP, as long as your application is already using the Atomic family of functions (`AtomicModify()`, `TestAndSet()`, etc.), you do not need to do anything else to make your accesses atomic.

It is worth noting that atomic operations are not defined to provide any implicit memory barrier effects. If you are implementing your own locking system using atomic operations, it is important that you provide appropriate acquire and release barrier semantics. An acquire barrier, when used in conjunction with an atomic operation, generally takes the following form:

```
ASP_SMPBarrier(ASP_BARRIER_IMPORT_ATOMIC_LOAD, ASP_BARRIER_LOADSTORE);
```

A release barrier generally takes the following form:

```
ASP_SMPBarrier(ASP_BARRIER_LOADSTORE, ASP_BARRIER_EXPORT_STORE);
```

- In the acquire case, this ensures that all accesses to normal memory that follow the lock acquisition in program order will actually follow the acquisition of the lock (or, more precisely, the stage in lock acquisition at which the lock was observed to be unowned).
- In the release case, this ensures that all accesses to normal memory that precede the lock release in program order will actually precede the release of the lock.
- Together, these ensure that data in normal memory always accessed under the same lock will appear consistent no matter which processor performs the access.

Note: The system-provided LocalMutexes are lightweight and fast for intra-AddressSpace locking, and provide the correct barrier semantics for accesses to normal memory. All other system-provided locking primitives also supply the correct barriers for accesses to normal memory.

Note: System-provided locking primitives are only guaranteed to order accesses to normal memory. When locking access to memory with the MEMORY_WT, MEMORY_UNCACHEABLE, or MEMORY_VOLATILE attributes, the user is responsible for providing a barrier to order accesses to the shared memory after the lock acquire and for providing a barrier to order accesses to the shared memory before the lock release. The general acquire barrier might take the form:

```
ASP_SMPBarrier(ASP_BARRIER_IMPORT_ATOMIC_LOAD, ASP_BARRIER_ALL);
```

The general release barrier might take the form:

```
ASP_SMPBarrier(ASP_BARRIER_ALL, ASP_BARRIER_EXPORT_STORE);
```

Because the platform may actually provide the barriers you need implicitly, the ASP_Barrier() API has separate operation flags for Atomic Loads and Atomic Stores. This allows the API to skip barrier instructions which are not required, given that the preceding or following memory operation is atomic. For example, the Intel x86 and x64 architectures provides a full memory barrier on all atomic read-modify-write instructions. The following contrived example code, when compiled for an x86 or x64 system, does not generate any code for any of the ASP_SMPBarrier() invocations.

```
void UnsensibleExampleFunction(Value *P, Value *A)
{
    Value old_A;
    Value *P;

    ASP_SMPBarrier(ASP_BARRIER_LOADSTORE, ASP_BARRIER_ATOMIC_LOAD);
    AtomicModify(A, &old_A, 0, 1);
    ASP_SMPBarrier(ASP_BARRIER_ATOMIC_STORE, ASP_BARRIER_LOADSTORE);
    P[0] = old_A;
    ASP_SMPBarrier(ASP_BARRIER_STORE, ASP_BARRIER_ATOMIC_STORE);
    TestAndSet(A, P[0], old_A);
}
```

For the sake of portability, we recommend that you add the necessary barriers to your application code even if you know that at present they are not needed.

21.4 Debugging SMP Systems

Debugging race conditions and timing-dependent behavior in a concurrent system can be difficult. The best approach is to use system-provided mutual exclusion or IPC primitives when sharing data between Tasks, which safely decouples their execution.

21.4.1 SMP Debugging Tools

The majority of the tools that work for debugging and tuning a uniprocessor system are also at your disposal for debugging and tuning SMP systems. However, they do present some limitations, which are discussed in this and the following section.

For an SMP system, the MULTI EventAnalyzer remains a powerful tool for understanding the behavior of a system over time, and can be very helpful in diagnosing race conditions and performance issues. However, interrupt logging is not supported and enabling interrupt logging may result in lost or corrupted data (as explained in the following section).

Profiling is also a useful tool for understanding the performance of your system and learning how to tune your applications.

21.4.2 SMP Run-mode Debugging Limitations

Run-mode debugging is supported. However, there are some limitations in the current implementation.

On an SMP target, when breakpoints are set in an AddressSpace with more than one Task, Tasks in that AddressSpace may be transiently bound to a single CPU in a manner that overrides the bindings set by the user through the INTEGRITY kernel API.

Run-mode event logging does not handle interrupt logging correctly for SMP and is not supported. Enabling interrupt logging may result in lost or corrupted data. Increasing the buffer size may mitigate these effects.

Debugging DLLs shared between multiple AddressSpaces is not supported.

System Halt is not supported.

Group breakpoints are not supported.

Chapter 22

Troubleshooting

This chapter lists problems and errors, possible causes, and some suggestions for about how to handle them. The following troubleshooting topics are covered:

- Running a Dynamically Downloadable Image
- Dynamic Download from Debugger Failed
- Debugging Problems Downloading with the Virtual Loader
- MULTI / rtserver Loss of Communication to the Board
- Undefined Symbol main()
- INTEGRITY Violation before main()
- No Kernel Banner
- Not Enough Target RAM Memory Allocation Errors
- Cannot Find Target File Error
- Task Defined in Integrate Configuration File Cannot Exit
- MULTI Project Manager Reports Incorrect Component Types for Legacy Projects
- Checksum Warning
- Stack Overflow
- CheckSuccess at Boot Time
- CheckSuccess: CannotRaiseTaskInterruptPriorityLevelBelowCurrentLevel
- Performance Bottlenecks

22.1 Running a Dynamically Downloadable Image

Problem:

After downloading your image, the program runs for a few instructions and then dies or jumps into random memory.

Possible Cause:

- An image that was built to be dynamically downloaded was loaded to the board RAM directly instead of downloaded to a running kernel. Consult the “Dynamic Download INTEGRITY Application Project” section in the “Building INTEGRITY Applications” chapter for more information about how to build an image appropriate for dynamic downloading.

Solutions:

- If dynamic download was intended, follow the steps in the “Dynamic Download INTEGRITY Application Project” section of the “Building INTEGRITY Applications” chapter which describes how to use MULTI to download the application.
- If it was intended to have the code included in KernelSpace, include the code in the KernelSpace program, then rebuild and load it.

