MULTI: Configuring Connections for ARM Targets

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

LEGAL NOTICES AND DISCLAIMERS

PubID: connect_arm-744926
Branch: http://toolsvc/branches/release-branch-2023-5-comp
Date: September 1, 2023

Contents

Preface

Contents

This preface discusses the purpose of the manual, the MULTI documentation set, and typographical conventions used.

About This Book

The MULTI Debugger can be used to debug applications that reside on native, embedded, or simulated *targets*. To debug these applications, MULTI must first connect to the target through a *debug server* that runs on the host. Each distribution of the MULTI Integrated Development Environment (IDE) includes multiple debug servers, each of which supports connections to a specific set of probes, in-circuit emulators, monitors, and/or target simulators. This book describes how to configure connections to your target using the appropriate debug server. Specifically:

- The chapters in this book provide connecting instructions, features, commands, and options specific to each debug server. This information supplements the general connection instructions described in the documentation about connecting to your target in the *MULTI: Debugging* book.
- Appendix A, "Green Hills Debug Server Command and Scripting Reference", documents the commands that are supported by some Green Hills debug servers and describes the scripting language and conventions used for writing target setup scripts in previous versions of MULTI. The scripting language described in this appendix has been deprecated beginning with MULTI 5.0. The new scripting conventions are described in the documentation about MULTI scripts in the *MULTI: Scripting* book.

Note

New or updated information may have become available while this book was in production. For additional material that was not available at press time, or for revisions that may have become necessary since this book was printed, please check your installation directory for release notes, **README** files, and other supplementary documentation.

Supported Target Connections for ARM

The following table lists the target connections supported for ARM. For detailed information about each type of target connection, see the appropriate chapter, as identified in the table.

Debugging interface, monitor, or simulator	Green Hills debug server	Debug server chapter
Green Hills Probe or SuperTrace Probe	**mpserv**	Chapter 2, “Green Hills Debug Probe (mpserv) Connections” on page 7
INDRT (Run-time OS debugging)	**rtserv**	the documentation about INDRT (rtserv) connections in the *MULTI: Debugging* book
INDRT2 (Run-time OS debugging)	**rtserv2**	the documentation about INDRT2 (rtserv2) connections in the *MULTI: Debugging* book
Simulator for ARM (simarm)	**simarm**	Chapter 3, “Simulator for ARM (simarm) Connections” on page 9

Note

If you are using MULTI with the Green Hills INTEGRITY RTOS, see the INTEGRITY document set you received with your INTEGRITY distribution. If you are using another supported OS target, see the appropriate book from the list below.

- *MULTI: Developing for ThreadX*

This list may not be comprehensive. Contact Green Hills Technical Support if your particular debugging interface, monitor, simulator, or OS does not appear in the list.

The MULTI Document Set

The primary documentation for using MULTI is provided in the following books:

- *MULTI: Getting Started* — Provides an introduction to the MULTI Integrated Development Environment and leads you through a simple tutorial.
- *MULTI: Licensing* — Describes how to obtain, install, and administer MULTI 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* — Describes 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.
- *MULTI: Toolchain Error Messages and Other Diagnostics* — Provides a sequential listing of MULTI diagnostic numbers and messages for the compiler, assembler, and linker. Where possible, includes information about associated resolutions and workarounds.

For a comprehensive list of the books provided with your MULTI installation, see the **Help → Manuals** menu accessible from most MULTI windows.

All books are available in one or more of the following formats:

- Print
- Online help, accessible from most MULTI windows via the **Help → Manuals** menu
- PDF, available in the **manuals** subdirectory of your MULTI or compiler installation

Conventions Used in the MULTI Document Set

All Green Hills documentation assumes that you have a working knowledge of your host operating system and its conventions, including its command line and graphical user interface (GUI).

Green Hills documentation uses a variety of notational conventions to present information and describe procedures. These conventions are described below.

Convention	Indication	Example
bold type	Filename or pathname	**C:\MyProjects**
	Command	The **setup** command
	Option	The **-G** option
	Window title	The **Breakpoints** window
	Menu name or menu choice	The **File** menu
	Field name	The **Name** field
	Button name	The **Browse** button
italic type	Replaceable text	where *version* is replaced by the version number
	A new term	A task may be called a *process* or a *thread*
	A book title	*MULTI: Debugging*
`monospace` type	Text you should enter as presented	Type `unload`
	A word or words used in a command or example	`Path` must be a valid C string
	Source code	`printf("Hello world!\n");`
	Input/output	`> print Test` `Test`
	A function	`GHS_System()`
ellipsis (...) (in command line instructions)	The preceding argument or option can be repeated zero or more times.	**debugbutton** [*name*]...
greater than sign (>)	Represents a prompt. Your actual prompt may be a different symbol or string. The > prompt helps to distinguish input from output in examples of screen displays.	`> print Test` `Test`
pipe (|) (in command line instructions)	One (and only one) of the parameters or options separated by the pipe or pipes should be specified.	**call** *func* | *expr*

Convention	Indication	Example
square brackets ([ ]) (in command line instructions)	Optional argument, command, option, and so on. You can either include or omit the enclosed elements. The square brackets should not appear in your actual command.	`.macro` *`name`* `[`*`list`*`]`

The following command description demonstrates the use of some of these typographical conventions.

gxyz [-*option*]*...filename*

The formatting indicates that:

- **gxyz** is a command.
- The option *`-option`* should either be replaced with one or more appropriate options or be omitted.
- The word *`filename`* should be replaced with the actual filename of an appropriate file.

