
RTEMS User Manual

Release 5.ac1773d-modified (4th March 2020)

RTEMS Documentation Project

Mar 04, 2020

CONTENTS

1	Introduction	3
1.1	Overview	4
1.2	Features	5
1.3	Ecosystem	8
1.3.1	Rational	8
1.3.2	Open Source	8
1.3.3	Deployment	9
1.4	Real-time Application Systems	10
1.5	Real-time Executive	11
2	Quick Start	13
2.1	Prepare Your Host Computer	14
2.2	Choose an Installation Prefix	15
2.3	Obtain the Sources	16
2.4	Install the Tool Suite	17
2.5	Bootstrap the RTEMS Sources	20
2.6	Build a Board Support Package (BSP)	21
2.7	Test a Board Support Package (BSP)	24
2.8	Build Your Application	25
3	Support and Contributing	27
3.1	RTEMS Project Support	28
3.1.1	Users Mailing List	28
3.1.2	Documentation	28
3.1.3	All Mailing Lists	28
3.1.4	IRC	28
3.2	Report Bugs	29
3.2.1	Search for Existing Bugs	29
3.2.2	Not RTEMS Bugs	29
3.2.3	Good Bug Reports	30
3.2.4	Nobody Fixes my Bug	31
3.3	Contributing	32
3.3.1	How to Contribute?	32
3.3.2	Preparing and Sending Patches	32
3.3.3	Checklist for Patches	32
3.3.4	Patch Review Process	33
3.3.5	Why Contribute?	33
3.3.6	Common Questions and Answers	35

3.4	Commercial Support Services	37
4	Host Computer	39
4.1	Host Operating Systems	40
4.2	POSIX Hosts	42
4.2.1	Root Access	42
4.2.2	Linux	42
4.2.2.1	ArchLinux	42
4.2.2.2	CentOS	43
4.2.2.3	Fedora	43
4.2.2.4	Raspbian	43
4.2.2.5	Ubuntu	43
4.2.2.6	Linux Mint	44
4.2.2.7	openSUSE	44
4.2.3	FreeBSD	44
4.2.4	NetBSD	44
4.3	Apple macOS	45
4.3.1	Catalina	45
4.3.2	Sierra	45
4.3.3	Mavericks	45
4.4	Microsoft Windows	46
4.4.1	Windows Path Length	46
4.4.2	Windows Spaces In Paths	46
4.4.3	Parallel Builds with Make	47
4.4.4	POSIX Support	47
4.4.5	Python	47
4.4.6	MSYS2	47
4.4.7	Cygwin	49
5	Installation	53
5.1	Releases	54
5.1.1	RTEMS Tools and Kernel	54
5.2	Developer (Unstable)	61
5.2.1	POSIX and OS X Host Tools Chain	61
5.2.2	Windows Host Tool Chain	66
5.2.2.1	RTEMS Windows Tools	66
5.2.2.2	Building the Kernel	69
5.3	RTEMS Kernel	75
5.3.1	Development Sources	75
5.3.2	Tools Path Set Up	75
5.3.3	Bootstrapping	75
5.3.4	Building a BSP	76
5.3.5	Installing A BSP	79
5.3.6	Contributing Patches	80
5.4	Project Sandboxing	82
6	Target Hardware	85
6.1	Targets	86
6.2	Architectures	87
6.3	Tiers	89
7	Board Support Packages	91

7.1	aarch64 (AArch64)	92
7.2	arm (ARM)	93
7.2.1	altera-cyclone-v (Intel Cyclone V)	93
7.2.1.1	Boot via U-Boot	93
7.2.1.2	Clock Driver	93
7.2.1.3	Console Driver	93
7.2.1.4	I2C Driver	93
7.2.1.5	Network Interface Driver	93
7.2.1.6	MMC/SDCard Driver	94
7.2.1.7	USB Host Driver	94
7.2.1.8	Caveats	94
7.2.2	atsam	94
7.2.3	beagle	94
7.2.3.1	Boot via U-Boot	94
7.2.3.2	Getting the Device Tree Blob	95
7.2.3.3	Writing the uEnv.txt file	95
7.2.3.4	I2C Driver	95
7.2.3.5	SPI Driver	95
7.2.4	csb336	96
7.2.5	edb7312	96
7.2.6	gdbarmsim	96
7.2.7	gumstix	96
7.2.8	imx (NXP i.MX)	96
7.2.8.1	Build Configuration Options	96
7.2.8.2	Boot via U-Boot	97
7.2.8.3	Clock Driver	97
7.2.8.4	Console Driver	97
7.2.8.5	I2C Driver	97
7.2.8.6	SPI Driver	97
7.2.8.7	Network Interface Driver	98
7.2.8.8	MMC/SDCard Driver	98
7.2.8.9	Caveats	98
7.2.9	lm3s69xx	98
7.2.10	lpc176x	98
7.2.11	imx (NXP i.MX)	99
7.2.11.1	Build Configuration Options	99
7.2.11.2	Boot via U-Boot	99
7.2.11.3	Clock Driver	100
7.2.11.4	Console Driver	100
7.2.11.5	I2C Driver	100
7.2.11.6	SPI Driver	100
7.2.11.7	Network Interface Driver	100
7.2.11.8	MMC/SDCard Driver	101
7.2.11.9	Caveats	101
7.2.12	raspberrypi	101
7.2.12.1	Setup SD card	101
7.2.12.2	Kernel image	101
7.2.12.3	Testing using QEMU	102
7.2.13	realview-pbx-a9	103
7.2.14	rtl22xx	103
7.2.15	smdk2410	103

7.2.16	tms570	103
7.2.17	xen (Xen on ARM)	103
7.2.17.1	Execution	103
7.2.17.2	Additional Information	104
7.2.18	xilinx-zynq	104
7.2.19	xilinx-zynqmp	104
7.3	bfin (Blackfin)	105
7.3.1	bf537Stamp	105
7.3.2	eZKit533	105
7.3.3	TLL6527M	105
7.4	epiphany (Epiphany)	106
7.4.1	epiphany_sim	106
7.5	i386	107
7.5.1	pc386	107
7.6	lm32 (LatticeMicro32)	108
7.6.1	lm32_evr	108
7.6.2	milkymist	108
7.7	m68k (Motorola 68000 / ColdFire)	109
7.7.1	av5282	109
7.7.2	csb360	109
7.7.3	gen68340	109
7.7.4	gen68360	109
7.7.5	genmcf548x	109
7.7.6	mcf5206elite	109
7.7.7	mcf52235	109
7.7.8	mcf5225x	109
7.7.9	mcf5235	109
7.7.10	mcf5329	109
7.7.11	mrm332	110
7.7.12	mvme147	110
7.7.13	mvme147s	110
7.7.14	mvme162	110
7.7.15	mvme167	110
7.7.16	uC5282	110
7.8	microblaze (Microblaze)	111
7.9	mips (MIPS)	112
7.9.1	csb350	112
7.9.2	hurricane	112
7.9.3	jmr3904	112
7.9.4	malta	112
7.9.5	rbtx4925	112
7.9.6	rbtx4938	112
7.10	moxie	113
7.10.1	moxiesim	113
7.11	nios2 (Nios II)	114
7.11.1	nios2_iss	114
7.12	or1k (OpenRISC 1000)	115
7.12.1	generic_or1k	115
7.13	powerpc (PowerPC)	116
7.13.1	beatnik	116
7.13.2	gen5200	116

7.13.3	gen83xx	116
7.13.4	haleakala	116
7.13.5	motorola_powerpc	116
7.13.5.1	Boot Image Generation	116
7.13.6	mpc55xxevb	116
7.13.7	mpc8260ads	116
7.13.8	mvme3100	116
7.13.9	mvme5500	116
7.13.10	psim	117
7.13.11	qemu_ppc	117
7.13.12	qoriq (QorIQ)	117
7.13.12.1	Boot via U-Boot	117
7.13.12.2	Clock Driver	117
7.13.12.3	Console Driver	118
7.13.12.4	Network Interface Driver	118
7.13.12.5	Topaz Hypervisor Guest	118
7.13.13	ss555	118
7.13.14	t32mppc	118
7.13.15	tqm8xx	118
7.13.16	virtex	118
7.13.17	virtex4	118
7.13.18	virtex5	118
7.14	riscv (RISC-V)	119
7.14.1	riscv	119
7.14.1.1	Build Configuration Options	119
7.14.1.2	Interrupt Controller	120
7.14.1.3	Clock Driver	120
7.14.1.4	Console Driver	120
7.14.2	griscv	120
7.15	sh (SuperH)	121
7.15.1	gensh1	121
7.15.2	gensh2	121
7.15.3	gensh4	121
7.15.4	shsim	121
7.16	sparc64 (SPARC V9)	122
7.16.1	niagara	122
7.16.2	usiii	122
7.17	sparc (SPARC / LEON)	123
7.17.1	erc32	123
7.17.2	leon2	123
7.17.3	leon3	123
7.18	v850 (V850)	124
7.18.1	gdbv850sim	124
7.19	x86_64	125
7.19.1	amd64	125
7.19.1.1	Build Configuration Options	125
7.19.1.2	Testing with QEMU	125
7.19.1.2.1	Building TianoCore's UEFI firmware, OVMF	125
7.19.1.3	Boot RTEMS via FreeBSD's bootloader	126
7.19.1.4	Paging	127
7.19.1.5	Interrupt Setup	127

7.19.1.6	Clock Driver	127
7.19.1.7	Console Driver	128
8	Executables	129
8.1	RTEMS Executable	130
8.2	Building an Application	131
8.2.1	Machine Flags and ABI	132
8.3	Target Execution	133
8.4	BSP Initialization	134
8.5	RTEMS Initialization	135
8.5.1	System Initialization Handlers	135
8.6	Debugging	137
8.7	Dynamic Loader	140
8.7.1	System Design	141
8.7.2	Loader Interface	142
8.7.3	Symbols	145
8.7.3.1	Base Image Symbols	146
8.7.3.2	Embedded Symbols	146
8.7.3.3	Loadable Symbols	147
8.7.4	Unresolved Symbols	147
8.7.5	Libraries	148
8.7.6	Large Memory	150
8.7.7	Allocator	150
8.7.8	Languages	152
8.7.9	Thread Local Storage	152
8.7.10	Architectures	153
8.7.10.1	ARM	153
8.7.10.2	PowerPC	153
8.8	Device Tree	154
8.8.1	Building the DTB	154
8.8.2	Using Device Tree Overlay	154
9	Testing	157
9.1	Test Banners	158
9.2	Test Controls	159
9.2.1	Expected Test States	159
9.2.2	Test Configuration	160
9.3	Test Builds	162
9.4	Tester Configuration	163
9.4.1	Defaults	163
9.4.2	BSP and User Configuration	163
9.4.3	Configuration Scripts	166
9.4.3.1	Console	166
9.4.3.2	Execute	167
9.4.3.3	GDB	167
9.4.3.4	TFTP	167
9.5	Coverage Analysis	169
9.6	Consoles	170
9.7	Simulation	171
9.8	GDB and JTAG	172
9.9	TFTP and U-Boot	173
9.9.1	Target Hardware	173

9.9.1.1	U-Boot Set Up	174
9.9.2	BSP Configuration	174
9.9.3	TFTP Sequences	176
10	Tracing	179
10.1	Introduction to Tracing	180
10.1.1	RTEMS Trace Using Trace Buffering	180
10.1.2	RTEMS Trace Using Printk	180
10.2	Tracing Examples	182
10.2.1	Features	182
10.2.2	Prerequisites	182
10.2.3	Demonstration	182
10.3	Capture Engine	186
10.3.1	Capture Engine Commands	186
10.3.2	Example	187
10.4	Trace Linker	189
10.4.1	Command Line	189
10.4.2	Configuration (INI) files	190
10.4.2.1	Tracer Section	190
10.4.2.2	Options section	191
10.4.2.3	Trace Section	192
10.4.2.4	Function Section	193
10.4.2.5	Generators	193
10.4.3	Development	196
10.5	Event Recording	198
11	Host Tools	199
11.1	RTEMS Linker	200
11.2	RTEMS Symbols	201
11.2.1	Symbol Table	201
11.2.2	2-Pass Linking	201
11.2.3	Command	202
11.2.4	Examples	202
11.3	RTEMS Executable Infomation	204
11.3.1	System Initialisation	204
11.3.2	Command	204
11.3.3	Examples	205
11.4	RTEMS BSP Builder	208
11.4.1	Developer Workflows	208
11.4.2	Build Characteristics	208
11.4.2.1	Profiles	209
11.4.2.2	Builds	209
11.4.2.2.1	All Build	210
11.4.3	Build Configurations	210
11.4.4	Performance	211
11.4.5	Command	212
11.4.5.1	Examples	213
11.5	RTEMS Tester and Run	216
11.5.1	Available BSP testers	216
11.5.2	Building RTEMS Tests	217
11.5.3	Running the Tests	220
11.5.4	Test Status	222

11.5.4.1	Pass	222
11.5.4.2	Fail	222
11.5.4.3	User-input	222
11.5.4.4	Expected-fail	222
11.5.4.5	Indeterminate	222
11.5.4.6	Benchmark	222
11.5.4.7	Timeout	222
11.5.4.8	Invalid	223
11.5.5	Logging	223
11.5.5.1	All	223
11.5.5.2	Failures	223
11.5.5.3	None	223
11.5.6	Reporting	224
11.5.7	Running Tests in Parallel	225
11.5.8	Command Line Help	225
11.6	RTEMS Boot Image	226
11.6.1	Boot Loaders	226
11.6.1.1	U-Boot	226
11.6.2	Hosts	227
11.6.2.1	FreeBSD	227
11.6.2.2	Linux	227
11.6.2.3	MacOS	227
11.6.3	Configuration	227
11.6.4	Command	228
11.6.5	Examples	229
11.7	RTEMS TFTP Proxy	233
11.7.1	Operation	233
11.7.2	Configuration	234
11.7.3	Command	234
11.7.4	Examples	235
12	Source Builder	237
12.1	Why Build from Source?	239
12.2	Project Sets	240
12.2.1	Bare Metal	240
12.2.2	RTEMS	241
12.2.3	Patches	242
12.2.3.1	Testing a Newlib Patch	243
12.3	Cross and Canadian Cross Building	245
12.3.1	Cross Building	245
12.3.2	Canadian Cross Building	245
12.4	Third-Party Packages	247
12.4.1	Vertical Integration	247
12.4.2	Building	247
12.4.3	Adding	248
12.4.4	Host and Build Flags	249
12.4.5	BSP Support	251
12.4.6	BSP Configuration	255
12.5	Configuration	257
12.5.1	Source and Patches	257
12.5.1.1	HTTP, HTTPS, and FTP	258

12.5.1.2	GIT	259
12.5.1.3	CVS	259
12.5.2	Macros and Defaults	260
12.5.2.1	Macro Maps and Files	260
12.5.2.2	Personal Macros	262
12.5.3	Report Mailing	262
12.5.4	Build Set Files	263
12.5.5	Configuration Control	263
12.5.6	Personal Configurations	264
12.5.7	New Configurations	265
12.5.7.1	Layering by Including	265
12.5.7.2	Configuration File Numbering	265
12.5.7.3	Common Configuration Scripts	265
12.5.7.4	DTC Example	265
12.5.7.5	Debugging	269
12.5.8	Scripting	269
12.5.8.1	Expanding	271
12.5.8.2	%prep	271
12.5.8.3	%build	273
12.5.8.4	%install	274
12.5.8.5	%clean	275
12.5.8.6	%include	275
12.5.8.7	%name	275
12.5.8.8	%summary	275
12.5.8.9	%release	276
12.5.8.10	%version	276
12.5.8.11	%buildarch	276
12.5.8.12	%source	276
12.5.8.13	%patch	277
12.5.8.14	%hash	278
12.5.8.15	%echo	278
12.5.8.16	%warning	278
12.5.8.17	%error	279
12.5.8.18	%select	279
12.5.8.19	%define	279
12.5.8.20	%undefine	279
12.5.8.21	%if	279
12.5.8.22	%ifn	281
12.5.8.23	%ifarch	281
12.5.8.24	%ifnarch	281
12.5.8.25	%ifos	281
12.5.8.26	%else	281
12.5.8.27	%endfi	281
12.5.8.28	%bconf_with	281
12.5.8.29	%bconf_without	281
12.6	Commands	282
12.6.1	Checker (sb-check)	282
12.6.2	Defaults (sb-defaults)	282
12.6.3	Set Builder (sb-set-builder)	283
12.6.4	Set Builder (sb-builder)	286
12.7	Building and Deploying Tool Binaries	288

12.8 Bugs, Crashes, and Build Failures	291
12.8.1 Contributing	291
12.9 History	292
13 Glossary	293
Index	297

Copyrights and License

© 2019 Vijay Kumar Banerjee
 © 2018 Amaan Cheval
 © 2018 Marçal Comajoan Cara
 © 2018 Vidushi Vashishth
 © 2017 Tanu Hari Dixit
 © 2016, 2019 embedded brains GmbH
 © 2016, 2019 Sebastian Huber
 © 2012, 2019 Chris Johns
 © 2012 Gedare Bloom
 © 1988, 2018 On-Line Applications Research Corporation (OAR)

This document is available under the [Creative Commons Attribution-ShareAlike 4.0 International Public License](https://creativecommons.org/licenses/by-sa/4.0/).

The authors have used their best efforts in preparing this material. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. No warranty of any kind, expressed or implied, with regard to the software or the material contained in this document is provided. No liability arising out of the application or use of any product described in this document is assumed. The authors reserve the right to revise this material and to make changes from time to time in the content hereof without obligation to notify anyone of such revision or changes.

The RTEMS Project is hosted at <https://www.rtems.org>. Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to the RTEMS Project community.

RTEMS Online Resources

Home	https://www.rtems.org
Documentation	https://docs.rtems.org
Mailing Lists	https://lists.rtems.org
Bug Reporting	https://devel.rtems.org/wiki/Developer/Bug_Reporting
Git Repositories	https://git.rtems.org
Developers	https://devel.rtems.org

INTRODUCTION

1.1 Overview

You are someone looking for a real-time operating system. This document

- presents the basic features of RTEMS, so that you can decide if it is worth to look at,
- gives you a *quick start* (page 13) to install all the tools necessary to work with RTEMS, and
- helps you to build an example application on top of RTEMS.

1.2 Features

The Real-Time Executive for Multiprocessor Systems (*RTEMS*) is a multi-threaded, single address-space, real-time operating system with no kernel-space/user-space separation. It is capable to operate in an *SMP* configuration providing a state of the art feature set.

RTEMS is licensed under a [modified GPL 2.0 or later license with an exception for static linking](#)¹. It exposes no license requirements on application code. The third-party software used and distributed by RTEMS which may be linked to the application is licensed under permissive open source licenses. Everything necessary to build RTEMS applications is available as open source software. This makes you completely vendor independent.

RTEMS provides the following basic feature set:

- *APIs*
 - *POSIX* with [pthreads](#) (enables a broad range of standard software to run on RTEMS)
 - [Classic](#)
 - *C11* (including [thread](#) support)
 - *C++11* (including [thread](#) support)
 - Newlib and *GCC* internal
- Programming languages
 - C/C++/OpenMP (RTEMS Source Builder, RSB)
 - Ada (RSB, `--with-ada`)
 - Erlang
 - Fortran (RSB, `--with-fortran`)
 - Python and MicroPython
- Parallel languages
 - *EMB*²
 - Google Go²
 - *OpenMP* 4.5
- Thread synchronization and communication
 - Mutexes with and without locking protocols
 - Counting semaphores
 - Binary semaphores
 - Condition variables
 - Events
 - Message queues
 - Barriers

¹ The goal is to use the [BSD 2-Clause license](#) for new code or code those copyright holder agreed to a license change, see [#3053](#) for the details.

² See [#2832](#).

- *Futex* (used by *OpenMP* barriers)
 - Epoch Based Reclamation (*libbsd*)
- Locking protocols
 - Transitive Priority Inheritance
 - *OMIP* (SMP feature)
 - Priority Ceiling
 - *MrsP* (SMP feature)
- Scalable timer and timeout support
- Lock-free timestamps (FreeBSD timecounters)
- Responsive interrupt management
- C11/C++11 *TLS*³
- Link-time configurable schedulers
 - Fixed-priority
 - Job-level fixed-priority (*EDF*)
 - Constant Bandwidth Server (experimental)
- Clustered scheduling (SMP feature)
 - Flexible link-time configuration
 - Job-level fixed-priority scheduler (*EDF*) with support for one-to-one and one-to-all thread to processor affinities (default SMP scheduler)
 - Fixed-priority scheduler
 - Proof-of-concept strong *APA* scheduler
- Focus on link-time application-specific configuration
- Linker-set based initialization (similar to global C++ constructors)
- Operating system uses fine-grained locking (SMP feature)
- Dynamic memory allocators
 - First-fit (default)
 - Universal Memory Allocator (*UMA*, *libbsd*)
- File systems
 - *IMFS*
 - *FAT*
 - *RFS*
 - *NFSv2*
 - *JFFS2* (NOR flashes)

³ Thread-local storage requires some support by the tool chain and the RTEMS architecture support, e.g. context-switch code. It is supported at least on ARM, PowerPC, RISC-V, SPARC and m68k. Check the [RTEMS CPU Architecture Supplement](#) if it is supported.

- YAFFS2 (NAND flashes, GPL or commercial license required)
- Device drivers
 - Termios (serial interfaces)
 - I2C (Linux user-space API compatible)
 - SPI (Linux user-space API compatible)
 - Network stacks (legacy, libbsd, lwIP)
 - USB stack (libbsd)
 - SD/MMC card stack (libbsd)
 - Framebuffer (Linux user-space API compatible, Qt)
 - Application runs in kernel-space and can access hardware directly
- libbsd
 - Port of FreeBSD user-space and kernel-space components to RTEMS
 - Easy access to FreeBSD software for RTEMS
 - Support to stay in synchronization with FreeBSD

1.3 Ecosystem

The RTEMS Ecosystem is the collection of tools, packages, code, documentation and online content provided by the RTEMS Project. The ecosystem provides a way to develop, maintain, and use RTEMS. Its parts interact with the user, the host environment, and each other to make RTEMS accessible, useable and predictable.

The ecosystem is for users, developers and maintainers and it is an ongoing effort that needs your help and support. The RTEMS project is always improving the way it delivers the kernel to you and your feedback is important so please join the mailing lists and contribute back comments, success stories, bugs and patches.

What the RTEMS project describes here to develop, maintain and use RTEMS does not dictate what you need to use in your project. You can and should select the work-flow that best suites the demands of your project and what you are delivering.

1.3.1 Rational

RTEMS is complex and the focus of the RTEMS Ecosystem is to simplify the complexity for users by providing a stable documented way to build, configure and run RTEMS. RTEMS is more than a kernel running real-time applications on target hardware, it is part of a project's and therefore team's workflow and every project and team is different.

RTEMS's ecosystem does not mandate a way to work. It is a series of parts, components, and items that are used to create a suitable development environment to work with. The processes explained in this manual are the same things an RTEMS maintainer does to maintain the kernel or an experienced user does to build their production system. It is important to keep this in mind when working through this manual. We encourage users to explore what can be done and to discover ways to make it fit their needs. The ecosystem provided by the RTEMS Project will not install in a single click of a mouse because we want users to learn the parts they will come to depend on as their project's development matures.

The RTEMS Ecosystem provides a standard interface that is the same on all supported host systems. Standardizing how a user interacts with RTEMS is important and making that experience portable is also important. As a result the ecosystem is documented at the command line level and we leave GUI and IDE integration for users and integrators.

Standardizing the parts and how to use them lets users create processes and procedures that are stable over releases. The RTEMS Ecosystem generates data that can be used to audit the build process so their configuration can be documented.

The ecosystem is based around the source code used in the various parts, components and items of the RTEMS development environment. A user can create an archive of the complete build process including all the source code for long term storage. This is important for projects with a long life cycle.

1.3.2 Open Source

RTEMS is an open source operating system and an open source project and this extends to the ecosystem. We encourage users to integrate the processes to build tools, the kernel and any third-party libraries into their project's configuration management processes.

All the parts that make up the ecosystem are open source. The ecosystem uses a package's source code to create an executable on a host so when an example RTEMS executable is created and run for the first time the user will have built every tool as well as the executable from source. The RTEMS Project believes the freedom this gives a user is as important as the freedom of having access to the source code for a package.

1.3.3 Deployment

The RTEMS Project provides the ecosystem as source code that users can download to create personalised development environments. The RTEMS Project does not provide packaging and deployment for a specific host environment, target architecture or BSP. The RTEMS Project encourages users and organizations to fill this role for the community. The *RTEMS Source Builder* (page 237) provides some aid to *build and deploy tool binaries* (page 288).

1.4 Real-time Application Systems

Real-time application systems are a special class of computer applications. They have a complex set of characteristics that distinguish them from other software problems. Generally, they must adhere to more rigorous requirements. The correctness of the system depends not only on the results of computations, but also on the time at which the results are produced. The most important and complex characteristic of real-time application systems is that they must receive and respond to a set of external stimuli within rigid and critical time constraints referred to as deadlines. Systems can be buried by an avalanche of interdependent, asynchronous or cyclical event streams.

Deadlines can be further characterized as either hard or soft based upon the value of the results when produced after the deadline has passed. A deadline is hard if the results have no value after the deadline has passed, or a catastrophic event results from their intended use if not completed on time. In contrast, results produced after a soft deadline may still have some value.

Another distinguishing requirement of real-time application systems is the ability to coordinate or manage a large number of concurrent activities. Since software is a synchronous entity, this presents special problems. One instruction follows another in a repeating synchronous cycle. Even though mechanisms have been developed to allow for the processing of external asynchronous events, the software design efforts required to process and manage these events and tasks are growing more complicated.

The design process is complicated further by spreading this activity over a set of processors instead of a single processor. The challenges associated with designing and building real-time application systems become very complex when multiple processors are involved. New requirements such as interprocessor communication channels and global resources that must be shared between competing processors are introduced. The ramifications of multiple processors complicate each and every characteristic of a real-time system.

1.5 Real-time Executive

Fortunately, real-time operating systems, or real-time executives, serve as a cornerstone on which to build the application system. A real-time multitasking executive allows an application to be cast into a set of logical, autonomous processes or tasks which become quite manageable. Each task is internally synchronous, but different tasks execute independently, resulting in an asynchronous processing stream. Tasks can be dynamically paused for many reasons resulting in a different task being allowed to execute for a period of time. The executive also provides an interface to other system components such as interrupt handlers and device drivers. System components may request the executive to allocate and coordinate resources, and to wait for and trigger synchronizing conditions. The executive system calls effectively extend the CPU instruction set to support efficient multitasking. By causing tasks to travel through well-defined state transitions, system calls permit an application to demand-switch between tasks in response to real-time events.

By properly grouping stimuli responses into separate tasks a system can now asynchronously switch between independent streams of execution. This allows the system to directly respond to external stimuli as they occur, as well as meet critical performance specifications that are typically measured by guaranteed response time and transaction throughput. The multiprocessor extensions of RTEMS provide the features necessary to manage the extra requirements introduced by a system distributed across several processors. It removes the physical barriers of processor boundaries from the world of the system designer, enabling more critical aspects of the system to receive the required attention. Such a system, based on an efficient real-time, multiprocessor executive, is a more realistic model of the outside world or environment for which it is designed. As a result, the system will always be more logical, efficient, and reliable.

By using the directives provided by RTEMS, the real-time applications developer is freed from the problem of controlling and synchronizing multiple tasks and processors. In addition, one need not develop, test, debug, and document routines to manage memory, pass messages, or provide mutual exclusion. The developer is then able to concentrate solely on the application. By using standard software components, the time and cost required to develop sophisticated real-time applications are significantly reduced.

QUICK START

Follow the sections of this chapter step by step to get started developing applications on top of RTEMS.

2.1 Prepare Your Host Computer

The *host computer* is a computer you use to develop applications. It runs all your tools, editors, documentation viewers, etc. To get started with RTEMS development you need tools from your host's operating system to build the RTEMS tool suite from source. This is not a one-click installation process, but there are *good reasons* (page 239) to build everything from source. You need a native C, C++ and Python development environment. Please make sure that you can build native C/C++ applications on your host computer. You must be able to build native Python C modules. Usually, you have to install a Python development package for this. Please have a look at the *Host Computer* (page 39) chapter for the gory details. In particular *Microsoft Windows* (page 46) user should do this.

2.2 Choose an Installation Prefix

You will see the term *prefix* referred to throughout this documentation and in a wide number of software packages you can download from the internet. It is also used in the [GNU Coding Standard](#). A *prefix* is the path on your host computer a software package is installed under. Packages that have a prefix will place all parts under the prefix path. Packages for your host computer typically use a default prefix of `/usr/local` on FreeBSD and Linux.

You have to select a prefix for your installation. You will build and install the RTEMS tool suite, an RTEMS kernel for a BSP and you may build and install third party libraries. You can build them all as a stack with a single prefix or you can

The RTEMS tool suite consists of a cross tool chain (Binutils, GCC, GDB, Newlib, etc.) for your target architecture and *other tools* (page 199) provided by the RTEMS Project. The RTEMS

You build and install the tool suite with the *RTEMS Source Builder (RSB)* (page 237). By default, the RSB will start the prefix path with a host operating system specific path plus `rtems` plus the RTEMS version, e.g. `/opt/rtems/5` on Linux and `/usr/local/rtems/5` on FreeBSD and macOS.

It is strongly recommended to run the RSB as a *normal user* and not with *root* privileges (also known as *super user* or *Administrator*). You have to make sure that your normal user has sufficient privileges to create files and directories under the prefix. For example, you can create a directory `/opt/rtems` and give it to a developer group with read, write and execute permissions. Alternatively, you can choose a prefix in your home directory, e.g. `$HOME/rtems/5` or with a project-specific component `$HOME/project-x/rtems/5`. For more ideas, see the *project sandboxing* (page 82) section. In this quick start chapter, we will choose `$HOME/quick-start/rtems/5` for the RTEMS tool suite prefix.

Warning: The prefix must not contain space characters.

2.3 Obtain the Sources

You have considered and chosen a suitable installation prefix in the previous section. We have chosen `$HOME/quick-start/rtems/5` as the installation prefix.

You need at least two source archives or Git repositories to work with RTEMS. You can download the source archives for a released RTEMS version or you can clone Git repositories to get all versions of RTEMS including the development head.

We will clone the Git repositories into `$HOME/quick-start/src`.

```
1 mkdir -p $HOME/quick-start/src
2 cd $HOME/quick-start/src
3 git clone git://git.rtems.org/rtems-source-builder.git rsb
4 git clone git://git.rtems.org/rtems.git
```

The rsb repository clone contains the *RTEMS Source Builder (RSB)* (page 237). We clone it into rsb to get shorter paths during the tool suite build. The rtems repository clone contains the RTEMS sources. These two repositories are enough to get started. There are [more repositories](#) available.

Alternatively, you can download the source archives of a released RTEMS version.

```
1 mkdir -p $HOME/quick-start/src
2 cd $HOME/quick-start/src
3 curl https://ftp.rtems.org/pub/rtems/releases/4.11/4.11.3/rtems-4.11.3.tar.xz | tar xJf -
4 curl https://ftp.rtems.org/pub/rtems/releases/4.11/4.11.3/rtems-source-builder-4.11.3.tar.
  ↪xz | tar xJf -
```

This quick start chapter focuses on working with the Git repository clones since this gives you some flexibility. You can switch between branches to try out different RTEMS versions. You have access to the RTEMS source history. The RTEMS Project welcomes contributions. The Git repositories enable you to easily create patches and track local changes. If you prefer to work with archives of a released RTEMS version, then simply replace the version number 5 used throughout this chapter with the version number you selected, e.g. 4.11.

2.4 Install the Tool Suite

You have chosen an installation prefix and cloned two RTEMS repositories in the previous sections. We have chosen `$HOME/quick-start/rtems/5` as the installation prefix and cloned the repositories in `$HOME/quick-start/src`.

You must select the *target architecture* (page 87) for which you need a tool suite. In this quick start chapter we choose *sparc-rtems5*. The *sparc-rtems5* is the tool suite name for the SPARC architecture and RTEMS version 5. The tool suite for RTEMS and the RTEMS sources are tightly coupled. For example, do not use a RTEMS version 5 tool suite with RTEMS version 4.11 sources and vice versa. We use the RSB in two steps. The first step is to download additional sources and patches. Afterwards, you could disconnect your host computer from the internet. It is no longer required to work with RTEMS.

```
1 cd $HOME/quick-start/src/rsb/rtems
2 ../source-builder/sb-set-builder --source-only-download 5/rtems-sparc
```

This command should output something like this (omitted lines are denoted by ...):

```
1 RTEMS Source Builder - Set Builder, 5 (98588a55961a)
2 warning: exe: absolute exe found in path: (__unzip) /usr/local/bin/unzip
3 Build Set: 5/rtems-sparc
4 ...
5 download: https://ftp.gnu.org/gnu/gcc/gcc-7.4.0/gcc-7.4.0.tar.xz -> sources/gcc-7.4.0.tar.
  ↳XZ
6 ...
7 Build Sizes: usage: 0.000B total: 141.738MB (sources: 141.559MB, patches: 183.888KB,
  ↳installed 0.000B)
8 Build Set: Time 0:01:17.613061
```

If you encounter errors in the first step, check your internet connection, firewall settings, virus scanners and the availability of the download servers. The seconds step is to build and install the tool suite.

```
1 cd $HOME/quick-start/src/rsb/rtems
2 ../source-builder/sb-set-builder --prefix=$HOME/quick-start/rtems/5 5/rtems-sparc
```

This command should output something like this (omitted lines are denoted by ...):

```
1 RTEMS Source Builder - Set Builder, 5 (98588a55961a)
2 warning: exe: absolute exe found in path: (__unzip) /usr/local/bin/unzip
3 Build Set: 5/rtems-sparc
4 ...
5 config: tools/rtems-gcc-7.4.0-newlib-3e24fbf6f.cfg
6 package: sparc-rtems5-gcc-7.4.0-newlib-3e24fbf6f-x86_64-freebsd12.0-1
7 building: sparc-rtems5-gcc-7.4.0-newlib-3e24fbf6f-x86_64-freebsd12.0-1
8 sizes: sparc-rtems5-gcc-7.4.0-newlib-3e24fbf6f-x86_64-freebsd12.0-1: 4.651GB (installed:
  ↳879.191MB)
9 cleaning: sparc-rtems5-gcc-7.4.0-newlib-3e24fbf6f-x86_64-freebsd12.0-1
10 ....
11 Build Sizes: usage: 5.618GB total: 1.105GB (sources: 141.559MB, patches: 185.823KB,
  ↳installed 989.908MB)
12 Build Set: Time 0:22:02.262039
```

In case the seconds step was successful, you can check if for example the cross C compiler works with the following command:

```
1 $HOME/quick-start/rtems/5/bin/sparc-rtems5-gcc --version --verbose
```

This command should output something like below. In this output the actual prefix path was replaced by \$PREFIX. The compiled by line depends on the native C++ compiler of your host computer. In the output you see the Git hash of the RSB. This helps you to identify the exact sources which were used to build the cross compiler of your RTEMS tool suite.

```
1 Using built-in specs.
2 COLLECT_GCC=$PREFIX/bin/sparc-rtems5-gcc
3 COLLECT_LTO_WRAPPER=$PREFIX/bin/./libexec/gcc/sparc-rtems5/7.4.0/lto-wrapper
4 sparc-rtems5-gcc (GCC) 7.4.0 20181206 (RTEMS 5, RSB_
  ↳ 98588a55961a92f5d27bfd756dfc9e31b2b1bf98, Newlib 3e24fbf6f)
5 Copyright (C) 2017 Free Software Foundation, Inc.
6 This is free software; see the source for copying conditions. There is NO
7 warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
8
9
10 Target: sparc-rtems5
11 Configured with: ../gcc-7.4.0/configure --prefix=$PREFIX --bindir=$PREFIX/bin --exec_
  ↳ prefix=$PREFIX --includedir=$PREFIX/include --libdir=$PREFIX/lib --libexecdir=$PREFIX/
  ↳ libexec --mandir=$PREFIX/share/man --infodir=$PREFIX/share/info --datadir=$PREFIX/share_
  ↳ --build=x86_64-freebsd12.0 --host=x86_64-freebsd12.0 --target=sparc-rtems5 --disable-
  ↳ libstdcxx-pch --with-gnu-as --with-gnu-ld --verbose --with-newlib --disable-nls --
  ↳ without-included-gettext --disable-win32-registry --enable-version-specific-runtime-
  ↳ libs --disable-lto --enable-newlib-io-c99-formats --enable-newlib-iconv --enable-newlib-
  ↳ iconv-encodings=big5,cp775,cp850,cp852,cp855,cp866,euc_jp,euc_kr,euc_tw,iso_8859_1,iso_
  ↳ 8859_10,iso_8859_11,iso_8859_13,iso_8859_14,iso_8859_15,iso_8859_2,iso_8859_3,iso_8859_
  ↳ 4,iso_8859_5,iso_8859_6,iso_8859_7,iso_8859_8,iso_8859_9,iso_ir_111,koi8_r,koi8_ru,koi8_
  ↳ u,koi8_uni,ucs_2,ucs_2_internal,ucs_2be,ucs_2le,ucs_4,ucs_4_internal,ucs_4be,ucs_4le,us_
  ↳ ascii,utf_16,utf_16be,utf_16le,utf_8,win_1250,win_1251,win_1252,win_1253,win_1254,win_
  ↳ 1255,win_1256,win_1257,win_1258 --enable-threads --disable-plugin --enable-libgomp --
  ↳ enable-languages=c,c++
12 Thread model: rtems
13 gcc version 7.4.0 20181206 (RTEMS 5, RSB 98588a55961a92f5d27bfd756dfc9e31b2b1bf98, Newlib_
  ↳ 3e24fbf6f) (GCC)
14 COLLECT_GCC_OPTIONS='--version' '-v' '-mcpu=v7'
15 $PREFIX/bin/./libexec/gcc/sparc-rtems5/7.4.0/cc1 -quiet -v -iprefix $PREFIX/bin/./lib/
  ↳ gcc/sparc-rtems5/7.4.0/ help-dummy -quiet -dumpbase help-dummy -mcpu=v7 -auxbase help-
  ↳ dummy -version --version -o /tmp//ccuAN1wc.s
16 GNU C11 (GCC) version 7.4.0 20181206 (RTEMS 5, RSB_
  ↳ 98588a55961a92f5d27bfd756dfc9e31b2b1bf98, Newlib 3e24fbf6f) (sparc-rtems5)
17     compiled by GNU C version 4.2.1 Compatible FreeBSD Clang 6.0.1 (tags/RELEASE_601/
  ↳ final 335540), GMP version 6.1.0, MPFR version 3.1.4, MPC version 1.0.3, isl version_
  ↳ isl-0.16.1-GMP
18
19 GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
20 COLLECT_GCC_OPTIONS='--version' '-v' '-mcpu=v7'
21 $PREFIX/bin/./lib/gcc/sparc-rtems5/7.4.0/../../../../sparc-rtems5/bin/as -v -s --
  ↳ version -o /tmp//ccFVGRAa.o /tmp//ccuAN1wc.s
22 GNU assembler version 2.32 (sparc-rtems5) using BFD version (GNU Binutils) 2.32
23 GNU assembler (GNU Binutils) 2.32
24 Copyright (C) 2019 Free Software Foundation, Inc.
25 This program is free software; you may redistribute it under the terms of
26 the GNU General Public License version 3 or later.
27 This program has absolutely no warranty.
28 This assembler was configured for a target of `sparc-rtems5'.
```

(continues on next page)

(continued from previous page)

```

29 COMPILER_PATH=$PREFIX/bin/./libexec/gcc/sparc-rtems5/7.4.0/:$PREFIX/bin/./libexec/gcc/:
   ↳ $PREFIX/bin/./lib/gcc/sparc-rtems5/7.4.0/../../../../sparc-rtems5/bin/
30 LIBRARY_PATH=$PREFIX/bin/./lib/gcc/sparc-rtems5/7.4.0/:$PREFIX/bin/./lib/gcc/:$PREFIX/
   ↳ bin/./lib/gcc/sparc-rtems5/7.4.0/../../../../sparc-rtems5/lib/
31 COLLECT_GCC_OPTIONS='--version' '-v' '-mcpu=v7'
32 $PREFIX/bin/./libexec/gcc/sparc-rtems5/7.4.0/collect2 --version $PREFIX/bin/./lib/gcc/
   ↳ sparc-rtems5/7.4.0/../../../../sparc-rtems5/lib/crt0.o -L$PREFIX/bin/./lib/gcc/sparc-
   ↳ rtems5/7.4.0 -L$PREFIX/bin/./lib/gcc -L$PREFIX/bin/./lib/gcc/sparc-rtems5/7.4.0/../../../../
   ↳ ../../sparc-rtems5/lib /tmp/ccFVgRAa.o -lgcc -lc -lgcc
33 collect2 version 7.4.0 20181206 (RTEMS 5, RSB 98588a55961a92f5d27bfd756dfc9e31b2b1bf98,
   ↳ Newlib 3e24fbf6f)
34 $PREFIX/bin/./lib/gcc/sparc-rtems5/7.4.0/../../../../sparc-rtems5/bin/ld --version
   ↳ $PREFIX/bin/./lib/gcc/sparc-rtems5/7.4.0/../../../../sparc-rtems5/lib/crt0.o -L$PREFIX/
   ↳ bin/./lib/gcc/sparc-rtems5/7.4.0 -L$PREFIX/bin/./lib/gcc -L$PREFIX/bin/./lib/gcc/
   ↳ sparc-rtems5/7.4.0/../../../../sparc-rtems5/lib /tmp/ccFVgRAa.o -lgcc -lc -lgcc
35 GNU ld (GNU Binutils) 2.32
36 Copyright (C) 2019 Free Software Foundation, Inc.
37 This program is free software; you may redistribute it under the terms of
38 the GNU General Public License version 3 or (at your option) a later version.
39 This program has absolutely no warranty.
40 COLLECT_GCC_OPTIONS='--version' '-v' '-mcpu=v7'

```

2.5 Bootstrap the RTEMS Sources

You installed the tool suite in your installation prefix and cloned two RTEMS repositories in the previous sections. We installed the tool suite in `$HOME/quick-start/rtems/5` and cloned the repositories in `$HOME/quick-start/src`.