22.2 Dynamic Download from Debugger Failed

Problem:

The attempt to download an image to the running target failed. An error message was displayed in the MULTI Target pane and/or on the INTEGRITY console.

Possible Causes:

- One common error is accidentally loading the virtual AddressSpace program instead of the final download image. The error indicating that a bad BootTable was detected will display. The final download image is run through Integrate and therefore has a valid BootTable. The stand-alone virtual AddressSpace program will not have a BootTable. An easy way to tell the difference is to look at the type of the **.gpj** file. The stand-alone virtual AddressSpace program will have type **Program** and the Dynamic Download image will have type **INTEGRITY Application**.
- Another common problem is running out of memory. If too many images are loaded for the available memory, the download will fail. Kernel memory pages are used for the new AddressSpaces’ code, data, etc. Some amount of heap is also used for accounting purposes.
- Another problem is loading an image that is too large for the available download area. For more information, consult the “Dynamic Downloading” chapter of this manual.

Solutions:

- If the error `Memory allocation failure in loader` or similar is displayed on the console, it indicates heap exhaustion. This can be corrected by either unloading some images already in the system or by increasing the kernel’s `.heap` section size and restarting.
- If this error message is not displayed, it indicates that available RAM has been exhausted, and as a result no new AddressSpaces can be created. This can be corrected by unloading images, or increasing RAM.

22.3 Debugging Problems Downloading with the Virtual Loader

Problem

Something is failing to download. “Download Failed” prints to the rtsserv2 Target pane.

Possible Cause #1

- The download region is not large enough for the image. The message “Load image too large for .download section” will be printed to the console, along with the image and .download sizes.

Solution #1

- Make sure the download region, which is set by __INTEGRITY_DownloadSize, is large enough. It should not have to be much bigger than the size of the image being downloaded.

Possible Cause #2

- The MemoryPool is not large enough. The download will fail with the message “Not enough physical memory for new AddressSpace” printed to the console.

Solution #2

- Make sure you have enough MemoryPool. For more information, see “MemoryPool Configuration” in the “Common Application Development Issues” chapter.

22.4 MULTI / rtsserv2 Loss of Communication to the Board

Problem:

Zero or sluggish response from the Debugger. If the Debugger is attempting to get information from the target and receives no response for an extended period of time (**SERVERTIMEOUT** variable), MULTI will ask the user if the connection should be terminated.

Possible Causes:

- An overloaded network can cause the debug protocol to lose packets. It should be able to recover after some delay, unless it is continually being bombarded.
- If freeze-mode debugging is occurring simultaneously, halting the kernel (for example, by hitting breakpoints, etc.) can cause delays in getting the rtsserv2 connection back online after the system has resumed. Try not to cause rtsserv2 events until the system is running for maximal debugging efficiency.
- If the problem persists, it may indicate something more serious.

Solution:

- Give the system a minute or so to recover. If the problem persists, reboot may be required.

22.5 Undefined Symbol main()

Problem:

Linker produces error indicating that main() is referenced but not defined.

Possible Cause:

- No main() function was specified by the user in a virtual AddressSpace project. Each virtual AddressSpace has an Initial Task which has a user-code entry point at main().

Solution:

- Include a main() function in the virtual AddressSpace program. Even if the virtual AddressSpace has other BootTable tasks that contain all the useful user code, include at least a stub main. The kernel will not allow other BootTable tasks to run until the Initial Task has initialized the AddressSpace appropriately.

22.6 INTEGRITY Violation Before main()

Problem:

The Initial Task sets up the AddressSpace before jumping to main(). Sometimes the task can encounter an INTEGRITY violation before reaching user code. The task will be marked as SUSPENDED and its priority will be 255 (or the AddressSpace Maximum Priority).

Possible Cause #1:

- The C or POSIX library setup has run out of memory. An error message should display on the serial console or rtserve2 I/O pane, possibly referencing “error 11” which is the errno EAGAIN.

Solution #1:

- Increase the size of the .heap section

Possible Cause #2

- The initialization code has run out of stack. See “Stack Overflow” later in this chapter.

Solution #2

- Increase the size of the Initial Task’s stack in the .int file.

Possible Cause #3:

- The program was not compiled for the correct chip. If the exception message is “Illegal Instruction” this is most likely the case. If the program was compiled for an AltiVec chip, the Initial Task will get an exception that it is trying to set the AltiVec state (even if the program does not use AltiVec instructions).

Solution #3

- Look at the build structure and make sure the -bsp option is correct.

22.7 No Kernel Banner

Problem:

When booting the kernel, the kernel banner does not appear.

Possible Cause:

- A memory region cannot be allocated by the BSP because the memory footprint of your application is too large. If your image has consumed all of RAM, run-time allocations such as BSP memory reservations will not be available, and the kernel will panic at startup rather than boot into an inconsistent state.

Solution:

- Decrease the memory footprint of your application.

22.8 Not Enough Target RAM Memory Allocation Errors

Problem:

When Building an INTEGRITY application, integrate issues warnings indicating that a MemoryRegion cannot be allocated and that there is not enough RAM for the target. Errors similar to the following are displayed:

```
intex: error: Failed to allocate memory request MemRegion0: size 0x80000000
intex: error: Not enough target RAM for request.
intex: error: Ran out of memory in AddressSpace KernelSpace when allocating
section MemRegion0
```

Possible Cause:

- The memory being requested is not in the range of memory declared in the .bsp file.

Solution:

- Check the .bsp file in use. The memory declared with the MinimumAddress/MaximumAddress keywords must match your board's memory. If it does not, modify these keywords as needed. If the memory declared in the .bsp file does match the board, you must modify your application to use less memory.

22.9 Cannot Find Target File Error

Problem:

An error similar to the following is displayed during a build:

```
Cannot find target file: c:\ghs\int40\ppc\default.bsp
intex: fatal: Integrate failed.
```

Possible Cause:

- This usually indicates that the proper MULTI Project Manager Target has not been chosen for the project.

Solution:

- See “target Directory” in the “INTEGRITY Installation and Directory Structure” chapter for more information about setting the build target.

22.10 Task Defined in Integrate Configuration File Cannot Exit

Problem:

If running in a virtual AddressSpace, the task may be running in the **INTERRUPT_Panic** routine due to a failure in the exit() or other function.