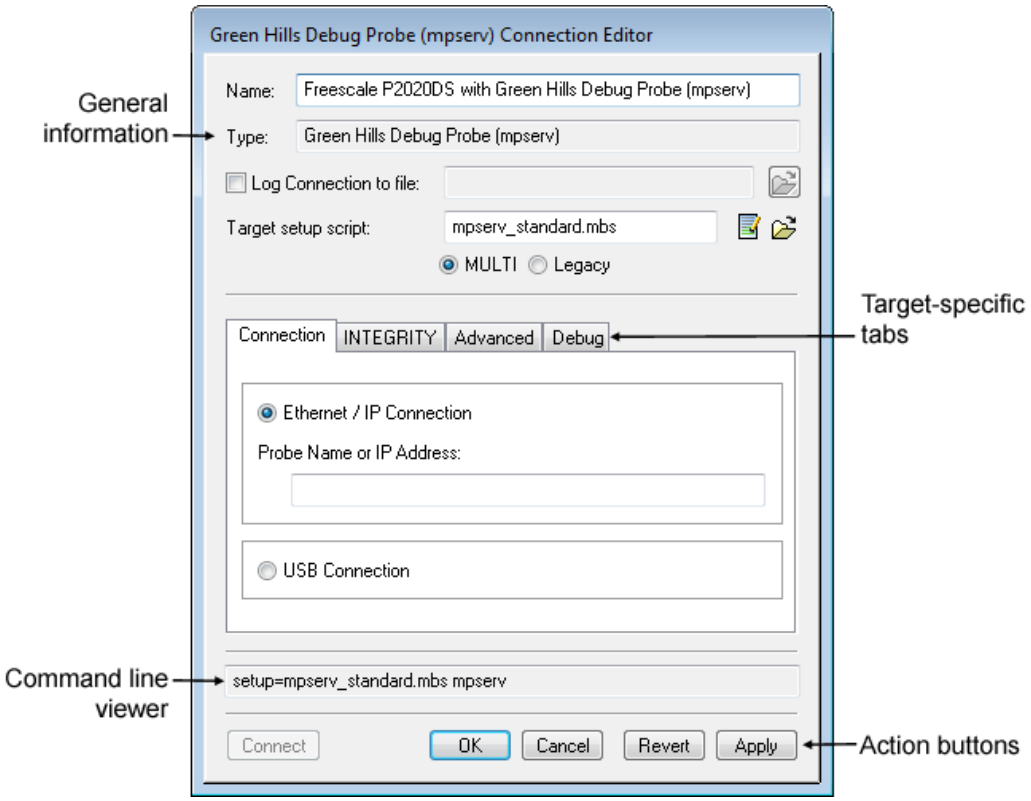
The square brackets and the ellipsis should not appear in the actual command you enter.

Chapter 1

Using the Connection Editor

Contents

The **Connection Editor** is a target specific dialog that you can use to configure and edit Connection Methods.



The **Connection Editor** window for Standard Connection Methods has four sections, which are called out in the above graphic and documented in the following sections.

General Information Settings

The first section of the **Connection Editor** contains fields that display and/or set basic properties of a Connection Method. These fields are standard and appear on nearly all **Connection Editor** windows. The following table describes each field.

Name Displays the name of the Standard Connection Method you define or edit. You can change the name of your method by editing this field.
Type Displays the type of debugging interface the Connection Method is for, which determines which debug server MULTI uses. The connection type is specified when you create a Standard Connection Method. You cannot edit this read-only field.
Log Connection to file Enables you to log transactions between MULTI and your debug server. To specify the file to write the log to, enter a pathname or click  to browse for the desired file. On Linux, if you check the option box but do not specify a filename, MULTI writes the log to standard error output. Logging is disabled by default.
Target Setup script Specifies your target setup script file. This is an optional field because not all targets require setup scripts. If you require a setup script to configure your board, specify the filename (and full path, if necessary) of the appropriate script or click  to browse for it. When this Connection Method is used, MULTI usually runs the specified script before each download. If you are editing a default Connection Method created by the Project Wizard for your processor-board combination, the necessary setup script file (if applicable) appears in this field automatically. To open the Editor on the target setup script, click  . If no setup script is specified in this field, the Editor opens on a file with a filename derived from the debug connection's name. For more information about target setup scripts, see the documentation about configuring your target hardware in the *MULTI: Debugging* book.

Scripting language radio buttons

Specify the scripting language so that MULTI knows how to interpret the specified target setup script, where:

- **MULTI** — Indicates that the specified target setup script was written with the MULTI scripting language. This is the default setting. For more information about MULTI scripting in general, see the documentation about MULTI scripts in the *MULTI: Scripting* book.
- **Legacy** — Indicates that the specified target setup script was written with the debug server scripting language. For more information, see Appendix A, "Green Hills Debug Server Command and Scripting Reference" on page 21.

In previous versions of MULTI, target setup scripts had **.dbs** extensions and used the Green Hills debug server scripting conventions and associated debug server commands. Beginning with the v4.0 release, that scripting language was deprecated, and support for it will be discontinued in a future release. The default setup scripts created by the Project Wizard have **.mbs** extensions and use MULTI Debugger commands and scripting conventions instead. You must indicate which scripting language your setup script uses so that MULTI interprets it properly. When using the MULTI scripting language, you can still specify debug server commands by using the MULTI **target** command. For information about this command, see the documentation about general target connection commands in the *MULTI: Debugging Command Reference* book.