If you use source archives of a released RTEMS version, then you can skip this section.

Before you can build a *Board Support Package (BSP)* (page 91) for your target hardware, you have to bootstrap the build system in the RTEMS sources. This is only necessary, if you use a Git repository clone of the RTEMS sources. You have to do this after a fresh repository clone and sometimes after build system file updates (e.g. after a `git pull`). If you are not a build system expert, then do the bootstrap after each update of build system files. This is a bit annoying, but improving the build system is a complex and time consuming undertaking. Feel free to help the RTEMS Project to improve it. For the bootstrap it is important that the right version of Autotools (autoconf and automake) are in your `$PATH`. The right version of Autotools is shipped with the RTEMS tool suite you already installed.

```
1 cd $HOME/quick-start/src/rtems
2 export PATH=$HOME/quick-start/rtems/5/bin:"$PATH"
3 ./bootstrap -c
4 $HOME/quick-start/src/rsb/source-builder/sb-bootstrap
```

These commands should output something like this (omitted lines are denoted by ...):

```
1 removing automake generated Makefile.in files
2 removing configure files
3 removing aclocal.m4 files
4 $ $HOME/quick-start/src/rsb/source-builder/sb-bootstrap
5 RTEMS Source Builder - RTEMS Bootstrap, 5 (f07504d27192)
6   1/120: autoreconf: configure.ac
7   2/120: autoreconf: c/configure.ac
8   3/120: autoreconf: c/src/configure.ac
9   4/120: autoreconf: c/src/lib/libbsp/arm/configure.ac
10  ...
11 120/120: autoreconf: testsuites/tmtests/configure.ac
12 Bootstrap time: 0:00:48.744222
```


2.6 Build a Board Support Package (BSP)

You installed the tool suite in your installation prefix, cloned two RTEMS repositories and bootstrapped the RTEMS sources in the previous sections. We installed the tool suite in `$HOME/quick-start/rtems/5` and cloned the repositories in `$HOME/quick-start/src`. We also bootstrapped the RTEMS sources.

You are now able to build *Board Support Packages (BSPs)* (page 91) for all architectures where you have an RTEMS tool suite installed. To build applications on top of RTEMS, you have to configure, build and install a BSP for your target hardware. To select a proper BSP for your target hardware consult the *BSPs* (page 91) chapter. We select the *erc32* BSP.

We configure, build and install the BSP in four steps. The first step is to create a build directory. It must be separate from the RTEMS source directory. We use `$HOME/quick-start/build/b-erc32`.

```
1 mkdir -p $HOME/quick-start/build/b-erc32
```

The second step is to configure the BSP. There are various configuration options available. Some configuration options are BSP-specific. Prepend the RTEMS tool suite binary directory to your `$PATH` throughout the remaining steps.

```
1 cd $HOME/quick-start/build/b-erc32
2 export PATH=$HOME/quick-start/rtems/5/bin:$PATH
3 $HOME/quick-start/src/rtems/configure \
4   --prefix=$HOME/quick-start/rtems/5 \
5   --enable-maintainer-mode \
6   --target=sparc-rtems5 \
7   --enable-rtemsbsp=erc32 \
8   --enable-tests
```

This command should output something like this (omitted lines are denoted by ...):

```
1 checking for gmake... gmake
2 checking for RTEMS Version... 5.0.0
3 checking build system type... x86_64-unknown-freebsd12.0
4 checking host system type... x86_64-unknown-freebsd12.0
5 checking target system type... sparc-unknown-rtems5
6 ...
7 config.status: creating Makefile
8
9 target architecture: sparc.
10 available BSPs: erc32.
11 'gmake all' will build the following BSPs: erc32.
12 other BSPs can be built with 'gmake RTEMS_BSP="bsp1 bsp2 ..."'
13
14 config.status: creating Makefile
```

Building the BSP is the third step.

```
1 cd $HOME/quick-start/build/b-erc32
2 make
```

This command should output something like this (omitted lines are denoted by ...). In this output the base directory `$HOME/quick-start` was replaced by `$BASE`.

```

1 Configuring RTEMS_BSP=erc32
2 checking for gmake... gmake
3 checking build system type... x86_64-unknown-freebsd12.0
4 checking host system type... sparc-unknown-rtems5
5 checking rtems target cpu... sparc
6 checking for a BSD-compatible install... /usr/bin/install -c
7 checking whether build environment is sane... yes
8 checking for sparc-rtems5-strip... sparc-rtems5-strip
9 checking for a thread-safe mkdir -p... $BASE/src/rtems/c/src/../../install-sh -c -d
10 checking for gawk... no
11 checking for mawk... no
12 checking for nawk... nawk
13 checking whether gmake sets $(MAKE)... yes
14 checking whether to enable maintainer-specific portions of Makefiles... yes
15 checking for RTEMS_BSP... erc32
16 checking whether CPU supports libposix... yes
17 configure: setting up make/custom
18 configure: creating make/erc32.cache
19 gmake[3]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'
20 ...
21 sparc-rtems5-gcc -mcpu=cypress -O2 -g -ffunction-sections -fdata-sections -Wall -
↳ Wmissing-prototypes -Wimplicit-function-declaration -Wstrict-prototypes -Wnested-
↳ externs -B./../../lib/libbsp/sparc/erc32 -B$BASE/src/rtems/bsps/sparc/erc32/start -
↳ specs bsp_specs -qrtems -L./../../cpukit -L$BASE/src/rtems/bsps/sparc/shared/start -Wl,-
↳ wrap=printf -Wl,--wrap=puts -Wl,--wrap=putchar -Wl,--gc-sections -o spwkspace.exe
↳ spwkspace/spwkspace-init.o ./../../lib/libbsp/sparc/erc32/librtemsbsp.a ./../../cpukit/
↳ librtemscpu.a
22 gmake[5]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32/testsuites/sptests'
23 gmake[4]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32/testsuites'
24 gmake[3]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'
25 gmake[2]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'
26 gmake[1]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c'
27 gmake[1]: Entering directory '$BASE/build/b-erc32'
28 gmake[1]: Nothing to be done for 'all-am'.
29 gmake[1]: Leaving directory '$BASE/build/b-erc32'

```

The last step is to install the BSP.

```

1 cd $HOME/quick-start/build/b-erc32
2 make install

```

This command should output something like this (omitted lines are denoted by ...). In this output the base directory \$HOME/quick-start was replaced by \$BASE.

```

1 Making install in sparc-rtems5/c
2 gmake[1]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c'
3 Making install in .
4 gmake[2]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c'
5 gmake[3]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c'
6 gmake[3]: Nothing to be done for 'install-exec-am'.
7 gmake[3]: Nothing to be done for 'install-data-am'.
8 gmake[3]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c'
9 gmake[2]: Leaving directory '$BASE/build/b-erc32/sparc-rtems5/c'
10 Making install in erc32
11 gmake[2]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'
12 gmake[3]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32'

```

(continues on next page)

(continued from previous page)

```
13 Making install-am in .
14 Making install-am in cpukit
15 gmake[4]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32/cpukit'
16 gmake[5]: Entering directory '$BASE/build/b-erc32/sparc-rtems5/c/erc32/cpukit'
17 gmake[5]: Nothing to be done for 'install-exec-am'.
18 $BASE/src/rtems/c/src/../../../../cpukit/../../install-sh -c -d '$BASE/rtems/5/sparc-rtems5/erc32/
  ↳ lib/include'
19 ...
20 $BASE/src/rtems/make/Templates/Makefile.lib '$BASE/rtems/5/share/rtems5/make/Templates'
21 $BASE/src/rtems/./install-sh -c -d '$BASE/rtems/5/make/custom'
22 /usr/bin/install -c -m 644 $BASE/src/rtems/make/custom/default.cfg '$BASE/rtems/5/make/
  ↳ custom'
23 gmake[2]: Leaving directory '$BASE/build/b-erc32'
24 gmake[1]: Leaving directory '$BASE/build/b-erc32'
```

2.7 Test a Board Support Package (BSP)

You built a BSP with tests in the previous section. We built the erc32 BSP in `$HOME/quick-start/build/b-erc32`.

You should run the RTEMS test suite on your target hardware. The RTEMS Project provides some support to do this, see the *Testing* (page 157) chapter for the details.

On the erc32 BSP we selected for this quick start chapter this is easy. Just run this command:

```
1 cd $HOME/quick-start/build/b-erc32
2 rtems-test --rtems-bsp=erc32-sis --rtems-tools=$HOME/quick-start/rtems/5 .
```

This command should output something like this (omitted lines are denoted by ...). In this output the base directory `$HOME/quick-start` was replaced by `$BASE`.

```
1 RTEMS Testing - Tester, 5.0.not_released
2 Command Line: $BASE/rtems/5/bin/rtems-test --rtems-bsp=erc32-sis --rtems-tools=$BASE/
  ↳ rtems/5 .
3 Python: 2.7.15 (default, Jan 10 2019, 01:14:47) [GCC 4.2.1 Compatible FreeBSD Clang 6.0.
  ↳ 1 (tags/RELEASE_601/final 335540)]
4 Host: FreeBSD-12.0-RELEASE-p2-amd64-64bit-ELF (FreeBSD Build_FreeBSD12 12.0-RELEASE-p2
  ↳ FreeBSD 12.0-RELEASE-p2 GENERIC amd64 amd64)
5 [ 1/589] p:0 f:0 u:0 e:0 I:0 B:0 t:0 i:0 W:0 | sparc/erc32: dhrystone.
  ↳ exe
6 ...
7 [589/589] p:574 f:0 u:5 e:0 I:0 B:3 t:0 i:0 W:0 | sparc/erc32: tmtimer01.
  ↳ exe
8
9 Passed:          580
10 Failed:         0
11 User Input:     5
12 Expected Fail:  0
13 Indeterminate:  0
14 Benchmark:      3
15 Timeout:        1
16 Invalid:        0
17 Wrong Version:  0
18 Wrong Build:    0
19 Wrong Tools:    0
20 -----
21 Total:          589
22 User Input:
23 monitor.exe
24 termios.exe
25 top.exe
26 fileio.exe
27 capture.exe
28 Benchmark:
29 whetstone.exe
30 linpack.exe
31 dhrystone.exe
32 Timeouts:
33 pppd.exe
34 Average test time: 0:00:00.437773
35 Testing time      : 0:04:17.848557
```

2.8 Build Your Application

TODO

SUPPORT AND CONTRIBUTING

3.1 RTEMS Project Support

3.1.1 Users Mailing List

RTEMS offers a variety of support options and ways to contribute to the project. Users can ask their questions on the [Users Mailing List](#). This is a low frequency mailing list intended for topics related to the use of RTEMS. If you are new to RTEMS, please join the list and ask whatever you want.

3.1.2 Documentation

You find the latest set of manuals at the [Documentation Site](#).

3.1.3 All Mailing Lists

We have several mailing lists for RTEMS users and developers:

- [Announce Mailing List](#): Announcements for major and other project-related issues.
- [Bugs Mailing List](#): Emails generated by the [Bugs Database](#).
- [Developers Mailing List](#): For developers of RTEMS itself.
- [Build Logs](#): Results from building and testing of RTEMS.
- [Users Mailing List](#): For users of RTEMS.
- [Version Control Mailing List](#): Commits to the RTEMS Project repositories.

3.1.4 IRC

RTEMS IRC is available on the Freenode network. See the [Freenode](#) web site for details on connecting, selecting a nickname, and general usage tips. If you are new to IRC it is recommended reading.

There is currently only one IRC channel available for RTEMS:

#rtems

This is a general channel for all things RTEMS. You can just hang out with other RTEMS users and developers to talk about RTEMS, using RTEMS or to make contact with other RTEMS users.

The #rtems channel is logged. You can find the logs at:

<http://www.rtems.org/irclogs/>

You can search the logs using Google by adding

site:rtems.org inurl:irclogs

to your search terms.

3.2 Report Bugs

The RTEMS Project uses a ticket system to deal with bugs, organize enhancement requests, and manage small tasks and projects. You can [submit a bug report](#) to the RTEMS Project ticket system. Before you do this, please read the following information. Good bug reports are more likely to get addressed quickly. If you have patches not specifically related to bugs or existing tickets, please have a look at the *Contributing* (page 32) guidelines.

3.2.1 Search for Existing Bugs

You can [search for existing bugs](#) in the RTEMS Project ticket system. Please try to avoid duplicate bug reports and search for an existing bug before you report a new bug. If you are unsure, please ask on the [Users Mailing List](#) and we will help you sort it out.

3.2.2 Not RTEMS Bugs

Some issues appear to be an RTEMS bug to you, but are actually the intended behaviour or in the scope of other projects.

- Bugs in the assembler, the linker, or the C library (Newlib) are not RTEMS bugs. These are separate projects, with separate mailing lists and different bug reporting procedures. The RTEMS Project is happy to work with you and those projects to resolve the problem but we must work with those projects. Bugs in those products must be addressed in the corresponding project. Report [assembler, linker, and GDB bugs to sourceware.org](#), [compiler bugs to GCC](#), and [Newlib bugs to the Newlib mailing list](#). If the bug was fixed, then you can update the *Source Builder* (page 237) to pick up the fix.
- Questions about the correctness or the expected behaviour of programming language constructs or calls to library routines that are not part of RTEMS belong somewhere else.
- The POSIX standard does *not* specify the default set of thread attributes. Thus, when passing a NULL for attributes to `pthread_create()`, the application is not guaranteed any particular thread behaviour.
- The defaults for all [RTEMS Application Configuration](#) parameters are intentionally small. Thus, it is common for RTEMS tasking and file related calls to return errors indicating out of resources until the configuration parameters are properly tuned for the application. For example, there are only three file descriptors available by default: `stdin`, `stdout`, and `stderr`. Any attempt to open a socket or file will fail unless more file descriptors are configured.
- When first developing a BSP, many users encounter an unexpected interrupt or exception immediately upon completion of RTEMS initialization. This occurs because interrupts are disabled during RTEMS initialization and are automatically initialized as part of switching to the first task. The interrupted user code will be in either `_CPU_Context_switch()` or `_Thread_Handler()`. This indicates that an interrupt source has not been properly initialized or masked.
- Some users encounter a random reset during BSP initialization. This usually indicates that the board has a watchdog timer that is not being properly serviced during the BSP initialization.
- Bugs in releases or snapshots of RTEMS not issued by the RTEMS Project. Report them to whoever provided you with the release.

3.2.3 Good Bug Reports

Please open the page to [submit a bug](#) to the RTEMS Project ticket system and follow the guidelines below to write a good bug report.

- Provide a useful single line **Summary**.
- Use [WikiFormatting](#) to structure the information you provide. It does help the readability of the information you provide.
- Add a description of the expected behaviour. The expected behaviour may be obvious to you, but maybe not to someone else reading the bug report.
- Add a description of the actual undesired behaviour.
- Name the *target hardware* (page 85) (processor architecture, chip family or model, and *BSP* (page 91)) in the description.
- Add the toolchain version used (GCC, Binutils, Newlib) to the description. Custom toolchain builds are discouraged. To avoid problems caused by custom builds of the toolchain, please build your toolchain with the *Source Builder* (page 237). If you use a custom build of the toolchain, then try to reproduce the bug first using a toolchain built by the RSB.
- Provide the configuration options used to build the RTEMS BSP in the description. This helps to reproduce the issue.
- Make the bug reproducible by others. Write a self-contained piece of source code which can be compiled and reproduces the bug. Avoid adding assembly files (*.s) produced by the compiler, or any binary files, such as object files, executables, core files, or precompiled header files. If it is difficult or time consuming to reproduce the bug, then it may not get the attention it deserves from others. Developing and debugging real-time embedded systems can be difficult. Exercise caution in reporting an error that occurs only some of the times a certain program is executed, such that retrying a sufficient number of times results in a successful compilation; this is often a symptom of a hardware problem or application issue, not of a RTEMS bug (sorry). We do recognise that sometimes a timing bug will exist in RTEMS, but we want you to exercise due diligence before pointing fingers.
- Only when your bug report requires multiple source files to be reproduced should you attach an archive. Otherwise, the uploaded individual source file or diff should contain the minimal source code needed to reproduce the bug. In any case, make sure the above are included in the body of your bug report as plain text, even if needlessly duplicated as part of an archive.
- Please try to reproduce the bug on the current Git master. If it is not reproducible on the Git master, you should figure out if the bug was already fixed. You can search the existing bugs once again, ask on the [Users Mailing List](#), or do a Git bisect to find a commit which fixed the bug.
- Include only information relevant to the bug.
- Write separate bug reports for different bugs.
- Select a **Type** for the ticket.
 - Use defect for a bug.
 - Use enhancement for a feature request in the software or an addition to the documentation.

- Note *infra* is used to report issues with the RTEMS servers at OSUOSL.
- Select a **Version** for the ticket. This should be the first RTEMS version which is affected by this bug. If this is the current Git master branch use the version of the next release. Please provide the exact version of RTEMS in the description. If you use an RTEMS release, then the release number. If you use a Git clone, then the commit hash. The commit hash should be present in an RTEMS Project repository. Commit hashes of private branches are not interesting.
- Select a **Component** for the ticket. Use unspecified if you are unsure.
- Select a **Severity** for the ticket.
- The fields **Milestone** and **Priority** will be most likely set by an RTEMS maintainer.
- You can relate your new bug to existing bugs through the **Blocked by** and **Blocking** fields.
- If you have any external files, such as screenshots or examples, please *attach* these as files to the ticket. *Do not use external hosting* because if you do use external hosting, then our historical record is broken when those files are no longer available.
- Some fields should only be set by the maintainers, as it is not always clear what they should be set to. Feel free to make your own choices.

When you have checked that your report meets the criteria for a good bug report, please click on the `Create ticket` button to submit it to the RTEMS Project ticket system.

If you fail to supply enough information for a bug report to be reproduced, someone will probably ask you to post additional information. In this case, please post the additional information and not just to the person who requested it, unless explicitly told so.

3.2.4 Nobody Fixes my Bug

Sometimes, you may notice that after some time your bug report gets no attention and the bug is not magically fixed. This may have several reasons

- the bug report is incomplete or confusing,
- the target hardware is not available to others,
- the bug is not reproducible on the Git master,
- the bug is not reproducible at all,
- the RTEMS version is quite old and no longer used by RTEMS maintainers, or
- fixing the bug has a low priority for others.

Please note that you do not have a service contract with the RTEMS Project. The RTEMS Project is run by volunteers and persons who take care about how RTEMS performs in their application domain. If your bug does not affect the interest of someone else, then you should try to fix the bug on your own, see the *Contributing* (page 32) guidelines. To change the priorities of others with respect to your bug, you may refer to the *Commercial Support Services* (page 37).

3.3 Contributing

3.3.1 How to Contribute?

You can contribute to the RTEMS Project in various ways, for example:

- participation in mailing list discussions, helping other users
- documentation updates, clarifications, consolidation, fixes
- bug fixes, bug report consolidation
- new BSPs
- new device drivers
- new CPU (processor architecture) ports
- improvements in the existing code base (code size, code clarity, test coverage, performance optimizations)
- new features
- RTEMS Tools improvements

Most contributions will end up in patches of the RTEMS source code or documentation sources. The patch integration into the RTEMS repositories is done through a *patch review process* (page 34) on the [Developers Mailing List](#).

3.3.2 Preparing and Sending Patches

The RTEMS Project uses Git for version control. Git has a special command to prepare patches intended for mailing lists: `git format-patch`. Create logically connected patches as a patch series ideally accompanied by a cover letter (`--cover-letter` option). You can send patches via email through a Git command: `git send-email`.

3.3.3 Checklist for Patches

Check the following items before you send a patch to the [Developers Mailing List](#):

- The author name of the patch is your full name.
- The author email of the patch is your valid email address.
- The licence conditions of the contributed content allow an integration into the RTEMS code base.
- If you are the copyright holder of the entire patch content, then please contribute it under the [BSD-2-Clause](#) license. For documentation use [CC BY-SA 4.0](#).
- Make sure you have a pregnant subject which does not exceed 50 characters in one line. Use a “topic: The pregnant subject” style. A topic could be the main component of patch. Just have a look at existing commit messages.
- The patch has a good commit message. It should describe the reason for the change. It should list alternative approaches and why they were not chosen.

- The code changes honour the coding style. At least do your changes in the style of the surrounding code.
- The patch contains no spelling mistakes and grammar errors.
- The patch is easy to review. It changes one thing only and contains no unrelated changes. Format changes should be separated from functional changes.
- If the patch corresponds to a ticket, it should have “Close #X.” or “Update #X.” as the last line in the commit message to update status once it is committed to the repository.
- The patch builds. All RTEMS tests link with this patch.
- The patch does not introduce new compiler warnings.
- The patch does not introduce new test failures in existing tests.

3.3.4 Patch Review Process

Patches sent to the [Developers Mailing List](#) undergo a *patch review process* (page 34). Once a patch series is accepted for integration into the RTEMS code base it is committed by an [RTEMS maintainer](#). The maintainers are usually quite busy with all sorts of stuff. If you do not get a response to a patch series submission to the mailing list after five work days, please send a reminder. It helps if you follow the *Checklist for Patches* (page 32). An easy to review patch series which meets the quality standards of the RTEMS Project will be more likely get integrated quickly.

3.3.5 Why Contribute?

If you are writing a major extension to RTEMS, such as a port to a new CPU family (processor architecture) or model, a new target board, a major rewrite of some existing component, or adding some missing functionality, please keep in mind the importance of keeping other developers informed. Part of being a good cooperating member of the RTEMS development team is the responsibility to consider what the other developers need in order to work effectively.

Nobody likes to do a lot of work and find it was duplicated effort. So when you work on a major new feature, you should tell [Users Mailing List](#) what you are working on, and give occasional reports of how far you have come and how confident you are that you will finish the job. This way, other developers (if they are paying attention) will be aware which projects would duplicate your effort, and can either join up with you, or at least avoid spending time on something that will be unnecessary because of your work. If, for whatever reason, you are not in a position to publicly discuss your work, please at least privately let an [RTEMS maintainer](#) know about it so they can look out for duplicated effort or possible collaborators.

You should also monitor the [Users Mailing List](#) and [Developers Mailing List](#) to see if someone else mentions working on a similar project to yours. If that happens, speak up!

If you are thinking of taking a contract to develop changes under a temporary delayed-release agreement, please negotiate the agreement so that you can give progress reports before the release date, even though you cannot release the code itself. Also please arrange so that, when the agreed-on date comes, you can release whatever part of the job you succeeded in doing, even if you have not succeeded in finishing it. Someone else may be able to finish the job.

Many people have done RTEMS ports or BSPs on their own, to a wide variety of processors, without much communication with the RTEMS development team. However, much of this

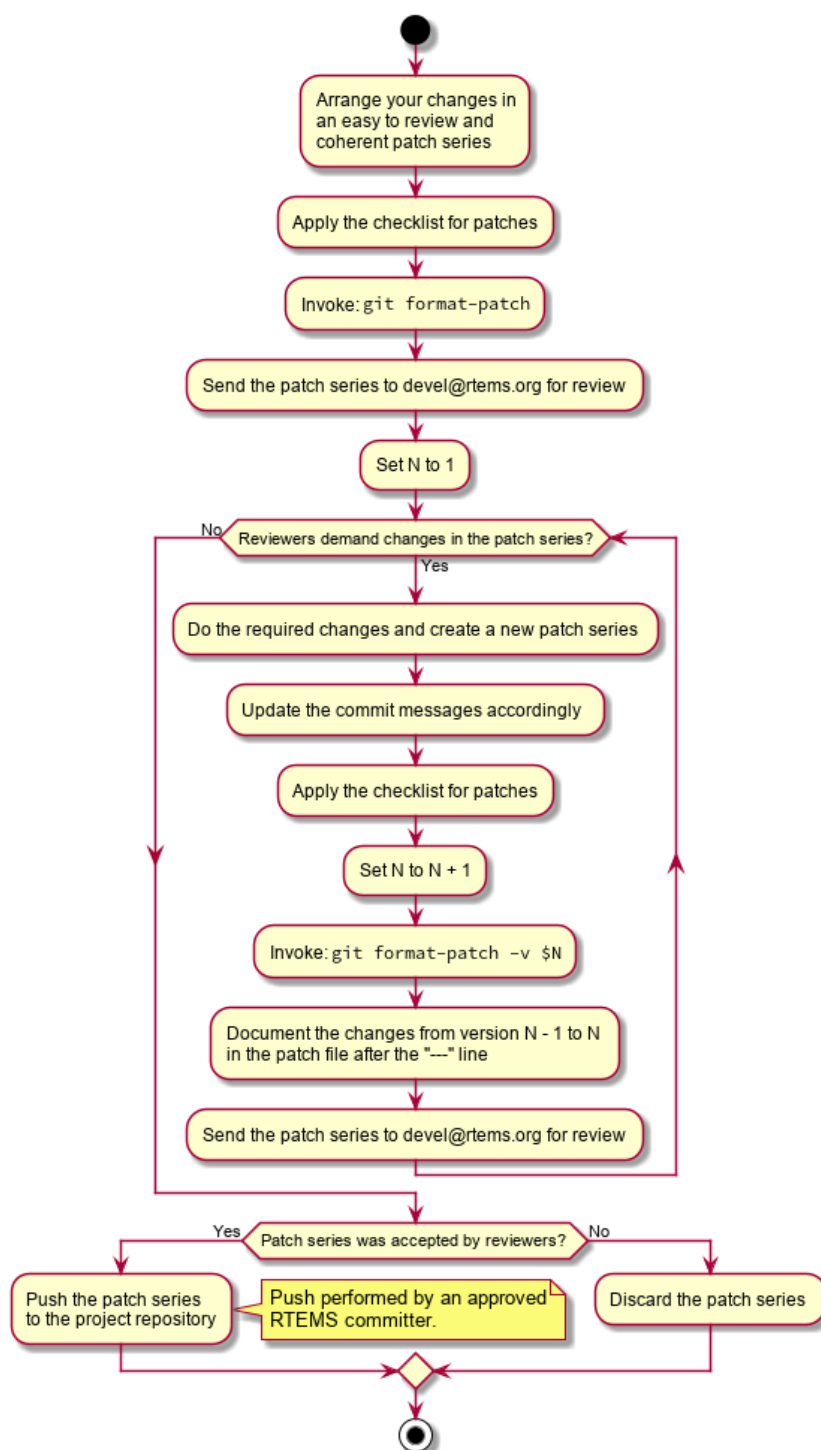


Fig. 1: RTEMS Patch Review Process.

work has been lost over time, or have proven very hard to integrate. So, what we are asking is that, to the maximum extent possible, you communicate with us as early on and as much as possible.

3.3.6 Common Questions and Answers

Here are some questions RTEMS porters may have with our answers to them. While the focus here is on new ports and BSPs, we believe that the issues are similar for other RTEMS development efforts including student efforts to implement new algorithmic optimizations.

Our engineers understand our target environment better than anyone else, and we have a tight schedule. Why should we work with the RTEMS developers, when we can get the code out faster by whacking it out on our own?

You understand your target environment better than anyone else. However, the RTEMS developers understand RTEMS better than anyone else; furthermore, the RTEMS developers tend to have a wide breadth of experience across a large number of processors, boards, peripherals, and application domains. It has been our experience that few problems encountered in embedded systems development are unique to a particular processor or application. The vast majority of the time an issue that arises in one project has also shown up in other projects.

The intimate knowledge of RTEMS internals as well as a wide breadth of embedded systems knowledge means that there is a good chance that at least one RTEMS developer has already addressed issues you are likely to face when doing your port, BSP, or application. The developers can help guide you towards a workable long term solution, possibly saving you significant time in your development cycle.

If getting the sources into the official RTEMS distributions is one of your goals, then engaging other RTEMS developers early will also likely shorten your development time. By interacting as early as possible you are more likely to write code which can be easily accepted into the official sources when you are finished. If you wait until you think you are done to begin interacting with the RTEMS team, you might find that you did some things wrong and you may have to rewrite parts of your RTEMS port, which is a waste of your valuable time.

Why should we care if our port is integrated into the official RTEMS sources? We can distribute it ourselves to whoever is interested.

Yes, the RTEMS licenses allows you to do that. But by doing so, you end up having to maintain that code yourself; this can be a significant effort over time as the RTEMS sources change rapidly.

You also lose the advantage of wider exposure by including your port in the official RTEMS sources maintained by the RTEMS Project. The wider exposure in the RTEMS developer and tester community will help keep your work up to date with the current sources. You may even find that volunteers will run the ever-growing test suite on your port and fix problems during the development cycle – sometimes without your intervention.

It has been our experience that integrated ports tend to ultimately be of better quality and stay up to date from release to release.

Why should we communicate up front? We are happy to let the RTEMS developers integrate our stuff later.

See above. It will save work for you over both the short and the long term, and it is the right thing to do.

Aspects of my target environment that my application exploits are still under NDA.

Nevertheless, if the target hardware is built of any commercial parts that are generally available including, but not limited to, the CPU or peripherals, then that portion of your work is still of general use. Similarly, if you have written software that adheres to existing API or interface standards, then that portion is also of general use. Our experience is that most embedded applications do utilize a custom mix of hardware and application, but they are built upon layers of hardware and software components that are in no way unique to the project.

If you are porting to an unreleased CPU family or model, then just announcing it is important because other RTEMS users may be planning to use it and some of them may already be trying to port RTEMS on their own. Your customers might be happier to know that your port will eventually be available. Also, there is no requirement that RTEMS include all features or ports at any particular time, so you are encouraged to submit discrete pieces of functionality in stages.

Assume that your processor has some new functionality or peripherals. However that functionality is still covered by NDA, but the basic core architecture is not. It is still to your advantage to go ahead and work with the developers early to provide a “base port” for the CPU family. That base port would only use the publicly available specifications until such time as the NDA is lifted. Once the NDA is lifted you can work with the developers to provide the code necessary to take advantage of the new functionality.

Ultimately, cooperating with the free software community as early as possible helps you by decreasing your development cycle, decreasing your long term maintenance costs and may help raise interest in your processor by having a free compiler implementation available to anyone who wants to take a look.

3.4 Commercial Support Services

The wider RTEMS community has developers and organizations who can provide commercial support services. These services range from training, implementing new features in RTEMS, deployment of RTEMS, helping establish a new project environment for a team, to application and system design.

The RTEMS Project does not endorse or promote any provider of these services and we recommend you use a search engine to locate a suitable provider. If you are unsure please contact a provider and see what is available.

If you develop a new feature or you have someone do this for you we recommend you have the work submitted to the project and merged. Once accepted into the project the work will be maintained as part of the development process within the project and this is a benefit for.

HOST COMPUTER

RTEMS applications are developed using cross-development tools running on a development computer, more often called the host computer. These are typically your desktop machine or a special build server. All RTEMS tools and runtime libraries are built from source on your host machine. The RTEMS Project does not maintain binary builds of the tools. This differs to what you normally experience with host operating systems, and it is, however this approach works well. RTEMS is not a host operating system and it is not a distribution. Deploying binary packages for every possible host operating system is too big a task for the RTEMS Project and it is not a good use of core developer time. Their time is better spent making RTEMS better and faster.

The RTEMS Project's aim is to give you complete freedom to decide on the languages used in your project, which version control system, and the build system for your application.

The rule for selecting a computer for a developer is *more is better* but we do understand there are limits. Projects set up different configurations, some have a development machine per developer while others set up a tightly controlled central build server. RTEMS Ecosystem is flexible and lets you engineer a development environment that suites you. The basic specs are:

- Multicore processor
- 8G bytes RAM
- 256G harddisk

RTEMS makes no demands on graphics.

If you are using a VM or your host computer is not a fast modern machine do not be concerned. The tools may take longer to build than faster hardware however building tools is something you do once. Once the tools and RTEMS is built all your time can be spent writing and developing your application. Over an hour can happen and for the ARM architecture and with all BSPs it can be many hours.

4.1 Host Operating Systems

GDB and Python

RTEMS uses Python in GDB to aid debugging which means GDB needs to be built with Python development libraries. Please check the RSB documentation and install the packages specified for your host. Make sure a python development package is included.

A wide range of host operating systems and hardware can be used. The host operating systems supported are:

- Linux
- FreeBSD
- NetBSD
- Apple OS X
- Windows
- Solaris

The functionality on a POSIX operating such as Linux and FreeBSD is similar and most features on Windows are supported but you are best to ask on the [Users Mailing List](#) if you have a specific question.

We recommend you maintain your operating system by installing any updates.

We also recommend you keep your environment to the bare minimum, particularly the PATH variable. Using environment variables has been proven over the years to be difficult to manage in production systems.

Warning: The RSB assumes your host is set up and the needed packages are installed and configured to work. If your host has not been set up please refer to section that covers your host's packages you need to install.

Path to use when building applications:

Do not forget to set the path before you use the tools, for example to build the RTEMS kernel.

The RSB by default will install (copy) the executables to a directory tree under the *prefix* you supply. To use the tools once finished just set your path to the bin directory under the *prefix* you use. In the examples that follow the *prefix* is `$HOME/development/rtems/4.11` and is set using the `--prefix` option so the path you need to configure to build applications can be set with the following in a BASH shell:

```
$ export PATH=$HOME/development/rtems/4.11/bin:$PATH
```

Make sure you place the RTEMS tool path at the front of your path so they are searched first. RTEMS can provide newer versions of some tools your operating system provides and placing the RTEMS tools path at the front means it is searched first and the RTEMS needed versions of the tools are used.

Warning: Do not put spaces or special characters in the directories you use to build RTEMS. Many of the packages built by the RSB use GNU *make*, which cannot handle spaces in pathnames. If there is a space in the pathname the build will fail. Special characters are also likely to confuse build systems.

Note: RSB and RTEMS have a matching *git branch* for each version of RTEMS. For example, if you want to build a toolchain for 4.11, then you should checkout the 4.11 branch of the RSB:

```
1 $ git checkout -t origin/4.11
```

Branches are available for the 4.9, 4.10, and 4.11 versions of RTEMS.

4.2 POSIX Hosts

POSIX hosts are most Unix operating systems such as Linux, FreeBSD and NetBSD. RTEMS development works well on Unix and can scale from a single user and a desktop machine to a team with decentralised or centralised development infrastructure.