Possible Cause:

- No Object entry for the task specified in the Integrate configuration file. For example, the following is not sufficient to create a task:

```
AddressSpace      myaspace
  Language        C
  Task            mytask
    Priority      100
    EntryPoint    mytask_func
  EndTask
EndAddressSpace
```

Solution:

- Add an Object entry for the task specified explicitly in the Integrate configuration file. For example:

```
AddressSpace      myaspace
  Language        C
  Task            mytask
    Priority      100
    EntryPoint    mytask_func
  EndTask
  Object          10
    Task mytask
  EndObject
EndAddressSpace
```

- Alternatively, you may specify:

```
AddressSpace      myaspace
  Language        C
  Object          10
    Task mytask
    Priority      100
    EntryPoint    mytask_func
  EndObject
EndAddressSpace
```

22.11 MULTI Project Manager Reports Incorrect Component Types for Legacy Projects

Problem:

The MULTI Project Manager reports incorrect component types for an INTEGRITY project that was created with an earlier version of MULTI.

Possible Cause:

- The MULTI Project Manager usually auto-detects components in projects created with versions of MULTI that did not have the Project Manager. However, there are situations where a component will not be auto-detected. For example, if a legacy INTEGRITY Monolith does not include an **.int** file, and **kernel.gpj** is not a child of the Monolith, the Project Manager may incorrectly identify the project as a Dynamic Download.

Solution:

- Add the missing component to the **.gpj** file by adding `#component component_name` after the `#!gbuild` line.

To find `component_name`, use the New Project Wizard to make a project of the desired type, and copy the appropriate `#component` line from the **.gpj** file. For example, in the case of a Monolith being mistakenly identified as a Dynamic Download, you would add the following line to the project's **.gpj** file:

```
#component integrity_monolith
```

22.12 Checksum Warning

Problem:

When Building an INTEGRITY Application with checksum enabled, a warning is issued by Integrate.

Possible Cause:

- The application was built with the **-checksum** option enabled in a virtual AddressSpace.

Solution:

- Enable the **-checksum** option in the kernel only, not in virtual AddressSpaces. For more information, see “Invoking CRC on the Kernel and Virtual AddressSpaces” in the “Building INTEGRITY Applications” chapter of this manual.

22.13 Stack Overflow

Problem:

When a task encounters an INTEGRITY violation and the task's current program counter (as viewed in the MULTI Debugger, in interleaved assembly view) references the first instruction in the current function to store into the stack, this is often the symptom of a stack overflow. For example, on the PowerPC, often one of the first instructions in the function looks like the following:

```
stwu    sp, -8(sp)
```

Possible Cause:

- This instruction simultaneously allocates the stack for the current function and stores the previous stack pointer at the bottom of this newly allocated area. This is often the instruction that causes the memory access exception when a stack overflow occurs.

Solution:

- Use a larger stack for the faulting task.

22.14 CheckSuccess at Boot Time

Problem:

The setup code fails when initializing. Generally a message such as `CheckSuccess failed with error n` will be printed to the serial port to indicate the failure.

Possible Cause #1:

- BSP Configuration file is not in sync with system.

Solution #1:

- Edit `InitialKernelObjects` in the `.bsp` file.

The `bspname\default.bsp` file (or custom file if included in project) contains a keyword `InitialKernelObjects` that needs to be greater than the `Initial Objects` value that gets printed out in the kernel banner.

For example, if `default.bsp` has `InitialKernelObject` set to 17 and an Object is added at index 18, but `Initial Objects` is 20, the following error will be generated:

```
Kernel.....INTEGRITY v11.0.0
BSP.....ISIM - INTEGRITY Simulator for PowerPC
IP Address.....unknown
RAM.....8 MB
Initial Objects.....20
```

```
Error: Objects 1 through 20 are reserved by the system.  
Integrate specified object 18 cannot be created.  
Increase the InitialKernelObjects keyword in the .bsp file  
to 21 and the ObjectIndex 18 specified in the .int file.  
CheckSuccess failed with error 7
```

Possible Cause #2:

- BSP Memory Table is not in sync with memory requested by image.

Solution #2

- The **bspname\default.bsp** file (or custom file if included in project) contains keyword pairs MinimumAddress/MaximumAddress to describe memory that can be used for making MemoryRegions requested by the **.int** file. If the requested memory is not included in the MemoryTable defined by the BSP, then it will fail. Failure codes for this case include:
 - SegmentIsNotInAnyMemoryRegion
 - SegmentAddressNotAvailable
- When mapping virtual memory to physical memory, the virtual attributes must match or be more strict than the physical attributes. The physical attributes are defined in the memory table and the virtual attributes are defined in the **.int** file. Failure codes for this case include:
 - PhysicalMemoryRegionHasLessPermissions
 - PhysicalMemoryRegionIsVolatile
 - PhysicalMemoryRegionIsVolatileVirtualMemoryRegionIsNot
 - PhysicalMemoryRegion IsNotMappable
 - CannotMapVirtualMemoryRegionAddress
 - CannotMapToPhysicalMemoryRegionAddress
 - ImplementationRestrictionOnAttributes
 - SegmentsMustBothBePagedOrBothNotPaged
 - IOSpaceNotAllowedOnAVirtualMemoryRegion
 - VirtualZeroPageIllegalInNonPagedSegments

Possible Cause #3:

- Not enough memory in system to satisfy image requirements

Solution #3:

- The system does not contain enough memory to load the image and create all its Objects. Double check to make sure that all available board memory is referenced in the memory table. Failure codes for this case include:

- MemoryAllocationFailed
- MemoryPoolExhausted
- NotEnoughPagesOnAddressSpaceFreeList
- ObjectTableIsFull
- NotEnoughPagesForMappingTables

22.15 CheckSuccess: CannotRaiseTaskInterruptPriorityLevelBelow-CurrentLevel

Problem:

A CheckSuccess message (as in the previous section) fails with Error CannotRaiseTaskInterruptPriorityLevelBelowCurrentLevel.

Possible Cause:

- A Task is calling into the C library from an Activity interrupt handler for an Activity with an interrupt priority level (IPL) that exceeds the C library Task IPL.

Solution:

- Use the SET_C_LIBRARY_TASK_IPL macro to specify a higher C library Task IPL.

The default is 1, meaning that a Task runs at IPL 1 while in the C library, so any Activity interrupt can preempt the C library. Then, for safety, any C library call from an Activity interrupt handler will CheckSuccess, which makes it impossible for it to re-enter the C library.

In order for the interrupt handler of an Activity with priority n to be able to call into the C library safely, the C library Task IPL must be at least n .