For more detailed information about editing and using setup scripts, see the documentation about configuring your target hardware in the *MULTI: Debugging* book.

Target-Specific Tabs

The second section of the **Connection Editor** contains one or more tabs that contain fields for setting options specific to your debugging interface, type of connection, and target architecture. The number of tabs and the fields that appear vary according to the type of Standard Connection Method you are editing.

When the **Connection Editor** is first displayed after you create a new Connection Method, the settings and options are set to default values. Settings and options that are not available on your host operating system may appear dimmed. Some of the fields may require user input before the Connection Method can be used.

The following table describes in general terms the types of settings and options that appear on the most common tabs.

Tab	Description
Connection tab	The options on this tab usually require input for the Connection Method to function properly. The settings you choose for the connection options are probably different for each new target you connect to.
INTEGRITY tab	The option on this tab is useful if you are debugging an INTEGRITY system. For more information, see the documentation for **set_runmode_partner** in *MULTI: Debugging Command Reference*.
Download tab	The options on this tab allow you to customize the process of downloading programs to the target. For example, this tab allows you to control what program sections are downloaded to the target.
Advanced tab	The options on this tab allow a high degree of control over the details of your connection. These options can often be left in their default states. Use this tab sparingly because changing the advanced options from their default settings may cause problems with your connection.
Debug tab	The options on this tab are provided to help troubleshoot your connection. You should not change these settings unless instructed to do so by Green Hills Technical Support.

Most **Connection Editors** include at least a **Connection** tab and a **Debug** tab, but many contain additional tabs. For most connections, you need to set or edit some of the fields on these tabs for your Standard Connection Method to work properly. For more information about the **Connection Editor** settings available for your particular target, see the chapter that documents your debug server, or if you are a Green Hills Probe or SuperTrace Probe user, see the *Green Hills Debug Probes User's Guide*.

The Command Line Viewer

The third section of the **Connection Editor** displays the command line equivalents of your GUI selections. Default selections do not always have corresponding commands or options on the command line.

The command line viewer is a read-only field. You cannot make changes to the command line by typing in this field.

Changes you make in the GUI will not be reflected in the command line viewer until you click **Apply** or **OK**.

Action Buttons

The fourth section of the **Connection Editor** contains buttons that allow you to discard your edits, apply your edits, or connect using a Connection Method. The following table describes each button.

Button	Description
Connect	Records your changes, connects to your target using the Standard Connection Method you are editing, and closes the **Connection Editor**.
OK	Records your changes and closes the **Connection Editor**.
Cancel	Discards your changes and closes the **Connection Editor**.
Revert	Discards your changes and leaves the **Connection Editor** open.
Apply	Records your changes and leaves the **Connection Editor** open.

Note

Clicking **Connect**, **OK**, or **Apply** in the **Connection Editor** saves your Standard Connection Method. In the future, the Connection Method will appear in the list of available Connection Methods in the **Connection Chooser** and the **Connection Organizer**.

Chapter 2

Green Hills Debug Probe (mpserv) Connections

The Green Hills family of debug probes includes the Green Hills Probe and the SuperTrace Probe. These advanced hardware debugging instruments communicate with your target over a standard JTAG or SWD test port and allow you to use the MULTI Debugger to control, debug, and trace your target CPU. Green Hills Probe and SuperTrace Probe target connections are all supported through the **mpserv** debug server. For complete instructions on how to set up these probes and make target connections using **mpserv**, see the *Green Hills Debug Probes User's Guide* that came with your probe.

Chapter 3

Simulator for ARM (simarm) Connections

Contents

MULTI includes a target simulator called **simarm** that allows you to execute and debug ARM code on your host system. This chapter supplements the general target connection information contained in Chapter 2, "Setting Up Your Target Hardware" of this book and the documentation about connecting to your target in the *MULTI: Debugging* book with information about how to connect to and use the **simarm** simulator.

Features available when using **simarm** with MULTI include:

- Symbolic disassembly
- Interactive breakpoint debugging
- Single-step execution into procedure calls
- Single-step execution over procedure calls

Installing the Simulator for ARM (simarm)

The **simarm** Simulator is installed automatically when you install the MULTI Compiler. No other installation steps are necessary.

By default, the **simarm** Simulator runs in fast mode, which is significantly faster than older versions of the simulator, but does not offer an identical feature set. If you require backwards compatibility to accomplish specific tasks (and you have the necessary licenses), you can also run the simulator in compatibility mode. For more information, see "Fast Mode and Compatibility Mode" on page 16.

Connecting to the Simulator for ARM (simarm)

To help you connect to targets (including simulators) quickly and easily, MULTI allows you to create and save Connection Methods that specify the parameters of your connection.

For general instructions that explain how to create and use Connection Methods, see the documentation about connecting to your target in the *MULTI: Debugging* book. The information in the following sections supplements the instructions provided there with information that is specific to Simulator for ARM (**simarm**) connections.