4.2.1 Root Access

You either have root access to your host development machine or you do not. Some users are given hardware that is centrally managed. If you do not have root access you can create your work environment in your home directory. You could use a prefix of `$HOME/development/rtems` or `$HOME/rtems`. Note, the `$HOME` environment variable can be substituted with `~`.

Choose an Installation Prefix (page 15) details using Prefixes to manage the installation.

RTEMS Tools and packages do not require root access to be built and we encourage you to not build the tools as root. If you need to control write access then it is best to manage this with groups assigned to users.

If you have root access you can decide to install the tools under any suitable prefix. This may depend on the hardware in your host development machine. If the machine is a centralised build server the prefix may be used to separate production versions from the test versions and the prefix paths may have restricted access rights to only those who manage and have configuration control of the machine. We call this project sandboxing and *Project Sandboxing* (page 82) explains this in more detail.

4.2.2 Linux

BSP Build will require pax package if RTEMS is configured with the `--enable-tests` option, see *Building RTEMS Tests* (page 217). This package is not installed, by default, on many Linux distributions, you can check for it using your package manager. Install it, if it is not present on your system.

A number of different Linux distributions are known to work. The following have been tested and report as working.

4.2.2.1 ArchLinux

The following packages are required on a fresh Archlinux 64bit installation:

```
1 # pacman -S base-devel gdb xz unzip ncurses git zlib
```

Archlinux, by default installs `texinfo-5` which is incompatible for building GCC 4.7 tree. You will have to obtain `texinfo-legacy` from AUR and provide a manual override:

```
1 # pacman -R texinfo
2 $ yaourt -S texinfo-legacy
3 # ln -s /usr/bin/makeinfo-4.13a /usr/bin/makeinfo
```

4.2.2.2 CentOS

The following packages are required on a minimal CentOS 6.3 64bit installation:

```
1 # yum install autoconf automake binutils gcc gcc-c++ gdb make patch \
2 bison flex xz unzip ncurses-devel texinfo zlib-devel python-devel git
```

The minimal CentOS distribution is a specific DVD that installs a minimal system. If you use a full system some of these packages may have been installed.

4.2.2.3 Fedora

The RTEMS Source Builder has been tested on Fedora 19 64bit with the following packages:

```
1 # yum install ncurses-devel python-devel git bison gcc cvs gcc-c++ \
2 flex texinfo patch perl-Text-ParseWords zlib-devel
```

4.2.2.4 Raspbian

This is the Debian distribution for the Raspberry Pi. The following packages are required:

```
1 $ sudo apt-get install autoconf automake bison flex binutils gcc g++ gdb \
2 texinfo unzip ncurses-dev python-dev git
```

It is recommended you get Model B of the Pi with 512M of memory and to mount a remote disk over the network. The tools can be built on the network disk with a prefix under your home directory as recommended and end up on the SD card.

4.2.2.5 Ubuntu

The latest version is Ubuntu 18.04.1 LTS 64-bit. This section also includes Xubuntu. A minimal installation was used and the following packages installed:

```
1 $ sudo apt-get build-dep build-essential gcc-defaults g++ gdb git \
2 unzip pax bison flex texinfo unzip python3-dev libpython-dev \
3 libncurses5-dev zlib1g-dev
```

Note that in previous versions of Ubuntu, the package libpython-dev was python2.7-dev. The name of packages changes over time. You need the package with Python development libraries for C/C++ programs. The following is needed for recent versions:

```
1 $ sudo apt-get install python-dev
```

It is likely necessary that you will have to enable the Ubuntu Source Repositories. Users have suggested the following web pages which have instructions:

- <https://askubuntu.com/questions/158871/how-do-i-enable-the-source-code-repositories/158872>
- <https://askubuntu.com/questions/496549/error-you-must-put-some-source-uris-in-your-sources-list>

4.2.2.6 Linux Mint

zlib package is required on Linux Mint. It has a different name (other than the usual `zlib-dev`):

```
1 # sudo apt-get install zlib1g-dev
```

4.2.2.7 openSUSE

This has been reported to work but no instructions were provided. This is an opportunity to contribute. Please submit any guidance you can provide.

4.2.3 FreeBSD

The RTEMS Source Builder has been tested on FreeBSD 9.1, 10.3, 11 and 12 64bit version. You need to install some ports. They are:

```
1 # cd /usr/ports
2 # portinstall --batch lang/python27
```

If you wish to build Windows (mingw32) tools please install the following ports:

```
1 # cd /usr/ports
2 # portinstall --batch devel/mingw32-binutils devel/mingw32-gcc
3 # portinstall --batch devel/mingw32-zlib devel/mingw32-pthreads
```

The `+zlib+` and `+pthreads+` ports for MinGW32 are used for building a Windows QEMU.

If you are on FreeBSD 10.0 and you have `pkgng` installed you can use `'pkg install'` rather than `'portinstall'`.

We recommend you run as root the following command to speed up Python 3's subprocess support:

```
1 # mount -t fdescfs none /dev/fd
```

This speeds up closing file descriptors when creating subprocesses.

4.2.4 NetBSD

The RTEMS Source Builder has been tested on NetBSD 6.1 i386. Packages to add are:

```
1 # pkg_add ftp://ftp.netbsd.org/pub/pkgsrc/packages/NetBSD/i386/6.1/devel/gmake-3.82nb7.tgz
2 # pkg_add ftp://ftp.netbsd.org/pub/pkgsrc/packages/NetBSD/i386/6.1/devel/bison-2.7.1.tgz
3 # pkg_add ftp://ftp.netbsd.org/pub/pkgsrc/packages/NetBSD/i386/6.1/archivers/xz-5.0.4.tgz
```


4.3 Apple macOS

Apple's macOS is fully supported. You need to download and install a recent version of the Apple developer application Xcode. Xcode is available in the App Store. Make sure you install the Command Line Tools add on available for download within Xcode and once installed open a Terminal shell and enter the command `cc` and accept the license agreement.

The normal prefix when working on macOS as a user is under your home directory. Prefixes of `$HOME/development/rtems` or `$HOME/rtems` are suitable.

Choose an Installation Prefix (page 15) details using Prefixes to manage the installation.

4.3.1 Catalina

In the [macOS Catalina 10.15 Release Notes](#) Apple deprecated several scripting language runtimes such as Python 2.7. See also [Xcode 11 Release Notes](#). Due to the deprecated Python 2.7 support, we recommend to install and use the [latest Python 3 release from python.org](#).

4.3.2 Sierra

The RSB works on Sierra with the latest Xcode.

4.3.3 Mavericks

The RSB works on Mavericks and the GNU tools can be built for RTEMS using the Mavericks clang LLVM tool chain. You will need to build and install a couple of packages to make the RSB pass the `sb-check`. These are CVS and XZ. You can get these tools from a packaging tool for macOS such as *MacPorts* or *HomeBrew*.

I do not use third-party packaging on macOS and prefer to build the packages from source using a prefix of `/usr/local`. There are good third-party packages around however they sometimes bring in extra dependence and that complicates my build environment and I want to know the minimal requirements when building tools. The following are required:

- . The XZ package's home page is <http://tukaani.org/xz/> and I use version 5.0.5. XZ builds and installs cleanly.

4.4 Microsoft Windows

RTEMS supports Windows as a development host and the tools for most architectures are available. The RTEMS Project relies on the GNU tools for compilers and debuggers and we use the simulators that come with GDB and QEMU. The Windows support for these tools varies and the RTEMS Project is committed to helping the open source community improve the Windows experience. If something is not working or supported please email the [Users Mailing List](#).

The RTEMS Project's Windows tools can be native Windows executables which give the user the best possible experience on Windows. Native Windows programs use the standard Windows DLLs and paths. Integration with standard Windows integrated development tools such as editors is straight forward. POSIX emulation environments such as Cygwin and the MSYS2 shell have special executables that require a POSIX emulation DLL and these emulation DLLs add an extra layer of complexity as well as a performance over-head. The RTEMS Project uses these POSIX emulation shells to run configure scripts that come with various open source packages such as gcc so they form an important and valued part of the environment we describe here. The output of this procedure forms the tools you use during your application development and they do not depend on the emulation DLLs.

The performance of a native Windows compiler is as good as you can have on Windows and the performance compiling a single file will be similar to that on a host like Linux or FreeBSD given the same hardware. Building the tools from source is much slower on Windows because POSIX shells and related tools are used and the POSIX emulation overhead is much much slower than a native POSIX operating system like Linux and FreeBSD. This overhead is only during the building of the tools and the RTEMS kernel and if you use a suitable build system that is native to Windows your application development should be similar to other operating systems.

Building is known to work on *Windows 7 64bit Professional* and *Windows 10 64bit*.

4.4.1 Windows Path Length

Windows path length is limited and can cause problems when building the tools. The standard Windows API has a MAX_PATH length of 260 characters. This can effect some of the tools used by RTEMS. It is recommended you keep the top level directories as short as possible when building the RTEMS tools and you should also keep an eye on the path length when developing your application. The RTEMS built tools can handle much longer path lengths however some of the GNU tools such as those in the binutils package cannot.

The release packages of the RSB when unpacked have top level file names that are too big to build RTEMS. You need to change or rename that path to something smaller to build. This is indicated in *Releases* (page 54).

4.4.2 Windows Spaces In Paths

Occasionally, a program will fail on Windows with errors that appear as if a directory or file name was partially parsed by some utility or program. This can be caused by having directories of file names with spaces. Programs written in scripting languages sometimes fail to properly quote file names and the space is incorrectly interpreted.

Parts of the PATH inherited from the native Windows environment often include directory names with spaces. Sometimes it is necessary to set the PATH explicitly to avoid these.

4.4.3 Parallel Builds with Make

The MSYS2 GNU make has problems when using the *jobs* option. The RSB defaults to automatically using as many cores as the host machine has. To get a successful build on Windows it is recommended you add the `--jobs=none` option to all RSB build set commands.

4.4.4 POSIX Support

Building the RTEMS compilers, debugger, the RTEMS kernel and a number of other third-party packages requires a POSIX environment. On Windows you can use Cygwin or MSYS2. This document focuses on MSYS2. It is smaller than Cygwin and comes with the Arch Linux package manager *pacman*.

MSYS2 provides MinGW64 support as well as a POSIX shell called MSYS2. The MinGW64 compiler and related tools produce 64bit native Windows executables. The shell is a standard Bourne shell and the MSYS2 environment is a stripped Cygwin shell with enough support to run the various configure scripts needed to build the RTEMS tools and the RTEMS kernel.

MSYS2 is built around the *pacman* packaging tool. This makes MSYS2 a distribution and that is a welcome feature on Windows. You get a powerful tool to manage your development environment on Windows.

4.4.5 Python

We need Python to build the tools as the RSB is written in Python and we need suitable Python libraries to link to GDB as RTEMS makes use of GDB's Python support. This places specific demands on the Python we need installed and available and MSYS2 provides suitable Python versions we can use. You need to make sure you have the correct type and version of Python installed.

We cannot use the Python executables created by the Python project (python.org) as they are built by Microsoft's C (MSC) compiler. Linking the MSC Python libraries with the MinGW64 executables is not easy and MSYS provides us with a simple solution so we do not support linking MSC libraries.

MSYS2 provides two types and two versions of Python executables, MinGW and MSYS and Python version 2 and 3. For Windows we need the MinGW executable so we have suitable libraries and we have to have Python version 2 because on Windows GDB only builds with Python2.

You also need to install the MSYS version of Python along with the MinGW64 Python2 package. The MSYS Python is version 3 and the RSB can support version 2 and 3 of Python and it helps handle some of the long paths building GCC can generate.

4.4.6 MSYS2

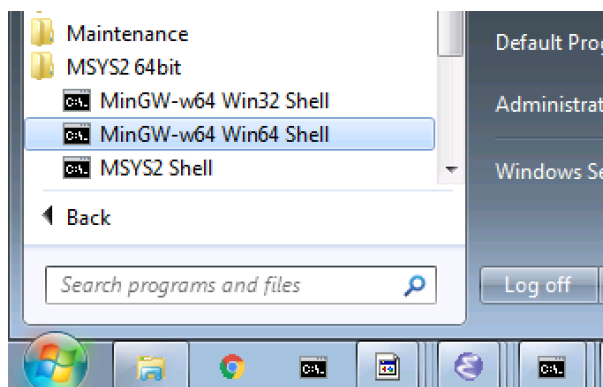
MSYS2 is installed on a new machine using the MSYS2 installer found on <https://msys2.github.io/>. Please select the `x86_64` variant for 64bit support. Run the installer following the 7 steps listed on the page.

MSYS2 uses the *pacman* package manager. The Arch Linux project has detailed documentation on how to use *pacman*. What is shown here is a just few examples of what you can do.

Pin MSYS2 Shell to Taskbar

Pin the MSYS2 64bit Shell to the Taskbar so you always use it rather than the 32bit Shell.

Open a 64bit MSYS shell from the Start Menu:



The packages we require are:

- python
- mingw-w64-x86_64-python2
- mingw-w64-x86_64-gcc
- git
- bison
- cvs
- diffutils
- make
- patch
- tar
- texinfo
- unzip

Note: The actual output provided may vary due to changes in the dependent packages or newer package versions.

Install the packages using pacman:

```
1 $ pacman -S python mingw-w64-x86_64-python2 mingw-w64-x86_64-gcc \  
2 bison cvs diffutils git make patch tar texinfo unzip  
3 resolving dependencies...  
4 looking for conflicting packages...  
5 .... output shortened for brevity ....
```

4.4.7 Cygwin

Building on Windows is a little more complicated because the Cygwin shell is used rather than the MSYS2 shell. The MSYS2 shell is simpler because the detected host triple is MinGW so the build is a standard cross-compiler build. A Canadian cross-build using Cygwin is supported if you would like native tools or you can use a Cygwin built set of tools.

Install a recent Cygwin version using the Cygwin setup tool. Select and install the groups and packages listed:

Table 1: Cygwin Packages

Group	Package
Archive	bsdtar
Archive	unzip
Archive	xz
Devel	autoconf
Devel	autoconf2.1
Devel	autoconf2.5
Devel	automake
Devel	binutils
Devel	bison
Devel	flex
Devel	gcc4-core
Devel	gcc4-g++
Devel	git
Devel	make
Devel	mingw64-x86_64-binutils
Devel	mingw64-x86_64-gcc-core
Devel	mingw64-x86_64-g++
Devel	mingw64-x86_64-runtime
Devel	mingw64-x86_64-zlib
Devel	patch
Devel	zlib-devel
MinGW	mingw-zlib-devel
Python	python

The setup tool will add a number of dependent package and it is ok to accept them.

Disabling Windows Defender improves performance if you have another up to date virus detection tool installed and enabled. The excellent Process Hacker 2 tool can monitor the performance and the Windows Defender service contributed a high load. In this case a third-party virus tool was installed so the Windows Defender service was not needed.

To build a MinGW tool chain a Canadian cross-compile (Cxc) is required on Cygwin because the host is Cygwin therefore a traditional cross-compile will result in Cygwin binaries. With a Canadian cross-compile a Cygwin cross-compiler is built as well as the MinGW RTEMS cross-compiler. The Cygwin cross-compiler is required to build the C runtime for the RTEMS target because we are building under Cygwin. The build output for an RTEMS 4.10 ARM tool set is:

```
1 chris@cygwin ~/development/rtems/src/rtems-source-builder/rtems
2 $ ../source-builder/sb-set-builder --log=l-arm.txt \
```

(continues on next page)

(continued from previous page)

```

3      --prefix=$HOME/development/rtems/4.10 4.10/rtems-arm
4 RTEMS Source Builder - Set Builder, v0.2
5 Build Set: 4.10/rtems-arm
6 config: expat-2.1.0-1.cfg
7 package: expat-2.1.0-x86_64-w64-mingw32-1
8 building: expat-2.1.0-x86_64-w64-mingw32-1
9 reporting: expat-2.1.0-1.cfg -> expat-2.1.0-x86_64-w64-mingw32-1.html
10 config: tools/rtems-binutils-2.20.1-1.cfg
11 package: arm-rtems4.10-binutils-2.20.1-1 <1>
12 building: arm-rtems4.10-binutils-2.20.1-1
13 package: (Cxc) arm-rtems4.10-binutils-2.20.1-1 <2>
14 building: (Cxc) arm-rtems4.10-binutils-2.20.1-1
15 reporting: tools/rtems-binutils-2.20.1-1.cfg ->
16 arm-rtems4.10-binutils-2.20.1-1.html
17 config: tools/rtems-gcc-4.4.7-newlib-1.18.0-1.cfg
18 package: arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1
19 building: arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1
20 package: (Cxc) arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1
21 building: (Cxc) arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1
22 reporting: tools/rtems-gcc-4.4.7-newlib-1.18.0-1.cfg ->
23 arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1.html
24 config: tools/rtems-gdb-7.3.1-1.cfg
25 package: arm-rtems4.10-gdb-7.3.1-1
26 building: arm-rtems4.10-gdb-7.3.1-1
27 reporting: tools/rtems-gdb-7.3.1-1.cfg -> arm-rtems4.10-gdb-7.3.1-1.html
28 config: tools/rtems-kernel-4.10.2.cfg
29 package: arm-rtems4.10-kernel-4.10.2-1
30 building: arm-rtems4.10-kernel-4.10.2-1
31 reporting: tools/rtems-kernel-4.10.2.cfg -> arm-rtems4.10-kernel-4.10.2-1.html
32 installing: expat-2.1.0-x86_64-w64-mingw32-1 -> /cygdrive/c/Users/chris/development/rtems/
33 ↪ 4.10
34 installing: arm-rtems4.10-binutils-2.20.1-1 -> /cygdrive/c/Users/chris/development/rtems/
35 ↪ 4.10 <3>
36 installing: arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1 -> /cygdrive/c/Users/chris/
37 ↪ development/rtems/4.10
38 installing: arm-rtems4.10-gdb-7.3.1-1 -> /cygdrive/c/Users/chris/development/rtems/4.10
39 installing: arm-rtems4.10-kernel-4.10.2-1 -> /cygdrive/c/Users/chris/development/rtems/4.
40 ↪ 10
41 cleaning: expat-2.1.0-x86_64-w64-mingw32-1
42 cleaning: arm-rtems4.10-binutils-2.20.1-1
43 cleaning: arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1
44 cleaning: arm-rtems4.10-gdb-7.3.1-1
45 cleaning: arm-rtems4.10-kernel-4.10.2-1
46 Build Set: Time 10:09:42.810547 <4>

```

Items:

1. The Cygwin version of the ARM cross-binutils.
2. The +(Cxc)+ indicates this is the MinGW build of the package.
3. Only the MinGW version is installed.
4. Cygwin is slow so please be patient. This time was on an AMD Athlon 64bit Dual Core 6000+ running at 3GHz with 4G RAM running Windows 7 64bit.

Warning: Cygwin documents the ‘Big List Of Dodgy Apps’ or ‘BLODA’. The link is <http://cygwin.com/faq/faq.html#faq.using.bloda> and it is worth a look. You will see a large number of common pieces of software found on Windows systems that can cause problems. My testing has been performed with NOD32 running and I have seen some failures. The list is for all of Cygwin so I am not sure which of the listed programs effect the RTEMS Source Biulder. The following FAQ item talks about *fork* failures and presents some technical reasons they cannot be avoided in all cases. Cygwin and it’s fork MSYS are fantastic pieces of software in a difficult environment. I have found building a single tool tends to work, building all at once is harder.

INSTALLATION

This section details how to set up and install the RTEMS Ecosystem. You will create a set of tools and an RTEMS kernel for your selected Board Support Package (BSP).

You will be asked to follow a few simple steps and when you have finished you will have a development environment set up you can use to build applications for RTEMS. You will have also created a development environment you and a team can adapt for a project of any size and complexity.

RTEMS applications are developed using cross-development tools running on a development computer, more commonly referred to as the host computer. These are typically your desktop machine or a special build server. All RTEMS tools and runtime libraries are built from source on your host machine. The RTEMS Project does not maintain binary builds of the tools. This may appear to be the opposite to what you normally experience with host operating systems, and it is, however this approach works well. RTEMS is not a host operating system and it is not a distribution. Providing binary packages for every possible host operating system is too big a task for the RTEMS Project and it is not a good use of core developer time. Their time is better spent making RTEMS better and faster.

The RTEMS Project base installation set ups the tools and the RTEMS kernel for the selected BSPs. The tools run on your host computer are used to compile, link, and format executables so they can run on your target hardware.

The RTEMS Project supports two set ups, release and developer environments. Release installations create the tools and kernel in a single pass ready for you to use. The tools and kernel are stable and only bug fixes are added creating new dot point releases. The developer set up tracks the Git repositories for the tools and kernel.

5.1 Releases

RTEMS releases provide a stable version of the kernel for the supported architectures. RTEMS maintains the current and previous releases. Support for older releases is provided using the RTEMS support channels.

Please read *Host Computer* (page 39) before continuing. The following procedure assumes you have installed and configured your host operating. It also assumes you have installed any dependent packages needed when building the tools and the kernel.

You need to select a location to build and install the RTEMS Tool chain and RTEMS. Make sure there is plenty of disk space and a fast disk is recommended. Our procedure will document building and installing the tools in a base directory called `/opt/rtems`. This path will require root access. If you are working on a machine you do not have root access to you can use a home directory, If building on Windows use `/c/opt/rtems` to keep the top level paths as short as possible. *Windows Path Length* (page 46) provides more detail about path lengths on Windows.

The location used to install the tools and kernel is called the *prefix*. *Choose an Installation Prefix* (page 15) explains prefixes and how to use them. It is best to have a *prefix* for each different version of RTEMS you are using. If you are using RTEMS 4.11 in production it is **not** a good idea to install a development version of 5 over the top by using the same *prefix* as the 4.11 build. A separate *prefix* for each version avoids this.

Released versions of the RTEMS Source Builder (RSB) downloads all source code for all packages from the [FTP File Server](#) rather than from the package's home site. Hosting all the source on the [FTP File Server](#) ensures the source is present for the life of the release on the [FTP File Server](#). If there is a problem accessing the RTEMS FTP the RSB will fall back to the packages home site.

The [FTP File Server](#) is hosted at the Oregon State University's The Open Source Lab (<http://osuosl.org/>). This is a nonprofit organization working for the advancement of open source technologies and RTEMS is very fortunate to be shosted here. It has excellent internet access and performance.

Note: Controlling the RTEMS Kernel Build

Building releases by default does not build the RTEMS kernel. To build the RTEMS kernel add the `--with-rtems` option to the RSB command line.

By default all the BSPs for an architecture are built. If you only wish to have a specific BSP built you can specify the BSP list by providing to the RSB the option `--with-rtemsbsp`. For example to build two BSPs for the SPARC architecture you can supply `--with-rtemsbsp="erc32 leon3"`. This can speed the build time up for some architectures that have a lot of BSPs.

Once you have built the tools and kernel you can move to the Packages section of the manual.

5.1.1 RTEMS Tools and Kernel

This procedure will build a SPARC tool chain. Set up a suitable workspace to build the release in. On Unix:

```

1 $ cd
2 $ mkdir -p development/rtems/releases
3 $ cd development/rtems/releases

```

If building on Windows:

```

1 $ cd /c
2 $ mkdir -p opt/rtems
3 $ cd opt/rtems

```

Note the paths on Windows will be different to those shown.

Download the RTEMS Source Builder (RSB) from the RTEMS FTP server:

```

1 $ wget https://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/rtems-source-builder-4.11.0.
   ↪tar.xz
2 --2016-03-21 10:50:04-- https://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/rtems-source-
   ↪builder-4.11.0.tar.xz
3 Resolving ftp.rtems.org (ftp.rtems.org)... 140.211.10.151
4 Connecting to ftp.rtems.org (ftp.rtems.org)|140.211.10.151|:443... connected.
5 HTTP request sent, awaiting response... 200 OK
6 Length: 967056 (944K) [application/x-xz]
7 Saving to: 'rtems-source-builder-4.11.0.tar.xz'
8
9 rtems-source-builder-4.1 100%[=====>] 944.39K 206KB/s   in_
   ↪5.5s
10
11 2016-03-21 10:50:11 (173 KB/s) - 'rtems-source-builder-4.11.0.tar.xz' saved [967056/
   ↪967056]

```

On Unix unpack the RSB release tar file using:

```

1 $ tar Jxf rtems-source-builder-4.11.0.tar.xz
2 $ cd rtems-source-builder-4.11.0/rtems/

```

On Windows you need to shorten the path (See *Windows Path Length* (page 46)) after you have unpacked the tar file:

```

1 $ tar Jxf rtems-source-builder-4.11.0.tar.xz
2 $ mv rtems-source-builder-4.11.0 4.110
3 $ cd 4.11.0

```

Build a tool chain for the SPARC architecture. We are using the SPARC architecture in our example because GDB has a good simulator that lets us run and test the samples RTEMS builds by default

If building on Windows add `--jobs=none` to avoid GNU make issues on Windows discussed in *Parallel Builds with Make* (page 47).

```

1 $ ../source-builder/sb-set-builder \
2   --prefix=/opt/rtems/4.11 4.11/rtems-sparc
3 Build Set: 4.11/rtems-sparc
4 Build Set: 4.11/rtems-autotools.bset
5 Build Set: 4.11/rtems-autotools-internal.bset
6 config: tools/rtems-autoconf-2.69-1.cfg
7 package: autoconf-2.69-x86_64-freebsd10.1-1

```

(continues on next page)

(continued from previous page)

```

8 Creating source directory: sources
9 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/autoconf-2.69.tar.gz
  ↳-> sources/autoconf-2.69.tar.gz
10 downloading: sources/autoconf-2.69.tar.gz - 1.8MB of 1.8MB (100%)
11 building: autoconf-2.69-x86_64-freebsd10.1-1
12 config: tools/rtems-automake-1.12.6-1.cfg
13 package: automake-1.12.6-x86_64-freebsd10.1-1
14 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/automake-1.12.6.tar.
  ↳gz -> sources/automake-1.12.6.tar.gz
15 downloading: sources/automake-1.12.6.tar.gz - 2.0MB of 2.0MB (100%)
16 Creating source directory: patches
17 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/automake-1.12.6-
  ↳bugzilla.redhat.com-1239379.diff -> patches/automake-1.12.6-bugzilla.redhat.com-1239379.
  ↳diff
18 downloading: patches/automake-1.12.6-bugzilla.redhat.com-1239379.diff - 408.0 bytes of
  ↳408.0 bytes (100%)
19 building: automake-1.12.6-x86_64-freebsd10.1-1
20 cleaning: autoconf-2.69-x86_64-freebsd10.1-1
21 cleaning: automake-1.12.6-x86_64-freebsd10.1-1
22 Build Set: Time 0:00:32.749337
23 Build Set: 4.11/rtems-autotools-base.bset
24 config: tools/rtems-autoconf-2.69-1.cfg
25 package: autoconf-2.69-x86_64-freebsd10.1-1
26 building: autoconf-2.69-x86_64-freebsd10.1-1
27 reporting: tools/rtems-autoconf-2.69-1.cfg -> autoconf-2.69-x86_64-freebsd10.1-1.txt
28 reporting: tools/rtems-autoconf-2.69-1.cfg -> autoconf-2.69-x86_64-freebsd10.1-1.xml
29 config: tools/rtems-automake-1.12.6-1.cfg
30 package: automake-1.12.6-x86_64-freebsd10.1-1
31 building: automake-1.12.6-x86_64-freebsd10.1-1
32 reporting: tools/rtems-automake-1.12.6-1.cfg -> automake-1.12.6-x86_64-freebsd10.1-1.txt
33 reporting: tools/rtems-automake-1.12.6-1.cfg -> automake-1.12.6-x86_64-freebsd10.1-1.xml
34 installing: autoconf-2.69-x86_64-freebsd10.1-1 -> /opt/work/rtems/4.11.0
35 installing: automake-1.12.6-x86_64-freebsd10.1-1 -> /opt/work/rtems/4.11.0
36 cleaning: autoconf-2.69-x86_64-freebsd10.1-1
37 cleaning: automake-1.12.6-x86_64-freebsd10.1-1
38 Build Set: Time 0:00:15.619219
39 Build Set: Time 0:00:48.371085
40 config: devel/expat-2.1.0-1.cfg
41 package: expat-2.1.0-x86_64-freebsd10.1-1
42 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/expat-2.1.0.tar.gz ->
  ↳ sources/expat-2.1.0.tar.gz
43 downloading: sources/expat-2.1.0.tar.gz - 549.4kB of 549.4kB (100%)
44 building: expat-2.1.0-x86_64-freebsd10.1-1
45 reporting: devel/expat-2.1.0-1.cfg -> expat-2.1.0-x86_64-freebsd10.1-1.txt
46 reporting: devel/expat-2.1.0-1.cfg -> expat-2.1.0-x86_64-freebsd10.1-1.xml
47 config: tools/rtems-binutils-2.26-1.cfg
48 package: sparc-rtems4.11-binutils-2.26-x86_64-freebsd10.1-1
49 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/binutils-2.26.tar.
  ↳bz2 -> sources/binutils-2.26.tar.bz2
50 downloading: sources/binutils-2.26.tar.bz2 - 24.4MB of 24.4MB (100%)
51 building: sparc-rtems4.11-binutils-2.26-x86_64-freebsd10.1-1
52 reporting: tools/rtems-binutils-2.26-1.cfg ->
53 sparc-rtems4.11-binutils-2.26-x86_64-freebsd10.1-1.txt
54 reporting: tools/rtems-binutils-2.26-1.cfg ->
55 sparc-rtems4.11-binutils-2.26-x86_64-freebsd10.1-1.xml
56 config: tools/rtems-gcc-4.9.3-newlib-2.2.0-20150423-1.cfg

```

(continues on next page)

(continued from previous page)

```

57 package: sparc-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-freebsd10.1-1
58 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/gcc-4.9.3.tar.bz2 ->_
    ↳sources/gcc-4.9.3.tar.bz2
59 downloading: sources/gcc-4.9.3.tar.bz2 - 85.8MB of 85.8MB (100%)
60 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/newlib-2.2.0.
    ↳20150423.tar.gz -> sources/newlib-2.2.0.20150423.tar.gz
61 downloading: sources/newlib-2.2.0.20150423.tar.gz - 16.7MB of 16.7MB (100%)
62 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/mpfr-3.0.1.tar.bz2 ->
    ↳ sources/mpfr-3.0.1.tar.bz2
63 downloading: sources/mpfr-3.0.1.tar.bz2 - 1.1MB of 1.1MB (100%)
64 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/mpc-0.8.2.tar.gz ->_
    ↳sources/mpc-0.8.2.tar.gz
65 downloading: sources/mpc-0.8.2.tar.gz - 535.5kB of 535.5kB (100%)
66 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/gmp-5.0.5.tar.bz2 ->_
    ↳sources/gmp-5.0.5.tar.bz2
67 downloading: sources/gmp-5.0.5.tar.bz2 - 2.0MB of 2.0MB (100%)
68 building: sparc-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-freebsd10.1-1
69 reporting: tools/rtems-gcc-4.9.3-newlib-2.2.0-20150423-1.cfg ->
70 sparc-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-freebsd10.1-1.txt
71 reporting: tools/rtems-gcc-4.9.3-newlib-2.2.0-20150423-1.cfg ->
72 sparc-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-freebsd10.1-1.xml
73 config: tools/rtems-gdb-7.9-1.cfg
74 package: sparc-rtems4.11-gdb-7.9-x86_64-freebsd10.1-1
75 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/gdb-7.9.tar.xz ->_
    ↳sources/gdb-7.9.tar.xz
76 downloading: sources/gdb-7.9.tar.xz - 17.0MB of 17.0MB (100%)
77 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0001-sim-erc32-
    ↳Disassembly-in-stand-alone-mode-did-not-wo.patch -> patches/0001-sim-erc32-Disassembly-
    ↳in-stand-alone-mode-did-not-wo.patch
78 downloading: patches/0001-sim-erc32-Disassembly-in-stand-alone-mode-did-not-wo.patch - 1.
    ↳9kB of 1.9kB (100%)
79 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0002-sim-erc32-
    ↳Corrected-wrong-CPU-implementation-and-ver.patch -> patches/0002-sim-erc32-Corrected-
    ↳wrong-CPU-implementation-and-ver.patch
80 downloading: patches/0002-sim-erc32-Corrected-wrong-CPU-implementation-and-ver.patch -_
    ↳827.0 bytes of 827.0 bytes (100%)
81 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0003-sim-erc32-
    ↳Perform-pseudo-init-if-binary-linked-to-no.patch -> patches/0003-sim-erc32-Perform-
    ↳pseudo-init-if-binary-linked-to-no.patch
82 downloading: patches/0003-sim-erc32-Perform-pseudo-init-if-binary-linked-to-no.patch - 2.
    ↳6kB of 2.6kB (100%)
83 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0004-sim-erc32-Use-
    ↳fenv.h-for-host-FPU-access.patch -> patches/0004-sim-erc32-Use-fenv.h-for-host-FPU-
    ↳access.patch
84 downloading: patches/0004-sim-erc32-Use-fenv.h-for-host-FPU-access.patch - 4.9kB of 4.9kB_
    ↳(100%)
85 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0005-sim-erc32-
    ↳Remove-unused-defines-in-Makefile-and-swit.patch -> patches/0005-sim-erc32-Remove-
    ↳unused-defines-in-Makefile-and-swit.patch
86 downloading: patches/0005-sim-erc32-Remove-unused-defines-in-Makefile-and-swit.patch -_
    ↳871.0 bytes of 871.0 bytes (100%)
87 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0006-sim-erc32-Fix-
    ↳incorrect-simulator-performance-report.patch -> patches/0006-sim-erc32-Fix-incorrect-
    ↳simulator-performance-report.patch
88 downloading: patches/0006-sim-erc32-Fix-incorrect-simulator-performance-report.patch - 5.
    ↳6kB of 5.6kB (100%)

```

(continues on next page)

(continued from previous page)

```

89 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0007-sim-erc32-File-
    ↳ loading-via-command-line-did-not-work.patch -> patches/0007-sim-erc32-File-loading-via-
    ↳ command-line-did-not-work.patch
90 downloading: patches/0007-sim-erc32-File-loading-via-command-line-did-not-work.patch - 1.
    ↳ 0kB of 1.0kB (100%)
91 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0008-sim-erc32-Added-
    ↳ v-command-line-switch-for-verbose-ou.patch -> patches/0008-sim-erc32-Added-v-command-
    ↳ line-switch-for-verbose-ou.patch
92 downloading: patches/0008-sim-erc32-Added-v-command-line-switch-for-verbose-ou.patch - 3.
    ↳ 6kB of 3.6kB (100%)
93 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0009-sim-erc32-
    ↳ Removed-type-mismatch-compiler-warnings.patch -> patches/0009-sim-erc32-Removed-type-
    ↳ mismatch-compiler-warnings.patch
94 downloading: patches/0009-sim-erc32-Removed-type-mismatch-compiler-warnings.patch - 1.9kB
    ↳ of 1.9kB (100%)
95 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0010-sim-erc32-
    ↳ Switched-emulated-memory-to-host-endian-or.patch -> patches/0010-sim-erc32-Switched-
    ↳ emulated-memory-to-host-endian-or.patch
96 downloading: patches/0010-sim-erc32-Switched-emulated-memory-to-host-endian-or.patch - 16.
    ↳ 0kB of 16.0kB (100%)
97 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0011-sim-erc32-use-
    ↳ SIM_AC_OPTION_HOSTENDIAN-to-probe-for-.patch -> patches/0011-sim-erc32-use-SIM_AC_
    ↳ OPTION_HOSTENDIAN-to-probe-for-.patch
98 downloading: patches/0011-sim-erc32-use-SIM_AC_OPTION_HOSTENDIAN-to-probe-for-.patch - 14.
    ↳ 8kB of 14.8kB (100%)
99 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0012-sim-erc32-Use-
    ↳ memory_iread-function-for-instruction-.patch -> patches/0012-sim-erc32-Use-memory_iread-
    ↳ function-for-instruction-.patch
100 downloading: patches/0012-sim-erc32-Use-memory_iread-function-for-instruction-.patch - 3.
    ↳ 8kB of 3.8kB (100%)
101 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0013-sim-erc32-Fix-a-
    ↳ few-compiler-warnings.patch-> patches/0013-sim-erc32-Fix-a-few-compiler-warnings.patch
102 downloading: patches/0013-sim-erc32-Fix-a-few-compiler-warnings.patch - 2.2kB of 2.2kB
    ↳ (100%)
103 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0014-sim-erc32-Use-
    ↳ gdb-callback-for-UART-I-O-when-linked-.patch -> patches/0014-sim-erc32-Use-gdb-callback-
    ↳ for-UART-I-O-when-linked-.patch
104 downloading: patches/0014-sim-erc32-Use-gdb-callback-for-UART-I-O-when-linked-.patch - 9.
    ↳ 2kB of 9.2kB (100%)
105 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0015-sim-erc32-
    ↳ Access-memory-subsystem-through-struct-mem.patch -> patches/0015-sim-erc32-Access-
    ↳ memory-subsystem-through-struct-mem.patch
106 downloading: patches/0015-sim-erc32-Access-memory-subsystem-through-struct-mem.patch - 22.
    ↳ 9kB of 22.9kB (100%)
107 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0016-sim-erc32-Use-
    ↳ readline.h-for-readline-types-and-func.patch -> patches/0016-sim-erc32-Use-readline.h-
    ↳ for-readline-types-and-func.patch
108 downloading: patches/0016-sim-erc32-Use-readline.h-for-readline-types-and-func.patch - 1.
    ↳ 5kB of 1.5kB (100%)
109 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0017-sim-erc32-Move-
    ↳ local-extern-declarations-into-sis.h.patch -> patches/0017-sim-erc32-Move-local-extern-
    ↳ declarations-into-sis.h.patch
110 downloading: patches/0017-sim-erc32-Move-local-extern-declarations-into-sis.h.patch - 5.
    ↳ 8kB of 5.8kB (100%)
111 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0018-sim-erc32-Add-
    ↳ support-for-LEON3-processor-emulation.patch -> patches/0018-sim-erc32-Add-support-for-
    ↳ LEON3-processor-emulation.patch

```

(continues on next page)

(continued from previous page)

```

112 downloading: patches/0018-sim-erc32-Add-support-for-LEON3-processor-emulation.patch - 66.
    ↳ 7kB of 66.7kB (100%)
113 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0019-sim-erc32-Add-
    ↳ support-for-LEON2-processor-emulation.patch -> patches/0019-sim-erc32-Add-support-for-
    ↳ LEON2-processor-emulation.patch
114 downloading: patches/0019-sim-erc32-Add-support-for-LEON2-processor-emulation.patch - 26.
    ↳ 1kB of 26.1kB (100%)
115 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0020-sim-erc32-
    ↳ Updated-documentation.patch -> patches/0020-sim-erc32-Updated-documentation.patch
116 downloading: patches/0020-sim-erc32-Updated-documentation.patch - 16.1kB of 16.1kB (100%)
117 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0021-sim-erc32-Add-
    ↳ data-watchpoint-support.patch -> patches/0021-sim-erc32-Add-data-watchpoint-support.
    ↳ patch
118 downloading: patches/0021-sim-erc32-Add-data-watchpoint-support.patch - 10.1kB of 10.1kB
    ↳ (100%)
119 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0022-Add-watchpoint-
    ↳ support-to-gdb-simulator-interface.patch -> patches/0022-Add-watchpoint-support-to-gdb-
    ↳ simulator-interface.patch
120 downloading: patches/0022-Add-watchpoint-support-to-gdb-simulator-interface.patch - 25.
    ↳ 5kB of 25.5kB (100%)
121 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/0023-sim-erc32-ELF-
    ↳ loading-could-fail-on-unaligned-sectio.patch -> patches/0023-sim-erc32-ELF-loading-
    ↳ could-fail-on-unaligned-sectio.patch
122 downloading: patches/0023-sim-erc32-ELF-loading-could-fail-on-unaligned-sectio.patch - 1.
    ↳ 3kB of 1.3kB (100%)
123 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/gdb-sim-arange-
    ↳ inline.diff -> patches/gdb-sim-arange-inline.diff
124 downloading: patches/gdb-sim-arange-inline.diff - 761.0 bytes of 761.0 bytes (100%)
125 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/gdb-sim-cgen-inline.
    ↳ diff -> patches/gdb-sim-cgen-inline.diff
126 downloading: patches/gdb-sim-cgen-inline.diff - 706.0 bytes of 706.0 bytes (100%)
127 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/sources/patch-gdb-python-
    ↳ python-config.py -> patches/patch-gdb-python-python-config.py
128 downloading: patches/patch-gdb-python-python-config.py - 449.0 bytes of 449.0 bytes (100%)
129 building: sparc-rtems4.11-gdb-7.9-x86_64-freebsd10.1-1
130 reporting: tools/rtems-gdb-7.9-1.cfg ->
131 sparc-rtems4.11-gdb-7.9-x86_64-freebsd10.1-1.txt
132 reporting: tools/rtems-gdb-7.9-1.cfg ->
133 sparc-rtems4.11-gdb-7.9-x86_64-freebsd10.1-1.xml
134 config: tools/rtems-tools-4.11-1.cfg
135 package: rtems-tools-4.11.0-1
136 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/rtems-tools-4.11.0.tar.xz ->
    ↳ sources/rtems-tools-4.11.0.tar.xz
137 downloading: sources/rtems-tools-4.11.0.tar.xz - 1.6MB of 1.6MB (100%)
138 building: rtems-tools-4.11.0-1
139 reporting: tools/rtems-tools-4.11-1.cfg -> rtems-tools-4.11.0-1.txt
140 reporting: tools/rtems-tools-4.11-1.cfg -> rtems-tools-4.11.0-1.xml
141 config: tools/rtems-kernel-4.11.cfg
142 package: sparc-rtems4.11-kernel-4.11.0-1
143 download: ftp://ftp.rtems.org/pub/rtems/releases/4.11/4.11.0/rtems-4.11.0.tar.xz ->
    ↳ sources/rtems-4.11.0.tar.xz
144 downloading: sources/rtems-4.11.0.tar.xz - 9.3MB of 9.3MB (100%)
145 building: sparc-rtems4.11-kernel-4.11.0-1
146 reporting: tools/rtems-kernel-4.11.cfg -> sparc-rtems4.11-kernel-4.11.0-1.txt
147 reporting: tools/rtems-kernel-4.11.cfg -> sparc-rtems4.11-kernel-4.11.0-1.xml
148 installing: expat-2.1.0-x86_64-freebsd10.1-1 -> /opt/work/rtems/4.11.0

```

(continues on next page)

(continued from previous page)

```
149 installing: sparc-rtems4.11-binutils-2.26-x86_64-freebsd10.1-1 -> /opt/work/rtems/4.11.0
150 installing: sparc-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-freebsd10.1-1 -> /opt/
    ↪work/rtems/4.11.0
151 installing: sparc-rtems4.11-gdb-7.9-x86_64-freebsd10.1-1 -> /opt/work/rtems/4.11.0
152 installing: rtems-tools-4.11.0-1 -> /opt/work/rtems/4.11.0
153 installing: sparc-rtems4.11-kernel-4.11.0-1 -> /opt/work/rtems/4.11.0
154 cleaning: expat-2.1.0-x86_64-freebsd10.1-1
155 cleaning: sparc-rtems4.11-binutils-2.26-x86_64-freebsd10.1-1
156 cleaning: sparc-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-freebsd10.1-1
157 cleaning: sparc-rtems4.11-gdb-7.9-x86_64-freebsd10.1-1
158 cleaning: rtems-tools-4.11.0-1
159 cleaning: sparc-rtems4.11-kernel-4.11.0-1
160 Build Set: Time 0:19:15.713662
```

You can now build a third-party library or an application as defaulted in TBD.