22.16 Performance Bottlenecks

Problem:

A performance bottleneck was found using MULTI tools. The following solutions may help you fix the problem.

Solution #1

- Adjust Compiler Optimizations — you can adjust the compiler optimizations in your application. For descriptions of the various optimization settings that can be applied and information about how to use them, see “Optimizing Your Programs” in the *MULTI: Building Applications* manual.

Note: Do not modify the compiler optimization in any **default.gpj** that ships with INTEGRITY. Changing MULTI Optimization settings from the defaults shipped in INTEGRITY build files may alter the code behavior in ways that are untested and unsupported by Green Hills Software.

Solution #2

- Move Code Into KernelSpace — you can move code into KernelSpace. However, this method trades off reliability for performance. Running code in KernelSpace is faster than code running in a virtual AddressSpace, however code running KernelSpace has complete access to all of the code and data in the entire system, including the kernel's. As a result, a bug in code that runs in KernelSpace could take down the entire system. This method is only appropriate when the performance of some portion of the code is time critical and it is well trusted.

Index

- About This Manual, **13**, **13**
- Access Beyond Allocated Area, **255**, **256**
- Activity list, OSA Explorer, **218**
- Add Virtual AddressSpace dialog, **126**, **126**
- Adding and Configuring Virtual AddressSpaces with the Project Manager, **78**, **109**, **112**, **117**, **118**, **125**, **126**
- Adding Items to INTEGRITY Projects, **107**, **125**
- Adding OS Modules with the Project Manager, **112**, **118**, **125**, **126**
- Additional Options for Building Applications, **67**, **99**
- AddressSpace Profiling, **202**, **202**
- Advanced Core Dumping, **269**, **271**
- Analyzing Trace Data with the TimeMachine Tool Suite, **66**, **66**
- Application Core Dumps, **269**, **270**, **270**, **271**
- Application Development Restrictions, **133**, **153**
- Application Memory Configuration, **133**, **140**
- application types, **67**
- application-level debug server, **63**
- application-level debugging, **171**, **183**, **183**
- Array Bounds, **249**, **251**, **252**, **252**
- Assignment Bounds, **251**, **251**
- Automatic Stack Overflow Detection, **258**, **258**
- Automatically Created Tasks, **133**, **138**
- Available Shared Libraries, **95**
- Bad Free, **255**, **257**, **257**, **260**
- bin Directory, **51**, **53**, **59**
- Binary Installation, **51**, **51**–**53**
- Block Coverage Profiling, **201**, **204**, **204**
- board setup, **48**, **48**
- Booting an INTEGRITY Kernel, **19**, **20**, **49**
- breakpoint limitations, **192**
- breakpoint types, **192**
- breakpoints, overlapping, **192**
- BSP, **55**
- BSP Source Directory, **51**, **53**, **55**, **55**, **56**, **71**
- BSP User’s Guide, **15**, **53**, **175**, **284**, **285**, **287**
- bspname, **30**, **38**, **48**, **49**, **52**–**54**, **55**, **55**–**59**, **71**, **83**, **105**, **131**, **165**, **167**, **168**, **172**, **175**, **198**, **308**, **309**
- BSPs, **55**
- Building a Dynamic Download INTEGRITY Application Project, **76**
- Building a KernelSpace Project, **82**
- Building a Monolith INTEGRITY Application Project, **74**
- Building in Run-Mode Debugging Support, **171**, **173**
- Building Installed BSPs, **19**, **25**, **29**, **37**
- Building INTEGRITY Applications, **40**, **49**, **53**, **56**, **67**, **67**, **99**, **104**, **105**, **108**, **115**, **119**, **125**, **140**, **142**, **152**, **153**, **168**, **195**, **198**, **255**, **302**, **307**
- Call Count Data, **201**, **207**, **207**–**209**
- call count profiling, **207**, **207**, **209**, **211**
- call graph profiling, **207**
- Callback Handlers, **283**, **285**
- Callback Restrictions, **283**, **287**
- callback, initialize, **284**
- callback, place, **285**
- Callbacks and IODevices, **283**, **286**
- Callbacks and KernelSpace Device Drivers, **283**, **286**
- Callbacks and Message Queues, **283**, **287**
- Callbacks and Stack Size Configuration, **288**
- Callbacks and Timers, **283**, **287**
- Calling API Functions, **288**
- Cannot Find Target File Error, **301**, **305**

- Case Label Bounds, 251, **252**, 252
 CheckSuccess at Boot Time, 301, **308**
 CheckSuccess: CannotRaiseTaskInterrupt-PriorityLevelBelowCurrentLevel, 301, **310**
 Checksum Warning, 301, **307**
 checksum, SHA-1, **99**
 Clearing Overruns and Handling Multiple Events Simultaneously, 133, **155**
 Clock list, OSA Explorer, **221**
 Close the Pizza Demo, **47**
 Code Coverage, **204**, 206
 Collecting and Viewing Profiling Data, 65, **202**, 202
 command-line driver, 99, **104**, 104, 153
 Command-Line Driver Examples, **105**
 Command-Line Driver Options, **104**, 104
 Common Application Development Issues, 87, **133**, 211, 267, 288, 303
 Common Features of Object View Windows, **225**
 common library source, **55**, 55, 56, 59, 87
 Common Library Source Directories, **55**, 59, 87
 Communication Media, **172**
 Compiler, Toolchain, and Object File Utilities, 61, **64**
 configuration file, integrate, **84**
 Configure Virtual AddressSpace‘, **126**
 Configuring Applications with the MULTI Project Manager, 20, 72, 75, 78, 81, **107**
 Configuring Dynamic Download Projects with the NPW, 75, 107, **115**
 Configuring KernelSpace Projects with the NPW, 81, 107, **119**
 Configuring Monolith Projects with the NPW, 72, 107, **108**
 Connecting to Multiple ISIM Targets, 163, **180**
 Connecting to rtserver, 57, 171, **174**, 195
 Connecting to rtserver Using a Custom Connection Command-Line, 174, **175**, **179**
 Connecting to rtserver Using the MULTI Connection Organizer, 174, **175**
 Connecting to Your Target, 166, **167**, **174**, 174
 Connecting with INDRT2 (rtserver2), 64, 133, 163, 166, **171**, 195, 201, 273, 280
 Connection Editor, 57, 167, 174, **175**, 175–177, 180
 Connection list, OSA Explorer, **217**
 Connection Organizer, 22, 166, 168, 174, **175**, 175, 178
 console redirection over host I/O, **170**
 Controlling Target Settings, **188**
 Conventions Used in Green Hills Documentation, **16**
 Converting a Dynamic Download to a Monolith, 113, **119**
 Converting a KernelSpace Project to a Monolith, **123**
 Converting a Monolith to a Dynamic Download, **113**, 119
 Converting and Upgrading Projects with the MULTI Project Manager, 107, **131**
 Copying Examples with the MULTI Project Manager, 20, 107, **128**
 core dumps, debugging, **271**
 Core File Debugging, **269**, 269
 CRC, **99**, 99, 100
 CRC, troubleshooting, **307**
 Creating a Dynamic Download Application with the New Project Wizard, 75, 115
 Creating a KernelSpace Project from Scratch, **83**
 Creating a KernelSpace Project with the New Project Wizard, **79**, 119
 Creating a Monolith INTEGRITY Project with the New Project Wizard, **72**, 108
 Creating an INTEGRITY Top Project, 67, **68**, 72, 75, 79
 Creating and Destroying INTEGRITY Tasks, **133**, 133
 Creating Custom Linker Directives Files, **87**
 Creating MULTI Workspaces for Installed BSPs, 19, **23**, 27, 29, 174
 Creating Tasks Dynamically via INTEGRITY API, 133, **134**