Using the Simulator for ARM (simarm) Connection Editor

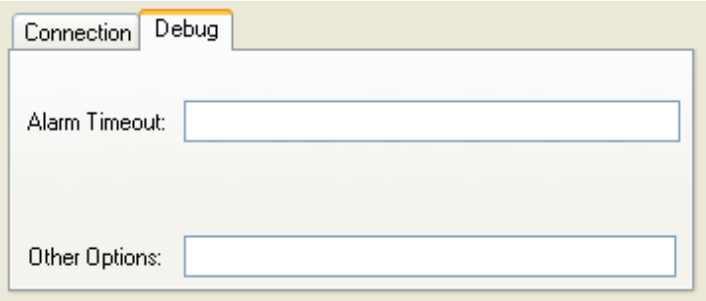
In addition to the generic fields that appear on all Connection Editors for Standard Connection Methods, (see Chapter 1, "Using the Connection Editor" on page 1) the **Simulator for ARM (simarm) Connection Editor** includes **Connection** and **Debug** tabs that provide settings and options specific to your simulated target and your host operating system.

When the **Connection Editor** is first displayed after you create a new Connection Method, the settings and options are set to default values. Settings and options that are not available on your host operating system may appear dimmed. Some of the fields may require user input before the Connection Method can be used.

Simulator for ARM (simarm) Connection Settings

Processor Specifies which target processor to simulate. This field defaults to **ARM7TM**.
ROM mode Specifies ROM Simulation Mode (as opposed to OS Simulation Mode). For more information about OS and ROM Simulation Modes, see "Simulation Environments" on page 14.
FPU Specifies whether the simulated target has a hardware floating-point coprocessor. If set to **<default>**, this option is based on the selected CPU.

Simulator for ARM (simarm) Debug Settings



Warning

Do not change the settings on the **Debug** tab unless you are instructed to do so by Green Hills Technical Support.

Other Options
Allows you to add other, optional arguments directly to the command line. You should only use this field if directed to do so by Green Hills Technical Support.

Using Custom Simulator for ARM (simarm) Connection Methods

To connect to the Simulator for ARM with a Custom Connection Method, type the command given below, with the appropriate parameters and options, into the **Start a Custom Connection** dialog box and click **Connect**.

simarm [*options*]...

where *options* can be any non-conflicting combination of the **simarm** options listed in “Options for Custom simarm Simulator Connection Methods” on page 12.

Note

You can also enter the above connection command from the Debugger's command pane, where it must be preceded by the **connect** command.

See the documentation about Custom Connection Methods in the *MULTI: Debugging* book for more information about Custom Connections.

Options for Custom simarm Simulator Connection Methods

The following options are available when creating a Custom **simarm** simulator Connection Method.

-allocmem
Allocates memory pages at run-time when simulating access to addresses outside the known extent of ROM or RAM. This option is enabled by default when either **-rom** or **-rom_use_entry** is enabled in the compatibility mode simulator. You must specify the option manually for the fast mode simulator. When **-allocmem** is not enabled, accesses to addresses outside the known extent of memory result in a `Memory Access Violation` or `Invalid Memory Read` error. See the release notes in the MULTI installation directory for more information.
-collect_stats
Enables collection of various run-time statistics (fast mode only).
-cpu=*cpu*
Specifies which target processor to simulate.

-fpu Specifies that the simulated target has a hardware floating point coprocessor.
-rom Enables ROM mode on the simulator and sets the simulator to start from the reset vector. For more information, see “ROM Mode” on page 15.
-rom_use_entry Enables ROM mode on the simulator and sets the simulator to start from the program's entry point. For more information, see “ROM Mode” on page 15.

Additional Commands for Simulator for ARM (simarm) Connections

The following commands, in addition to some of the commands listed in “Generic Debug Server Commands” on page 22, are available to **simarm**.

You can enter all of these commands directly into the **simarm Target** pane. You can also enter these commands into the MULTI Debugger command pane using the **target** command. All of the commands for **simarm** are case-insensitive.

alloc *addr size* Allocates `size` bytes at `addr`
assert_fiq Assert FIQ interrupt. Supported in ROM mode. FIQ interrupts must be enabled in status register. This command is not supported for Cortex-M.
assert_irq Assert IRQ interrupt. Supported in ROM mode. IRQ interrupts must be enabled in status register. This command is not supported for Cortex-M.
help Lists available commands
insts Displays instruction profile. In fast mode, this command is only enabled if you pass the **-collect_stats** option to **simarm** when connecting.
reset Causes processor to reset trap

set *key value* Sets the target-specific parameter `key` to `value`. For a list of different keys, run this command without specifying `key` or `value`.
timer *count* **fiq** Assert FIQ every `count` instructions. Supported in ROM mode. FIQ interrupts must be enabled in status register. This command is not supported for Cortex-M.
timer *count* **irq** Assert IRQ every `count` instructions. Supported in ROM mode. IRQ interrupts must be enabled in status register. This command is not supported for Cortex-M.

Simulation Environments

The **simarm** simulator has two simulation environments:

- *OS Simulation Mode* — Simulates an environment similar to a user process running in Linux. This is the default mode.
- *ROM Mode* — Simulates just the bare minimum hardware, such as the CPU and memory systems. To enable ROM mode, connect using a Custom Connection Method and include the **-rom** or **-rom_use_entry** option in the Connection Method command.

The **simarm** Simulator is designed to be applicable for user level applications, but processor features such as caches, memory management unit, and any special peripherals and registers are generally not implemented.

OS Simulation Mode

The OS Simulator Mode (which is the default mode of the **simarm** simulator) provides an environment similar to a user process running under Linux. The simulator allocates memory for the text and initialized and uninitialized data of the simulated program, loads it into memory, and starts executing at the entry point specified at link time. The stack pointer is initialized and any program arguments are passed to the main program via the normal `argc`/`argv` convention. A limited set of system calls is supported for I/O purposes. This level of simulation is suitable

for debugging user-level programs, but if your programs need a more accurate simulation of the actual CPU behavior, you should run in ROM mode (see “ROM Mode” on page 15).