5.2 Developer (Unstable)

RTEMS provides open access to its development processes. We call this the developer set up. The project encourages all users to inspect, review, comment and contribute to the code base. The processes described here are the same processes the core development team use when developing and maintaining RTEMS.

Warning: The development version is not for use in production and it can break from time to time.

Please read *Host Computer* (page 39) before continuing. The following procedure assumes you have installed and configured your host operating system. It also assumes you have installed any dependent packages needed when building the tools and the kernel.

You need to select a location to build and install the RTEMS Tool chain and RTEMS. Make sure there is plenty of disk space and a fast disk is recommended. Our procedure will document building and installing the tools in a home directory called `development/rtems`. Using a home directory means you can do this without needing to be root. You can also use `/opt/rtems/build` if you have access to that path.

The location used to install the tools and kernel is called the *prefix*. It is best to have a *prefix* for each different version of RTEMS you are using. If you are using RTEMS 4.11 in production it is not a good idea to install a development version of 5 over the top. A separate *prefix* for each version avoids this.

The RTEMS tool chain changes less often than the RTEMS kernel. One method of working with development releases is to have a separate *prefix* for the RTEMS tools and a different one for the RTEMS kernel. You can then update each without interacting with the other. You can also have a number of RTEMS versions available to test with.

Downloading the source

You need an internet connection to download the source. The downloaded source is cached locally and the RSB checksums it. If you run a build again the download output will be missing. Using the RSB from git will download the source from the upstream project's home site and this could be *http*, *ftp*, or *git*.

5.2.1 POSIX and OS X Host Tools Chain

This procedure will build a SPARC tool chain.

Clone the RTEMS Source Builder (RSB) repository:

```

1 $ cd
2 $ mkdir -p development/rtems
3 $ cd development/rtems
4 $ git clone git://git.rtems.org/rtems-source-builder.git rsb
5 Cloning into 'rsb'...
6 remote: Counting objects: 5837, done.
7 remote: Compressing objects: 100% (2304/2304), done.
```

(continues on next page)

(continued from previous page)

```

8 remote: Total 5837 (delta 4014), reused 5056 (delta 3494)
9 Receiving objects: 100% (5837/5837), 2.48 MiB | 292.00 KiB/s, done.
10 Resolving deltas: 100% (4014/4014), done.
11 Checking connectivity... done.

```

Check all the host packages you need are present. Current libraries are not checked and this includes checking for the python development libraries GDB requires:

```

1 $ cd rsb
2 $ ./source-builder/sb-check
3 RTEMS Source Builder - Check, 5 (089327b5dcf9)
4 Environment is ok

```

If you are unsure how to specify the build set for the architecture you wish to build, just ask the tool:

```

1 $ ../source-builder/sb-set-builder --list-bsets <1>
2 RTEMS Source Builder - Set Builder, v4.11.0
3 Examining: config
4 Examining: ../source-builder/config <2>
5 4.10/rtems-all.bset <3>
6 4.10/rtems-arm.bset <4>
7 4.10/rtems-autotools.bset
8 4.10/rtems-avr.bset
9 4.10/rtems-bfin.bset
10 4.10/rtems-h8300.bset
11 4.10/rtems-i386.bset
12 4.10/rtems-lm32.bset
13 4.10/rtems-m32c.bset
14 4.10/rtems-m32r.bset
15 4.10/rtems-m68k.bset
16 4.10/rtems-mips.bset
17 4.10/rtems-nios2.bset
18 4.10/rtems-powerpc.bset
19 4.10/rtems-sh.bset
20 4.10/rtems-sparc.bset
21 4.11/rtems-all.bset
22 4.11/rtems-arm.bset
23 4.11/rtems-autotools.bset
24 4.11/rtems-avr.bset
25 4.11/rtems-bfin.bset
26 4.11/rtems-h8300.bset
27 4.11/rtems-i386.bset
28 4.11/rtems-lm32.bset
29 4.11/rtems-m32c.bset
30 4.11/rtems-m32r.bset
31 4.11/rtems-m68k.bset
32 4.11/rtems-microblaze.bset
33 4.11/rtems-mips.bset
34 4.11/rtems-moxie.bset
35 4.11/rtems-nios2.bset
36 4.11/rtems-powerpc.bset
37 4.11/rtems-sh.bset
38 4.11/rtems-sparc.bset
39 4.11/rtems-sparc64.bset

```

(continues on next page)

(continued from previous page)

```

40 4.11/rtems-v850.bset
41 4.9/rtems-all.bset
42 4.9/rtems-arm.bset
43 4.9/rtems-autotools.bset
44 4.9/rtems-i386.bset
45 4.9/rtems-m68k.bset
46 4.9/rtems-mips.bset
47 4.9/rtems-powerpc.bset
48 4.9/rtems-sparc.bset
49 gnu-tools-4.6.bset
50 rtems-4.10-base.bset <5>
51 rtems-4.11-base.bset
52 rtems-4.9-base.bset
53 rtems-base.bset <5>

```

Items:

1. Only option required is `--list-bsets`
2. The paths inspected. See *Configuration* (page 257).
3. A build set to build all RTEMS 4.10 supported architectures.
4. The build set for the ARM architecture on RTEMS 4.10.
5. Support build set file with common functionality included by other build set files.

Build a tool chain for the SPARC architecture. We are using the SPARC architecture because GDB has a good simulator that lets us run and test the samples RTEMS builds by default. The current development version is 5 and is on master:

```

1 $ cd rtems
2 $ ../source-builder/sb-set-builder --prefix=/usr/home/chris/development/rtems/5 5/rtems-
  ↪sparc
3 RTEMS Source Builder - Set Builder, 5 (089327b5dcf9)
4 Build Set: 5/rtems-sparc
5 Build Set: 5/rtems-autotools.bset
6 Build Set: 5/rtems-autotools-internal.bset
7 config: tools/rtems-autoconf-2.69-1.cfg
8 package: autoconf-2.69-x86_64-linux-gnu-1
9 Creating source directory: sources
10 download: ftp://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz -> sources/autoconf-2.69.
  ↪tar.gz
11 downloading: sources/autoconf-2.69.tar.gz - 1.8MB of 1.8MB (100%)
12 building: autoconf-2.69-x86_64-linux-gnu-1
13 config: tools/rtems-automake-1.12.6-1.cfg
14 package: automake-1.12.6-x86_64-linux-gnu-1
15 download: ftp://ftp.gnu.org/gnu/automake/automake-1.12.6.tar.gz -> sources/automake-1.12.
  ↪6.tar.gz
16 downloading: sources/automake-1.12.6.tar.gz - 2.0MB of 2.0MB (100%)
17 Creating source directory: patches
18 download: https://git.rtems.org/rtems-tools/plain/tools/5/automake/automake-1.12.6-
  ↪bugzilla.redhat.com-1239379.diff -> patches/automake-1.12.6-bugzilla.redhat.com-1239379.
  ↪diff
19 downloading: patches/automake-1.12.6-bugzilla.redhat.com-1239379.diff - 408.0 bytes of
  ↪408.0 bytes (100%)

```

(continues on next page)

(continued from previous page)

```

20 building: automake-1.12.6-x86_64-linux-gnu-1
21 cleaning: autoconf-2.69-x86_64-linux-gnu-1
22 cleaning: automake-1.12.6-x86_64-linux-gnu-1
23 Build Set: Time 0:00:12.713221
24 Build Set: 5/rtems-autotools-base.bset
25 config: tools/rtems-autoconf-2.69-1.cfg
26 package: autoconf-2.69-x86_64-linux-gnu-1
27 building: autoconf-2.69-x86_64-linux-gnu-1
28 reporting: tools/rtems-autoconf-2.69-1.cfg -> autoconf-2.69-x86_64-linux-gnu-1.txt
29 reporting: tools/rtems-autoconf-2.69-1.cfg -> autoconf-2.69-x86_64-linux-gnu-1.xml
30 config: tools/rtems-automake-1.12.6-1.cfg
31 package: automake-1.12.6-x86_64-linux-gnu-1
32 building: automake-1.12.6-x86_64-linux-gnu-1
33 reporting: tools/rtems-automake-1.12.6-1.cfg -> automake-1.12.6-x86_64-linux-gnu-1.txt
34 reporting: tools/rtems-automake-1.12.6-1.cfg -> automake-1.12.6-x86_64-linux-gnu-1.xml
35 installing: autoconf-2.69-x86_64-linux-gnu-1 -> /usr/home/chris/development/rtems/5
36 installing: automake-1.12.6-x86_64-linux-gnu-1 -> /usr/home/chris/development/rtems/5
37 cleaning: autoconf-2.69-x86_64-linux-gnu-1
38 cleaning: automake-1.12.6-x86_64-linux-gnu-1
39 Build Set: Time 0:00:09.105363
40 Build Set: Time 0:00:21.822083
41 config: devel/expat-2.1.0-1.cfg
42 package: expat-2.1.0-x86_64-linux-gnu-1
43 download: http://downloads.sourceforge.net/project/expat/expat/2.1.0/expat-2.1.0.tar.gz ->
  ↳ sources/expat-2.1.0.tar.gz
44 redirect: https://vorboss.dl.sourceforge.net/project/expat/expat/2.1.0/expat-2.1.0.tar.gz
45 downloading: sources/expat-2.1.0.tar.gz - 549.4kB of 549.4kB (100%)
46 building: expat-2.1.0-x86_64-linux-gnu-1
47 reporting: devel/expat-2.1.0-1.cfg -> expat-2.1.0-x86_64-linux-gnu-1.txt
48 reporting: devel/expat-2.1.0-1.cfg -> expat-2.1.0-x86_64-linux-gnu-1.xml
49 config: tools/rtems-binutils-2.29-1.cfg
50 package: sparc-rtems5-binutils-2.29-x86_64-linux-gnu-1
51 download: ftp://ftp.gnu.org/gnu/binutils/binutils-2.29.tar.bz2 -> sources/binutils-2.29.
  ↳ tar.bz2
52 downloading: sources/binutils-2.29.tar.bz2 - 27.7MB of 27.7MB (100%)
53 download: https://devel.rtems.org/raw-attachment/ticket/3091/0001-Fix-Binutils-2.29-
  ↳ PR21884.patch -> patches/0001-Fix-Binutils-2.29-PR21884.patch
54 downloading: patches/0001-Fix-Binutils-2.29-PR21884.patch - 8.8kB of 8.8kB (100%)
55 building: sparc-rtems5-binutils-2.29-x86_64-linux-gnu-1
56 reporting: tools/rtems-binutils-2.29-1.cfg -> sparc-rtems5-binutils-2.29-x86_64-linux-gnu-
  ↳ 1.txt
57 reporting: tools/rtems-binutils-2.29-1.cfg -> sparc-rtems5-binutils-2.29-x86_64-linux-gnu-
  ↳ 1.xml
58 config: tools/rtems-gcc-7.2.0-newlib-2.5.0.20170922-1.cfg
59 package: sparc-rtems5-gcc-7.2.0-newlib-2.5.0.20170922-x86_64-linux-gnu-1
60 download: https://ftp.gnu.org/gnu/gcc/gcc-7.2.0/gcc-7.2.0.tar.xz -> sources/gcc-7.2.0.tar.
  ↳ xz
61 downloading: sources/gcc-7.2.0.tar.xz - 59.4MB of 59.4MB (100%)
62 download: https://gcc.gnu.org/git/?p=gcc.git;a=commitdiff_plain;
  ↳ h=62ffbc7502f0ff88ff7566cd6d7c59c0483ecc0 -> patches/gcc-
  ↳ 62ffbc7502f0ff88ff7566cd6d7c59c0483ecc0.patch
63 downloading: patches/gcc-62ffbc7502f0ff88ff7566cd6d7c59c0483ecc0.patch - 1.8kB
64 download: https://gcc.gnu.org/git/?p=gcc.git;a=blobdiff_plain;f=gcc/config.gcc;
  ↳ h=593631849bb5e0df5cc4ff42c1a1cc34b7eec2f8;hp=a9196cd26d9ec24c2e3f6026f63348cae3734861;
  ↳ hb=e840389000b8339a63bee56d8b3...<see log> -> patches/gcc-
  ↳ 593631849bb5e0df5cc4ff42c1a1cc34b7eec2f8.patch

```

(continues on next page)

(continued from previous page)

```

65 downloading: patches/gcc-593631849bb5e0df5cc4ff42c1a1cc34b7eec2f8.patch - 806.0 bytes
66 download: https://gcc.gnu.org/git/?p=gcc.git;a=blobdiff_plain;f=gcc/config/rs6000/rtems.h;
    ↪ h=7ea9ebdb77b6a9b7060ad2362318e0e12b9058ae;hp=8a62fdcbaf321d616021c4c396619b7f56cf5ed2;
    ↪ hb=e840389000b8339a...<see log> -> patches/gcc-7ea9ebdb77b6a9b7060ad2362318e0e12b9058ae.
    ↪ patch
67 downloading: patches/gcc-7ea9ebdb77b6a9b7060ad2362318e0e12b9058ae.patch - 3.2kB
68 download: ftp://sourceware.org/pub/newlib/newlib-2.5.0.20170922.tar.gz -> sources/newlib-
    ↪ 2.5.0.20170922.tar.gz
69 downloading: sources/newlib-2.5.0.20170922.tar.gz - 17.3MB of 17.3MB (100%)
70 download: https://devel.rtems.org/raw-attachment/ticket/2514/0001-RTEMS-Self-contained-
    ↪ POSIX-objects.patch -> patches/0001-RTEMS-Self-contained-POSIX-objects.patch
71 downloading: patches/0001-RTEMS-Self-contained-POSIX-objects.patch - 5.7kB of 5.7kB (100%)
72 download: https://sourceware.org/git/gitweb.cgi?p=newlib-cygwin.git;a=patch;
    ↪ h=c165a27c0147471977377acd8918ab3b446f947a -> patches/newlib-cygwin-git-
    ↪ c165a27c0147471977377acd8918ab3b446f947a.patch
73 downloading: patches/newlib-cygwin-git-c165a27c0147471977377acd8918ab3b446f947a.patch -
    ↪ 986.0 bytes
74 download: https://sourceware.org/git/gitweb.cgi?p=newlib-cygwin.git;a=patch;
    ↪ h=ce189d8afef720b0977b5cae7f9eabf5d49b530c -> patches/newlib-cygwin-git-
    ↪ ce189d8afef720b0977b5cae7f9eabf5d49b530c.patch
75 downloading: patches/newlib-cygwin-git-ce189d8afef720b0977b5cae7f9eabf5d49b530c.patch - 3.
    ↪ 4kB
76 download: https://ftp.gnu.org/gnu/mpfr/mpfr-3.1.4.tar.bz2 -> sources/mpfr-3.1.4.tar.bz2
77 downloading: sources/mpfr-3.1.4.tar.bz2 - 1.2MB of 1.2MB (100%)
78 download: https://ftp.gnu.org/gnu/mpc/mpc-1.0.3.tar.gz -> sources/mpc-1.0.3.tar.gz
79 downloading: sources/mpc-1.0.3.tar.gz - 654.2kB of 654.2kB (100%)
80 download: https://ftp.gnu.org/gnu/gmp/gmp-6.1.0.tar.bz2 -> sources/gmp-6.1.0.tar.bz2
81 downloading: sources/gmp-6.1.0.tar.bz2 - 2.3MB of 2.3MB (100%)
82 building: sparc-rtems5-gcc-7.2.0-newlib-2.5.0.20170922-x86_64-linux-gnu-1
83 reporting: tools/rtems-gcc-7.2.0-newlib-2.5.0.20170922-1.cfg -> sparc-rtems5-gcc-7.2.0-
    ↪ newlib-2.5.0.20170922-x86_64-linux-gnu-1.txt
84 reporting: tools/rtems-gcc-7.2.0-newlib-2.5.0.20170922-1.cfg -> sparc-rtems5-gcc-7.2.0-
    ↪ newlib-2.5.0.20170922-x86_64-linux-gnu-1.xml
85 config: tools/rtems-gdb-8.0.1-1.cfg
86 package: sparc-rtems5-gdb-8.0.1-x86_64-linux-gnu-1
87 download: http://ftp.gnu.org/gnu/gdb/gdb-8.0.1.tar.xz -> sources/gdb-8.0.1.tar.xz
88 downloading: sources/gdb-8.0.1.tar.xz - 18.7MB of 18.7MB (100%)
89 download: https://gaisler.org/gdb/gdb-8.0.1-sis-leon2-leon3.diff -> patches/gdb-8.0.1-sis-
    ↪ leon2-leon3.diff
90 downloading: patches/gdb-8.0.1-sis-leon2-leon3.diff - 224.5kB of 224.5kB (100%)
91 building: sparc-rtems5-gdb-8.0.1-x86_64-linux-gnu-1
92 reporting: tools/rtems-gdb-8.0.1-1.cfg -> sparc-rtems5-gdb-8.0.1-x86_64-linux-gnu-1.txt
93 reporting: tools/rtems-gdb-8.0.1-1.cfg -> sparc-rtems5-gdb-8.0.1-x86_64-linux-gnu-1.xml
94 config: tools/rtems-tools-5-1.cfg
95 package: rtems-tools-HEAD-1
96 Creating source directory: sources/git
97 git: clone: git://git.rtems.org/rtems-tools.git -> sources/git/rtems-tools.git
98 git: reset: git://git.rtems.org/rtems-tools.git
99 git: fetch: git://git.rtems.org/rtems-tools.git -> sources/git/rtems-tools.git
100 git: checkout: git://git.rtems.org/rtems-tools.git => HEAD
101 git: pull: git://git.rtems.org/rtems-tools.git
102 building: rtems-tools-HEAD-1
103 reporting: tools/rtems-tools-5-1.cfg -> rtems-tools-HEAD-1.txt
104 reporting: tools/rtems-tools-5-1.cfg -> rtems-tools-HEAD-1.xml
105 config: tools/rtems-kernel-5.cfg
106 package: sparc-rtems5-kernel-5-1

```

(continues on next page)

(continued from previous page)

```

107 building: sparc-rtems5-kernel-5-1
108 reporting: tools/rtems-kernel-5.cfg -> sparc-rtems5-kernel-5-1.txt
109 reporting: tools/rtems-kernel-5.cfg -> sparc-rtems5-kernel-5-1.xml
110 installing: expat-2.1.0-x86_64-linux-gnu-1 -> /usr/home/chris/development/rtems/5
111 installing: sparc-rtems5-binutils-2.29-x86_64-linux-gnu-1 -> /usr/home/chris/development/
    ↳ rtems/5
112 installing: sparc-rtems5-gcc-7.2.0-newlib-2.5.0.20170922-x86_64-linux-gnu-1 -> /usr/home/
    ↳ chris/development/rtems/5
113 installing: sparc-rtems5-gdb-8.0.1-x86_64-linux-gnu-1 -> /usr/home/chris/development/
    ↳ rtems/5
114 installing: rtems-tools-HEAD-1 -> /usr/home/chris/development/rtems/5
115 installing: sparc-rtems5-kernel-5-1 -> /usr/home/chris/development/rtems/5
116 cleaning: expat-2.1.0-x86_64-linux-gnu-1
117 cleaning: sparc-rtems5-binutils-2.29-x86_64-linux-gnu-1
118 cleaning: sparc-rtems5-gcc-7.2.0-newlib-2.5.0.20170922-x86_64-linux-gnu-1
119 cleaning: sparc-rtems5-gdb-8.0.1-x86_64-linux-gnu-1
120 cleaning: rtems-tools-HEAD-1
121 cleaning: sparc-rtems5-kernel-5-1
122 Build Set: Time 0:39:33.988995

```

5.2.2 Windows Host Tool Chain

This section details how you create an RTEMS development environment on Windows. The installation documented here is on *Windows 7 64bit Professional*. Building on *Windows 10* has been reported as working.

Please see *Microsoft Windows* (page 46) before continuing.

Note: If the RSB reports error: no hosts defaults found; please add you have probably opened an MSYS2 32bit Shell. Close all 32bit Shell windows and open the MSYS2 64bit Shell.

5.2.2.1 RTEMS Windows Tools

Create a workspace for RTEMS using the following shell command:

Creating Tool Archives

Add `--bset-tar-file` to the `sb-set-builder` command line to create tar files of the built package set.

```

1 ~
2 $ mkdir -p /c/opt/rtems

```

The `/c` path is an internal MSYS2 mount point of the C: drive. The command creates the RTEMS work space on the C: drive. If you wish to use another drive please substitute `/c` with your drive letter.

We build and install all RTEMS packages under the *prefix* we just created. Change to that directory and get a copy of the RSB:

```

1 ~
2 $ cd /c/opt/rtems
3 /c/opt/rtems
4 $ git clone git://git.rtems.org/rtems-source-builder.git rsb
5 Cloning into 'rsb'...
6 remote: Counting objects: 5716, done.
7 remote: Compressing objects: 100% (2183/2183), done.
8 remote: Total 5716 (delta 3919), reused 5071 (delta 3494)
9 Receiving objects: 100% (5716/5716), 2.46 MiB | 656.00 KiB/s, done.
10 Resolving deltas: 100% (3919/3919), done.
11 Checking connectivity... done.
12 Checking out files: 100% (630/630), done.
13 /c/opt/rtems
14 $ cd rsb

```

We are building RTEMS 4.11 tools so select the *4.11* branch:

```

1 /c/opt/rtems/rsb
2 $ git checkout 4.11
3 Branch 4.11 set up to track remote branch 4.11 from origin.
4 Switched to a new branch '4.11'
5 /c/opt/rtems/rsb
6 $

```

Check the RSB has a valid environment:

```

1 /c/opt/rtems/rsb
2 $ cd rtems
3 /c/opt/rtems/rsb/rtems
4 $ ../source-builder/sb-check
5 RTEMS Source Builder - Check, 4.11 (01ac76f2f90f)
6 Environment is ok
7 /c/opt/rtems/rsb/rtems
8 $

```

To build a set of RTEMS tools for the Intel i386 architecture. The build runs a single job rather than a job per CPU in your machine and will take a long time so please be patient. The RSB creates a log file containing all the build output and it will be changing size. The RSB command to build i386 tools is:

```

1 /c/opt/rtems/rsb/rtems
2 $ ../source-builder/sb-set-builder --prefix=/c/opt/rtems/4.11 \
3                                     --jobs=none 4.11/rtems-i386
4 RTEMS Source Builder - Set Builder, 4.11 (01ac76f2f90f)
5 Build Set: 4.11/rtems-i386
6 Build Set: 4.11/rtems-autotools.bset
7 Build Set: 4.11/rtems-autotools-internal.bset
8 config: tools/rtems-autoconf-2.69-1.cfg
9 package: autoconf-2.69-x86_64-w64-mingw32-1
10 Creating source directory: sources
11 download: ftp://ftp.gnu.org/gnu/autoconf/autoconf-2.69.tar.gz -> sources/autoconf-2.69.
12 ↪tar.gz
13 downloading: sources/autoconf-2.69.tar.gz - 1.8MB of 1.8MB (100%)
14 building: autoconf-2.69-x86_64-w64-mingw32-1
15 config: tools/rtems-automake-1.12.6-1.cfg
16 package: automake-1.12.6-x86_64-w64-mingw32-1

```

(continues on next page)

(continued from previous page)

```

16 download: ftp://ftp.gnu.org/gnu/automake/automake-1.12.6.tar.gz -> sources/automake-1.12.
   ↳ 6.tar.gz
17 downloading: sources/automake-1.12.6.tar.gz - 2.0MB of 2.0MB (100%)
18 building: automake-1.12.6-x86_64-w64-mingw32-1
19 cleaning: autoconf-2.69-x86_64-w64-mingw32-1
20 cleaning: automake-1.12.6-x86_64-w64-mingw32-1
21 Build Set: Time 0:00:42.515625
22 Build Set: 4.11/rtems-autotools-base.bset
23 config: tools/rtems-autoconf-2.69-1.cfg
24 package: autoconf-2.69-x86_64-w64-mingw32-1
25 building: autoconf-2.69-x86_64-w64-mingw32-1
26 reporting: tools/rtems-autoconf-2.69-1.cfg -> autoconf-2.69-x86_64-w64-mingw32-1.txt
27 reporting: tools/rtems-autoconf-2.69-1.cfg -> autoconf-2.69-x86_64-w64-mingw32-1.xml
28 config: tools/rtems-automake-1.12.6-1.cfg
29 package: automake-1.12.6-x86_64-w64-mingw32-1
30 building: automake-1.12.6-x86_64-w64-mingw32-1
31 reporting: tools/rtems-automake-1.12.6-1.cfg -> automake-1.12.6-x86_64-w64-mingw32-1.txt
32 reporting: tools/rtems-automake-1.12.6-1.cfg -> automake-1.12.6-x86_64-w64-mingw32-1.xml
33 tarball: tar/rtems-4.11-autotools-x86_64-w64-mingw32-1.tar.bz2
34 installing: autoconf-2.69-x86_64-w64-mingw32-1 -> C:\opt\rtems\4.11
35 installing: automake-1.12.6-x86_64-w64-mingw32-1 -> C:\opt\rtems\4.11
36 cleaning: autoconf-2.69-x86_64-w64-mingw32-1
37 cleaning: automake-1.12.6-x86_64-w64-mingw32-1
38 Build Set: Time 0:00:37.718750
39 Build Set: Time 0:01:20.234375
40 config: devel/expat-2.1.0-1.cfg
41 package: expat-2.1.0-x86_64-w64-mingw32-1
42 download: http://downloads.sourceforge.net/project/expat/expat/2.1.0/expat-2.1.0.tar.gz ->
   ↳ sources/expat-2.1.0.tar.gz
43 redirect: http://iweb.dl.sourceforge.net/project/expat/expat/2.1.0/expat-2.1.0.tar.gz
44 downloading: sources/expat-2.1.0.tar.gz - 549.4kB of 549.4kB (100%)
45 building: expat-2.1.0-x86_64-w64-mingw32-1
46 reporting: devel/expat-2.1.0-1.cfg -> expat-2.1.0-x86_64-w64-mingw32-1.txt
47 reporting: devel/expat-2.1.0-1.cfg -> expat-2.1.0-x86_64-w64-mingw32-1.xml
48 config: tools/rtems-binutils-2.24-1.cfg
49 package: i386-rtems4.11-binutils-2.24-x86_64-w64-mingw32-1
50 download: ftp://ftp.gnu.org/gnu/binutils/binutils-2.24.tar.bz2 -> sources/binutils-2.24.
   ↳ tar.bz2
51 downloading: sources/binutils-2.24.tar.bz2 - 21.7MB of 21.7MB (100%)
52 building: i386-rtems4.11-binutils-2.24-x86_64-w64-mingw32-1
53 reporting: tools/rtems-binutils-2.24-1.cfg -> i386-rtems4.11-binutils-2.24-x86_64-w64-
   ↳ mingw32-1.txt
54 reporting: tools/rtems-binutils-2.24-1.cfg -> i386-rtems4.11-binutils-2.24-x86_64-w64-
   ↳ mingw32-1.xml
55 config: tools/rtems-gcc-4.9.3-newlib-2.2.0-20150423-1.cfg
56 package: i386-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-w64-mingw32-1
57 download: ftp://ftp.gnu.org/gnu/gcc/gcc-4.9.3/gcc-4.9.3.tar.bz2 -> sources/gcc-4.9.3.tar.
   ↳ bz2
58 downloading: sources/gcc-4.9.3.tar.bz2 - 85.8MB of 85.8MB (100%)
59 download: ftp://sourceware.org/pub/newlib/newlib-2.2.0.20150423.tar.gz -> sources/newlib-
   ↳ 2.2.0.20150423.tar.gz
60 downloading: sources/newlib-2.2.0.20150423.tar.gz - 16.7MB of 16.7MB (100%)
61 download: http://www.mpfr.org/mpfr-3.0.1/mpfr-3.0.1.tar.bz2 -> sources/mpfr-3.0.1.tar.bz2
62 downloading: sources/mpfr-3.0.1.tar.bz2 - 1.1MB of 1.1MB (100%)
63 download: http://www.multiprecision.org/mpc/download/mpc-0.8.2.tar.gz -> sources/mpc-0.8.
   ↳ 2.tar.gz

```

(continues on next page)

(continued from previous page)

```

64 downloading: sources/mpc-0.8.2.tar.gz - 535.5kB of 535.5kB (100%)
65 download: ftp://ftp.gnu.org/gnu/gmp/gmp-5.0.5.tar.bz2 -> sources/gmp-5.0.5.tar.bz2
66 downloading: sources/gmp-5.0.5.tar.bz2 - 2.0MB of 2.0MB (100%)
67 building: i386-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-w64-mingw32-1
68 reporting: tools/rtems-gcc-4.9.3-newlib-2.2.0-20150423-1.cfg ->
69 i386-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-w64-mingw32-1.txt
70 reporting: tools/rtems-gcc-4.9.3-newlib-2.2.0-20150423-1.cfg ->
71 i386-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-w64-mingw32-1.xml
72 config: tools/rtems-gdb-7.9-1.cfg
73 package: i386-rtems4.11-gdb-7.9-x86_64-w64-mingw32-1
74 download: http://ftp.gnu.org/gnu/gdb/gdb-7.9.tar.xz -> sources/gdb-7.9.tar.xz
75 downloading: sources/gdb-7.9.tar.xz - 17.0MB of 17.0MB (100%)
76 download: https://git.rtems.org/rtems-tools/plain/tools/4.11/gdb/gdb-sim-arange-inline.
  ↪diff -> patches/gdb-sim-arange-inline.diff
77 downloading: patches/gdb-sim-arange-inline.diff - 761.0 bytes of 761.0 bytes (100%)
78 download: https://git.rtems.org/rtems-tools/plain/tools/4.11/gdb/gdb-sim-cgen-inline.diff ↪
  ↪-> patches/gdb-sim-cgen-inline.diff
79 downloading: patches/gdb-sim-cgen-inline.diff - 706.0 bytes of 706.0 bytes (100%)
80 building: i386-rtems4.11-gdb-7.9-x86_64-w64-mingw32-1
81 reporting: tools/rtems-gdb-7.9-1.cfg ->
82 i386-rtems4.11-gdb-7.9-x86_64-w64-mingw32-1.txt
83 reporting: tools/rtems-gdb-7.9-1.cfg ->
84 i386-rtems4.11-gdb-7.9-x86_64-w64-mingw32-1.xml
85 config: tools/rtems-tools-4.11-1.cfg
86 package: rtems-tools-4.11-1
87 Creating source directory: sources/git
88 git: clone: git://git.rtems.org/rtems-tools.git -> sources/git/rtems-tools.git
89 git: reset: git://git.rtems.org/rtems-tools.git
90 git: fetch: git://git.rtems.org/rtems-tools.git -> sources/git/rtems-tools.git
91 git: checkout: git://git.rtems.org/rtems-tools.git => 4.11
92 git: pull: git://git.rtems.org/rtems-tools.git
93 building: rtems-tools-4.11-1
94 reporting: tools/rtems-tools-4.11-1.cfg -> rtems-tools-4.11-1.txt
95 reporting: tools/rtems-tools-4.11-1.cfg -> rtems-tools-4.11-1.xml
96 config: tools/rtems-kernel-4.11.cfg
97 installing: expat-2.1.0-x86_64-w64-mingw32-1 -> C:\opt\rtems\4.11
98 installing: i386-rtems4.11-binutils-2.24-x86_64-w64-mingw32-1 -> C:\opt\rtems\4.11
99 installing: i386-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-w64-mingw32-1 -> ↪
  ↪C:\opt\rtems\4.11
100 installing: i386-rtems4.11-gdb-7.9-x86_64-w64-mingw32-1 -> C:\opt\rtems\4.11
101 installing: rtems-tools-4.11-1 -> C:\opt\rtems\4.11
102 cleaning: expat-2.1.0-x86_64-w64-mingw32-1
103 cleaning: i386-rtems4.11-binutils-2.24-x86_64-w64-mingw32-1
104 cleaning: i386-rtems4.11-gcc-4.9.3-newlib-2.2.0.20150423-x86_64-w64-mingw32-1
105 cleaning: i386-rtems4.11-gdb-7.9-x86_64-w64-mingw32-1
106 cleaning: rtems-tools-4.11-1
107 Build Set: Time 1:32:58.972919
108 /c/opt/rtems/rsb/rtems
109 $

```

5.2.2.2 Building the Kernel

We can now build the RTEMS kernel using the RTEMS tools we have just built. First we need to set the path to the tools:

```

1 /c
2 $ cd /c/opt/rtems
3 /c/opt/rtems
4 $ export PATH=/c/opt/rtems/4.11/bin:$PATH
5 /c/opt/rtems
6 $