Creating Tasks Dynamically via POSIX API, 133, **135**
Creating Tasks Statically via Integrate, 133, **135**

data consistency, **99**
Data Table, 262, 265, **266**, 266, 267
Debug Agent Commands, 139, 147, 189, **273**
Debug Entities Size, **185**
Debug Entities, number, **185**
debug server, run-mode, **63**
debug server, system level, **63**
debug support, **173**
Debugging Core Dumps, 269, **271**
Debugging Out of RAM, 165, **166**, 167, 168
Debugging Out of ROM/Flash, 165, **167**
Debugging Problems Downloading with the Virtual Loader, 301, **303**
Debugging SMP Systems, 291, **299**
Debugging Tasks with the OSA Explorer, **216**
debugging, kernel aware, **213**
debugging, OS aware, **213**
Default Dynamic Download INTEGRITY Project, **76**
Default INTEGRITY Top Project, **70**
Default KernelSpace Project, **81**
Default Monolith INTEGRITY Project, **73**
default.gpj, **57**
default.ld, 56, 71, 74, 82, **87**, 87, 89, 91, 143, 160, 185, 196, 208, 269
default.ld Example, **89**
Detecting Stack Overflow, 147, 249, **258**
Developing Symmetric Multiprocessing (SMP) Applications, **291**
Development Guide, **14**, 14
Dining Philosophers Demo, 29, **31**
DISABLE_HEAP_EXTEND, **143**, 143
Disconnecting from rtserv2, **181**
Divide by Zero, 251, **253**, 253
Download Area Requirements, **196**
Dynamic Download from Debugger Failed, 301, **302**
Dynamic Download INTEGRITY Application Project, 67, 68, **75**, 75, 77, 152, 195, 198, 302

Dynamic Download project, build, **76**
Dynamic Download project, create, **75**
Dynamic Download project, default, **76**
Dynamic Download Status, 189, 195, **198**
Dynamic Download, troubleshooting, **301**
Dynamic Downloading, 29, 64, 77, 171, **195**, 195, 302
Dynamic Memory Allocation, 260, **288**, 288
dynamically create Tasks, 79, **134**
Dynamically Created Tasks, 144, **146**

employing shared memory, **148**
Engineer Task, 35, 36, 39, 42, 44, 45, **46**, 46
error checking options, 250, **251**
error checking, run-time, **249**
EventAnalyzer, **65**
EventAnalyzer security, **281**
EventAnalyzer User's Guide, **15**, 15, 65, 281
Execution Time Profiling, **202**
exit vs. Exit, **137**
extended physical memory, **141**, 141, 144
ExtendedMemoryPool Configuration, 140, **141**
ExtendedMemoryPool Size Configuration, **142**

flag stack overflow, **258**
flash.ld, 56, 71, 79, **87**, 87, 89, 90, 160, 168, 169
Floating Point Registers and Other Extended Register Sets, **289**
Freeze-Mode Debug Server, 61, **63**, 63, 169
Freeze-Mode Debugging, 13, 48, 63, **165**, 165, 166, 169, 183, 191, 303
Freeze-Mode Debugging Using Host I/O, **165**, **169**
Freeze-Mode Debugging Using Probes, **165**, 165

gdump, 95, **151**, 151
Generating Profile Data Programmatically, 201, **210**
Getting Started with INTEGRITY, **19**, 174
GILA Boot Loader Guide, **48**
global_table.c, 58, 74, **82**, 82, 170
Green Hills Debug Probes User's Guide, **165**, 165, 169, 175

Green Hills Standard C Library Functions, **284**, 284, 289
 gstack, **147**, 147, 258
 gstack Utility Program, **147**, 147

Halt On Attach, **188**, 188
 hardbrk, **169**, 169
 hardware breakpoints, 168, 192, **193**, 193
 Heap Configuration, 140, **142**, 211
 Heap Growth, **143**, 144
 Heap Size Configuration, 87, **142**, 208
 HEAP_EXTEND_CONTIGUOUS, **143**, 143, 144
 Host I/O, 95, 163, 169, 170, **190**, 190, 191, 205
 Host I/O, console redirection, **170**
 Host I/O, over freeze-mode debug, **169**
 Host I/O, over rtserve2, **190**

I/O pane, 161, 179, **190**
 Idle Task, **138**, 138, 139, 184, 265
 Implementation-Dependent Behavior Guide, **15**
 index, **311**
 INDRT2 security, **280**
 INDRT2, connecting with, **171**
 Information Task, 35, 39, **42**, 42–44
 Initial Task, **138**
 Initializing a Callback, 283, **284**
 Initiating a Dynamic Download, 195, **197**
 Installation Guide, **14**, 14
 Instrumented Stack Checking, **258**, 258
 Integrate Configuration File, 40, 67, 77, **84**, 84, 92, 106, 135–137, 140, 142–145, 149, 306
 Integrate User’s Guide, **15**, 15, 57, 77, 84, 93, 104, 135, 140, 142, 144, 145, 152, 208
 Integrate, creating Tasks with, **135**
 Integrate, troubleshooting, **306**
 INTEGRITY 11.7 Document Set, **14**
 INTEGRITY Application Types, **67**, 67, 125
 INTEGRITY BSP User’s Guide, **15**, 15, 93, 100, 286, 291, 293, 294
 INTEGRITY Development Guide, 13, **14**, 14
 INTEGRITY Directory Structure, 51, **53**