While running in the simulator, your program can make the system calls listed in the table below.

access	alarm	brk	close	creat	exit
fstat	getpid	link	lseek	open	read
rename	stat	time	unlink	write	

Be aware that while only these system calls are implemented, many library functions use a combination of these calls to achieve their goal.

To implement these system calls, the simulator cooperates with the Green Hills Libraries by setting a breakpoint in the `.syscall` section. The simulator is then able to trap any call into this special section and treat it as a system call request by the library.

In OS simulation mode, the only memory available is that allocated initially, plus any created by using the `brk` system call, such as using the `malloc` function in the libraries. References to other memory addresses will be trapped by the simulator and cause the program to abort. Similarly, attempts to use exception generating instructions also cause the program to abort. You should use ROM mode if the simulation must closely reflect actual hardware behavior.

ROM Mode

ROM Mode simulates only the bare minimum hardware, such as CPU and memory systems. In ROM mode, you can set the simulator to start at either the reset vector or at the program's entry point. To cause the simulator to always start execution at the address of the system reset vector (address `0x0`), as if the CPU were given a hard reset, and to ignore any program startup address specified at link time, connect using a Custom Connection Method and include the **-rom** option in the Connection Method command. To start at the program's entry point instead, use a Custom Connection Method and include the **-rom_use_entry** option in the Connection Method command..

Other features of ROM mode include:

- Exceptions are handled as in the hardware, vectoring to the appropriate address. User code must be supplied to handle the exception appropriately.
- No arguments are passed to the running program.
- The stack pointer must be initialized by user startup code.
- A limited set of system calls for I/O is supported.

Since the simulator handles very few things for you in ROM mode, you can use this mode for writing and testing exception handlers, and even parts of a real operating system. You should write at least some start-up code in pure assembly language, since this mode simulates a bare bones processor.

Note

If you are using the new simulator in ROM mode (with **simarm -rom** or **simarm -rom_use_entry**), memory pages are not allocated at run-time when simulating accesses to memory outside the known extent of RAM. This behavior differs from the behavior of the old simulator. If you encounter `Invalid memory read/write` errors, you can:

- Add the extra memory pages you need as a section in the link map (**.ld** file) for the program you are simulating and re-link your program.
- Pass the **-allocmem** option to the simulator (for example, `simarm -rom_use_entry -allocmem`).

Fast Mode and Compatibility Mode

The MULTI Compiler ships with two different simulator modes:

- *fast mode* — Runs much faster than compatibility mode.
- *compatibility mode* — Supports legacy features.

If you require backwards compatibility for specific tasks, you may need to connect to the simulator in compatibility mode. For more information about the features supported in this mode, see “Features Supported by the ARM Compatibility Mode Simulator” on page 18.

When you connect to **simarm**, MULTI checks for available licenses for both fast mode and compatibility mode. If it finds licenses for fast mode, it will connect in fast mode. If it only finds compatibility mode licenses, it will connect to the simulator in compatibility mode instead. To check which mode you are using:

1. Connect to **simarm** using any of the methods described in the previous sections.
2. Type the **info** Debugger command. If you are running in fast mode, you will see:

   ```
   Target: Simulated ARM
   ```

   If you are running in compatibility mode, you will see:

   ```
   Target: simarm_compat
   ```

If you want to explicitly specify compatibility mode, use **simarm_compat** in a custom connection method. For example:

```
multi -connect=simarm_compat
```

Profiling with simarm

You can use MULTI to profile with the **simarm** simulator. The simulated clock rate is 20 MHz, but the clock cycles used by each instruction are approximations. For more information about profiling, see the documentation about recording and viewing profiling data in the *MULTI: Debugging* book.

Connecting to the Simulator for ARM (simarm) as a Stand-alone Debug Server

To connect to the Simulator for ARM from the command line, independently of MULTI, enter:

```
simarm [options] image_file
```

where:

- *options* — Can be any non-conflicting combination of the options listed in "Options for Custom simarm Simulator Connection Methods" on page 12.
- *image_file* — Specifies an ELF-style executable file.

Connecting to the simulator in this way allows you to simulate your program, but does not offer the interactive debugging capabilities available through MULTI.

Features Supported by the ARM Compatibility Mode Simulator

The following sections list additional options and commands available when using compatibility mode.

Additional Compatibility Mode Options

If you are using Linux, the compatibility mode simulator allows you to abort simulation after a specified number of seconds using the **-T** command line option.

Additional Compatibility Mode Commands

This section contains additional commands available when using the compatibility mode simulator. You can enter all of these commands directly into the **simarm Target** pane. You can also enter these commands into the MULTI Debugger **Cmd** pane using the **target** command. All of the commands are case-sensitive.

cycles Displays number of cycles executed.
cycles reset Resets number of cycles executed.
freq *f* Sets clock frequency to *f* (the default is 0 MHz).

Chapter 4

Simulator for INTEGRITY on ARM (isimarm)

The **isimarm** simulator is a specialized version of the simulator for ARM intended for simulating a target running the INTEGRITY Real-Time Operating System. For more information, see the *INTEGRITY Development Guide*.

Appendix A

Green Hills Debug Server Command and Scripting Reference

Contents

This chapter describes generic debug server commands and a full scripting language that can be used with most Green Hills debug servers.

These commands and the scripting language are used only in legacy **.dbs** scripts, and are deprecated.