```

We currently build RTEMS from the git release branch for 4.11:

```

1 /c/opt/rtems
2 $ mkdir kernel
3 /c/opt/rtems
4 $ cd kernel
5 /c/opt/rtems/kernel
6 $ git clone git://git.rtems.org/rtems.git rtems
7 Cloning into 'rtems'...
8 remote: Counting objects: 482766, done.
9 remote: Compressing objects: 100% (88781/88781), done.
10 remote: Total 482766 (delta 389610), reused 475155 (delta 383437)
11 Receiving objects: 100% (482766/482766), 69.77 MiB | 697.00 KiB/s, done.
12 Resolving deltas: 100% (389610/389610), done.
13 Checking connectivity... done.
14 Checking out files: 100% (10626/10626), done.
15 /c/opt/rtems/kernel
16 $ cd rtems
17 /c/opt/rtems/kernel/rtems
18 $ git checkout 4.11
19 Checking out files: 100% (2553/2553), done.
20 Branch 4.11 set up to track remote branch 4.11 from origin.
21 Switched to a new branch '4.11'
22 /c/opt/rtems/kernel
23 $

```

The kernel code cloned from git needs to be *bootstrapped*. Bootstrapping creates autoconf and automake generated files. To bootstrap we first clean away any files, then generate the pre-install header file lists and finally we generate the autoconf and automake files using the RSB's bootstrap tool. First we clean any generated files that exist:

```

1 /c/opt/rtems/kernel/rtems
2 $ ./bootstrap -c
3 removing automake generated Makefile.in files
4 removing configure files
5 removing aclocal.m4 files

```

Then we generate the pre-install header file automake make files:

```

1 /c/opt/rtems/kernel/rtems
2 $ ./bootstrap -p
3 Generating ./src/ada/preinstall.am
4 Generating ./src/lib/libbsp/arm/altera-cyclone-v/preinstall.am
5 Generating ./src/lib/libbsp/arm/atsam/preinstall.am
6 Generating ./src/lib/libbsp/arm/beagle/preinstall.am
7 Generating ./src/lib/libbsp/arm/csb336/preinstall.am
8 Generating ./src/lib/libbsp/arm/csb337/preinstall.am
9 Generating ./src/lib/libbsp/arm/edb7312/preinstall.am
10 Generating ./src/lib/libbsp/arm/gdbarmsim/preinstall.am

```

(continues on next page)

(continued from previous page)

```

11 .....
12 Generating ./cpukit/score/cpu/mips/preinstall.am
13 Generating ./cpukit/score/cpu/moxie/preinstall.am
14 Generating ./cpukit/score/cpu/nios2/preinstall.am
15 Generating ./cpukit/score/cpu/no_cpu/preinstall.am
16 Generating ./cpukit/score/cpu/or1k/preinstall.am
17 Generating ./cpukit/score/cpu/powerpc/preinstall.am
18 Generating ./cpukit/score/cpu/sh/preinstall.am
19 Generating ./cpukit/score/cpu/sparc/preinstall.am
20 Generating ./cpukit/score/cpu/sparc64/preinstall.am
21 Generating ./cpukit/score/cpu/v850/preinstall.am
22 Generating ./cpukit/score/preinstall.am
23 Generating ./cpukit/telnetd/preinstall.am
24 Generating ./cpukit/wrapup/preinstall.am
25 Generating ./cpukit/zlib/preinstall.am
26 /c/opt/rtems/kernel/rtems

```

Finally we run the RSB's parallel bootstrap command:

```

1 $ /c/opt/rtems/rsb/source-builder/sb-bootstrap
2 RTEMS Source Builder - RTEMS Bootstrap, 4.11 (76188ee494dd)
3 1/139: autoreconf: configure.ac
4 2/139: autoreconf: c/configure.ac
5 3/139: autoreconf: c/src/configure.ac
6 4/139: autoreconf: c/src/ada-tests/configure.ac
7 5/139: autoreconf: c/src/lib/libbsp/arm/configure.ac
8 6/139: autoreconf: c/src/lib/libbsp/arm/altera-cyclone-v/configure.ac
9 7/139: autoreconf: c/src/lib/libbsp/arm/atsam/configure.ac
10 8/139: autoreconf: c/src/lib/libbsp/arm/beagle/configure.ac
11 9/139: autoreconf: c/src/lib/libbsp/arm/csb336/configure.ac
12 10/139: autoreconf: c/src/lib/libbsp/arm/csb337/configure.ac
13 11/139: autoreconf: c/src/lib/libbsp/arm/edb7312/configure.ac
14 .....
15 129/139: autoreconf: testsuites/samples/configure.ac
16 130/139: autoreconf: testsuites/smptests/configure.ac
17 131/139: autoreconf: testsuites/sptests/configure.ac
18 132/139: autoreconf: testsuites/tmtests/configure.ac
19 133/139: autoreconf: testsuites/tools/configure.ac
20 134/139: autoreconf: testsuites/tools/generic/configure.ac
21 135/139: autoreconf: tools/build/configure.ac
22 136/139: autoreconf: tools/cpu/configure.ac
23 137/139: autoreconf: tools/cpu/generic/configure.ac
24 138/139: autoreconf: tools/cpu/nios2/configure.ac
25 139/139: autoreconf: tools/cpu/sh/configure.ac
26 Bootstrap time: 0:20:38.759766
27 /c/opt/rtems/kernel/rtems
28 $

```

We will build the RTEMS kernel for the i386 target and the pc686 BSP. You can check the available BSPs by running the `rtems-bsps` command found in the top directory of the RTEMS kernel source. We build the Board Support Package (BSP) outside the kernel source tree:

```

1 /c/opt/rtems/kernel/rtems
2 $ cd ..
3 /c/opt/rtems/kernel

```

(continues on next page)

(continued from previous page)

```

4 $ mkdir pc686
5   /c/opt/rtems/kernel
6 $ cd pc686
7   /c/opt/rtems/kernel/pc686
8 $

```

Configure the RTEMS kernel to build pc686 BSP for the i386 target with networking disabled, We will build the external libBSD stack later:

```

1  /c/opt/rtems/kernel/pc686
2 $ /c/opt/rtems/kernel/rtems/configure --prefix=/c/opt/rtems/4.11 \
3     --target=i386-rtems4.11 --disable-networking --enable-rtemsbsp=pc686
4 checking for gmake... no
5 checking for make... make
6 checking for RTEMS Version... 4.11.99.0
7 checking build system type... x86_64-pc-mingw64
8 checking host system type... x86_64-pc-mingw64
9 checking target system type... i386-pc-rtems4.11
10 checking for a BSD-compatible install... /usr/bin/install -c
11 checking whether build environment is sane... yes
12 checking for a thread-safe mkdir -p... /usr/bin/mkdir -p
13 checking for gawk... gawk
14 checking whether make sets $(MAKE)... yes
15 checking whether to enable maintainer-specific portions of Makefiles... no
16 checking that generated files are newer than configure... done
17 configure: creating ./config.status
18 configure: configuring in ./tools/build
19     ....
20 checking whether make sets $(MAKE)... yes
21 checking whether to enable maintainer-specific portions of Makefiles... no
22 checking that generated files are newer than configure... done
23 configure: creating ./config.status
24 config.status: creating Makefile
25
26 target architecture: i386.
27 available BSPs: pc686.
28 'make all' will build the following BSPs: pc686.
29 other BSPs can be built with 'make RTEMS_BSP="bsp1 bsp2 ..."'
30
31 config.status: creating Makefile
32 /c/opt/rtems/kernel/pc686
33 $

```

Build the kernel:

```

1  /c/opt/rtems/kernel/pc686
2 $ make
3 Making all in tools/build
4 make[1]: Entering directory '/c/opt/rtems/kernel/pc686/tools/build'
5 make  all-am
6 make[2]: Entering directory '/c/opt/rtems/kernel/pc686/tools/build'
7 gcc -DHAVE_CONFIG_H -I. -I/c/opt/rtems/kernel/rtems/tools/build      -g -O2 -MT
8 cklength.o -MD -MP -MF .deps/cklength.Tpo -c -o cklength.o
9 /c/opt/rtems/kernel/rtems/tools/build/cklength.c
10 gcc -DHAVE_CONFIG_H -I. -I/c/opt/rtems/kernel/rtems/tools/build      -g -O2 -MT

```

(continues on next page)

(continued from previous page)

```

11 eolstrip.o -MD -MP -MF .deps/eolstrip.Tpo -c -o eolstrip.o
12 /c/opt/rtems/kernel/rtems/tools/build/eolstrip.c
13 .....
14 i386-rtems4.11-objcopy -O binary nsecs.nxe nsecs.bin
15 ../../../../pc686/build-tools/bin2boot -v nsecs.ralf 0x00097E00
16 ../../../../pc686/lib/start16.bin 0x00097C00 0 nsecs.bin 0x00100000 0
17 header address      0x00097e00, its memory size 0xzx
18 first image address 0x00097c00, its memory size 0x00000200
19 second image address 0x00100000, its memory size 0x0003d800
20 rm -f nsecs.nxe
21 make[6]: Leaving directory '/c/opt/rtems/kernel/pc686/i386-rtems4.11/c/pc686/testsuites/
  ↳ samples/nsecs'
22 make[5]: Leaving directory '/c/opt/rtems/kernel/pc686/i386-rtems4.11/c/pc686/testsuites/
  ↳ samples'
23 make[4]: Leaving directory '/c/opt/rtems/kernel/pc686/i386-rtems4.11/c/pc686/testsuites/
  ↳ samples'
24 make[4]: Entering directory '/c/opt/rtems/kernel/pc686/i386-rtems4.11/c/pc686/testsuites'
25 make[4]: Nothing to be done for 'all-am'.
26 make[4]: Leaving directory '/c/opt/rtems/kernel/pc686/i386-rtems4.11/c/pc686/testsuites'
27 make[3]: Leaving directory '/c/opt/rtems/kernel/pc686/i386-rtems4.11/c/pc686/testsuites'
28 make[2]: Leaving directory '/c/opt/rtems/kernel/pc686/i386-rtems4.11/c/pc686'
29 make[1]: Leaving directory '/c/opt/rtems/kernel/pc686/i386-rtems4.11/c'
30 make[1]: Entering directory '/c/opt/rtems/kernel/pc686'
31 make[1]: Nothing to be done for 'all-am'.
32 make[1]: Leaving directory '/c/opt/rtems/kernel/pc686'
33 /c/opt/rtems/kernel/pc686
34 $

```

Install the kernel to our prefix:

```

1 $ make install
2 Making install in tools/build
3 make[1]: Entering directory '/c/opt/rtems/kernel/pc686/tools/build'
4 make[2]: Entering directory '/c/opt/rtems/kernel/pc686/tools/build'
5 /usr/bin/mkdir -p '/c/opt/rtems/4.11/bin'
6 /usr/bin/install -c cklength.exe eolstrip.exe packhex.exe unhex.exe
7 rtems-bin2c.exe '/c/opt/rtems/4.11/bin'
8 /usr/bin/mkdir -p '/c/opt/rtems/4.11/bin'
9 /usr/bin/install -c install-if-change '/c/opt/rtems/4.11/bin'
10 make[2]: Nothing to be done for 'install-data-am'.
11 make[2]: Leaving directory '/c/opt/rtems/kernel/pc686/tools/build'
12 make[1]: Leaving directory '/c/opt/rtems/kernel/pc686/tools/build'
13 Making install in tools/cpu
14 make[1]: Entering directory '/c/opt/rtems/kernel/pc686/tools/cpu'
15 Making install in generic
16 make[2]: Entering directory '/c/opt/rtems/kernel/pc686/tools/cpu/generic'
17 make[3]: Entering directory '/c/opt/rtems/kernel/pc686/tools/cpu/generic'
18 make[3]: Nothing to be done for 'install-exec-am'.
19 make[3]: Nothing to be done for 'install-data-am'.
20 make[3]: Leaving directory '/c/opt/rtems/kernel/pc686/tools/cpu/generic'
21 make[2]: Leaving directory '/c/opt/rtems/kernel/pc686/tools/cpu/generic'
22 make[2]: Entering directory '/c/opt/rtems/kernel/pc686/tools/cpu'
23 make[3]: Entering directory '/c/opt/rtems/kernel/pc686/tools/cpu'
24 make[3]: Nothing to be done for 'install-exec-am'.
25 make[3]: Nothing to be done for 'install-data-am'.
26 .....

```

(continues on next page)

(continued from previous page)

```
27 make[2]: Entering directory '/c/opt/rtems/kernel/pc686'
28 make[2]: Nothing to be done for 'install-exec-am'.
29 /usr/bin/mkdir -p '/c/opt/rtems/4.11/make'
30 /usr/bin/install -c -m 644 /c/opt/rtems/kernel/rtems/make/main.cfg
31 /c/opt/rtems/kernel/rtems/make/leaf.cfg '/c/opt/rtems/4.11/make'
32 /usr/bin/mkdir -p '/c/opt/rtems/4.11/share/rtems4.11/make/Templates'
33 /usr/bin/install -c -m 644
34 /c/opt/rtems/kernel/rtems/make/Templates/Makefile.dir
35 /c/opt/rtems/kernel/rtems/make/Templates/Makefile.leaf
36 /c/opt/rtems/kernel/rtems/make/Templates/Makefile.lib
37 '/c/opt/rtems/4.11/share/rtems4.11/make/Templates'
38 /usr/bin/mkdir -p '/c/opt/rtems/4.11/make/custom'
39 /usr/bin/install -c -m 644 /c/opt/rtems/kernel/rtems/make/custom/default.cfg
40 '/c/opt/rtems/4.11/make/custom'
41 make[2]: Leaving directory '/c/opt/rtems/kernel/pc686'
42 make[1]: Leaving directory '/c/opt/rtems/kernel/pc686'
43 /c/opt/rtems/kernel/pc686
44 $
```

5.3 RTEMS Kernel

RTEMS is an open source real-time operating system. As a user you have access to all the source code. The RTEMS Kernel section will show you how you build the RTEMS kernel on your host.

5.3.1 Development Sources

Create a new location to build the RTEMS kernel:

```
1 $ cd
2 $ cd development/rtems
3 $ mkdir kernel
4 $ cd kernel
```

Clone the RTEMS repository:

```
1 $ git clone git://git.rtems.org/rtems.git rtems
2 Cloning into 'rtems'...
3 remote: Counting objects: 483342, done.
4 remote: Compressing objects: 100% (88974/88974), done.
5 remote: Total 483342 (delta 390053), reused 475669 (delta 383809)
6 Receiving objects: 100% (483342/483342), 69.88 MiB | 1.37 MiB/s, done.
7 Resolving deltas: 100% (390053/390053), done.
8 Checking connectivity... done.
```

5.3.2 Tools Path Set Up

We need to set our path to include the RTEMS tools we built in the previous section. The RTEMS tools needs to be first in your path because RTEMS provides specific versions of the autoconf and automake tools. We want to use the RTEMS version and not your host's versions:

```
1 $ export PATH=$HOME/development/rtems/5/bin:$PATH
```

5.3.3 Bootstrapping

The developers version of the code from git requires we bootstrap the source code. This is an autoconf and automake bootstrap to create the various files generated by autoconf and automake. RTEMS does not keep these generated files under version control. The bootstrap process is slow so to speed it up the RSB provides a command that can perform the bootstrap in parallel using your available cores. We need to enter the cloned source directory then run the bootstrap commands:

```
1 $ cd rtems
2 $ ./bootstrap -c && $HOME/development/rtems/rsb/source-builder/sb-bootstrap
3 removing automake generated Makefile.in files
4 removing configure files
5 removing aclocal.m4 files
6 RTEMS Source Builder - RTEMS Bootstrap, 5 (089327b5dcf9)
7 1/139: autoreconf: configure.ac
8 2/139: autoreconf: cpukit/configure.ac
9 3/139: autoreconf: tools/cpu/configure.ac
```

(continues on next page)

(continued from previous page)

```

10 4/139: autoreconf: tools/cpu/generic/configure.ac
11 5/139: autoreconf: tools/cpu/sh/configure.ac
12 6/139: autoreconf: tools/cpu/nios2/configure.ac
13 7/139: autoreconf: tools/build/configure.ac
14 8/139: autoreconf: doc/configure.ac
15 .....
16 124/139: autoreconf: c/src/make/configure.ac
17 125/139: autoreconf: c/src/librtems++/configure.ac
18 126/139: autoreconf: c/src/ada-tests/configure.ac
19 127/139: autoreconf: testsuites/configure.ac
20 128/139: autoreconf: testsuites/libtests/configure.ac
21 129/139: autoreconf: testsuites/mptests/configure.ac
22 130/139: autoreconf: testsuites/fstests/configure.ac
23 131/139: autoreconf: testsuites/sptests/configure.ac
24 132/139: autoreconf: testsuites/tmtests/configure.ac
25 133/139: autoreconf: testsuites/smptests/configure.ac
26 134/139: autoreconf: testsuites/tools/configure.ac
27 135/139: autoreconf: testsuites/tools/generic/configure.ac
28 136/139: autoreconf: testsuites/psxtests/configure.ac
29 137/139: autoreconf: testsuites/psxtmtests/configure.ac
30 138/139: autoreconf: testsuites/rhealstone/configure.ac
31 139/139: autoreconf: testsuites/samples/configure.ac
32 Bootstrap time: 0:02:47.398824

```

5.3.4 Building a BSP

We build RTEMS in a directory outside of the source tree we have just cloned and bootstrapped. You cannot build RTEMS while in the source tree. Lets create a suitable directory using the name of the BSP we are going to build:

```

1 $ cd ..
2 $ mkdir erc32
3 $ cd erc32

```

Configure RTEMS using the configure command. We use a full path to configure so the object files built contain the absolute path of the source files. If you are source level debugging you will be able to access the source code to RTEMS from the debugger. We will build for the erc32 BSP with POSIX enabled and the networking stack disabled:

```

1 $ $HOME/development/rtems/kernel/rtems/configure --prefix=$HOME/development/rtems/5 \
2           --target=sparc-rtems5 --enable-rtemsbsp=erc32 --enable-posix \
3           --disable-networking
4 checking for gmake... no
5 checking for make... make
6 checking for RTEMS Version... 4.11.99.0
7 checking build system type... x86_64-pc-linux-gnu
8 checking host system type... x86_64-pc-linux-gnu
9 checking target system type... sparc-unknown-rtems5
10 checking for a BSD-compatible install... /usr/bin/install -c
11 checking whether build environment is sane... yes
12 checking for a thread-safe mkdir -p... /bin/mkdir -p
13 checking for gawk... no
14 checking for mawk... mawk

```

(continues on next page)

(continued from previous page)

```

15 checking whether make sets $(MAKE)... yes
16 checking whether to enable maintainer-specific portions of Makefiles... no
17 checking that generated files are newer than configure... done
18 .....
19 checking target system type... sparc-unknown-rtems5
20 checking rtems target cpu... sparc
21 checking for a BSD-compatible install... /usr/bin/install -c
22 checking whether build environment is sane... yes
23 checking for sparc-rtems5-strip... sparc-rtems5-strip
24 checking for a thread-safe mkdir -p... /bin/mkdir -p
25 checking for gawk... no
26 checking for mawk... mawk
27 checking whether make sets $(MAKE)... yes
28 checking whether to enable maintainer-specific portions of Makefiles... no
29 checking that generated files are newer than configure... done
30 configure: creating ./config.status
31 config.status: creating Makefile
32
33 target architecture: sparc.
34 available BSPs: erc32.
35 'make all' will build the following BSPs: erc32.
36 other BSPs can be built with 'make RTEMS_BSP="bsp1 bsp2 ..."'
37
38 config.status: creating Makefile

```

Build RTEMS using two cores:

```

1 $ make -j 2
2 Making all in tools/build
3 make[1]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/build'
4 make all-am
5 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/build'
6 gcc -DHAVE_CONFIG_H -I. -I/home/chris/development/rtems/kernel/rtems/tools/build -g -
  ↪O2 -MT cklength.o -MD -MP -MF .deps/cklength.Tpo -c -o cklength.o /home/chris/
  ↪development/rtems/kernel/rtems/tools/build/cklength.c
7 gcc -DHAVE_CONFIG_H -I. -I/home/chris/development/rtems/kernel/rtems/tools/build -g -
  ↪O2 -MT eolstrip.o -MD -MP -MF .deps/eolstrip.Tpo -c -o eolstrip.o /home/chris/
  ↪development/rtems/kernel/rtems/tools/build/eolstrip.c
8 mv -f .deps/cklength.Tpo .deps/cklength.Po
9 mv -f .deps/eolstrip.Tpo .deps/eolstrip.Po
10 gcc -DHAVE_CONFIG_H -I. -I/home/chris/development/rtems/kernel/rtems/tools/build -g -
  ↪O2 -MT compat.o -MD -MP -MF .deps/compat.Tpo -c -o compat.o /home/chris/development/
  ↪rtems/kernel/rtems/tools/build/compat.c
11 gcc -DHAVE_CONFIG_H -I. -I/home/chris/development/rtems/kernel/rtems/tools/build -g -
  ↪O2 -MT packhex.o -MD -MP -MF .deps/packhex.Tpo -c -o packhex.o /home/chris/development/
  ↪rtems/kernel/rtems/tools/build/packhex.c
12 mv -f .deps/compat.Tpo .deps/compat.Po
13 gcc -DHAVE_CONFIG_H -I. -I/home/chris/development/rtems/kernel/rtems/tools/build -g -
  ↪O2 -MT unhex.o -MD -MP -MF .deps/unhex.Tpo -c -o unhex.o /home/chris/development/rtems/
  ↪kernel/rtems/tools/build/unhex.c
14 mv -f .deps/packhex.Tpo .deps/packhex.Po
15 gcc -DHAVE_CONFIG_H -I. -I/home/chris/development/rtems/kernel/rtems/tools/build -g -
  ↪O2 -MT rtems-bin2c.o -MD -MP -MF .deps/rtems-bin2c.Tpo -c -o rtems-bin2c.o /home/chris/
  ↪development/rtems/kernel/rtems/tools/build/rtems-bin2c.c
16 mv -f .deps/unhex.Tpo .deps/unhex.Po
17 gcc -DHAVE_CONFIG_H -I. -I/home/chris/development/rtems/kernel/rtems/tools/build -g -
  ↪O2 -MT binpatch.o -MD -MP -MF .deps/binpatch.Tpo -c -o binpatch.o /home/chris/
  ↪development/rtems/kernel/rtems/tools/build/binpatch.c

```

(continued from previous page)

```

18 mv -f .deps/rtems-bin2c.Tpo .deps/rtems-bin2c.Po
19 gcc -g -O2 -o cklength cklength.o
20 mv -f .deps/binpatch.Tpo .deps/binpatch.Po
21 gcc -g -O2 -o eolstrip eolstrip.o compat.o
22 gcc -g -O2 -o packhex packhex.o
23 gcc -g -O2 -o rtems-bin2c rtems-bin2c.o compat.o
24 gcc -g -O2 -o unhex unhex.o compat.o
25 gcc -g -O2 -o binpatch binpatch.o
26 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/build'
27 make[1]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/build'
28 Making all in tools/cpu
29 make[1]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
30 Making all in generic
31 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/cpu/generic'
32 make[2]: Nothing to be done for 'all'.
33 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/cpu/generic'
34 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
35 make[2]: Nothing to be done for 'all-am'.
36 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
37 make[1]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
38 Making all in testsuites/tools
39 make[1]: Entering directory '/home/chris/development/rtems/kernel/erc32/testsuites/tools'
40 Making all in generic
41 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32/testsuites/tools/
↳ generic'
42 make[2]: Nothing to be done for 'all'.
43 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32/testsuites/tools/
↳ generic'
44 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32/testsuites/tools'
45 make[2]: Nothing to be done for 'all-am'.
46 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32/testsuites/tools'
47 make[1]: Leaving directory '/home/chris/development/rtems/kernel/erc32/testsuites/tools'
48 Making all in sparc-rtems5/c
49 make[1]: Entering directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/c'
50 Making all in .
51 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/c'
52 Configuring RTEMS_BSP=erc32
53 checking for gmake... no
54 checking for make... make
55 checking build system type... x86_64-pc-linux-gnu
56 checking host system type... sparc-unknown-rtems5
57 .....
58 sparc-rtems5-gcc -B../../../../../../erc32/lib/ -specs bsp_specs -qrtems -DHAVE_CONFIG_H -I. -
↳ -I/home/chris/development/rtems/kernel/rtems/c/src/../../../../testsuites/samples/nsecs -I. -
↳ -I/home/chris/development/rtems/kernel/rtems/c/src/../../../../testsuites/samples/./support/
↳ include -mcpu=cypress -O2 -g -ffunction-sections -fdata-sections -Wall -Wmissing-
↳ prototypes -Wimplicit-function-declaration -Wstrict-prototypes -Wnested-externs -MT.
↳ init.o -MD -MP -MF .deps/init.Tpo -c -o init.o /home/chris/development/rtems/kernel/
↳ rtems/c/src/../../../../testsuites/samples/nsecs/init.c
59 sparc-rtems5-gcc -B../../../../../../erc32/lib/ -specs bsp_specs -qrtems -DHAVE_CONFIG_H -I. -
↳ -I/home/chris/development/rtems/kernel/rtems/c/src/../../../../testsuites/samples/nsecs -I. -
↳ -I/home/chris/development/rtems/kernel/rtems/c/src/../../../../testsuites/samples/./support/
↳ include -mcpu=cypress -O2 -g -ffunction-sections -fdata-sections -Wall -Wmissing-
↳ prototypes -Wimplicit-function-declaration -Wstrict-prototypes -Wnested-externs -MT.
↳ empty.o -MD -MP -MF .deps/empty.Tpo -c -o empty.o /home/chris/development/rtems/kernel/
↳ rtems/c/src/../../../../testsuites/samples/nsecs/empty.c

```

(continues on next page)

(continued from previous page)

```

60 mv -f .deps/empty.Tpo .deps/empty.Po
61 mv -f .deps/init.Tpo .deps/init.Po
62 sparc-rtems5-gcc -B../../../../../../erc32/lib/ -specs bsp_specs -qrtems -mcpu=cypress -O2 -
↳ g -ffunction-sections -fdata-sections -Wall -Wmissing-prototypes -Wimplicit-function-
↳ declaration -Wstrict-prototypes -Wnested-externs -Wl,--gc-sections -mcpu=cypress -o_
↳ nsecs.exe init.o empty.o
63 sparc-rtems5-nm -g -n nsecs.exe > nsecs.num
64 sparc-rtems5-size nsecs.exe
65      text    data    bss    dec    hex filename
66 121392    1888    6624 129904  1fb70 nsecs.exe
67 cp nsecs.exe nsecs.ralf
68 make[6]: Leaving directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/ c/
↳ erc32/testsuites/samples/nsecs'
69 make[5]: Leaving directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/ c/
↳ erc32/testsuites/samples'
70 make[4]: Leaving directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/ c/
↳ erc32/testsuites/samples'
71 make[4]: Entering directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/ c/
↳ erc32/testsuites'
72 make[4]: Nothing to be done for 'all-am'.
73 make[4]: Leaving directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/ c/
↳ erc32/testsuites'
74 make[3]: Leaving directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/ c/
↳ erc32/testsuites'
75 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/ c/
↳ erc32'
76 make[1]: Leaving directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/c'
77 make[1]: Entering directory '/home/chris/development/rtems/kernel/erc32'
78 make[1]: Nothing to be done for 'all-am'.
79 make[1]: Leaving directory '/home/chris/development/rtems/kernel/erc32'

```

5.3.5 Installing A BSP

All that remains to be done is to install the kernel. Installing RTEMS copies the API headers and architecture specific libraries to a location under the *prefix* you provide. You can install any number of BSPs under the same *prefix*. We recommend you have a separate *prefix* for different versions of RTEMS. Do not mix versions of RTEMS under the same *prefix*. Make installs RTEMS with the following command:

```

1 $ make install
2 Making install in tools/build
3 make[1]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/build'
4 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/build'
5 /bin/mkdir -p '/home/chris/development/rtems/5/bin'
6 /usr/bin/install -c cklength eolstrip packhex unhex rtems-bin2c '/home/chris/development/
↳ rtems/5/bin'
7 /bin/mkdir -p '/home/chris/development/rtems/5/bin'
8 /usr/bin/install -c install-if-change '/home/chris/development/rtems/5/bin'
9 make[2]: Nothing to be done for 'install-data-am'.
10 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/build'
11 make[1]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/build'
12 Making install in tools/cpu
13 make[1]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
14 Making install in generic

```

(continues on next page)

(continued from previous page)

```

15 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/cpu/generic'
16 make[3]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/cpu/generic'
17 make[3]: Nothing to be done for 'install-exec-am'.
18 make[3]: Nothing to be done for 'install-data-am'.
19 make[3]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/cpu/generic'
20 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/cpu/generic'
21 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
22 make[3]: Entering directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
23 make[3]: Nothing to be done for 'install-exec-am'.
24 make[3]: Nothing to be done for 'install-data-am'.
25 make[3]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
26 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
27 make[1]: Leaving directory '/home/chris/development/rtems/kernel/erc32/tools/cpu'
28 ....
29 make[1]: Leaving directory '/home/chris/development/rtems/kernel/erc32/sparc-rtems5/c'
30 make[1]: Entering directory '/home/chris/development/rtems/kernel/erc32'
31 make[2]: Entering directory '/home/chris/development/rtems/kernel/erc32'
32 make[2]: Nothing to be done for 'install-exec-am'.
33 /bin/mkdir -p '/home/chris/development/rtems/5/make'
34 /usr/bin/install -c -m 644 /home/chris/development/rtems/kernel/rtems/make/main.cfg /home/
  ↳ chris/development/rtems/kernel/rtems/make/leaf.cfg '/home/chris/development/rtems/5/make
  ↳ '
35 /bin/mkdir -p '/home/chris/development/rtems/5/share/rtems5/make/Templates'
36 /usr/bin/install -c -m 644 /home/chris/development/rtems/kernel/rtems/make/Templates/
  ↳ Makefile.dir /home/chris/development/rtems/kernel/rtems/make/Templates/Makefile.leaf /
  ↳ /home/chris/development/rtems/kernel/rtems/make/Templates/Makefile.lib '/home/chris/
  ↳ development/rtems/5/share/rtems5/make/Templates'
37 /bin/mkdir -p '/home/chris/development/rtems/5/make/custom'
38 /usr/bin/install -c -m 644 /home/chris/development/rtems/kernel/rtems/make/custom/default.
  ↳ cfg '/home/chris/development/rtems/5/make/custom'
39 make[2]: Leaving directory '/home/chris/development/rtems/kernel/erc32'
40 make[1]: Leaving directory '/home/chris/development/rtems/kernel/erc32'

```

5.3.6 Contributing Patches

RTEMS welcomes fixes to bugs and new features. The RTEMS Project likes to have bugs fixed against a ticket created on our [Developer Site](#). Please raise a ticket if you have a bug. Any changes that are made can be tracked against the ticket. If you want to add a new a feature please post a message to [Developers Mailing List](#) describing what you would like to implement. The RTEMS maintainer will help decide if the feature is in the best interest of the project. Not everything is and the maintainers need to evaluate how much effort it is to maintain the feature. Once accepted into the source tree it becomes the responsibility of the maintainers to keep the feature updated and working.

Changes to the source tree are tracked using git. If you have not made changes and enter the source tree and enter a git status command you will see nothing has changed:

```

1 $ cd ../rtems
2 $ git status
3 On branch master
4 Your branch is up-to-date with 'origin/master'.
5 nothing to commit, working directory clean

```

We will make a change to the source code. In this example I change the help message to the

RTEMS shell's halt command. Running the same git status command reports:

```

1 $ git status
2 On branch master
3 Your branch is up-to-date with 'origin/master'.
4 Changes not staged for commit:
5   (use "git add <file>..." to update what will be committed)
6   (use "git checkout -- <file>..." to discard changes in working directory)
7
8       modified:   cpukit/libmisc/shell/main_halt.c
9
10 no changes added to commit (use "git add" and/or "git commit -a")

```

As an example I have a ticket open and the ticket number is 9876. I commit the change with the follow git command:

```

1 $ git commit cpukit/libmisc/shell/main_halt.c

```

An editor is opened and I enter my commit message. The first line is a title and the following lines form a body. My message is:

```

1 shell: Add more help detail to the halt command.
2
3 Closes #9876.
4
5 # Please enter the commit message for your changes. Lines starting
6 # with '#' will be ignored, and an empty message aborts the commit.
7 # Explicit paths specified without -i or -o; assuming --only paths...
8 #
9 # Committer: Chris Johns <chrisj@rtems.org>
10 #
11 # On branch master
12 # Your branch is up-to-date with 'origin/master'.
13 #
14 # Changes to be committed:
15 #       modified:   cpukit/libmisc/shell/main_halt.c

```

When you save and exit the editor git will report the commit's status:

```

1 $ git commit cpukit/libmisc/shell/main_halt.c
2 [master 9f44dc9] shell: Add more help detail to the halt command.
3 1 file changed, 1 insertion(+), 1 deletion(-)

```

You can either email the patch to [Developers Mailing List](#) with the following git command, and it is *minus one* on the command line:

```

1 $ git send-email --to=devel@rtems.org -1
2 <add output here>

```

Or you can ask git to create a patch file using:

```

1 $ git format-patch -1
2 0001-shell-Add-more-help-detail-to-the-halt-command.patch

```

This patch can be attached to a ticket.

5.4 Project Sandboxing

Project specific sandboxes let you have a number of projects running in parallel with each project in its own sandbox. You simply have a *prefix* per project and under that prefix you create a simple yet repeatable structure.

As an example lets say I have a large disk mounted under `/bd` for *Big Disk*. As root create a directory called `projects` and give the directory suitable permissions to be writable by you as a user.

Lets create a project sandbox for my *Box Sorter* project. First create a project directory called `/bd/projects/box-sorter`. Under this create `rtems` and under that create `rtems-4.11.0`. Under this path you can follow the *Releases* (page 54) procedure to build a tool set using the prefix of `/bd/projects/box-sorter/rtems/4.11.0`. You are free to create your project specific directories under `/bd/projects/box-sorter`. The top level directories would be:

/bd/projects

Project specific development trees.

/bd/projects/box-sorter

Box Sorter project sandbox.

/bd/projects/box-sorter/rtems/4.11.0

Project prefix for RTEMS 4.11.0 compiler, debuggers, tools and installed Board Support Package (BSP).

A variation is to use the `--without-rtems` option with the RSB to not build the BSPs when building the tools and to build RTEMS specifically for each project. This lets you have a production tools installed at a top level on your disk and each project can have a specific and possibly customised version of RTEMS. The top level directories would be:

/bd/rtems

The top path to production tools.

/bd/rtems/4.11.0

Production prefix for RTEMS 4.11.0 compiler, debuggers and tools.

/bd/projects

Project specific development trees.

/bd/projects/box-sorter

Box Sorter project sandbox.

/bd/projects/box-sorter/rtems

Box Sorter project's custom RTEMS kernel source and installed BSP.

A further variation if there is an RTEMS kernel you want to share between projects is it to move this to a top level and share. In this case you will end up with:

/bd/rtems

The top path to production tools and kernels.

/bd/rtems/4.11.0

Production prefix for RTEMS 4.11.0.

/bd/rtems/4.11.0/tools

Production prefix for RTEMS 4.11.0 compiler, debuggers and tools.

/bd/rtems/4.11.0/bsps

Production prefix for RTEMS 4.11.0 Board Support Packages (BSPs).

/bd/projects

Project specific development trees.

/bd/projects/box-sorter

Box Sorter project sandbox.

Finally you can have a single set of *production* tools and RTEMS BSPs on the disk under /bd/rtems you can share between your projects. The top level directories would be:

/bd/rtems

The top path to production tools and kernels.

/bd/rtems/4.11.0

Production prefix for RTEMS 4.11.0 compiler, debuggers, tools and Board Support Packages (BSPs).

/bd/projects

Project specific development trees.

/bd/projects/box-sorter

Box Sorter project sandbox.

The project sandoxing approach allows you move a specific production part into the project's sandbox to allow you to customise it. This is useful if you are testing new releases. The typical dependency is the order listed above. You can test new RTEMS kernels with production tools but new tools will require you build the kernel with them. Release notes with each release will let know what you need to update.

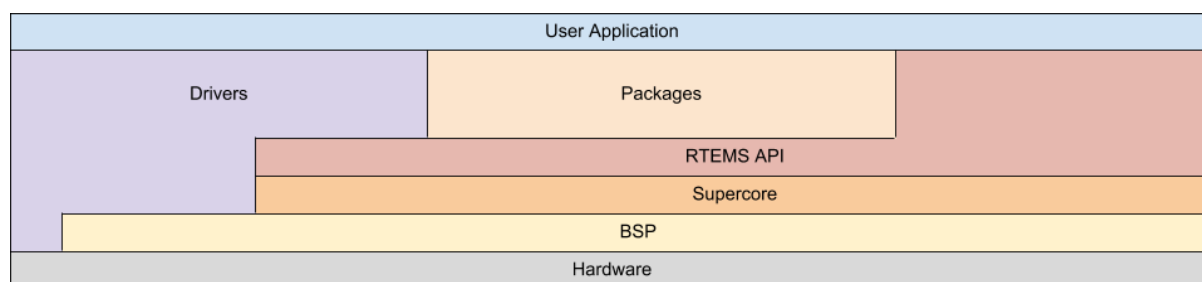
If the machine is a central project development machine simply replace projects with users and give each user a personal directory.

TARGET HARDWARE

6.1 Targets

Target hardware that can run RTEMS is often referred to simply as the *target* because RTEMS is specifically aimed at that target hardware. An RTEMS executable is statically linked and executes in a single address space on the target hardware. A statically linked executable means the RTEMS Kernel, drivers, third-party packages and application code is linked into a single executable image. A single address space means no virtual memory and no memory protected process address space is running within the RTEMS arena and the RTEMS executive, drivers and application have unprotected access to the whole address space and all hardware.

Target hardware supported by RTEMS has a *Board Support Package* (page 91) or BSP. A BSP is a specific instance of an RTEMS architecture that allows the creation of an RTEMS executable. You can view the layering as:



RTEMS targets are grouped by architectures and within an architecture there are a number of Board Support Packages or BSPs. An architecture is a specific class or family of processors and can be large such as ARM or specific such as the NIOS-II or Microblaze.

RTEMS is designed to be ported to new target hardware easily and efficiently.