INTEGRITY document Set, 13, **14**, 15, 52, 55
 INTEGRITY EventAnalyzer User’s Guide, **15**
 INTEGRITY GILA Boot Loader Guide, **48**, 48
 INTEGRITY Implementation-Dependent Behavior Guide, **15**, 15, 173
 INTEGRITY Include Directory, 51, 53, **54**
 INTEGRITY Installation and Directory Structure, 49, **51**, 306
 INTEGRITY Installation Guide, **14**, 14, 52
 INTEGRITY Kernel Reference Guide, 14, **15**, 134, 140, 146, 258, 297
 INTEGRITY Kernel User’s Guide, 14, **15**, 148
 INTEGRITY kernel, load, **49**
 INTEGRITY Libraries and Utilities Reference Guide, **15**, 15, 135, 148, 269, 271, 287
 INTEGRITY Libraries and Utilities User’s Guide, **15**, 15, 49, 93, 135, 146, 149, 153, 195, 280, 281, 287
 INTEGRITY Networking Guide, **15**, 15, 48, 82, 142, 172, 263, 277, 280
 INTEGRITY Shared Memory Configuration, 140, **148**
 INTEGRITY Simulator (ISIM), 19, 23, 27, 48, **49**, **64**
 INTEGRITY Top Project, default, **70**
 INTEGRITY Violation Before main(), 301, 304
 INTEGRITY.h, **54**, 54, 134, 295
 INTEGRITY.ld, 53, 71, 73, 76, 78, **87**, 87, 88, 132, 208
 INTEGRITY_version.h, 54, **103**, 103
 integritykernel image, **81**
 integritykernel.gpj, **81**
 Introduction, **13**, 14, 16
 Introduction to Dynamic Downloading, **195**, 195
 Introduction to ISIM, **159**, 159
 Introduction to OSA Debugging, **213**, 213
 Introduction to rtserve2 and INDRT2, **171**, 171
 Introduction to SMP, **291**, 291

- Introduction to the MULTI
ResourceAnalyzer, **261**, 261
- Invoking CRC on the Kernel and Virtual AddressSpaces, **99**, 307
- Invoking SHA-1 on an INTEGRITY Monolith, **100**
- IODevice list, OSA Explorer, **221**
- ISIM, **27**, **159**
- ISIM - INTEGRITY Simulator, 19, 64, **159**
- ISIM Caveats, 159, **163**
- ISIM Features, 159, **160**
- ISIM Socket Emulation, 159, **162**
- ISIM Socket Port Remapping, **162**, 181
- ISIM Target BSPs, 159, **160**
- ISIM, run, **161**
- Kernel Aware Debugging, **213**, 213, 216
- Kernel Lock Contention, **292**
- kernel physical memory, **141**, 141
- Kernel Reference Guide, **15**
- Kernel Source Installation, 13, 14, **52**, 52, 192, 250
- Kernel User's Guide, **15**
- KernelSpace Project, 58, 67, 68, 74, **79**, 82, 119, 121–123, 146, 168, 173
- KernelSpace project, build, **82**
- KernelSpace project, create, **79**
- KernelSpace Project, create from scratch, **83**
- KernelSpace Project, default, **81**
- Launcher, start, **21**
- libc.so, 95, **96**, 97, 98
- libcore.a, 58, **269**, 269, 270
- libdebug.a, 29, 58, 79, **173**, 173, 185, 192, 211, 263, 279, 280
- libdebug.a Configuration, **173**
- libcxx.so, **96**, 96
- libcxx_e.so, **96**, 96
- libINTEGRITY.so, **95**, 95, 97, 98
- Libraries and Utilities Reference Guide, **15**, 146, 287
- Libraries and Utilities User's Guide, **15**, 15
- libs Directory, 51, 53, **59**
- libscxx.so, **96**, 96–98
- libscxx_e.so, **96**, 96
- Link list, OSA Explorer, **220**
- Linker Directives File, 53, 54, 56, 67, 70, 71, 78, 79, **87**, 87–89, 92, 94, 142, 146, 185, 193, 208, 288
- linker directives file, custom, **87**
- Linker, troubleshooting, **304**
- LoaderTask, 79, 137, 187, 195, **196**, 196–199
- Loading and Running an INTEGRITY Kernel, **49**
- makefiles, 99, **104**, 104
- Managing Shared Data, 291, **294**
- Managing Workspaces and Shortcuts with the Launcher, **23**, 23
- Memory Allocation Errors, 249, 254, **255**, 255
- memory configuration, application, **140**
- Memory Leak, 249, **255**, 255, 277
- Memory Region list, OSA Explorer, **219**
- Memory Regions, **92**, 92, 93
- MemoryPool Configuration, **140**, 140, 267, 303
- MemoryPool Size Configuration, **140**
- Modify Project sub-menu, **125**
- Modifying Dynamic Download Projects with the Project Manager, 107, **118**
- Modifying KernelSpace Projects with the Project Manager, 107, 112, **122**
- Modifying Linker Constants, **91**, 91
- Modifying Monolith Projects with the Project Manager, 107, **112**
- Monolith INTEGRITY Application Project, 67, **72**, 72, 74, 79, 152
- Monolith project, build, **74**
- Monolith project, create, **72**
- Monolith project, default, **73**
- mr_Link Section, **92**
- MULTI / rtserver Loss of Communication to the Board, 301, **303**
- MULTI Builder and INTEGRITY Documentation Restrictions, **153**
- MULTI Document Set, **15**, 16
- MULTI documentation, **15**, 63, 99
- MULTI EventAnalyzer, **65**
- MULTI IDE, 16, 27, **61**, 61, 63
- MULTI Integrated Development Environment, 19, 20, **61**, 61