Green Hills Debug Server Commands

The commands described below are available to most Green Hills debug servers, but do not apply to simulator or operating system connections.

Most Green Hills debug servers also support additional commands. See "Additional Debug Server Commands" on page 28 for a listing of where to find descriptions of the additional commands (if any) available for the debug server that supports your specific target.

Generic Debug Server Commands

Unless otherwise noted, the commands listed in this section are available for all of the debug servers in this book.

All Green Hills debug server commands are case-insensitive.

addressof *symbol*

Returns the address of `symbol`. This command requires that an image be loaded in the MULTI Debugger.

amask [*mask*] [*value*]

If no arguments are specified, the current settings of the download mask and value are displayed. The default settings of `mask` and `value` are `0xffffffffffffffff` and `0`, respectively.

If `mask` and `value` are specified, **amask** sets the download mask and value to `mask` and `value`. When the debug server downloads a program, it will bitwise AND `mask` and bitwise OR `value` with the download addresses, as follows:

```
address = (address & mask) | value
```

The **amask** command is useful for shifting download target addresses without relinking a program. However, the program being downloaded still retains its original relocations and might not run correctly at its new destination address without further support on the target.

close *fd* Closes the specified file descriptor, `fd`.
debug *n* Sets the debugging bit flags. Do not use this command unless directed to do so by Green Hills Technical Support.
delrange *addr* Deletes a checked address range starting at `addr`. See **setrange** below for more information about checked memory ranges.
delrangeall Deletes all checked address ranges. See **setrange** below for more information about checked memory ranges.
echo [on | off] If no argument is specified, the current echo mode is printed. If `on` or `off` is specified, printing of commands executed in a script are enabled or disabled.
fprint *fd string* Prints *string* to the specified file, *fd*, with script variable expansion.
fprintb *fd integer* Prints `integer` to the specified file, `fd`, in binary mode.
fread *fd identifier* Reads one line of text from the specified file, `fd`. The line is stored as a variable with the specified `identifier`. The number of bytes read is returned. If there is an error, `-1` is returned.
freadb *fd identifier* Reads an integer from the specified file, `fd`, in binary mode. The integer is stored as a variable with the specified `identifier`. The number of bytes read is returned. If there is an error, `-1` is returned.
getenv *envName identifier* Stores the value of the environment variable, `envName`, in the script variable with the specified `identifier`.
halt Halts execution of the target CPU and forces it into debugging mode.

help [*command* | *group*]

(See also the device-specific **help** command listed in "Additional Debug Server Commands" on page 28.)

If no argument is specified, general help information and instructions for finding more detailed and specific help is printed.

If a *command* is specified, detailed help information about the specified *command* is printed.

If a *group* is specified, a list of all of the commands in the group with information about the arguments each command takes is printed. Valid command groups include **target**, **server**, and **scripting**.

Example 1:

```
> help server
help [<command> | <group>]
debug <n>
playdead
```

Example 2:

```
> help help
help [<command> | <group>]
Prints help information
```

listrange

Lists all checked ranges. See **setrange** below for more information about checked memory ranges.

listvars

Prints all variable identifiers in no particular order.

Example:

```
> str1="foo"
> str2="bar"
> i=100
> listvars
i
str1
str2
```

load [all] [text] [data] [bss]

If no argument is specified, displays the current **load** settings.

If one or more arguments are specified, **load** instructs which sections to include in host-to-target downloads. `.text` sections contain code, `.data` sections contain initialized variables, and `.bss` sections contain uninitialized data. Setting the `all` option includes all of these sections in the download.

You can combine the **load** command with the **noload** command.

Green Hills startup code clears all `.bss` sections so you do not need to download them. In most cases, the default setting is `text data`, but the default varies according to your debug server.

Standard Example:

```
> load all
Download Options: text data bss
```

Advanced Example:

```
> load all noload bss
Download Options: text data
```

m [-d*size*] *address*[=*val*]

If =*val* is not specified, the indicated target memory *address* is read.

If =*val* is specified, *val* is written to the indicated target memory *address*.

Memory addresses and values must be specified in hexadecimal (with or without a leading `0x`).

The optional `-d` argument can be used to set the access size to byte (`-d1`), short (`-d2`), or long (`-d4`). The default is `-d4`.

Examples:

```
> m 1000
7ca62b78
> m 1000=12345678
> m -d2 1000
1234
```

nofail *command*

Executes *command* without returning an error. This can be useful to prevent a command-specific error from aborting a script.

noload [all] [text] [data] [bss]

If no argument is specified, the current **noload** settings are displayed.

If one or more arguments are specified, **noload** specifies which sections to exclude from host-to-target downloads. `.text` sections contain code, `.data` sections contain initialized variables, and `.bss` sections contain uninitialized data. Setting the `all` option excludes all of these sections from the download.

Use the command **noload bss** to exclude `.bss` sections from downloads. These sections are cleared to all zeros by programs compiled with Green Hills tools; therefore, downloading them to the target is usually unnecessary.

open *file*

Opens the specified *`file`* for writing and returns a file descriptor.

print *string*

Prints *`string`* with script variable expansion.

Example:

```
> print Test
Test
```

quiet [on | off | -- *command*]

Controls output from script commands, globally or selectively.

If `on` or no argument is specified, suppresses all script command output.

If `--` *`command`* is specified, runs *`command`* and suppresses any resulting output.

random *max*

Generates and returns a pseudo-random number between `0` and *`max`*`-1`.

run [*address*]

If *`address`* is not specified, the processor is run from the current program counter(PC).