6.2 Architectures

An RTEMS architecture is a class or family of a processor architecture that RTEMS supports. The RTEMS architecture model follows the architecture model of GCC. An architecture in GCC results in a specific RTEMS GCC compiler. This compiler may support a range of processors in the family that may have differences in instructions sets, floating point support or other aspects. RTEMS configures GCC to create separate runtime libraries for each supported instruction set, floating point unit, vector unit, word size (e.g. 32-bit and 64-bit), endianness, code model, *ABI*, processor errata workarounds, and so on in the architecture. This is termed *multilib*. Multilibs are chosen automatically by GCC via selecting a specific set of machine options.

You can query the multilibs of a specific RTEMS GCC compiler via the `-print-multi-lib` option:

```

1 $ sparc-rtems5-gcc -print-multi-lib
2 .;
3 soft;@msoft-float
4 v8;@mcpu=v8
5 leon3;@mcpu=leon3
6 leon3v7;@mcpu=leon3v7
7 leon;@mcpu=leon
8 leon3/gr712rc;@mcpu=leon3@mfix-gr712rc
9 leon3v7/gr712rc;@mcpu=leon3v7@mfix-gr712rc
10 leon/ut699;@mcpu=leon@mfix-ut699
11 leon/at697f;@mcpu=leon@mfix-at697f
12 soft/v8;@msoft-float@mcpu=v8
13 soft/leon3;@msoft-float@mcpu=leon3
14 soft/leon3v7;@msoft-float@mcpu=leon3v7
15 soft/leon;@msoft-float@mcpu=leon
16 soft/leon3/gr712rc;@msoft-float@mcpu=leon3@mfix-gr712rc
17 soft/leon3v7/gr712rc;@msoft-float@mcpu=leon3v7@mfix-gr712rc
18 soft/leon/ut699;@msoft-float@mcpu=leon@mfix-ut699
19 soft/leon/at697f;@msoft-float@mcpu=leon@mfix-at697f

```

Each printed line represents a multilib. The `.` corresponds to the default multilib. It is used if a set of machine options does not match to a specialized multilib. The string before the `;` describes the directory in the GCC installation used for the particular multilib. After the `;` the set of machine options for this multilib follows separated by `@` characters.

You can figure out the multilib selected by GCC for a set of machine options with the `-print-multi-directory` option:

```

1 $ sparc-rtems5-gcc -print-multi-directory -mcpu=leon3
2 leon3

```

It is crucial that the RTEMS BSP, support libraries and the application code are compiled consistently with a compatible set of machine options. Otherwise, in the best case errors during linking will occur or you may end up silently with undefined behaviour which results in sporadic run-time crashes. A wrong set of machine options may result in a running application, however, with degraded performance, e.g. hardware floating point unit is not used by the mathematical library.

For a list of architectures supported by RTEMS please have a look at the sections of the *Board Support Packages* (page 91) chapter.

RTEMS executables (page 129) are statically linked for a specific target therefore a precise and exact match can be made for the hardware that extracts the best possible performance. The

compiler supports the variants to the instruction set and RTEMS extends the specialization to specific processors in an architecture. This specialization gives RTEMS a finer resolution of features and capabilities a specific device may offer allowing the kernel, drivers and application to make the most of those resources. The trade off is portability however this is not important because the executable are statically linked for a single target.

Note: RTEMS support dynamically load code through the dlopen interface. Loading code via this interface results in an executable image that is equivalent to statically linked executable of the same code. Dynamic loading is a system level tool for system architects.

6.3 Tiers

RTEMS has a tiered structure for architecture and BSPs. It provides:

1. A way to determine the state of a BSP in RTEMS.
2. A quaility measure for changes entering the RTEMS source code.

The RTEMS project supports RTEMS Architecture Tiers. Each architecture resided in one of the numbered tiers. The tiers are number 1 to 4 where Tier 1 is the highest tier and Tier 4 is the lowest. Architectures move between tiers based on the level of support and the level of testing that is performed. An architecture requires continual testing and reporting of test results to maintain a tier level. The RTEMS Project's continuous integration testing program' continually monitors and reports the test results.

The RTEMS Architecture Tier system provides a defined way to determine the state of an architecture in RTEMS. Architectures age and support for them drops off and the RTEMS Project needs a way to determine if an architecture should stay and be supported or depreciated and removed. The tier system also provides users with a clear understanding of the state of an architecture in RTEMS, often useful when deciding on a processor for a new project. It can also let a user know the RTEMS Project needs support to maintain a specific architecture. Access to hardware to perform testing is a large and complex undertaking and the RTEMS Project is always looking for user support and help. If you can help please contact someone and let us know.

The tier structure in RTEMS is support by the Buildbot continuous integration server. Changes to RTEMS are automatically built and tested and the results indicate if a BSP currently meets its tier status. As the RTEMS Project does not own hardware for every BSP, it is critical that users provide test results on hardware of interest.

The rules for Tiers are:

1. A BSP can only be in one of the following tiers:

Tier	Description
1	<ul style="list-style-type: none"> • The RTEMS Kernel must build without error. • Tests are run on target hardware.
2	<ul style="list-style-type: none"> • The RTEMS Kernel must build without error. • Tests can be run on simulation.
3	<ul style="list-style-type: none"> • The RTEMS Kernel must build without error. • There are no test results.
4	<ul style="list-style-type: none"> • The RTEMS Kernel does not build.
5	<ul style="list-style-type: none"> • The BSP is to be removed after the next release.

2. An architecture's tier is set by the highest BSP tier reached.
3. The tier level for a BSP is set by the RTEMS Project team. Movement of BSP between tier level requires agreement. The Buildbot results indicate the minimum current tier level.
4. Changes to RTEMS may result in a BSP not meeting its tier are acceptable if the change is accompanied by an announcement and a plan on how this is to be resolved. Temporary drops in tier are expected and should be brief.
5. Test results are set on a per BSP basis by the RTEMS Project team. Changes to the test result values requires agreement. The test results are defined as:
 1. Passes
 2. Expected Failures

Expected failures must be explicitly listed. A BSP is required to have a valid test result entry on target hardware to reach tier 1.

BOARD SUPPORT PACKAGES

A Board Support Package or BSP is the software that glues a specific target or board or piece of hardware to RTEMS so its services are available to applications.

RTEMS contains a large number of BSPs for commonly available simulators and target hardware.

You can see the current BSP list in the RTEMS sources by asking RTEMS with:

```
1 $ ./rtems-bsps
```

7.1 aarch64 (AArch64)

There are no AArch64 BSPs yet.

7.2 arm (ARM)

7.2.1 altera-cyclone-v (Intel Cyclone V)

This BSP offers only one variant, the *altcycv_devkit*. This variant supports the Intel Cyclone V system on chip. The basic hardware initialization is not performed by the BSP. A boot loader with device tree support must be used to start the BSP, e.g. U-Boot.

The BSP is known to run on these boards:

- [Cyclone V SoC Development Kit](#)
- [Enclustra Mars MA3 SoC Module](#)
- [Terasic DE10-Standard Development Kit](#)

7.2.1.1 Boot via U-Boot

The application executable file (ELF file) must be converted to an U-Boot image. Use the following commands:

```
1 arm-rtems5-objcopy -O binary app.exe app.bin
2 gzip -9 -f -c app.bin > app.bin.gz
3 mkimage -A arm -O linux -T kernel -a 0x00300000 -e 0x00300000 -n RTEMS -d app.bin.gz app.
  ↪img
```

Use the following U-Boot commands to boot an application via TFTP download:

```
1 tftpboot ${loadaddr} app.img && run loadfdt && bootm ${loadaddr} - ${fdt_addr} ; reset
```

The `loadfdt` command may be not defined in your U-Boot environment. Just replace it with the appropriate commands to load the device tree at `${fdt_addr}`.

7.2.1.2 Clock Driver

The clock driver uses the *Cortex-A9 MPCore Global Timer*.

7.2.1.3 Console Driver

The console driver supports up to two on-chip NS16550 UARTs. The console driver does not configure the pins.

7.2.1.4 I2C Driver

There is a legacy I2C driver. It should be converted to the I2C driver framework.

7.2.1.5 Network Interface Driver

The network interface driver is provided by the *libbsd*. It is initialized according to the device tree. It supports checksum offload.

7.2.1.6 MMC/SDCard Driver

The MMC/SDCard driver is provided by the *libbsd*. It is initialized according to the device tree. Pin re-configuration according to the serial clock frequency is not supported. DMA transfers are supported.

7.2.1.7 USB Host Driver

The USB host driver is provided by the *libbsd*. It is initialized according to the device tree. The driver works in polled mode.

7.2.1.8 Caveats

The clock and pin configuration support is quite rudimentary and mostly relies on the boot loader.

7.2.2 atsam

TODO.

7.2.3 beagle

This BSP supports four variants, *beagleboardorig*, *beagleboardxm*, *beaglebonewhite* and *beagleboneblack*. The basic hardware initialization is not performed by the BSP. A boot loader with device tree support must be used to start the BSP, e.g., U-Boot.

TODO(These drivers are present but not documented yet):

- Clock driver.
- Network Interface Driver.
- SDcard driver.
- GPIO Driver.
- Console driver.
- PWM Driver.
- RTC driver.

7.2.3.1 Boot via U-Boot

To boot via uboot, the ELF must be converted to a U-Boot image like below:

```
1 arm-rtems5-objcopy hello.exe -O binary app.bin
2 gzip -9 app.bin
3 mkimage -A arm -O linux -T kernel -a 0x80000000 -e 0x80000000 -n RTEMS -d app.bin.gz
  ↪ rtems-app.img
```

7.2.3.2 Getting the Device Tree Blob

The Device Tree Blob (DTB) is needed to load the device tree while starting up the kernel. We build the dtb from the FreeBSD source matching the commit hash from the libbsd HEAD of freebsd-org. For example if the HEAD is at “19a6ceb89dbacf74697d493e48c388767126d418” Then the right Device Tree Source (DTS) file is: <https://github.com/freebsd/freebsd/blob/19a6ceb89dbacf74697d493e48c388767126d418/sys/gnu/dts/arm/am335x-boneblack.dts>

Please refer to the *Device Tree* (page 154) to know more about building and applying the Device Trees.

7.2.3.3 Writing the uEnv.txt file

The uEnv.txt file is needed to set any environment variable before the kernel is loaded. Each line is a u-boot command that the uboot will execute during start up.

Add the following to a file named uEnv.txt:

```
1 setenv bootdelay 5
2 uenvcmd=run boot
3 boot=fatload mmc 0 0x80800000 rtems-app.img ; fatload mmc 0 0x88000000 am335x-boneblack.
  ↪ dtb ; bootm 0x80800000 - 0x88000000
```

7.2.3.4 I2C Driver

The Beagle has the *i2c-0* device registered at initialization. For registering *i2c-1* and *i2c-2* bbb_register_i2c_1() and bbb_register_i2c_2() wrapper functions are respectively used.

For registering an I2C device with a custom path (say */dev/i2c-3*) the function am335x_i2c_bus_register() has to be used.

The function prototype is given below:

```
1 int am335x_i2c_bus_register(
2   const char      *bus_path,
3   uintptr_t       register_base,
4   uint32_t        input_clock,
5   rtems_vector_number irq
6 );
```

7.2.3.5 SPI Driver

The SPI device */dev/spi-0* can be registered with bbb_register_spi_0()

For registering with a custom path, the bsp_register_spi() can be used.

The function prototype is given below:

```
1 rtems_status_code bsp_register_spi(
2   const char      *bus_path,
3   uintptr_t       register_base,
4   rtems_vector_number irq
5 );
```

7.2.4 csb336

TODO.

7.2.5 edb7312

TODO.

7.2.6 gdbarmsim

TODO.

7.2.7 gumstix

TODO.

7.2.8 imx (NXP i.MX)

This BSP offers only one variant, the *imx7*. This variant supports the i.MX 7Dual processor. The basic hardware initialization is not performed by the BSP. A boot loader with device tree support must be used to start the BSP, e.g. U-Boot.

7.2.8.1 Build Configuration Options

The following options are available at the configure command line.

BSP_PRESS_KEY_FOR_RESET

If defined to a non-zero value, then print a message and wait until pressed before resetting board when application terminates.

BSP_RESET_BOARD_AT_EXIT

If defined to a non-zero value, then reset the board when the application terminates.

BSP_PRINT_EXCEPTION_CONTEXT

If defined to a non-zero value, then print the exception context when an unexpected exception occurs.

BSP_FDT_BLOB_SIZE_MAX

The maximum size of the device tree blob in bytes (default is 262144).

CONSOLE_USE_INTERRUPTS

Use interrupt driven mode for console devices (enabled by default).

IMX_CCM_IPG_HZ

The IPG clock frequency in Hz (default is 67500000).

IMX_CCM_UART_HZ

The UART clock frequency in Hz (default is 24000000).

IMX_CCM_AHB_HZ

The AHB clock frequency in Hz (default is 135000000).

7.2.8.2 Boot via U-Boot

The application executable file (ELF file) must be converted to an U-Boot image. Use the following commands:

```
1 arm-rtems5-objcopy -O binary app.exe app.bin
2 gzip -9 -f -c app.bin > app.bin.gz
3 mkimage -A arm -O linux -T kernel -a 0x80200000 -e 0x80200000 -n RTEMS -d app.bin.gz app.
  ↪img
```

Use the following U-Boot commands to boot an application via TFTP download:

```
1 tftpboot ${loadaddr} app.img && run loadfdt && bootm ${loadaddr} - ${fdt_addr} ; reset
```

The loadfdt command may be not defined in your U-Boot environment. Just replace it with the appropriate commands to load the device tree at \${fdt_addr}.

7.2.8.3 Clock Driver

The clock driver uses the *ARMv7-AR Generic Timer*.

7.2.8.4 Console Driver

The console driver supports up to seven on-chip UARTs. They are initialized according to the device tree. The console driver does not configure the pins.

7.2.8.5 I2C Driver

I2C drivers are registered by the `i2c_bus_register_imx()` function. The I2C driver does not configure the pins.

```
1 #include <assert.h>
2 #include <bsp.h>
3
4 void i2c_init(void)
5 {
6     int rv;
7
8     rv = i2c_bus_register_imx("/dev/i2c-0", "i2c0");
9     assert(rv == 0);
10 }
```

7.2.8.6 SPI Driver

SPI drivers are registered by the `spi_bus_register_imx()` function. The SPI driver configures the pins according to the `pinctrl-0` device tree property. SPI transfers with a continuous chip select are limited by the FIFO size of 64 bytes. The driver has no DMA support.

```
1 #include <assert.h>
2 #include <bsp.h>
3
4 void spi_init(void)
5 {
6     int rv;
7
8     rv = spi_bus_register_imx("/dev/spi-0", "spi0");
9     assert(rv == 0);
10 }
```

7.2.8.7 Network Interface Driver

The network interface driver is provided by the *libbsd*. It is initialized according to the device tree. It supports checksum offload and interrupt coalescing. IPv6 transmit checksum offload is not implemented. The interrupt coalescing uses the MII/GMII clocks and can be controlled by the following system controls:

- `dev.ffec.<unit>.int_coal.rx_time`
- `dev.ffec.<unit>.int_coal.rx_count`
- `dev.ffec.<unit>.int_coal.tx_time`
- `dev.ffec.<unit>.int_coal.tx_count`

A value of zero for the time or count disables the interrupt coalescing in the corresponding direction.

7.2.8.8 MMC/SDCard Driver

The MMC/SDCard driver (uSDHC module) is provided by the *libbsd*. It is initialized according to the device tree. Pin re-configuration according to the serial clock frequency is not supported. Data transfers are extremely slow. This is probably due to the missing DMA support.

7.2.8.9 Caveats

The clock and pin configuration support is quite rudimentary and mostly relies on the boot loader. For a pin group configuration see `imx_iomux_configure_pins()`. There is no power management support.

7.2.9 lm3s69xx

TODO.

7.2.10 lpc176x

TODO.

7.2.11 imx (NXP i.MX)

This BSP offers only one variant, the *imx7*. This variant supports the i.MX 7Dual processor. The basic hardware initialization is not performed by the BSP. A boot loader with device tree support must be used to start the BSP, e.g. U-Boot.

7.2.11.1 Build Configuration Options

The following options are available at the configure command line.

BSP_PRESS_KEY_FOR_RESET

If defined to a non-zero value, then print a message and wait until pressed before resetting board when application terminates.

BSP_RESET_BOARD_AT_EXIT

If defined to a non-zero value, then reset the board when the application terminates.

BSP_PRINT_EXCEPTION_CONTEXT

If defined to a non-zero value, then print the exception context when an unexpected exception occurs.

BSP_FDT_BLOB_SIZE_MAX

The maximum size of the device tree blob in bytes (default is 262144).

CONSOLE_USE_INTERRUPTS

Use interrupt driven mode for console devices (enabled by default).

IMX_CCM_IPG_HZ

The IPG clock frequency in Hz (default is 67500000).

IMX_CCM_UART_HZ

The UART clock frequency in Hz (default is 24000000).

IMX_CCM_AHB_HZ

The AHB clock frequency in Hz (default is 135000000).

7.2.11.2 Boot via U-Boot

The application executable file (ELF file) must be converted to an U-Boot image. Use the following commands:

```
1 arm-rtems5-objcopy -O binary app.exe app.bin
2 gzip -9 -f -c app.bin > app.bin.gz
3 mkimage -A arm -O linux -T kernel -a 0x80200000 -e 0x80200000 -n RTEMS -d app.bin.gz app.
  ↪img
```

Use the following U-Boot commands to boot an application via TFTP download:

```
1 tftpboot ${loadaddr} app.img && run loadfdt && bootm ${loadaddr} - ${fdt_addr} ; reset
```

The loadfdt command may be not defined in your U-Boot environment. Just replace it with the appropriate commands to load the device tree at `${fdt_addr}`.

7.2.11.3 Clock Driver

The clock driver uses the *ARMv7-AR Generic Timer*.

7.2.11.4 Console Driver

The console driver supports up to seven on-chip UARTs. They are initialized according to the device tree. The console driver does not configure the pins.

7.2.11.5 I2C Driver

I2C drivers are registered by the `i2c_bus_register_imx()` function. The I2C driver does not configure the pins.

```
1 #include <assert.h>
2 #include <bsp.h>
3
4 void i2c_init(void)
5 {
6     int rv;
7
8     rv = i2c_bus_register_imx("/dev/i2c-0", "i2c0");
9     assert(rv == 0);
10 }
```

7.2.11.6 SPI Driver

SPI drivers are registered by the `spi_bus_register_imx()` function. The SPI driver configures the pins according to the `pinctrl-0` device tree property. SPI transfers with a continuous chip select are limited by the FIFO size of 64 bytes. The driver has no DMA support.

```
1 #include <assert.h>
2 #include <bsp.h>
3
4 void spi_init(void)
5 {
6     int rv;
7
8     rv = spi_bus_register_imx("/dev/spi-0", "spi0");
9     assert(rv == 0);
10 }
```

7.2.11.7 Network Interface Driver

The network interface driver is provided by the *libbsd*. It is initialized according to the device tree. It supports checksum offload and interrupt coalescing. IPv6 transmit checksum offload is not implemented. The interrupt coalescing uses the MII/GMII clocks and can be controlled by the following system controls:

- `dev.ffec.<unit>.int_coal.rx_time`
- `dev.ffec.<unit>.int_coal.rx_count`

- `dev.ffec.<unit>.int_coal.tx_time`
- `dev.ffec.<unit>.int_coal.tx_count`

A value of zero for the time or count disables the interrupt coalescing in the corresponding direction.

7.2.11.8 MMC/SDCard Driver

The MMC/SDCard driver (uSDHC module) is provided by the *libbsd*. It is initialized according to the device tree. Pin re-configuration according to the serial clock frequency is not supported. Data transfers are extremely slow. This is probably due to the missing DMA support.

7.2.11.9 Caveats

The clock and pin configuration support is quite rudimentary and mostly relies on the boot loader. For a pin group configuration see `imx_iomux_configure_pins()`. There is no power management support.

7.2.12 raspberrypi

This BSP supports *Raspberry Pi 1* and *Raspberry Pi 2* currently. The support for *Raspberry Pi 3* is work under progress. The default bootloader on the Raspberry Pi which is used to boot Raspbian or other OS can be also used to boot RTEMS. U-boot can also be used.

7.2.12.1 Setup SD card

The Raspberry Pis have an unconventional booting mechanism. The GPU boots first, initializes itself, runs the bootloader and starts the CPU. The bootloader looks for a kernel image, by default the kernel images must have a name of the form `kernel*.img` but this can be changed by adding `kernel=<img_name>` to `config.txt`.

You must provide the required files for the GPU to proceed. These files can be downloaded from [the Raspberry Pi Firmware Repository](#). You can remove the `kernel*.img` files if you want too, but don't touch the other files.

Copy these files in to a SD card with FAT filesystem.

7.2.12.2 Kernel image

The following steps show how to run `hello.exe` on a Raspberry Pi 2. The same instructions can be applied to Raspberry Pi 1 also. Other executables can be processed in a similar way.

To create the kernel image:

```
1 $ arm-rtems5-objcopy -Obinary hello.exe kernel.img
```

Copy the kernel image to the SD card.

Make sure you have these lines below, in your `config.txt`.

```
1 enable-uart=1
2 kernel_address=0x200000
3 kernel=kernel.img
```

7.2.12.3 Testing using QEMU

QEMU can be built using RSB. Navigate to <SOURCE_BUILDER_DIR>/rtems and run this command.

```
1 $ ../source-builder/sb-set-builder --prefix=<TOOLCHAIN_DIR> devel/qemu4.bset
```

Note: Replace <SOURCE_BUILDER_DIR> and <TOOLCHAIN_DIR> with the correct path of the directories. For example, if you used quick-start section as your reference, these two will be \$HOME/quick-start/src/rsb and \$HOME/quick-start/rtems/5 respectively,

QEMU along with GDB can be used for debugging, but it only supports Raspberry Pi 2 and the emulation is also incomplete. So some of the features might not work as expected.

Make sure your version of QEMU is newer than v2.6, because older ones don't support Raspberry Pis.

```
1 $ qemu-system-arm -M raspi2 -m 1G -kernel hello.exe -serial mon:stdio -nographic -S -s
```

This starts QEMU and creates a socket at port localhost:1234 for GDB to connect.

The Device Tree Blob (DTB) is needed to load the device tree while starting up the kernel. The BSP uses information from this file to initialize the drivers.

Make sure you pass in the correct DTB file. There are currently two version of DTB for the Raspberry Pi 2 bcm2709-rpi-2-b.dtb and bcm2710-rpi-2-b.dtb. The bcm2709-rpi-2-b.dtb is for Raspberry Pi 2 Model B and bcm2710-rpi-2-b.dtb is for Raspberry Pi 2 Model B v1.2

We need to pass in the DTB file to GDB before running the example.

In a new terminal, run GDB using

```
1 $ arm-rtems5-gdb hello.exe
```

This will open GDB and will load the symbol table from hello.exe. Issue the following commands in the GDB prompt.

```
1 (gdb) tar remote:1234
2 (gdb) load
3 (gdb) restore bcm2709-rpi-2-b.dtb binary 0x2ef00000
4 (gdb) set $r2 = 0x2ef00000
```

This will connect GDB to QEMU and will load the DTB file and the application.

```
1 (gdb) continue
```

The continue command will run the executable.

Note: Add set scheduler-locking on in GDB if you have any issues running the examples.

7.2.13 realview-pbx-a9

TODO.

7.2.14 rtl22xx

TODO.

7.2.15 smdk2410

TODO.

7.2.16 tms570

TODO.

7.2.17 xen (Xen on ARM)

This BSP enables RTEMS to run as a guest virtual machine in AArch32 mode on the Xen hypervisor for ARMv8 platforms.

Drivers:

- Clock: ARMv7-AR Generic Timer
- Console: Virtual PL011 device
- Interrupt: GICv2

BSP variants:

- `xen_virtual`: completely virtualized guest with no dependence on underlying hardware

The `xen_virtual` BSP variant relies on standard Xen features, so it should be able to run on any ARMv8 platform.

Xen allows for the passthrough of hardware peripherals to guest virtual machines. BSPs could be added in the future targeting specific hardware platforms and include the appropriate drivers.

This BSP was tested with Xen running on the Xilinx Zynq UltraScale+ MPSoC using the Virtuosity distribution maintained by DornerWorks.

7.2.17.1 Execution

This procedure describes how to run the ticker sample application that should already be built with the BSP.

The `ticker.exe` file can be found in the BSP build tree at:

```
1 arm-rtems5/c/xen_virtual/testsuites/samples/ticker.exe
```

The `ticker.exe` elf file must be translated to a binary format.

```
1 arm-rtems5-objcopy -O binary ticker.exe ticker.bin
```

Then place the `ticker.bin` file on the `dom0` filesystem.

From the `dom0` console, create a configuration file `ticker.cfg` with the following contents.

```
1 name = "ticker"1G
2 kernel = "ticker.bin"
3 memory = 8
4 vcpus = 1
5 gic_version = "v2"
6 uart = "sbsa_uart"
```

Create the virtual machine and attach to the virtual `vpl011` console.

```
1 xl create ticker.cfg && xl console -t uart ticker
```

To return back to the `dom0` console, press both `Ctrl` and `]` on your keyboard.

7.2.17.2 Additional Information

- [Virtuosity distribution](#)

7.2.18 xilinx-zynq

TODO.

7.2.19 xilinx-zynqmp

This BSP supports the Xilinx Zynq UltraScale+ MPSoC platform.

7.3 bfin (Blackfin)

7.3.1 bf537Stamp

TODO.

7.3.2 eZKit533

TODO.

7.3.3 TLL6527M

TODO.

7.4 epiphany (Epiphany)

7.4.1 epiphany_sim

TODO.

7.5 i386

7.5.1 pc386

TODO.

7.6 lm32 (LatticeMicro32)

7.6.1 lm32_evr

TODO.

7.6.2 milkymist

TODO.

7.7 m68k (Motorola 68000 / ColdFire)

7.7.1 av5282

TODO.

7.7.2 csb360

TODO.

7.7.3 gen68340

TODO.

7.7.4 gen68360

TODO.

7.7.5 genmcf548x

TODO.

7.7.6 mcf5206elite

TODO.

7.7.7 mcf52235

TODO.

7.7.8 mcf5225x

TODO.

7.7.9 mcf5235

TODO.

7.7.10 mcf5329

TODO.

7.7.11 mrm332

TODO.

7.7.12 mvme147

TODO.

7.7.13 mvme147s

TODO.

7.7.14 mvme162

TODO.

7.7.15 mvme167

TODO.

7.7.16 uC5282

TODO.

7.8 microblaze (Microblaze)

There are no Microblaze BSPs yet.

7.9 mips (MIPS)

7.9.1 csb350

TODO.

7.9.2 hurricane

TODO.

7.9.3 jmr3904

TODO.

7.9.4 malta

TODO.

7.9.5 rbtx4925

TODO.

7.9.6 rbtx4938

TODO.

7.10 moxie

7.10.1 moxiesim

TODO.

7.11 nios2 (Nios II)

7.11.1 nios2_iss

TODO.

7.12 or1k (OpenRISC 1000)

7.12.1 generic_or1k

TODO.

7.13 powerpc (PowerPC)

7.13.1 beatnik

TODO.

7.13.2 gen5200

TODO.

7.13.3 gen83xx

TODO.

7.13.4 haleakala

TODO.

7.13.5 motorola_powerpc

7.13.5.1 Boot Image Generation

The application executable file (ELF file) must be converted to a boot image. Use the following commands:

```
1 powerpc-rtems5-objcopy -O binary -R .comment -S ticker.exe rtems
2 gzip -9 -f rtems
3 powerpc-rtems5-ld -o ticker.boot bootloader.o --just-symbols=ticker.exe -b binary rtems.
  ↪ gz -T ppcboot.lds -no-warn-mismatch
4 powerpc-rtems5-objcopy -O binary ticker.boot ticker.bin
```

7.13.6 mpc55xxevb

TODO.

7.13.7 mpc8260ads

TODO.

7.13.8 mvme3100

TODO.

7.13.9 mvme5500

TODO.

7.13.10 psim

TODO.

7.13.11 gemuppc

TODO.

7.13.12 qorIQ (QorIQ)

The BSP for the **QorIQ** chip family offers three variants. The *qorIQ_e500* variant supports the P-series chips such as P1020, P2010 and P2020. The *qorIQ_e6500_32* (32-bit ISA) and *qorIQ_e6500_64* (64-bit ISA) variants support the T-series chips such as T2080 and T4240. The basic hardware initialization is not performed by the BSP. A boot loader with device tree support must be used to start the BSP, e.g. U-Boot.

The BSP is known to run on these boards:

- NXP P1020RDB
- MicroSys miriac MPX2020 (System on Module)
- Artesyn MVME2500 (VME64x SBC)
- NXP T2080RDB
- NXP T4240RDB
- MEN G52A (CompactPCI Serial)

The *qorIQ_core_0* and *qorIQ_core_1* variants should be used with care. They are intended for a *RTEMS_MULTIPROCESSING* configuration on the P1020.

7.13.12.1 Boot via U-Boot

The application executable file (ELF file) must be converted to an U-Boot image. Use the following commands:

```
1 powerpc-rtems5-objcopy -O binary app.exe app.bin
2 gzip -9 -f -c app.bin > app.bin.gz
3 mkimage -A ppc -O linux -T kernel -a 0x4000 -e 0x4000 -n RTEMS -d app.bin.gz app.img
```

Use the following U-Boot commands to boot an application via TFTP download:

```
1 tftpboot ${loadaddr} app.img && run loadfdt && bootm ${loadaddr} - ${fdt_addr} ; reset
```

7.13.12.2 Clock Driver

The clock driver uses two MPIC global timer (QORIQ_CLOCK_TIMER and QORIQ_CLOCK_TIMECOUNTER). In case QORIQ_IS_HYPERVISOR_GUEST is defined, then the PowerPC decremter is used.

7.13.12.3 Console Driver

The console driver supports the on-chip NS16550 compatible UARTs. In case `QORIQ_IS_HYPERVISOR_GUEST` is defined, then the EPAPR byte channel is used for the console device.

7.13.12.4 Network Interface Driver

The network interface driver is provided by the *libbsd*. The DPAA is supported including 10Gbit/s Ethernet.

7.13.12.5 Topaz Hypervisor Guest

For a Topaz hypervisor guest configuration use:

```
1 ../configure --enable-rtemsbsp=qoriq_e6500_32 \  
2   QORIQ_IS_HYPERVISOR_GUEST=1 \  
3   QORIQ_UART_0_ENABLE=0 \  
4   QORIQ_UART_1_ENABLE=0 \  
5   QORIQ_TLB1_ENTRY_COUNT=16
```

You may have to adjust the linker command file according to your partition configuration.

7.13.13 ss555

TODO.

7.13.14 t32mppe

TODO.

7.13.15 tqm8xx

TODO.

7.13.16 virtex

TODO.

7.13.17 virtex4

TODO.

7.13.18 virtex5

TODO.

7.14 riscv (RISC-V)

7.14.1 riscv

This BSP offers 13 variants:

- rv32i
- rv32iac
- rv32im
- rv32imac
- rv32imafc
- rv32imafd
- rv32imafdc
- rv64imac
- rv64imac_medany
- rv64imafd
- rv64imafd_medany
- rv64imafdc
- rv64imafdc_medany
- frdme310arty

Each variant corresponds to a GCC multilib. A particular variant reflects an ISA with ABI and code model choice.

The basic hardware initialization is not performed by the BSP. A boot loader with device tree support must be used to start the BSP, e.g. BBL. The BSP must be started in machine mode.

The reference platform for this BSP is the Qemu *virt* machine.

7.14.1.1 Build Configuration Options

The following options are available at the configure command line.

BSP_PRESS_KEY_FOR_RESET

If defined to a non-zero value, then print a message and wait until pressed before resetting board when application terminates.

BSP_RESET_BOARD_AT_EXIT

If defined to a non-zero value, then reset the board when the application terminates.

BSP_PRINT_EXCEPTION_CONTEXT

If defined to a non-zero value, then print the exception context when an unexpected exception occurs.

BSP_FDT_BLOB_SIZE_MAX

The maximum size of the device tree blob in bytes (default is 65536).

BSP_CONSOLE_BAUD

The default baud for console driver devices (default 115200).

RISCV_MAXIMUM_EXTERNAL_INTERRUPTS

The maximum number of external interrupts supported by the BSP (default 64).

RISCV_ENABLE_HTIF_SUPPORT

Enables the HTIF support if defined to a non-zero value, otherwise it is disabled (disabled by default).

RISCV_CONSOLE_MAX_NS16550_DEVICES

The maximum number of NS16550 devices supported by the console driver (2 by default).

RISCV_RAM_REGION_BEGIN

The begin of the RAM region for linker command file (default is 0x70000000 for 64-bit with -mcmode=medlow and 0x80000000 for all other).

RISCV_RAM_REGION_SIZE

The size of the RAM region for linker command file (default 64MiB).

RISCV_ENABLE_FRDME310ARTY_SUPPORT

Enables support sifive Freedom E310 Arty board if defined to a non-zero value, otherwise it is disabled (disabled by default)

7.14.1.2 Interrupt Controller

Exactly one Core Local Interruptor (CLINT) and exactly one Platform-Level Interrupt Controller (PLIC) are supported. The maximum number of external interrupts supported by the BSP is defined by the RISCV_MAXIMUM_EXTERNAL_INTERRUPTS BSP option.

7.14.1.3 Clock Driver

The clock driver uses the CLINT timer.

7.14.1.4 Console Driver

The console driver supports devices compatible to

- “ucb,htif0” (depending on the RISCV_ENABLE_HTIF_SUPPORT BSP option),
- “ns16550a” (see RISCV_CONSOLE_MAX_NS16550_DEVICES BSP option), and
- “ns16750” (see RISCV_CONSOLE_MAX_NS16550_DEVICES BSP option).
- “sifive,uart0” (see RISCV_ENABLE_FRDME310ARTY_SUPPORT BSP option).

They are initialized according to the device tree. The console driver does not configure the pins or peripheral clocks. The console device is selected according to the device tree “/chosen/stdout-path” property value.

7.14.2 griscv

This RISC-V BSP supports chips using the [GRLIB](#).

7.15 sh (SuperH)

7.15.1 gensh1

TODO.

7.15.2 gensh2

TODO.

7.15.3 gensh4

TODO.

7.15.4 shsim

TODO.

7.16 sparc64 (SPARC V9)

7.16.1 niagara

TODO.

7.16.2 usiii

TODO.

7.17 sparc (SPARC / LEON)

7.17.1 erc32

TODO.

7.17.2 leon2

TODO.

7.17.3 leon3

TODO.

7.18 v850 (V850)

7.18.1 gdbv850sim

TODO.

7.19 x86_64

7.19.1 amd64

This BSP offers only one variant, amd64. The BSP can run on UEFI-capable systems by using FreeBSD's bootloader, which then loads the RTEMS executable (an ELF image).

Currently only the console driver and context initialization and switching are functional (to a bare minimum), but this is enough to run the `hello.exe` sample in the RTEMS testsuite.

7.19.1.1 Build Configuration Options

There are no options available to configure at build time, at the moment.

7.19.1.2 Testing with QEMU

To test with QEMU, we need to:

- Build / install QEMU (most distributions should have it available on the package manager).
- Build UEFI firmware that QEMU can use to simulate an x86-64 system capable of booting a UEFI-aware kernel, through the `--bios` flag.

Building TianoCore's UEFI firmware, OVMF

Complete detailed instructions are available at [TianoCore's Github's wiki](#).

Quick instructions (which may fall out of date) are:

```
1 $ git clone git://github.com/tianocore/edk2.git
2 $ cd edk2
3 $ make -C BaseTools
4 $ . edksetup.sh
```

Then edit `Conf/target.txt` to set:

```
1 ACTIVE_PLATFORM      = OvmfPkg/OvmfPkgX64.dsc
2 TARGET               = DEBUG
3 TARGET_ARCH          = X64
4 # You can use GCC46 as well, if you'd prefer
5 TOOL_CHAIN_TAG       = GCC5
```

Then run build in the `edk2` directory - the output should list the location of the `OVMF.fd` file, which can be used with QEMU to boot into a UEFI shell.

You can find the `OVMF.fd` file like this as well in the `edk2` directory:

```
1 $ find . -name "*.fd"
2 ./Build/OvmfX64/DEBUG_GCC5/FV/MEMFD.fd
3 ./Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd # the file we're looking for
4 ./Build/OvmfX64/DEBUG_GCC5/FV/OVMF_CODE.fd
5 ./Build/OvmfX64/DEBUG_GCC5/FV/OVMF_VARS.fd
```

7.19.1.3 Boot RTEMS via FreeBSD's bootloader

The RTEMS executable produced (an ELF file) needs to be placed in the FreeBSD's `/boot/kernel/kernel`'s place.

To do that, we first need a hard-disk image with FreeBSD installed on it. [Download FreeBSD's installer "memstick" image for amd64](#) and then run the following commands, replacing paths as appropriate.

```
1 $ qemu-img create freebsd.img 8G
2 $ OVMF_LOCATION=/path/to/ovmf/OVMF.fd
3 $ FREEBSD_MEMSTICK=/path/to/FreeBSD-11.2-amd64-memstick.img
4 $ qemu-system-x86_64 -m 1024 -serial stdio --bios $OVMF_LOCATION \
5   -drive format=raw,file=freebsd.img \
6   -drive format=raw,file=$FREEBSD_MEMSTICK
```

The first time you do this, continue through and install FreeBSD. [FreeBSD's installation guide may prove useful](#) if required.

Once installed, build your RTEMS executable (an ELF file), for eg. `hello.exe`. We need to transfer this executable into `freebsd.img`'s filesystem, at either `/boot/kernel/kernel` or `/boot/kernel.old/kernel` (or elsewhere, if you don't mind user FreeBSD's loader's prompt to boot your custom kernel).

If your host system supports mounting UFS filesystems as read-write (eg. FreeBSD), go ahead and:

1. Mount `freebsd.img` as read-write
2. Within the filesystem, back the existing FreeBSD kernel up (i.e. effectively `cp -r /boot/kernel /boot/kernel.old`).
3. Place your RTEMS executable at `/boot/kernel/kernel`

If your host doesn't support mounting UFS filesystems (eg. most Linux kernels), do something to the effect of the following.