- MULTI Launcher, 21–25, 27, 29, 49, 54, **61**, 61, 62, 68, 72, 75, 79
- MULTI Launcher, start, **21**
- MULTI Profile Window, 64, **65**, 65, 171, 201, 202, 206
- MULTI Profiling, **201**, 201–203, 209, 211
- MULTI Project Manager, 25, 29, 37, 40, 53–55, 57, 61, **62**, 62, 64, 68–70, 72, 74, 75, 77–84, 87, 88, 94, 104, 107, 111, 117, 121, 123, 125, 127, 128, 131, 160, 166, 168, 173, 175, 179, 307
- MULTI Project Manager Reports Incorrect Component Types for Legacy Projects, 301, **307**
- MULTI Project Manager Target, **54**, 306
- MULTI Tools for Finding Performance Bottlenecks, 61, **65**
- Multi-Level Security, 279, **281**
- MULTI: Building Applications, **16**, 16, 54, 64, 144, 147, 153, 284, 289, 311
- MULTI: Configuring Connections, **16**, 16, 160, 165, 166
- MULTI: Debugging, **16**, 16, 65, 66, 166–168, 174, 202, 207, 209, 249, 255
- MULTI: Debugging Command Reference, **16**, 16, 169
- MULTI: Getting Started, 14, **16**, 16
- MULTI: Licensing, **16**, 16
- MULTI: Managing Projects and Configuring the IDE, **16**, 16, 23, 64, 107
- MULTI: Scripting, **16**, 16
- Multiple Core Dumps, 269, **270**
- Networking Guide, **15**
- networking software security, **280**
- Nil Pointer Dereference, 251, **253**, 253, 255
- No Kernel Banner, 301, **305**
- Not Enough Target RAM Memory Allocation Errors, 301, **305**
- Number of Debug Entities, **185**, 185, 193
- Object list, OSA Explorer, **222**
- Object Number Usage, **84**
- Object Structure Aware Debugging, 148, **213**
- Observing Stack Usage During Execution, **147**
- Optimizing Your Programs, **311**, 311
- OS Module Selection dialog, 117, **126**
- OSA Explorer, 147, 213, **214**, 214–216, 218, 224
- OSA Explorer Activity Tab, **218**
- OSA Explorer Clock Tab, **221**
- OSA Explorer Connection Tab, **217**
- OSA Explorer IODevice Tab, **221**
- OSA Explorer Link Tab, **220**
- OSA Explorer Memory Region Tab, **219**
- OSA Explorer Object Tab, **222**
- OSA Explorer Resource Tab, **222**
- OSA Explorer Semaphore Tab, **218**
- OSA Explorer Task Tab, 148, **215**
- OSA Object Viewer, 213, **224**, 224, 225, 228, 232, 233, 237, 239–241, 243, 244, 247
- Overhead of Run-time Error Checking, 249, **260**
- overlapping breakpoints, **192**, 192
- PC samples, 201, **202**, 202, 210, 211
- Performance Bottlenecks, 65, 201, 301, **310**
- PhoneCompany Task, 35, **39**, 39–42
- Pizza Demo, 29, **35**, 35, 46
- PizzaHut Task, 35, 36, **44**, 44, 45
- Placing a Callback, 283, **285**
- POSIX Demo, **29**, 29
- POSIX Task creation, **135**
- Preparing Your Target, **167**, 167, 168
- privileged instructions, **154**, 154
- probe run mode, 166, 172, 174, **175**, 175
- Processor Binding, **292**
- Producer/Consumer Demo, 29, **32**
- Profile window, **201**, 203, 205, 207–210
- Profiling Overhead and Limitations, 201, **211**
- Profiling with rtserv2, 64, 65, **201**
- profiling, call counts, **207**
- profiling, PC samples, **202**
- profiling, with call graph support, **207**
- Program Counter (PC) Samples, **202**, 202
- Programming Flash Memory, **168**, 168
- Project Manager, configuration, **107**
- RAM debugging, **166**

ram_sections.ld, 56, **87**, 87, 89, 142, 196, 269
Redirecting the Console over Host I/O, **170**
reset_fixup, **169**, 169
Resource list, OSA Explorer, **222**
ResourceAnalyzer, **65**
ResourceAnalyzer data table, **266**
ResourceAnalyzer Selected Task/AddressSpace panel, **265**
ResourceAnalyzer System Total panel, **265**
ResourceAnalyzer, MULTI, **261**
ResourceAnalyzer, using, **263**
ResourceManager and Callbacks, **287**
ResourceManager security, **280**
Restrictions on BSPs and Drivers, **288**
Restrictions on Hand-Written Assembly Code, **153**
Restrictions on KernelSpace Application Code, **154**
Restrictions on MULTI Builder and Driver Options, **153**
restrictions, application development, **153**
ROM/flash debugging, **167**
rom_sections.ld, 56, **87**, 87, 89
rtserv2, build in debug support, **173**
rtserv2, connect, **174**
rtserv2, connect with command-line, **179**
rtserv2, connecting with, **171**
rtserv2, Connection Organizer, **175**
rtserv2, debugging with, **183**
rtserv2, host I/O, **190**
rtserv2, profiling, **201**
rtserv2, troubleshooting, **303**
Run-Mode Debug Server, 61, **63**, 171, 201
Run-Mode Debugging, 13, 64, 109, 116, 126, 165, 169, 171, 173, 178, **183**, 183, 185, 216, 249, 250, 263, 280, 299
Run-Mode Debugging Limitations, **183**, 183
run-mode partner, 99, **174**, 174, 175
Run-time Error Checking, 147, **249**, 249, 250, 260
Run-time Error Checking Options, 250, **251**, 258
run-time error checking overhead, **260**, 260
run-time error checking, enable, **250**
Run-Time Error Checks, **249**, 249–253
Run-Time Memory Checking, 249, 254, **255**, 255
run-time memory checking, enable, **254**
Running a Dynamically Downloadable Image, **301**, 301
Running ISIM, 159, 160, **161**
Running the Kernel in ISIM and Connecting with rtserve2, 19, **27**, 29, 37
Security and INTEGRITY Kernel, **279**, 279
Security and Networking Software, 279, **280**
Security and ResourceManagers, 279, **280**
Security and the EventAnalyzer, 279, **281**
Security and the INDRT2 Debug Agent, 171, 185, 279, **280**
Security Issues, 171, 185, **279**, 281
security, multi-level, **281**
Select Item to Add Dialog, **125**, 125
Selected Task/AddressSpace panel, 262, **265**, 265
Semaphore list, OSA Explorer, **218**
Set Run-Mode Partner dialog box, **174**, 174
Setting Options for C++ Shared Libraries, **96**
Settings for Dynamic Download Dialog, **118**, 118
Settings for Monolith Dialog, **112**, 112
Settings for OS Module Selection dialog, **126**
Shared Libraries, 19, 25, 67, **94**, 94–96, 98, 109, 112, 116, 118, 207, 255, 292
shared memory, **148**, 148–151, 298
Shared Memory Consistency, **294**
shared memory, static, **149**
Sharing Libraries Across Multiple INTEGRITY Applications, **97**
Simulator for ARM (simarm) Connections, **160**, 160
Simulator for PowerPC (simpcc) Connections, **160**, 160
SMP, **291**
SMP and Shared Libraries, **292**
SMP Atomic Operations, **297**
SMP Debugging Tools, **299**
SMP Run-mode Debugging Limitations, **299**
socket emulation, in ISIM, **162**