If *`address`* is specified, it sets the program counter to *`address`* and then runs the processor.

Use this command very carefully. It is possible to run a program on the board when MULTI thinks it is halted, which causes unpredictable results.

A common use for this command is to run a ROM monitor program so that the monitor can set up the board properly before MULTI downloads a program.

script *file*

Executes the commands in the specified script file, *`file`*, as if they were typed in line by line.

setrange *addr length*

Sets a checked address range beginning at *addr* and extending for *length* bytes.

A checked address range is a range of addresses that are considered inaccessible. Any memory access, read, or write to a checked address range can fail.

The **setrange** command is useful for restricting access to target memory that might be sensitive or volatile.

setup *file*

Specifies a script file, *file*, to be automatically run immediately prior to every host-to-target download.

sleep *n*

Suspends the debug server for *n* seconds.

status

Prints and returns the current status of the debug server using the following status codes:

```
0 Running
1 Stopped by breakpoint
2 Single step completed
3 Exception
4 Halted
5 Process exited
6 Process terminated
7 No process
8 (Unassigned)
9 Stopped by hardware breakpoint
10 Failure
11 Process ready to run
12 Host system call in progress
13 Target reset
```

step

Single steps the target CPU from the current program counter location.

stepint [nomask | mask]

Turns interrupts during single stepping on (`nomask`) or off (`mask`).

When set, **stepint** masks all interrupts during a single step. For example, on Power Architecture, interrupt masking is accomplished by masking the EE bit in the MSR. Stepping over code that alters the MSR with **stepint** on will result in unpredictable values in the MSR.

If no argument is specified, this command displays the current value.

undef *variable*

Removes *variable* and releases any memory associated with it.

Example:

```
> x=5
> undef x
> print $x
Error: variable undefined!
```

Additional Debug Server Commands

The following table indicates where you can find information about additional commands (if any) available for your particular debugging interface.

Debugging interface	Where to find additional commands for this target
Green Hills Debug Probe (Green Hills Probe or SuperTrace Probe)	*Green Hills Debug Probes User's Guide*

The Green Hills Debug Server Scripting Language

In addition to the commands listed in “Generic Debug Server Commands” on page 22 and the additional commands available to your specific debug server (see “Additional Debug Server Commands” on page 28), a full scripting language is available from the debug server command line for all of the debug servers in this book except the Simulator for ARM (**simarm**).

You can run scripts by:

- Entering scripts one line at a time at the command prompt. When entering a script line by line, nothing is executed until the top-level enclosing **while** or **if** statement is terminated.
- Storing commands in a script file and then running the file using the **script** command.
- Storing commands in a script file and then using the **-setup** option or **setup** command (or the **Target Setup Script** field in some Connection Editors) to

ensure that the script file is automatically executed immediately prior to every host-to-target download.

General Notes

When writing scripts, keep the following points in mind:

- There can be no more than one statement per line.
- Any line that has the # character as the first non-whitespace character is treated as a comment.
- Variables do not need to be declared before they are used.
- Variable types are determined automatically. For example, an identifier that is bound to an integer variable can later be assigned to a string variable.
- Variable and function names are not case-sensitive.

Scripting Syntax

The following sections describe and give examples of some features of the Green Hills debug server scripting language.

Expressions

Expressions in the Green Hills debug server scripting language are similar to C language expressions. Unlike C, each operator has its own precedence (`10/2*5` evaluates to 1, not 25), and there are no unary operators (such as ~ and unary -). The following table contains, in order of precedence, the valid operators you can use in expressions. Note that a string is treated like an integer if it contains a string representation of an integer.

Operator	Integer Function	String Function
`( )`	Grouping of operators to ensure desired evaluation	Grouping of operators to ensure desired evaluation
`*`	Multiplication	Invalid
`/`	Division	Invalid
`%`	Modulus	Invalid

Operator	Integer Function	String Function
+	Addition	Concatenation
-	Subtraction	Invalid
<<	Bitwise left shift	Invalid
>>	Bitwise right shift	Invalid
<	Less than	Alphabetic less than
<=	Less than or equal to	Alphabetic less than or equal to
>	Greater than	Alphabetic greater than
>=	Greater than or equal to	Alphabetic greater than or equal to
==	Equality test	Equality test
!=	Inequality test	Inequality test
&	Bitwise AND	Invalid
^	Bitwise XOR	Invalid
|	Bitwise OR	Invalid
&&	Logical AND	Invalid
||	Logical OR	Invalid

Assignments

The syntax for an assignment is:

```
identifier = expression
```

The *expression* is evaluated and the result is stored as a variable with the given *identifier*. String, integer, and array variables are supported. Identifiers can contain alphanumeric characters and the underscore (_) character, but cannot begin with a number, or have the same name as a command.

Arrays

Arrays are indexed lists of variables. Each cell in an array can contain a string, an integer, or an array. Array indexing begins with the index `0`. An array can be created by assigning an entire array to an identifier or by assigning a string, an integer, or an array to one cell of an array. To reference a cell, follow the array identified by

the index contained in square brackets. If an array cell contains another array, the elements in the second array are accessed by appending an additional index in square brackets. The following example demonstrates the various methods of array access.

```
endl="\n"
bar[3] = 42
foo = { bar, 7, "hello" }
print $foo[2] world.$endl
if(foo[0][3]==bar[2+1])
    print Array indexing works.$endl
endif
```

Arrays are dynamically allocated in a sparse fashion. For example, making an assignment to `foo[0]` and then to `foo[100]` only allocates two array cells, and no space is used for the undefined array cells 1 through 99. After an array element has been allocated, it can only be deallocated by using the **undef** command on the top-level array identifier.