On the host

```
1 # Upload hello.exe anywhere accessible within the host
2 $ curl --upload-file hello.exe https://transfer.sh/rtems
```

Then on the guest (FreeBSD), login with root and

```
1 # Back the FreeBSD kernel up
2 $ cp -r /boot/kernel/ /boot/kernel.old
3 # Bring networking online if it isn't already
4 $ dhclient em0
5 # You may need to add the --no-verify-peer depending on your server
6 $ fetch https://host.com/path/to/rtems/hello.exe
7 # Replace default kernel
8 $ cp hello.exe /boot/kernel/kernel
9 $ reboot
```

After rebooting, the RTEMS kernel should run after the UEFI firmware and FreeBSD's bootloader. The `-serial stdio` QEMU flag will let the RTEMS console send its output to the host's stdio stream.

7.19.1.4 Paging

During the BSP's initialization, the paging tables are setup to identity-map the first 512GiB, i.e. virtual addresses are the same as physical addresses for the first 512GiB.

The page structures are set up statically with 1GiB super-pages.

Note: Page-faults are not handled.

Warning: RAM size is not detected dynamically and defaults to 1GiB, if the configuration-time `RamSize` parameter is not used.

7.19.1.5 Interrupt Setup

Interrupt vectors 0 through 32 (i.e. 33 interrupt vectors in total) are setup as “RTEMS interrupts”, which can be hooked through `rtems_interrupt_handler_install`.

The Interrupt Descriptor Table supports a total of 256 possible vectors (0 through 255), which leaves a lot of room for “raw interrupts”, which can be hooked through `_CPU_ISR_install_raw_handler`.

Since the APIC needs to be used for the clock driver, the PIC is remapped (IRQ0 of the PIC is redirected to vector 32, and so on), and then all interrupts are masked to disable the PIC. In this state, the PIC may *still* produce spurious interrupts (IRQ7 and IRQ15, redirected to vector 39 and vector 47 respectively).

The clock driver triggers the initialization of the APIC and then the APIC timer.

The I/O APIC is not supported at the moment.

Note: IRQ32 is reserved by default for the APIC timer (see following section).

IRQ255 is reserved by default for the APIC's spurious vector.

Warning: Besides the first 33 vectors (0 through 32), and vector 255 (the APIC spurious vector), no other handlers are attached by default.

7.19.1.6 Clock Driver

The clock driver currently uses the APIC timer. Since the APIC timer runs at the CPU bus frequency, which can't be detected easily, the PIT is used to calibrate the APIC timer, and then the APIC timer is enabled in periodic mode, with the initial counter setup such that interrupts fire at the same frequency as the clock tick frequency, as requested by `CONFIGURE_MICROSECONDS_PER_TICK`.

7.19.1.7 Console Driver

The console driver defaults to using the COM1 UART port (at I/O port 0x3F8), using the NS16550 polled driver.

EXECUTABLES

This section discusses what an RTEMS executable is and what happens when you execute it in a target. The section discusses how an application executable is created, what happens when an executable is loaded and run, debugging an executable, and creating and dynamically loading code.

8.1 RTEMS Executable

Running executables is the most important part of working with RTEMS, it is after all how you run your application and use the RTEMS kernel services.

An RTEMS executable is embedded in a target and executing an embedded executable has challenges not faced when executing software on a desktop or server computer. A desktop or server operating system kernel provides all the support needed to bring an executable's code and data into a process's address space passing control to it and cleaning up when it exits. An embedded target has to provide similar functionality to execute an embedded executable.

An RTEMS Source Builder (RSB) built RTEMS tool chain is used to create RTEMS executables. The tool chain executable creates a fixed position statically linked Extendable Loader Format (ELF) file that contains the RTEMS kernel, standard libraries, third-party libraries and application code. RTEMS executes in a single address space which means it does not support the fork or exec system calls so statically linking all the code is the easiest and best way to create an executable.

An RTEMS application is constructed vertically with the RTEMS kernel, BSP support code and drivers close to the hardware, above which sit the RTEMS Application Programming Interfaces (API) for control of threads, mutex and other resources an application may use. Middle-ware services like networking, interpreted languages, and protocol stacks sit between the RTEMS APIs and the application components. The software built into an executable can be seen as a vertical software stack.

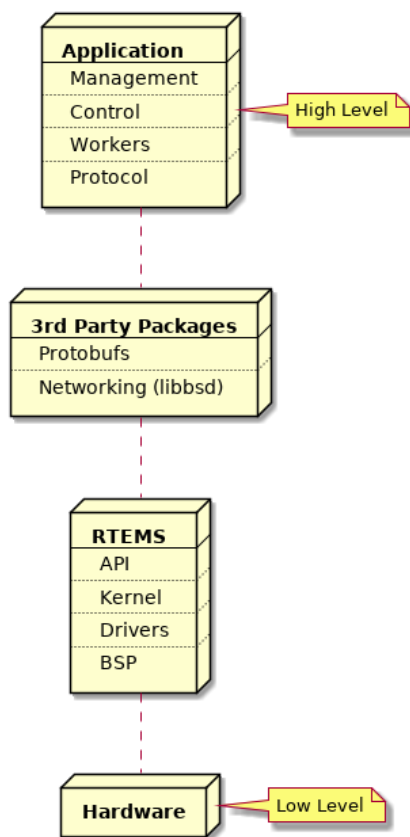


Fig. 1: Vertical Software Stack

8.2 Building an Application

RTEMS views any code it is running and using it's interfaces as an application. RTEMS conforms to a number of international standards such as POSIX and can build and run portable code written in languages such as C, C++ and Ada.

Applications are built from source into ELF object files, third-party packages can be built as libraries or they can be imported as source into an application code base. The application, third-party packages, RTEMS and standard libraries are linked to create the RTEMS executable. The executable is transferred to the target and a bootloader loads it from the non-volatile storage into RAM or the code is executed in place in the non-volatile storage. The target hardware defines what happens.

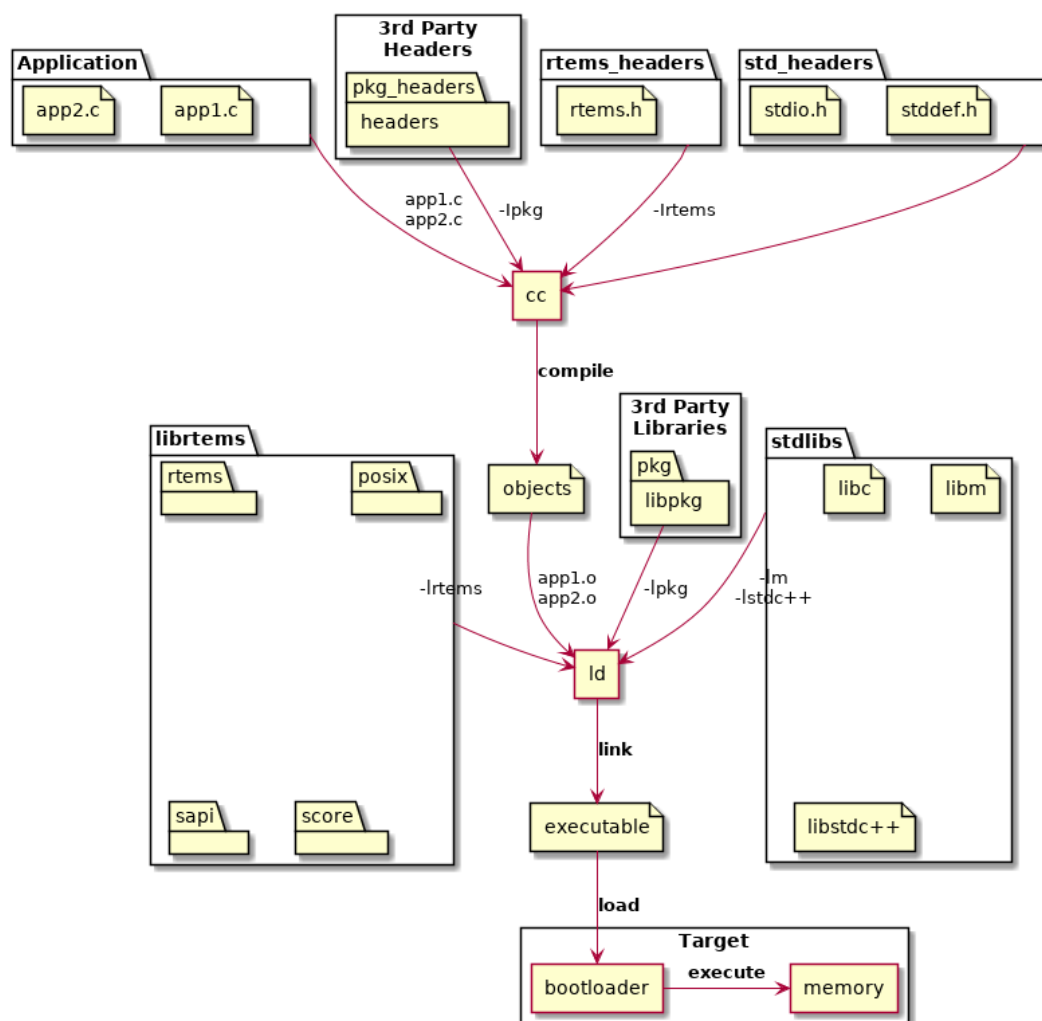


Fig. 2: Building an Application

The standard and third-party libraries are a collection of object files built using the same set of tools the application source is compiled with. The package collects it's object files into an archive or library.

RTEMS does not provide a standard application build system. The RTEMS ecosystem provides support so a range of build systems can be used. Applications can be built with make, autotools, cmake, waf and more. User should select a build system that meets their project, system, corporate or personal needs.

8.2.1 Machine Flags and ABI

All code in an RTEMS executable must be built with the same machine flags. The machine flags control the instruction set and application binary interface (ABI) the compiler generates. As the executable is statically linked all code must use the same instruction set the hardware is configured to support and all code must conform to the same ABI. Any variation can result in unpredictable behavior such as crashes, failures or lock ups. It is recommend an executable is built with the same or equivalent tool set. Mixing of tool set versions can also result in undefined behavior. The RTEMS tool `rtems-execinfo` can audit an RTEMS executable and list the machine flags and compilers used.

RTEMS by default does not support instruction emulation for unsupported instructions. RTEMS applications are normally built from source so binary compatibility is not as important as performance. Instruction emulation is costly to execute and rebuilding the executable with the correct instruction set only needs to be done once.

8.3 Target Execution

Fixed position statically linked executables have a fixed address in a target's address space. The location in the address space for code, data and read-only data is fixed. The BSP defines the memory map and it is set by the BSP developer based on the target's hardware requirements and its bootloader.

Targets typically contains a bootloader that is executed after the target's processor exits reset. A bootloader is specific to a target's processor and hardware configuration and is responsible for the low level initialization of the hardware resources needed to load and execute an operating system's kernel. In the case of RTEMS this is the RTEMS executable.

Bootloaders vary in size, complexity and functionality. Some architectures have a number of bootloader stages and others have only minimal support. An example of a high end system is Xilinx's Zynq processor with three stages. First a mask ROM in the System On Chip (SOC) executes after reset loading a first stage bootloader (FSBL) from an SD card, QSPI flash or NAND flash depending on signals connected to the device. The FSBL loads a second stage bootloader (SSBL) such as U-Boot and this loads the kernel. U-Boot can be configured to load a kernel from a range of media and file system formats as well as over a network using a number of protocols. This structure provides flexibility at the system level to support development environments such as a workshop or laboratory through to tightly control production configurations.

Bootloaders often have custom formats for the executable image they load. The formats can be simple to keep the bootloader simple or complex to support check-sums, encryption or redundancy in case an image becomes corrupted. A bootloader often provides a host tool that creates the required file from the RTEMS executable's ELF file.

If RTEMS is to run from RAM the bootloader reads the image and loads the code, initialized data and read-only data into the RAM and then jumps to a known entry point. If the code is executed from non-volatile storage the process to write the image into that storage will have extracted the various binary parts and written those to the correct location.

The important point to note is the binary parts of the executable are somehow loaded into the target's address space ready to execute. The way this done may vary but the out come is always the same, the binary code, data and read-only data is resident in the processor's address space at the BSP defined addresses.

8.4 BSP Initialization

The bootloader jumps or calls the RTEMS executable's entry point, normally a fixed address. The BSP entry point or start up code performs:

1. Low level processor specific initialization that such as setting control registers so the processor is operating in a mode RTEMS is built for
2. Cache flushing, clearing and invalidation
3. Memory management unit (MMU) set up if required
4. Clear the uninitialized data section
5. Process a command line if supported by the bootloader
6. Call bootcard which disabled interrupts, saves away a command line if the BSP supports it then call the RTEMS kernel early initialize entry point `rtems_initialize_executive`. This call never returns.

Further BSP initialization happens as part of RTEMS kernel's System Initialization process. The following handlers are declared and if provided are placed at the beginning of the initialization handler list. The BSP can provides:

bsp_work_area_initialize

This function determines the amount of memory that can be given to RTEMS for the workspace and the C library heap which malloc uses. The call typically uses the `bsp_work_area_initialize_default` to perform actually perform the initialization.

bsp_start

This function is specialized for each architecture and even for some BSPs. It performs the low level initialization RTEMS needs so it can run on the architecture and BSP.

bsp_predriver_hook

This function can be used to initialize hardware drivers depend on such as configuring an interrupt controller. The default version is empty and does nothing.

BSPs all perform similar operations with common functionality and the RTEMS kernel provides common code that can be shared between BSPs. The use of the common code is encouraged for all new BSPs.

8.5 RTEMS Initialization

The RTEMS kernel initialization is:

1. Invoke the registered system initialization handlers
2. Set the system state to **up**
3. If the kernel supports SMP request multitasking start. All online cores are transferred to the **ready to start multitasking** state.
4. Start threaded multitasking. RTEMS starts multitasking by getting the first thread to run and dispatching it.

C++ static object constructors are called in the context of the first running thread before the thread body is entered.

8.5.1 System Initialization Handlers

RTEMS supports the automatic registration of services used in applications. This method of initialization automatically configures RTEMS with only the services used in an application. There is no manual configuration of services used and no updating of initialization function tables.

RTEMS uses specialized sections in the ELF executable to perform this task. The system is based on the [FreeBSD SYSINT Framework](#). Ordered initialization is performed before multitasking is started.

The RTEMS Tool `rtems-exeinfo` can provide some detail about the registered handlers. The following shows the initialization handlers for the *hello world* sample application in the RTEMS kernel's testsuite:

```
1 .. code-block:: none
```

```
$ rtems-exeinfo -init arm-rtems5/c/xilinx_zynq_zedboard/testsuites/samples/hello.exe
RTEMS Executable Info 5.5416cfa39dd6 $ rtems-exeinfo -init arm-
rtems5/c/xilinx_zynq_zedboard/testsuites/samples/hello.exe exe: arm-
rtems5/c/xilinx_zynq_zedboard/testsuites/samples/hello.exe
```

Compilation:

Producers: 2

```
GNU AS 2.31.1: 14 objects
GNU C11 7.3.0 20180125 (RTEMS 5, RSB
e55769c64cf1a201588565a5662deafe3f1ccdcc, Newlib
103b055035fea328f8bc7826801760fb1c055683): 284 objects
```

Common flags: 4

```
-march=armv7-a -mthumb -mfpu=neon -mfloat-abi=hard
```

Init sections: 2

.init_array

```
0x001047c1 frame_dummy
```

.rtemsroset

```
0x00104c05      bsp_work_area_initialize      0x00104c41      bsp_start
0x0010eb45 zynq_debug_console_init 0x0010ec19 rtems_counter_sysinit
0x0010b779                                     _User_extensions_Handler_initialization
0x0010c66d      rtems_initialize_data_structures      0x00107751
_RTEMS_tasks_Manager_initialization      0x0010d4f5
_POSIX_Keys_Manager_initialization      0x0010dd09      _Thread_Create_idle
0x0010cf01      rtems_libio_init      0x001053a5      rtems_filesystem_initialize
0x0010546d _Console_simple_Initialize 0x0010c715 _IO_Initialize_all_drivers
0x001076d5      _RTEMS_tasks_Initialize_user_tasks_body      0x0010cfa9
rtems_libio_post_driver
```

The section `.rtemsroset` lists the handlers called in order. The handlers can be split into the BSP initialization handlers that start the BSP:

- `bsp_work_area_initialize`
- `bsp_start`
- `zynq_debug_console_init`
- `rtems_counter_sysinit`

And the remainder are handlers for services used by the application. The list varies based on the services the application uses.

8.6 Debugging

An RTEMS executable is debugged by loading the code, data and read-only data into a target with a debugger connected. The debugger running on a host computer accesses the ELF file reading the debug information it contains.

The executable being debugged needs to be built with the compiler and linker debug options enabled. Debug information makes the ELF executable file large but it does not change the binary footprint of the executable when resident in the target. Target boot loaders and file conversion tools extract the binary code, data and read-only data to create the file embedded on the target.

An ELF executable built with debug information contains DWARF debug information. DWARF is a detailed description of the executable a debugger uses to locate functions, find data, understand the type and structure of a variable, and know how much entry code every call has. The debugger uses this information to set break points, step functions, step instructions, view the data and much more.

We recommend the compiler and linker debug options are always enabled. An ELF file with debug information can be used to investigate a crash report from a production system if the production ELF image is archived. The RTEMS tools chain provides tools that can take an address from a crash dump and find the corresponding instruction and source line. The extra size the debug information adds does not effect the target footprint and the extra size on a host is small compared to the benefits it brings.

A desktop or server operating system's kernel hosts the executable being debugged handling the interaction with the executable and the debugger. The debugger knows how to communicate to the kernel to get the information it needs. Debugging an embedded executable needs an extra piece called an agent to connect the target to the debugger. The agent provides a standard remote interface to the debugger and an agent specific connection to the target.



Fig. 3: Embedded Executable Debugging

The RTEMS tool chain provides the GNU debugger GDB. GDB has a remote protocol that can run over networks using TCP and UDP protocols. The GDB remote protocol is available in a number of open source and commercial debugging solutions. Network debugging using the remote protocol helps setting up a laboratory, the targets can be remote from the developers desktop allowing for better control of the target hardware while avoiding the need to plug devices in to an expensive desktop or server machine.

The following are some examples of GDB and GDB server environments RTEMS supports.

QEMU contains a debugging agent for the target being simulated. A QEMU command line option enables a GDB server and the simulator manages the interaction with the target processor and it's memory and caches.

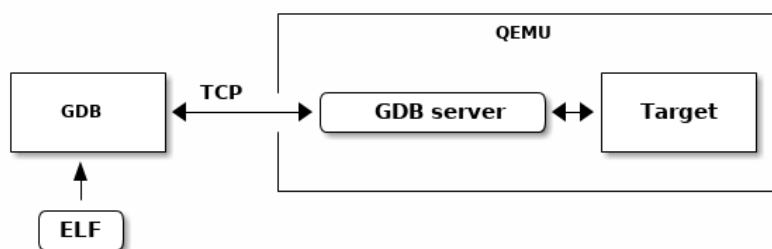


Fig. 4: QEMU Executable Debugging

OpenOCD is a JTAG debugging package that interfaces to a wide of JTAG pods. JTAG is a low level high speed serial interface modern processors provide as a means of controlling the core processing logic. The features available depend on the architecture and processor. Typical functions include:

1. Processor control and register access
2. System level register access to allow SOC initialization
3. General address space access
4. Cache and MMU control
5. Break and watch points

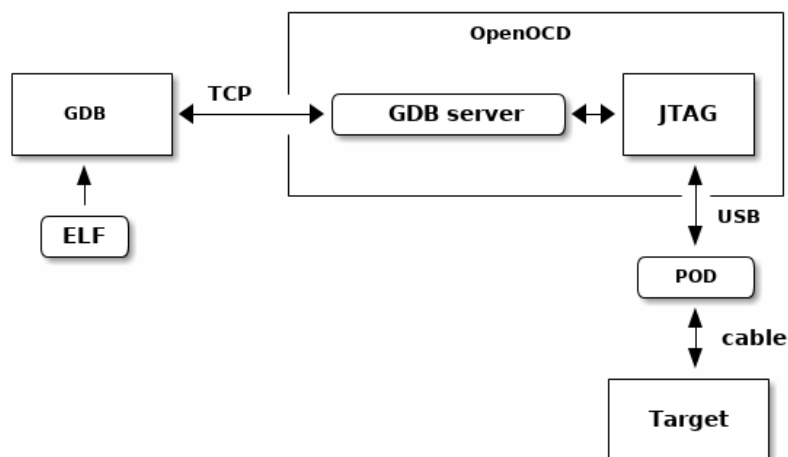


Fig. 5: OpenOCD JTAG Executable Debugging

The RTEMS kernel has a debugging agent called libdebugger. This is a software based agent that runs within RTEMS using network services to provide a remote GDB protocol interface. A growing number of architectures are supported. The RTEMS debugging agent is for application development providing thread aware stop model debug experience.

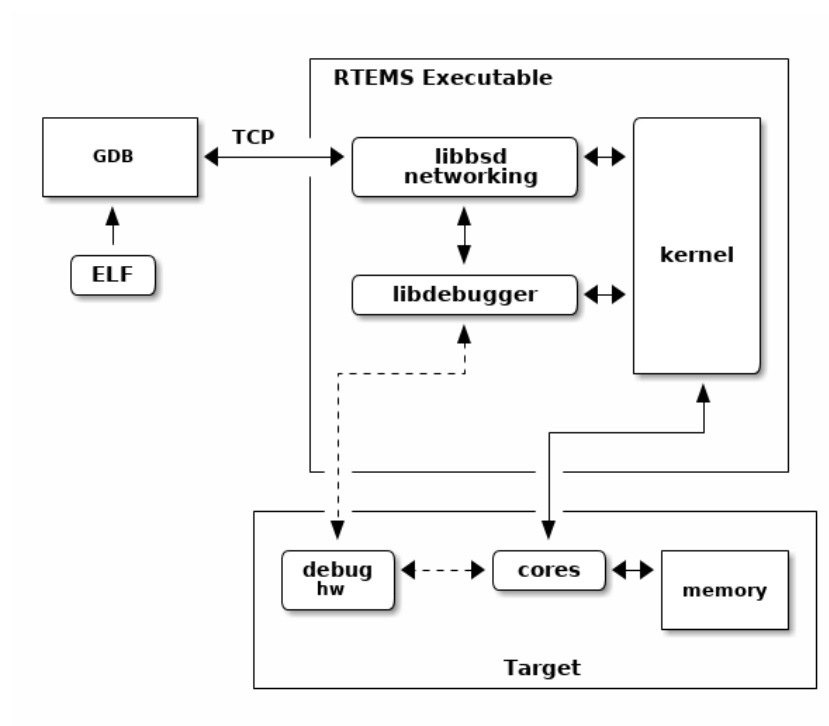


Fig. 6: Libdebugger Executable Debugging

8.7 Dynamic Loader

RTEMS supports dynamically loading of executable code and data in the form of object files into a running system where the run-time loaded code can be executed and data accessed

This section describes RTEMS loader, preparing and loading executable code into a running system, the supported architectures and any limitation that may exist with an architecture.

The RTEMS operating system contains a link editor that runs on the target. The link editor supports loading Extendable Linker Format (ELF) relocatable executable object files locating the code and data in the target's address space as it is loaded. An executable object file's external references to function identifiers and data object identifiers are resolved and any external symbols can be made available in the global symbol table. The executing performance of dynamically loaded code is similar to the same code statically linked into an executable. This is a core requirement of the RTEMS link editor.

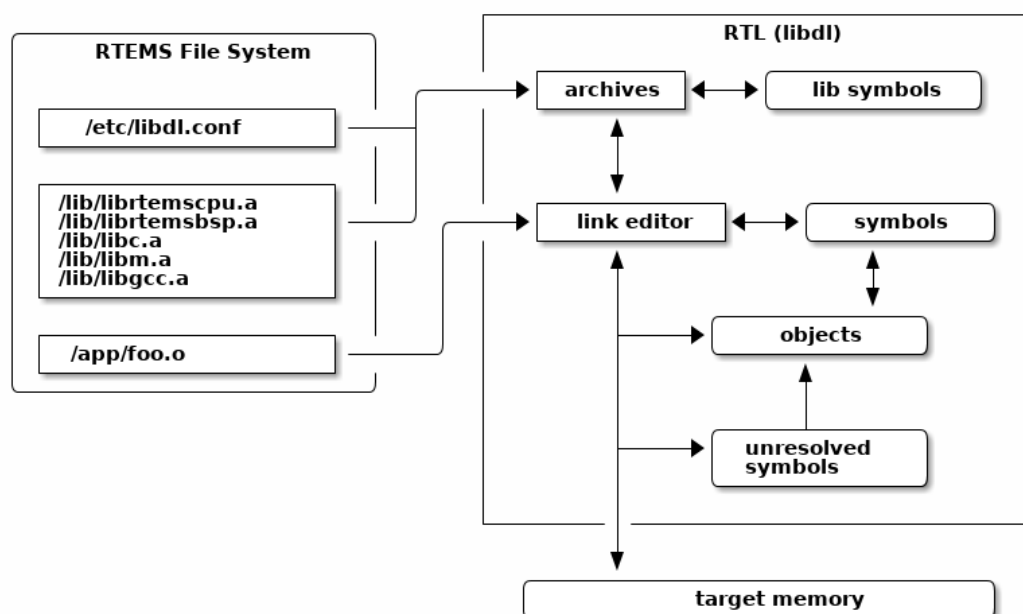


Fig. 7: Run Time Loader (libdl)

The RTEMS operating system's dynamic loader is not the same as the dynamic shared library support Unix or Windows have. Those operating systems use dynamic loading to share code between processes and this is an important feature in their design. RTEMS is a single address space operating system and that means there is no ability to share code at run-time. As a result code is loaded in a similar manner to static linking removing the need for any overheads sharing code may have.

To load an executable object file it must be resident on a target and accessible by RTEMS's file system. The executable object file can be a single file or a collection in a library stored using the Unix standard archive format. The RTEMS loader supports the extended GNU format for long file names in archives.

The RTEMS developers do not see dynamically loading of code as a real-time activity. A system should not respond to real-time external events by loading code. The loading of code should happen before a system is considered available and the activity the system is experiencing is

low and stable.

The statically linked executable that is loaded and run after reset is called the *base image*. The *base image* contains your base application that is used to dynamically load code, a global symbol table, the parts of the RTEMS operating system code used in the base image as well as functions and data from the tool suite libraries and packages you are using. Only the software referenced is used to create the base image. The parts of the libraries not referenced are not part of the executable or present in the global symbol table.

Application software can locate a symbol by name and call the address or reference the data at that address. A function identifier located by a symbol does not have its signatures checked, it is the responsibility of the caller to make sure the function is called with the correct arguments. It is the same for data objects, there is no type checking. Symbol versioning is not supported and supporting it does not make sense within the RTEMS operating system. An RTEMS target system is closed to normal users and software needs to be built from the same tool set and header files used to build the base image.

An executable object file's text or code has to be built for the target's architecture it is loaded on and it must be built with the same ABI flags the base image is built with. See *Machine Flags and ABI* (page 132).

8.7.1 System Design

The use of dynamic loading in a project is a system design decision. Some systems will have strict requirements where loading code into a live system is not allowed while other projects will benefit from the system level flexibility dynamically loading code provides.

Code loaded at run time needs to be resident or accessible to the target via RTEMS's file system. Targets that have suitable media or a network interface to NFS servers to hold the executable object and library files are best suited.

Dynamically loading code uses more memory than statically linking the same code into the base image. The link editor maintains symbol tables where each symbol is a string, an address, and some additional data. The executable object files resident in memory each have data to manage them, the memory they use, and any dependencies they might have. The link editor is designed to minimize the memory overheads however only statically linked executables have no memory overhead.

The link editor relocates the code and data into RAM fixing it to the load address as it is loaded. A target needs to have suitably configured memory available for the executable object file to reside in. The memory must be able to support read, write and executable type access. Fine control of the memory and its modes can be supported using a custom allocator. Examples are systems that have a custom memory map, specialized memory for the execution of code or a requirement for read-only executable sections.

The load address of an executable object file is determined by the load order and the allocator used. The default allocator for the link editor is the system heap which means the location a specific executable object file is loaded at depends on the memory allocated before it is loaded and when in the load order it is loaded. A statically linked executable's address map is fixed and this is considered important in some systems. A dynamically loaded system can be loaded in a repeatable manner if the load order is the same and the initialization sequence of the system is controlled. A custom allocator may also help.

Management of dynamically loadable object files and libraries adds to the configuration management of the hosts in a project. The loadable files need to be released and tracked in a

suitable configuration management process just like the base image is. Executable object files and libraries are specific to a version of RTEMS and cannot be mixed or moved and this needs to be carefully managed. Currently there are no checks an executable object file matches the version of the base image it is being loaded on. These extra configuration controlled items add to the overheads of a project and need to be considered.

Dynamically loadable systems have a number of features that benefit some systems and products. Systems can be built on a base of trusted or *golden* modules. A number of projects using a common base of hardware can make use of proven modules reducing the testing and qualification overhead for each new release. A tested base image with libraries for common and available boards provides a simple and fast way for new users to trial and use RTEMS.

A project can use dynamic loading during development, shipping statically linked executables in production. Hardware used by a development team can have more memory, extra media for disk drives, or a network interface.

8.7.2 Loader Interface

Run-time executable object file loading and management is via the standard's based calls provided by the header file `<dlfcn.h>`. The details of the calls follow.

void* dlopen(const char* path, int mode);

The `dlopen()` function makes the symbols (function identifiers and data object identifiers) in the executable object file specified by *file* available to the calling program.

The executable object files eligible for this operation are in the ELF format.

The link loader may load embedded dependencies in executable object files. In such cases, a `dlopen()` operation may load those dependencies in addition to the executable object file specified by *file*.

A successful `dlopen()` returns a *handle* which the caller may use on subsequent calls to `dlsym()`, `dlinfo()` and `dlclose()`.

The value of the *handle* should not be interpreted in any way by the caller.

Subsequent calls to `dlopen()` for the same executable object file increases the references to it.

The *file* argument is used to construct a pathname to the executable object file or archive library of executable object files. If the *file* argument contains a colon (:) the name of the executable object file in the library follows and this file name may optionally end with @ followed by a number which is the absolute offset in the library file where the executable object file starts. If an executable object file is not detected at the offset the archive library's file table is searched.

If *file* is a null pointer, `dlopen()` returns a global symbol table handle. This *handle* provides access to the global symbols from an ordered set of executable object files consisting of the original base image file, the set of executable object files loaded using `dlopen()` operations with the `RTLD_GLOBAL` flag, and any dependencies loaded. As the latter sets of executable object files can change during execution, the set of symbols made available by this *handle* can also change dynamically.

Only a single copy of an executable object file is brought into the address space, even if `dlopen()` is invoked multiple times in reference to the executable object file, and even if different pathnames are used to reference the executable object file.

Unresolved external symbols do not cause an error to be returned allowing the loading of jointly dependent executable object files.

If `dlopen()` fails, it returns a null pointer, and sets an error condition which may be interrogated with `dlerror()`.

The *mode* parameter describes how `dlopen()` operates upon *file* with respect to the processing of relocations and the scope of visibility of the symbols provided within *file*. When an executable object file is brought into the address space, it may contain references to symbols whose addresses are not known until the executable object file is loaded.

If a loaded executable object file and any dependent executable object files loaded with it contain any initialiser functions, they are called in the order loaded before `dlopen()` returns.

The modes `RTLD_LAZY` and `RTLD_NOW` do not effect the type of relocation performed, it is same for both modes. All relocations of an executable object file and any dependent executable object files loaded with it are completed before the `dlopen()` call returns. The execution performance of the code loaded can be considered deterministic once `dlopen()` has returned.

Any executable object file loaded by `dlopen()` can reference global symbols in the base image, any executable object files loaded included in the same `dlopen()` invocation, and any executable object files that were loaded in any `dlopen()` invocation and which specified the `RTLD_GLOBAL` flag. To determine the scope of visibility for the symbols loaded with a `dlopen()` invocation, the *mode* parameter should be a bitwise-inclusive OR with one of the following values:

RTLD_GLOBAL

The executable object file's symbols are made available for relocation processing of any other executable object file. In addition, symbol lookup using `dlopen(NULL, mode)` and an associated `dlsym()` allows executable object files loaded with this mode to be searched.

RTLD_LOCAL

The executable object file's symbols shall not be made available for relocation processing of any other executable object files.

If neither `RTLD_GLOBAL` nor `RTLD_LOCAL` is specified, the default behavior is unspecified.

If `RTLD_GLOBAL` has been specified, the executable object file maintains its `RTLD_GLOBAL` status regardless of any previous or future specification of `RTLD_LOCAL`, as long as the executable object file remains in the address space.

Symbols introduced through calls to `dlopen()` may be used in relocation activities. Symbols that duplicate symbols already defined by the base image or previous `dlopen()` calls are treated as an error and the object file is not loaded. Symbols introduced through loading dependent executable object files are ignored or not loaded depending on the method used to build the executable object files.

The symbols introduced by `dlopen()` operations and available through `dlsym()` are at a minimum those which are exported as identifiers of global scope by the executable object file. Typically, such identifiers shall be those that were specified in (for example) C source code as having extern linkage.

int dlclose(void* handle);

Releases a reference to the executable object file referenced by *handle*. If the reference count drops to 0, the executable object file's global symbol table is made unavailable. When all references to the global symbols the executable object file provided have been removed the object file is removed from the address space.

If the executable object being removed has any termination routines in it they are called.

void* dlsym(void* handle, const char* symbol);

The `dlsym()` function obtains the address of a symbol (a function identifier or a data object identifier) defined in the symbol table identified by the `handle` argument. The `handle` argument is a symbol table handle returned from a call to `dlopen()` (and which has not since been released by a call to `dlclose()`), and `name` is the symbol's name as a character string. The return value from `dlsym()`, cast to a pointer to the type of the named symbol, can be used to call (in the case of a function) or access the contents of (in the case of a data object) the named symbol.

The `dlsym()` function searches for the named symbol in the symbol table referenced by `handle` and returns the address of the code or data location specified by the null-terminated character string `symbol`. Which libraries and objects are searched depends on the *handle* parameter.

Upon successful completion, if `name` names a function identifier, `dlsym()` returns the address of the function converted from type pointer to function to type pointer to void; otherwise, `dlsym()` shall return the address of the data object associated with the data object identifier named by `name` converted from a pointer to the type of the data object to a pointer to void. If *handle* does not refer to a valid symbol table handle or if the symbol named by `name` cannot be found in the symbol table associated with *handle*, `dlsym()` shall return a null pointer.

int dlinfo(void* handle, int request, void* args);

The `dlinfo()` function provides information about dynamically loaded object. The action taken by `dlinfo()` and exact meaning and type of the argument *args* depend on value of the *request* argument provided by the caller.

RTLD_DI_UNRESOLVED

Return 1 in an indexer value pointed to by *args* if the symbol table handle has unresolved relocation records to symbols. If the *handle* is the global symbol table handle or `RTLD_SELF` return 1 if any unresolved relocation records to symbols are present in any loaded executable object files..

const char *dlerror(void);

The `dlerror()` function returns a null-terminated character string (with no trailing <newline>) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of `dlerror()`, `dlerror()` returns NULL. Thus, invoking `dlerror()` a second time, immediately following a prior invocation, results in NULL being returned.

This example opens an object file, checks for any unresolved symbols the object file may have, locates a global symbol in the object file, calls it then closes the object file:

```

1 #include <stdbool.h>
2 #include <stdio.h>
3 #include <dlfcn.h>
4
5 typedef int (*call_sig)(void);
6
7 bool load_object (void)
8 {
9     void*    handle;
10    call_sig call;
11    int      unresolved;
12
```

(continues on next page)

(continued from previous page)

```

13 handle = dlopen ("/code.o", RTLD_NOW | RTLD_GLOBAL);
14 if (handle == NULL)
15 {
16     printf ("dlopen failed: %s\n", dlerror ());
17     return false;
18 }
19
20 if (dlinfo (handle, RTLD_DI_UNRESOLVED, &unresolved) < 0)
21 {
22     printf ("dlinfo failed: %s\n", dlerror ());
23     dlclose (handle);
24     return false;
25 }
26
27 if (unresolved != 0)
28 {
29     printf ("object.o has unresolved external symbols\n");
30     dlclose (handle);
31     return false;
32 }
33
34 call = dlsym (handle, "foo");
35 if (call == NULL)
36 {
37     printf("dlsym failed: symbol 'foo' not found\n");
38     dlclose (handle);
39     return false;
40 }
41
42 printf ("foo() returns: %i\n", call ());
43
44 if (dlclose (handle) < 0)
45 {
46     printf("dlclose failed: %s\n", dlerror());
47     return false;
48 }
49
50 return true;
51 }

```

8.7.3 Symbols

The RTEMS link editor manages the symbols for the base image and all resident executable object files. A symbol is an identifier string and a pointer value to a function identifier or a data object identifier. The symbols held in the symbol tables are used in the relocation of executable object files or they can be accessed by application code using the *dlsym()* (page 144) call.

An executable object file's symbols are removed from the global symbol table when it is closed or orphaned. An executable object file cannot be unloaded if a symbol it provides is referenced by another object and that object is still resident. An executable object file that has no references to any of its symbols and was not explicitly loaded using the *dlopen()* (page 142) call is orphaned and automatically removed from the address space.

8.7.3.1 Base Image Symbols

The base image symbol table provides access to the function and data objects statically linked into the base image. Loaded executable object files can be directly linked to the code and data resident in the base image.

A statically linked RTEMS executable does not contain a symbol table, it has to be generated and either embedded into the executable or loaded as a specially created executable object file.

The base image symbol table is dependent on the contents of the base image and this is not known until it has been linked. This means the base image symbol table needs to be constructed after the base image executable has been linked and the list of global symbols is known.