- software breakpoints, 168, 185, **192**, 192
 sparse downloading, **196**, 197
 Specifying Memory Regions in the Linker Directives File, **92**
 Stack Configuration, 136, 140, **144**, 288
 Stack Overflow, 144, 147, 148, 249, 258, 301, 304, **308**, 308
 Stack Overflow Detection, 147, **148**, 258, 260
 stack overflow, troubleshooting, **308**
 stack size, 96, 136, 144–146, **147**, 147, 215, 261, 265, 274, 275
 stack usage, observe, **147**
 Standard Calls, 201, **202**, 203
 Standard Source Installation, **51**, 51–53
 Start the Demo, 29, **37**, 37
 Starting the MULTI Launcher, 19, **21**, 25
 static memory allocation, **149**
 statically create Tasks, **135**
 Statically Defined Tasks, **144**, 144, 145
 Stop on Task Creation, 30, **188**, 188
 stop-mode debug server, **63**
 stop-mode debugging, **165**, 191
 Supported Debug Agent Commands, **273**
 Synchronous Callbacks, **283**
 System Core Dumps, **269**, 269
 system debugging, **165**, 165
 System Profiling, **202**, 202
 System Total and Selected
 Task/AddressSpace Graphical Panels, 262, **265**
 System Total panel, 262, **265**, 265
 system-level debug server, **63**
 system-level debugging, **165**
 Target Board Setup, 19, 20, **48**
 Target Build Files, **54**, 54
 target Directory, 51, 53, **54**, 306
 target file, troubleshooting, **305**
 Target list, 28, 30–32, 34, 38, 39, 42, 44, 47, 63, 78, 171, 178, **186**, 199, 202, 204, 216, 224
 Target list, MULTI Debugger, **186**
 Target pane, 34, 47, 161, 163, 179, **189**, 198, 199, 302, 303
 Target Settings, **188**
 Task and AddressSpace Naming, 109, 116, 126, 133, **186**
 Task Defined in Integrate Configuration File
 Cannot Exit, 301, **306**
 Task list, OSA Explorer, **215**
 Task Priority Does Not Provide Mutual Exclusion, 292, **294**
 Task Profiling, **202**, 202
 Tasks, create and destroy, **133**
 The Help Viewer Window, **14**, 14
 The INTEGRITY Real-Time Operating System, **13**
 The Project Manager GUI Reference, **107**, 107
 Time Efficiency, **288**
 Top Level INTEGRITY Directory, **53**, 53, 87, 88
 Trace Data, **66**, 66
 Troubleshooting, 65, **301**, 301
 Troubleshooting Copy/Paste in the Project Manager, **130**
 Tutorials, 19, 20, 25, 26, 28, **29**, 29, 58, 84
 Typographical Conventions, **16**, 16
 Undefined Symbol main(), 301, **304**
 Understanding Task Scheduling, 291, **292**
 Unloading or Reloading an Application, **199**
 Using Breakpoints with Run-Mode
 Debugging, 183, 185, **192**
 Using Callbacks with INTEGRITY, **283**, 283
 Using Checksum and SHA-1 Utilities for Data Consistency, **99**
 Using Exit() vs. exit(), 133, **137**
 Using Hardware Breakpoints with rtserv2, **193**
 Using Integrate to Configure Statically Shared Memory, **149**
 Using INTEGRITY Tutorials, 19, 25, **29**, 58
 Using INTEGRITY Version Constants to Conditionalize Code, **103**
 Using ISIM, **27**
 Using makefiles, **104**
 Using MULTI to Develop INTEGRITY Applications, **61**
 Using Probe Run Mode, 166, 172, 174, **175**
 Using Run-Time Error Checks, **249**, 249

- Using Run-time Memory Checks, 249, **254**, 259
Using Software Breakpoints with rtserv2, **192**
Using the Command-Line Driver, **104**, 104, 153
Using the gstack Utility to Determine Stack Size Requirement, **147**
Using the I/O Pane, 183, **190**
Using the Memory Allocations Window, **249**, 249, 255
Using the MULTI Debugger with rtserv2, 178, 183, **186**, 216
Using the MULTI Profile Window, **201**, 201
Using the MULTI ResourceAnalyzer, 65, **261**
Using the OSA Explorer, 213, **214**, 214
Using the OSA Object Viewer, 213, **224**
Using the Run-Mode Partner, **174**, 174
Using the Target Pane, 183, **189**
- Viewing Activity Objects, **244**
Viewing AddressSpace Objects, **228**
Viewing Binary Semaphore Objects, **239**
Viewing Clock Objects, **241**
Viewing Connection Objects, **243**
Viewing Counting Semaphore Objects, **237**
Viewing Highest Locker Semaphore Objects, **240**
Viewing IODevice Objects, **247**
Viewing Link Objects, **247**
Viewing MemoryRegion Objects, **232**
Viewing Profiling Information in the Debugger, **207**, 207, 209
Viewing Semaphore Objects, **237**
Viewing Task Objects, **233**
violation before main(), troubleshooting, **304**
Virtual AddressSpace Program, 74, 77, **78**, 78, 145, 302, 304
Virtual AddressSpace Project, 53, 59, 67, 68, 72–77, **78**, 78, 88, 304
virtual memory breakpoints, **193**, 193
Virtual to Physical Mapping for Code and Data, 133, **151**
Virtual to Physical Mapping for Dynamic Download Application, **152**
- Virtual to Physical Mapping for Monolith Applications, **151**
Weak Symbols, 133, **157**
Working with the ResourceAnalyzer, 261, 262, **263**
Workspaces, create, **23**
Write to Watchpoint, 251, **253**, 253