Conditionals

The following table explains the syntax for conditionals.

Syntax	Effect
if *expression* *statements* **endif**	If *expression* evaluates to zero, nothing happens. Otherwise, the block of statements between the **if** and **endif** lines are executed.

if *expression*
statements
else
statements
endif
	If *expression* evaluates to zero, the debug server executes the block of statements between the **else** and **endif** lines. Otherwise, the block of statements between the **if** and **else** lines are executed.

Syntax	Effect
if *expression1* *statements* **elif *expression2*** *statements* [**elif *expressionX*** *statements*]... **endif**	If *expression1* does not evaluate to zero, the debug server executes the block of statements between the **if** and **elif** lines. If *expression1* does evaluate to zero and *expression2* does not evaluate to zero, the debug server executes the block of statements between the **elif** and **endif** lines. If both *expression1* and *expression2* evaluate to zero, the debug server continues evaluating each of the subsequent **elif** expressions (if any) that occur before the **endif** statement and will execute the block of statement associated with each **elif** that does not evaluate to zero. If none of the **elif** expressions evaluates to non-zero, the debug server will not execute anything.

Loops

The following table explains the syntax for loops.

Syntax	Effect
while *expression* *statements* **endwhile**	The statements between the **while** and **endwhile** lines are executed as long as *expression* does not evaluate to zero.
do *statements* **dowhile *expression***	The statements between **do** and **while** are executed one time, then continue to execute as long as *expression* does not evaluate to zero.

Be careful to avoid infinite loops. If an infinite loop occurs, you must shut down and restart the debug server.

Variable Expansion

To use script variables as arguments to Green Hills debug server commands, you must prepend the variable name with one or two `$` characters.

- To pass a variable to a command in its default text representation, prepend the variable name with a single `$` character. This passes a decimal string for integer variables and passes the string itself for string variables. Integers are expanded

as two's complement signed 32-bit integers. An entire array cannot be given as an argument to a command.

- To pass an integer variable to a command as a hexadecimal string, prepend the variable name with two `$` characters. Use this method with commands such as **m** that require arguments in hexadecimal form. The hexadecimal string does not include a leading `0x`.

Variable expansion must be unambiguous. The script parser attempts to use the longest legal identifier name following the `$` character. In the following example, the user has attempted to print the string `bar` after the expansion of the variable `foo`. The parser interprets this as printing the value of the variable `foobar` and reports that the variable is not defined:

```
>foo="foo"
>print $foobar
Error: variable undefined!
```

Example Scripts

You can use the following examples of the Green Hills debug server scripting language as a guide when writing your own scripts.

Example A.1. Testing Script File Access in ASCII Mode

For this example, assume that the file **test.ascii** contains the following text:

```
This is a test of script file access in ascii mode.
This should print 3 lines of which this is the second.
And this is the third.
```

The following script commands access the file **test.ascii**:

```
file = open test.ascii
filecontents=""
totalchars=0
while(numlinechars=fread file line)
    filecontents=filecontents+line
    totalchars=totalchars+numlinechars
endwhile
close file
endl="\n"
```

```
print Read: $filecontents$endl
print Total of $totalchars characters read.$endl
```

The output of this example script on a Linux host is:

```
Read: This is a test of script file access in ascii mode.
This should print 3 lines of which this is the second.
And this is the third.

Total of 130 characters read.
```

Example A.2. Accessing a File

The following script is another example of accessing a file:

```
i=100
file = open temp.bin
while(i>0)
    fprintb file $i
    i=i-1
endwhile
close file
file = open temp.bin
sum=0
while(freadb file i)
    sum=sum+i
endwhile
close file
endl="\n"
print The numbers between 1 and 100 sum to $sum!$endl
```

The output of this example script is:

```
The numbers between 1 and 100 sum to 5050!
```

Example A.3. Calculating a CRC32 Value

The following script calculates the CRC32 value of the memory range `0x010000` to `0x010100` and prints the result in the **Target** pane. This script demonstrates the use of loops, conditional statements, expressions, variables, and other debug server scripting constructs in a real application.

```
# Change the following values to specify the memory
# range you want to calculate a CRC32 for.
# Note: locations from memstart to memend-1 are used
```

```
# to compute the CRC32 value.
memstart=0x010000
memend=0x010100
# This is the CRC32 polynomial. This is the same as
# is used in ethernet packets.
p=2+4+16+32+128+256+1024+2048+4096+65536+4194304+8388608+67108864
r=0
ptr=memstart
while(ptr<memend)
    currbyte=m -d1 $$ptr
    currbit=128
    while(currbit)
        test=r&(1<<31)
        r=r<<1
        r=r|(currbyte&currbit)
        if(test)
            r=r^p
        endif
        currbit=currbit>>1
    endwhile
    ptr=ptr+1
endwhile
# This loop is for the 32 zeros appended to the
# original memory contents
i=0
while(i<32)
    test=r&(1<<31)
    r=r<<1
    if(test)
        r=r^p
    endif
    i=i+1
endwhile
# Now the resulting 32 bit CRC is in r
# Many of the same ASCII control codes that are used
# in C are supported in debug server scripts.
endl="\n"
print CRC32 = $$r$endl
```

Index

U

W