The RTEMS Tools command **rtems-syms** (see *RTEMS Symbols* (page 201)) extracts the global and weak symbols from an RTEMS static executable file, creates a C file and compiles it creating a relocatable executable object file. This file can be linked with the static executable's object files and libraries to create a static executables with an embedded symbol table or the executable file can be loaded dynamically at run-time. The following needs to be observed:

1. The option **-e** or **--embedded** to **rtems-syms** creates an executable object file to be embedded in the base image and not providing either of these options creates a symbols executable object file that is loaded at run-time. The same executable object file cannot be used to embedded or load.
2. The target C compiler and machine options need to be provided to make sure the correct ABI for the target is used. See *Machine Flags and ABI* (page 132).

8.7.3.2 Embedded Symbols

An embedded symbol table is *embedded* within the base image executable file and loaded when the static executable is loaded into memory by the bootloader. The symbol table is automatically added to the link editor's global symbol table when the first executable object file is loaded.

The process to embed the symbol table requires linking the base image twice. The first link is to create a static executable that collects together the symbols to make the symbol table. The RTEMS Tools command **rtems-syms** extracts the global and weak symbols from the static executable ELF file, creates a C file and compiles it to create an executable object file. The base image is linked a second time and this time the symbol table executable object file is added to the list of object files.

Embedding the symbol table means the chances of the symbol table and base image not matching is low, however it also means the symbol table is always present in the kernel image when dynamic loading may be optional. A project's build system is made more complex as it needs to have extra steps to link a second time.

This example shows creating an embedded symbol table object file and linking it into the base image.

```

1 $ sparc-rtems5-gcc -mcpu=cypress foo.o -lrtemsbsp -lrtemscpu -o foo.pre
2 $ rtems-syms -e -C sparc-rtems5-gcc -c "-mcpu=cypress" -o foo-sym.o foo.pre
3 $ sparc-rtems5-gcc -mcpu=cypress foo.o foo-sym.o -lrtemsbsp -lrtemscpu -o foo.exe

```

The link command line steps in this example are not complete.

8.7.3.3 Loadable Symbols

A run-time loaded symbol table is the default for the command **rtems-syms**. The symbol table executable object file is packaged with the other files to be dynamically loaded at run-time and placed on the target's file system. It needs to be loaded before any other executable object file are loaded or unresolved symbols can occur that will not be resolved.

A run-time loaded symbol table does not consume any target resources until it is loaded. This is useful in a system that optionally needs to dynamically load code, for example as a development environment. The symbol table executable needs to exactly match the base image loading it or the behavior is unpredictable. No checks are made.

The example shows creating and loading a symbol table executable object file. First create the symbol table's executable object file:

```
1 $ sparc-rtems5-gcc -mcpu=cypress foo.o -lrtemsbasp -lrtemscpu -o foo.exe
2 $ rtems-syms -C sparc-rtems5-gcc -c "-mcpu=cypress" -o foo-sym.o foo.exe
```

The link command line steps in this example are not complete.

Load the symbol table:

```
1 #include <stdbool.h>
2 #include <stdio.h>
3 #include <dlfcn.h>
4
5 bool load (void)
6 {
7     void* handle = dlopen ("/foo-sym.o", RTLD_NOW | RTLD_GLOBAL);
8     if (handle == NULL)
9     {
10         printf ("failed to load the symbol table: %s\n", dlerror ());
11         return false;
12     }
13     return true;
14 }
```

8.7.4 Unresolved Symbols

The RTEMS link editor does not return an error when an executable object file is loaded with unresolved symbols. This allows dependent object files to be loaded. For example an executable object file `foo.o` contains the function `foo()` and that function calls `bar()` and an executable object file `bar.o` contains a function `bar()` that calls the function `foo()`. Either of these executable object files can be loaded first as long both are loaded before any symbols are accessed.

The link editor defers the resolution of unresolved symbols until the symbol is available in the global symbol table. Executing code or accessing data in a loaded executable object file with unresolved external symbols results in unpredictable behavior.

All unresolved symbols are checked after an executable object file has been loaded. If a symbol is found and resolved any relocations that reference the symbol are fixed. If valid library files have been configured the symbol table's of each library are searched and if the symbol is found the dependent executable object file is loaded. This process repeats until no more symbols can be resolved.

The `dlinfo()` call can be used to see if a loaded executable object file has any unresolved symbols:

```

1 #include <stdbool.h>
2 #include <stdio.h>
3 #include <dlfcn.h>
4
5 bool has_unresolved(void* handle)
6 {
7     int unresolved;
8     if (dlinfo (handle, RTLD_DI_UNRESOLVED, &unresolved) < 0)
9     {
10         printf ("dlinfo failed: %s\n", dlerror ());
11         return false;
12     }
13     return unresolved != 0;
14 }
```

The handle `RTLD_SELF` checks for any unresolved symbols in all resident object files:

```

1 if (has_unresolved(RTLD_SELF))
2     printf("system has unsolved symbols\n");
```

8.7.5 Libraries

The RTEMS link editor supports loading executable object files from libraries. Executable object files can be explicitly loaded from a library using a specific path to `dlopen()` (page 142) and treated the same as loading a stand alone executable object file. Libraries can be searched and an executable object file containing the search symbol can be loaded automatically as a dependent executable object file. A dependent executable object file loaded from a library with no symbol references to it's symbols is orphaned and automatically unloaded and removed from the address space.

A library is an archive format file created using the RTEMS architecture prefixed **ar** command. The RTEMS tool suite provides the **ar** program and system libraries such as `libc.a` and `libm.a` for each architecture and ABI. Libraries used by the RTEMS link editor for searching must contain a symbol table created by the **ranlib** program from the RTEMS tool suite.

Searching a library's symbol table and loading an executable object file containing the symbol is called *dependent loading*. Dependent loading provides a simple way to manage the dependencies when loading an executable object file. If code in an executable object file references functions or data objects that are part of a library and the symbols are not part of the base image those symbols will not resolve unless the library is on the target and available for searching and loading. Dependent loading from libraries on the target provides a simple and understandable way to manage the dependency issue between the base image, loaded code and the system libraries.

The RTEMS link editor checks for the configuration file `/etc/libdl.conf` on each call to `dlopen()` (page 142). If the file has changed since the last check it is loaded again and the contents processed. The file format is:

1. Comments start with the `#` character.
2. A line is a wildcard path of libraries to search for. The wildcard search uses the `fnmatch()` call. The `fnmatch()` function matches patterns according to the rules used by a shell.

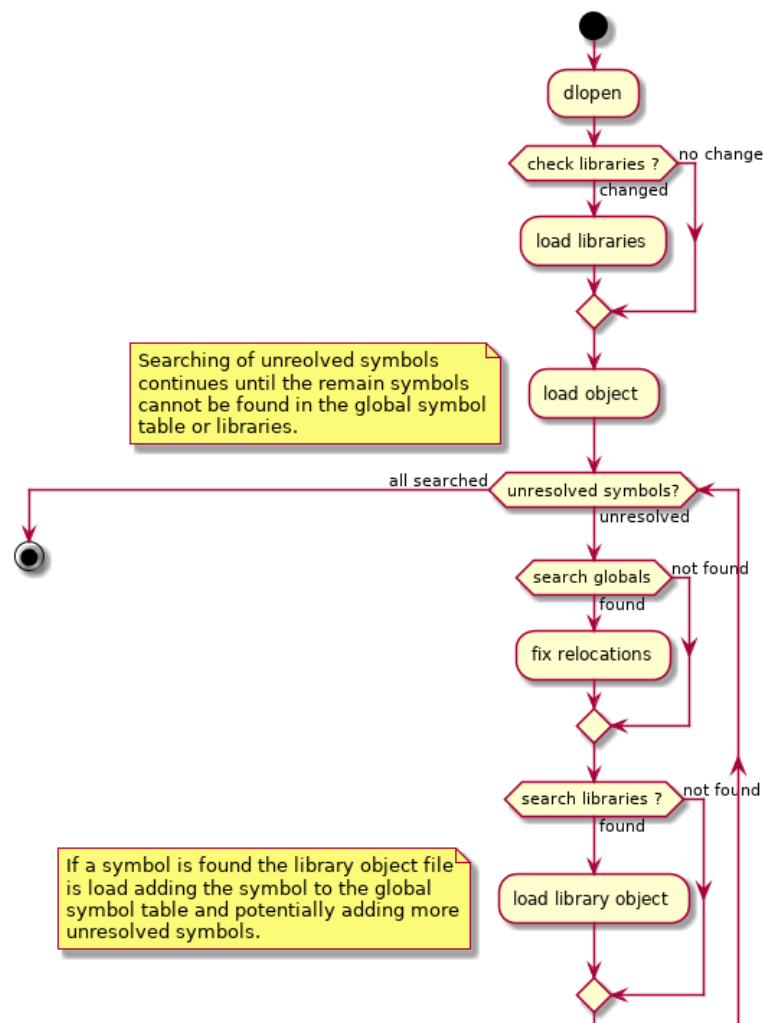


Fig. 8: Loading Executable Object Files

Files that match the search pattern are verified as a library and if a symbol table is found it is loaded and the symbols it contains made search-able.

A call to *dlopen()* (page 142) checks the global symbols table and any references to relocation symbols not found are *unresolved* and added to the unresolved symbol table. Once the executable object file is loaded the link editor attempts to resolve any unresolved symbols. The unresolved symbol resolver checks each unresolved symbol against the global symbol table and if not found the available library symbol tables are searched. If a symbol is found in a library the dependent executable object file is loaded. The process repeats until all unresolved symbols have been resolved and the remaining unresolved symbols are not in the global symbol table or any libraries. The loading of a library executable object file will resolve at least one symbol and it may add more unresolved symbols requiring further searching of the libraries.

A library of executable object files built by the RTEMS Tool suite can contain debug information and this should be stripped before loading on to the target. The tool suite's command **strip** can strip all the object files in a library with a single command.

```
1 $ sparc-rtems5-strip libc.a
```

8.7.6 Large Memory

The RTEMS link editor supports large memory relocations. Some architectures have instructions where the relative branch or jump offset from the instruction to the target address is limited. These instructions provide improved performance because less code generated compared to larger instructions which contain full address space references. The compact code helps lower cache pressure as well and providing improved performance for localized functions and loops. The compiler defaults to generating the smaller instruction and as the final address map not known when generating the code, linkers need to provide glue code to extend the small address range to the entire address space. This is called a trampoline. A trampoline is transparent to the execution of the code.

The link editor parses an executable object file's relocation records to determine the number of trampolines needed. Added to this value are all unresolved symbols present in an executable object file after it is loaded. There is a slot allocated even if the symbol ends up being within range as there is no way to determine a symbol's address until it is loaded and the range calculated.

The trampoline table is allocated a separate block of memory to the executable object file's text, data and constants memory. The trampoline parsing requires the executable object file's instructions (text) be in memory as the instructions are inspected by the architecture specific relocation support to determine an instruction's range. As a result the allocation for the trampoline table has to occur after the text memory has been allocated. Most instructions have relative offsets and the trampoline table is allocated at one end limiting the size of an object to half the maximum range.

Trampolines support is available for the ARM and PowerPC architectures. The SPARC and Intel x86 architectures do not need trampolines and MIPS needs support added.

8.7.7 Allocator

The RTEMS link editor supports custom allocators. A custom allocator lets you manage the memory used by the RTEMS link editor as it runs. Allocators could provide:

1. Support for the various types of memory that can be allocated allowing specialised target support for specific use cases.
2. Locking of read-only memory. The link editor unlocks read-only memory when it needs to write to it.
3. Separation of memory holding code and data from the heap.

The allocator can be hooked using the `rtems_rtl_alloc_hook` call before any calls to `dlopen()` (page 142) are made. The hook call returns the current allocate allowing the allocators to be chained.

The default allocator uses the heap.

The allocator tags specify the type of memory the allocator is handling. The tag used to allocate memory at an address must be used when making allocator calls. The `rtems_rtl_alloc_tags` are:

RTEMS_RTL_ALLOC_OBJECT

Allocate a generic object. The link editor uses this memory for data structures it uses to manage the linking process and resident executable object files.

RTEMS_RTL_ALLOC_SYMBOL

Allocate memory to hold symbol data.

RTEMS_RTL_ALLOC_EXTERNAL

Allocate memory for unresolved external symbols.

RTEMS_RTL_ALLOC_READ

Allocate memory for read-only data such as constants and exception tables.

RTEMS_RTL_ALLOC_READ_WRITE

Allocate memory for read-write data such as a initialised, uninitialized and common variables.

RTEMS_RTL_ALLOC_READ_EXEC

Allocate memory for code to be executed in. The address space is configure for read and execute.

The commands are used to control the action the allocator performs. The `rtems_rtl_alloc_cmd` are:

RTEMS_RTL_ALLOC_NEW

Allocate memory of the tag type. Returns NULL if the allocation fails.

RTEMS_RTL_ALLOC_DEL

Delete a previous allocation freeing the memory. The tag has to match address of the memory being deleted.

RTEMS_RTL_ALLOC_WR_ENABLE

Enable writes to a region of memory previously allocated. The tag has to match the address of the memory being write enabled. The link editor may call issue this command for memory that is already write enabled.

RTEMS_RTL_ALLOC_WR_DISABLE

Disable writes to a region of memory previously allocated. The tag has to match address of the memory being write disabled. The link editor may call issue this command for memory that is writable and not to be write disabled. The allocator need to manage this case.

The allocator handler is a single call to handle all allocator requests. The handler called on every allocation action made by the link editor. The type of the function you need is:

```

1 typedef void (*rtcms_rtl_allocator)(rtcms_rtl_alloc_cmd cmd,
2                                   rtcms_rtl_alloc_tag tag,
3                                   void**          address,
4                                   size_t          size);

```

The arguments are:

cmd

The command to action. See *rtcms_rtl_alloc_cmd* (page 151).

tag

The type of memory the command is for. The tag must match the address for commands other than RTEMS_RTL_ALLOC_OBJECT. See *rtcms_rtl_alloc_tags* (page 151).

address

Pointer to the address. This is set of the RTEMS_RTL_ALLOC_OBJECT command and read for the other commands. The tag must match the address for commands that read the address from the pointer.

size

The size of the memory to allocate. This is only valid for the RTEMS_RTL_ALLOC_OBJECT command.

The call to hook the allocator is:

```

1 rtcms_rtl_allocator rtcms_rtl_alloc_hook (rtcms_rtl_allocator handler);

```

The current allocator is returned. You can provide a full allocator or you can filter commands.

8.7.8 Languages

C is supported.

C++ is supported. Initializer functions are called when an object is loaded and finalizer functions are called before it is unloaded and removed. Static constructions are initializer functions and static destructors are finalizer functions.

C++ exceptions are handled across modules. The compiler generated exception tables present in an executable object file are registered with the architecture specific mechanism when loaded and deregistered when unloaded. An exception thrown in loaded code can be caught in the base image or another loaded module. If you are using C++ and exceptions it is recommended some exception code is added to the base image to place the architecture specific support in the base image.

8.7.9 Thread Local Storage

Thread local storage (TLS) is currently not supported by the RTEMS link editor. The RTEMS executive needs to have a special allocator added to manage dynamically allocating TLS variables in a thread.

If you need TLS support in dynamically loaded code please consider the RTEMS support options.

8.7.10 Architectures

The following architectures are supported:

- ARM
- Blackfin
- H8300
- Intel x86 (i386)
- LM32
- M68K
- MIPS
- Moxie
- PowerPC
- SPARC
- V850

8.7.10.1 ARM

The ARM relocation backend supports veneers which is trampolines.

The veneer implementation is a single instruction and a 32bit target address making the overhead 8 bytes for each veneer. The performance overhead is a single instruction.

8.7.10.2 PowerPC

The PowerPC relocation backend support trampolines and small data.

The trampoline is four instructions and uses register 12 which the PowerPC ABI reserves for scratch use. The implementation loads the counter register and branches to the address it contains. The trampoline size is 16 bytes. The performance overhead is four instructions.

The PowerPC relocation backend also supports small data. The sections of an executable object file are parsed and small data are tagged as needing architecture specific allocations. These sections are not allocated as part of the standard section allocation. Small data sections are allocated in the global small data region of memory. The size of this region is defined in the BSP's linker command file by setting `bsp_section_small_data_area_size` variable:

```
1 bsp_section_small_data_area_size = 65536;
```

The maximum size of the small data region is 65536 bytes. It is recommended code built for loading uses the same settings for small base as the base image.

8.8 Device Tree

A Device Tree is a data structure that is used to describe properties of non-discoverable hardware instead of hardcoding them in the kernel. The device tree data is generally stored in a *.dts* or a Device Tree Source (DTS) file. This file is then compiled into a binary format called Device Tree Blob (DTB) with *.dtb* extension. RTEMS preferably uses a DTB built from the FreeBSD source tree matching the *freebsd-org* HEAD commit hash in *libBSD*.

8.8.1 Building the DTB

A single DTB file can be built using the *dtc* tool in *libfdt* using the following command:

```
1 dtc -@ -I dts -O dtb -o my-devicetree.dtb my-devicetree.dts
```

For building the DTB from the FreeBSD source, the *make_dtb.sh* script from *freebsd/sys/tools/fdt* must be used as most of the DTS files in FreeBSD have included *.dtsi* files from their source tree. An example is given below as a reference for how to build the device tree from the FreeBSD source.

NOTE: The following example uses FreeBSD master branch from github mirror as an example. It is advised to always use the source from the commit matching the freebsd-org HEAD in libBSD.

```
1  #We're using the script from freebsd/sys/tools/make_dtb.sh
2  #Target device: Beaglebone Black.
3  #Architecture: Arm.
4  #DTS source name: am335x-boneblack.dts
5
6  #The make_dtb.sh script uses environment variable MACHINE
7  export MACHINE='arm'
8
9  SCRIPT_DIR=$HOME/freebsd/sys/tools/fdt
10
11 #The arguments to the script are
12 # $1 -> Build Tree (This is the path to freebsd/sys/ directory)
13 # $2 -> DTS source file
14 # $3 -> output path of the DTB file
15
16 ${SCRIPT_DIR}/make_dtb.sh ${SCRIPT_DIR}/../../../../ \
17 ${SCRIPT_DIR}/../../../../gnu/dts/arm/am335x-boneblack.dts \
18 $(pwd)
```

8.8.2 Using Device Tree Overlay

Device tree overlay is used either to add properties or devices to the existing device tree. Adding any property to DTS using an overlay will override the current values in the DTB. The Overlays enable us to modify the device tree using a small maintainable plugin without having to edit the whole Base Tree.

There are two ways of applying an overlay on top of the built DTB.

1. Use *fdtoverlay* from *libfdt*
2. Add the overlay in the root partition of the SD card and apply it using U-Boot

The `fdtoverlay` command can be used as follows:

```
1 fdtoverlay -i my-base-tree.dtb -o output-tree.dtb my-overlay.dtbo
```

To apply it from U-Boot during system initialization we have to add the device tree overlay file in the root directory of the SD card and use U-Boot commands to apply the overlay.

Below is given the series of U-Boot commands that can be used to apply the overlay, given that the overlay blob (.dtbo) file is already in the card.

```
1 fatload mmc 0:1 0x80800000 rtems-app.img
2 fatload mmc 0:1 0x88000000 my-base-tree.dtb
3 fdt addr 0x88000000
4 fatload mmc 0:1 0x88100000 my-overlay.dtbo
5 fdt resize 0x1000
6 fdt apply 0x88100000
7 bootm 0x80800000-0x88000000
```


TESTING

RTEMS developers run test executables when adding new features or testing a bug fix. All tests are run to make sure changes do not introduce regressions. Users can run the RTEMS tests to be certain the build of the kernel they have is functioning.

The section describes using and configuring the RTEMS Tester and RTEMS Run tools, the types of laboratory set ups supported and how to add your BSP to the framework. The tools command line interfaces are detailed in *RTEMS Tester and Run* (page 216).

An RTEMS Test is an RTEMS executable where the application code is a test. Tests in RTEMS print banners to the console to indicate the configuration of the test and if it has start and finished.

The RTEMS Tools Project provides the RTEMS Tester and RTEMS Run tools. The RTEMS Tester command is `rtems-test` and the RTEMS Run command is `rtems-run`. These commands manage the complexity of running embedded executables. The commands provide a consistent command line interface to a testing framework that supports the various run time and testing scenarios we encounter such as simulators, GDB and executing directly on target hardware.

The RTEMS kernel code contains an extensive set of tests to exercise and test the RTEMS kernel. The tests check functionality, provide coverage testing and make sure the kernel is operating as intended on your target system. The testsuite has support to make creating a test simple and uniform.

The tests are built by adding `--enable-tests` to the RTEMS build configuration command line. There are over 600 tests and building them does extend the RTEMS kernel's build time and use more disk space but it worth building and running them. The RTEMS test executables have the `.exe` file extension.

9.1 Test Banners

All test output banners or strings are embedded in each test and the test outputs the banners to the BSP's console as it executes. The RTEMS Tester captures the BSP's console and uses this information to manage the state of the executing test. The banner strings are:

***** BEGIN TEST <name> *****

The test has loaded, RTEMS has initialized and the test specific code is about to start executing. The <name> field is the name of the test. The test name is internal to the test and may not match the name of the executable. The test name is informative and not used by the RTEMS Tester.

***** END TEST <name> *****

The test has finished without error and has passed. The <name> field is the name of the test. See the *Test Begin Banner* (page 158) for details about the name.

***** TEST VERSION: <version>**

The test prints the RTEMS version return by the RTEMS Version API as <version>. All tests must match the first test's version or the Wrong Version error count is incremented.

***** TEST STATE: <state>**

The test is tagged in the RTEMS sources with a special <state> for this BSP. See *Test States* (page 158) for the list of possible states. The state banner lets the RTEMS Tester categorize and manage the test. For example a user input test typically needing user interaction may never complete producing an *invalid* test result. A user input test is terminated to avoid extended delays in a long test run.

***** TEST BUILD: <build>**

The test prints the RTEMS build as a space separated series of labels as <build>. The build labels are created from the configuration settings in the Super Score header file `rtems/score/cputops.h`. All tests must match the first test's build or the Wrong Build error count is incremented.

***** TEST TOOLS: <version>**

The test prints the RTEMS tools version returned the GGC internal macro `_VERSION_` as <version>. All tests must match the first test's tools version string or the Wrong Tools error count is incremented.

9.2 Test Controls

The tests in the RTEMS kernel testsuite can be configured for each BSP. The expected state of the test can be set as well as any configuration parameters a test needs to run on a BSP.

The test states are:

passed

The test start and end banners have been sent to the console.

failure

The test start banner has been sent to the console and no end banner has been seen when a target restart is detected.

excepted-fail

The test is tagged as `excepted-fail` in the RTEMS sources for this BSP and outputs the banner `*** TEST STATE: EXPECTED_FAIL`. The test is known not to pass on this BSP. The RTEMS Tester will let the test run as far as it can and if the test passes it is recorded as a pass in the test results otherwise it is recorded as *excepted-fail*.

indeterminate

The test is tagged as `indeterminate` in the RTEMS sources for this BSP and outputs the banner `*** TEST STATE: INDETERMINATE`. The test may or may not pass so the result is not able to be determined. The RTEMS Tester will let the test run as far as it can and record the result as `indeterminate`.

user-input

The test is tagged as `user-input` in the RTEMS sources and outputs the banner `*** TEST STATE: USER_INPUT`. The RTEMS Tester will reset the target if the target's configuration provides a target reset command.

benchmark

The test is tagged as `benchmark` in the RTEMS sources and outputs the banner `*** TEST STATE: BENCHMARK`. Benchmarks can take a while to run and performance is not regression tested in RTEMS. The RTEMS Tester will reset the target if the target's configuration provides a target reset command.

timeout

The test start banner has been sent to the console and no end banner is seen within the *timeout* period and the target has not restart. A default *timeout* can be set in a target configuration, a user configuration or provide on the RTEMS Tester's command line using the `--timeout` option.

invalid

The test did not output a start banner and the RTEMS Tester has detected the target has restarted. This means the executable did not load correctly, the RTEMS kernel did not initialize or the RTEMS kernel configuration failed for this BSP.

9.2.1 Expected Test States

A test's expected state is set in the RTEMS kernel's testsuite. The default for a tested is to pass. If a test is known to fail it can have its state set to `excepted-fail`. Setting tests that are known to fail to `excepted-fail` lets everyone know a failure is not to be countered and consider a regression.

Expected test states are listed in test configuration files

9.2.2 Test Configuration

Tests can be configured for each BSP using test configuration files. These files have the file extension `.tcfg`. The testsuite supports global test configurations in the `testsuite/testdata` directory. Global test states are applied to all BSPs. BSPs can provide a test configuration that applies to just that BSP and these files can include subsets of test configurations.

The configuration supports:

1. Including test configuration files to allow sharing of common configurations.
2. Excluding tests from being built that do not build for a BSP.
3. Setting the test state if it is not passed.
4. Specifying a BSP specific build configuration for a test.

The test configuration file format is:

```
1 state: test
```

where the state is state of the test and test is a comma separated list of tests the state applies too. The configuration format is:

```
1 flags: test: configuration
```

where flags is the type of flags being set, the test is a comma separate list of regular expresions that match the test the configuration is applied too and the configuration is the string of flags.

The state is one of:

include

The test list is the name of a test configuration file to include

exclude

The tests listed are not build. This can happen if a BSP cannot support a test. For example it does not have enough memory.

expected-fail

The tests listed are set to expected fail. The test will fail on the BSP being built.

user-input

The tests listed require user input to run and are not supported by automatic testers.

indeterminate

The tests listed may pass or may not, the result is not reliable.

benchmark

The tests listed are benchmarks. Benchmarks are flagged and not left to run to completion because they may take too long.

Specialized filtering using regular expressions is supported using:

rexclude

The test matching the regular expression are excluded.

rinclude

The tests matching the regular expression are included.

By default all tests are included, specific excluded tests using the exclude state are excluded and cannot be included again. If a test matching a rexclude regular it is excluded unless it is included using a rinclude regular expression. For example to build only the hello sample application you can:

```
1 rexclude .*
2 rinclude hello
```

Test configuration flags can be applied to a range of tests using flags. Currently only cflags is support. Tests need to support the configuration for it to work. For example to configure a test:

```
1 cflags: tm.*: -DTEST_CONFIG=42
2 cflags: sp0[456]: -DA_SET_OF_TESTS=1
```

Flags setting are joined together and passed to the compiler's command line. For example:

```
1 cflags: tm.*: -DTEST_CONFIG=42
2 cflags: tm03: -DTEST_TM03_CONFIG=1
```

would result in the command line to the test tm03 being:

```
1 -DTEST_CONFIG=42 -DTEST_TM03_CONFIG=1
```

Specific flags can be set for one test in a group. For example to set a configuration for all timer tests and a special configuraiton for one test:

```
1 cflags: (?!tm02)tm.*: -DTEST_CONFIG=one
2 cflags: tm02: -DTEST_CONFIG=two
```

9.3 Test Builds

The test reports the build of RTEMS being tested. The build are:

default

The build is the default. No RTEMS configure options have been used.

posix

The build includes the POSIX API. The RTEMS configure option `--enable-posix` has been used. The `cpuopts.h` define `RTEMS_POSIX` has defined and it true.

smp

The build is an SMP kernel. The RTEMS configure option `--enable-smp` has been used. The `cpuopts.h` define `RTEMS_SMP` has defined and it true.

mp

The build is an MP kernel. The RTEMS configure option `--enable-multiprocessing` has been used. The `cpuopts.h` define `RTEMS_MULTIPROCESSING` has defined and it true.

paravirt

The build is a paravirtualization kernel. The `cpuopts.h` define `RTEMS_PARAVIRT` has defined and it true.

debug

The build includes kernel debugging support. The RTEMS configure option `--enable-debug` has been used. The `cpuopts.h` define `RTEMS_DEBUG` has defined and it true.

profiling

The build include profiling support. The RTEMS configure option `--enable-profiling` has been used. The `cpuopts.h` define `RTEMS_PROFILING` has defined and it true.

9.4 Tester Configuration

The RTEMS Tester and RTEMS Run are controlled by configuration data and scripts. The user specifies a BSP on the command line using the `--rtems-bsp` option as well as optionally specifying a user configuration file using `--user-config`.

The Figure *RTEMS Tester and Run Configuration Files* (page 163) shows the various sources of configuration data and their format. The ini files are the standard INI format, the mc are the internal RTEMS Toolkit's Macro format, and cfg is the RTEMS Toolkit's Configuration script format, the same format used by the RTEMS Source Builder.

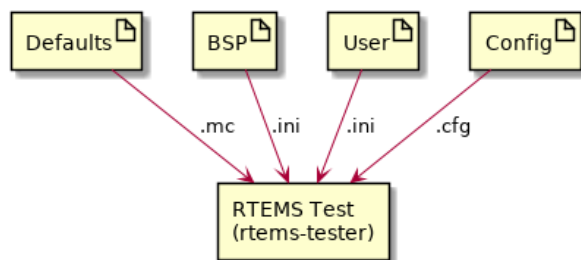


Fig. 1: RTEMS Tester and Run Configuration Files

Configuration data is held in a macro database keyed on the macro name. Macros can be expanded in configuration scripts using the syntax `%{name}`. The macro database is layered using maps. The defaults and values created when a configure script runs live in the global map. Values read from the BSP and User INI configuration files are loaded into maps based on the BSP name. This lets a single User configuration file contain specialized configuration values for a number of BSPs and the tester and run commands select the values based on the selected BSP. Macros are expanded using the BSP map first giving those values the highest priority. User defined values are loaded after the BSP configuration values overwriting them letting a user speckle a BSP's default configuration for their local needs.

Figure *RTEMS Tester and Run Configuration Load and Execute Sequence* (page 164) shows the configuration loading and script execution order.

9.4.1 Defaults

The RTEMS Tester and RTEMS Run are primed using defaults from the file `rtems/testing/testing.mc`. All default settings can be overridden in a BSP or User configuration file.

9.4.2 BSP and User Configuration

The BSP and User configuration files are INI format files. The BSP configuration file has to have an INI section that is the name of the BSP passed on the command line. The section has the following mandatory values:

bsp

The name of the BSP. The BSP name is used to create a macro map to hold the BSP's configuration data. Typically this is the same as the BSP name used on the command line.

arch

The name of the BSP architecture. This is needed for the GDB configuration scripts where the

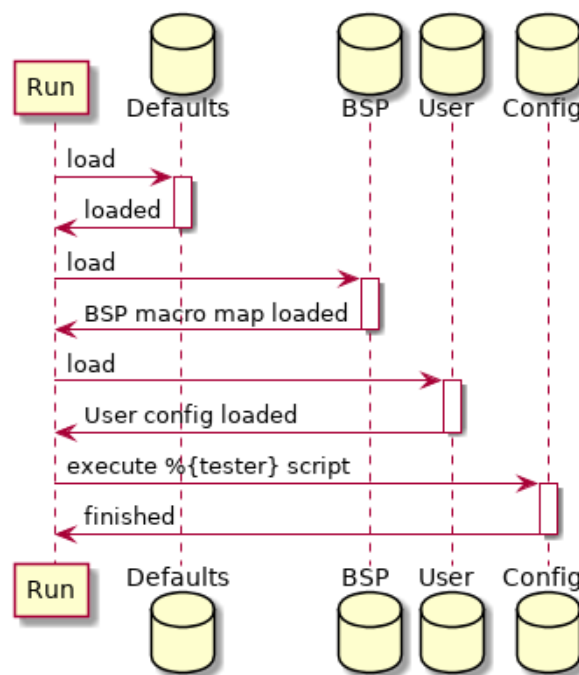


Fig. 2: RTEMS Tester and Run Configuration Load and Execute Sequence

architecture specific GDB needs to run. It is mandatory so the *arch/bsp* standard RTEMS BSP string can be used.

tester

The tester or run configuration script. This is the name of the configuration script the RTEMS Tester or RTEMS Run executes as a back end. The tester value is typically of the form `%{_rtscripts}/<script>` where `<script>` is name of the back end script to be run.

Target commands support expansion of specific tags to provide a convenient way for users to customize a local test environment. The parameters expanded are:

@ARCH@

The BSP architecture.

@BSP@

The BSP's name set by the bsp value.

@EXE@

The executable name as an absolute path

@FEXE@

The filtered executable if a `target_exe_filter` is provided else the executable's file name.

The following are optional and depend on the back end being used and the local target hardware set up:

jobs

The jobs value sets the number of jobs that can be run at once. This setting only effects the RTEMS Tester. The tester can run up to the jobs value of tests concurrently. If the tester back end is a simulator running a job on each available core lowers the total test time. Overloading a machine with too many simulators running in parallel can slow down each simulation and test timeouts may be recorded.

bsp_tty_dev

The BSP's tty device. This can be a real device on the host machine the executable is being run from or it can be a telnet server and port defined using the stand host format. See *Consoles* (page 170) for details.

target_pretest_command

The pre-test command is a host shell command that is called before each test runs. It can be used to construct a suitable environment or image needed by a simulator or target. The RTEMS executable being run is provided as an argument and the bootloader specific format is the output.

target_posttest_command

The post-test command is a host shell command that is called after each test has finished. It can be used to destroy any environment or image created by the pre-test command.

target_exe_filter

The target executable filter transforms the executable name into a filtered executable name. This filter lets the tester or run command track the name of any generated file a pre-test command may generate. The syntax is a simplified sed regular expression. The first character is a delimiter and there must be 2 sections therefore 3 delimiters. The first section is a Python regular expression and the second section is plain text that replaces anywhere the regular expression matches. For example `/\.exe/.exe.img/` will search for `.exe` in the executable name and replace it with `.exe.img`. Note, there is no need to escape the text in the second part, it is just plain text.

test_restarts

The number of restarts before the test is considered invalid. Currently not used.

target_reset_regex

The target reset regular expression. This is a [Python regular expression](#) used to filter the console input. If a match is made something has happened during the boot process that requires a reset. The `target_reset_command` is issued to perform the reset. Typically this field looks for boot loader error messages that indicate the boot process as failed.

target_start_regex

The target start regular expression. This is a Python regular expression to filter the console input to asynchronously detect if a target has reset. If a board crashes running a test or at any point reset this filter detects the restart and ends the test with a suitable result.

target_on_command

The target on command is a host shell command that is called before the first test. This command powers on a target. Targets should be left powered off when not running tests or the target may request TFTP downloads that are for another target interfering with those test results. We recommend you implement this command as a target off command, a pause, then a target on command.

target_off_command

The target off command is a host shell command that is called after the last test powering off the target.

target_reset_command

The target reset command is a host shell command that is called when the target needs to be reset. This command can power cycle the target or toggle a reset signal connected to the

target. If you are power cycling a target make sure you have a suitable pause to let the target completely power down.

9.4.3 Configuration Scripts

Configuration scripts are provided for each supported RTEMS Tester and RTEMS Run back end and console management. The scripts are in the standard RTEMS Toolkit Configuration Script format. Please refer to the RTEMS Source Builder documentation for the basic scripting syntax and usage.

The RTEMS Tester and RTEMS Run specializes the standard configuration syntax providing a directive for the console and each supported back end. The supported directives are:

- %console
- %execute
- %gdb
- %tftp

9.4.3.1 Console

The %console configures the console used to access the target's console. The console can be a process's stdout, a termios tty on Unix and MacOS and Telnet on all hosts. The directive accepts:

stdio

The standard output stream from the executing processing.

tty <dev> <settings>

The name of the tty to open and use. The tty device or <dev> can be a *termio* device and the <settings> are standard termios values.

The Python termios document provides details of the settings that can be controlled. The settings are a single string where prefix the value with ~ negates the setting. Setting are:

- B115200 (an example buadrate)
- BRKINT
- IGNBRK
- IGNCR
- ICANON
- ISIG
- IEXTEN
- ECHO
- CLOCAL
- CRTSCTS
- VMIN=<value>
- VTIME=<value>

A example in a configuration script is:

```
1 %define bsp_tty_dev      /dev/ttyUSB2
2 %define bsp_tty_settings B115200,~BRKINT,IGNBRK,IGNCR,~ICANON,~ISIG,~IEXTEN,~ECHO,CLOCAL,~
  ↪ CRTSCTS,VMIN=1,VTIME=2
```

A example BSP or User configuration file is:

```
1 [bsp-special]
2 bsp                = example-bsp
3 bsp_tty_dev        = /dev/ttyUSB2
4 bsp_tty_settings   = B115200,~BRKINT,IGNBRK,IGNCR,~ICANON,~ISIG,~IEXTEN,~ECHO,CLOCAL,~
  ↪ CRTSCTS,VMIN=1,VTIME=2
```

The console directive is managed in the `%{_rtscripts}/console.cfg` configuration script. If the `%{console_stdio}` is defined the console will be `stdio` else the console will be the BSP console or `%{bsp_tty_dev}`.

Telnet can be combined with the `ser2net` daemon to remotely access a target's physical serial UART interface.

9.4.3.2 Execute

The `%execute` directive executes a command for each rest. The execute forks the command and arguments supplied to the execute directive and captures the `stdout` stream as the console. If the console directive is set to `stdout` the sub-processes `stdout` stream is used as the console.

The RTEMS Tester will run parallel tests as jobs.

An example is:

```
1 %execute %{run_cmd} %{run_opts} %{test_executable} %{test_executable_opts}
```

9.4.3.3 GDB

The `%gdb` directive executes GDB in the machine interface mode give the RTEMS Tester and RTEMS Run commands control. The console is taken from GDB if it is `stdout`.

The RTEMS Tester will run parallel tests as jobs.

An example is:

```
1 %gdb %{gdb_cmd} %{test_executable} %{gdb_script}
```

9.4.3.4 TFTP

The `%tftp` directive starts a TFTP session on a specified port sending the test executable to the target over a networking using the TFTP protocol.

The RTEMS Tester will run only one test at a time. There is just one physical board running the test.

An example is:

```
1 %tftp %{test_executable} %{tftp_port}
```

9.5 Coverage Analysis

RTEMS is used in many critical systems. It is important that the RTEMS Project ensure that the RTEMS product is tested as thoroughly as possible. With this goal in mind, the RTEMS test suite was expanded with the goal that 100% of the RTEMS executive is tested.

RTEMS-TESTER takes the following arguments to produce coverage reports:

–coverage :

When the coverage option is enabled the tester produces coverage reports for all the symbols in cpukit. To generate a coverage report for a specific symbol-set (e.g.: score) the symbol-set is passed as an argument to the option, e.g.: –coverage=score.

–no-clean :

Tells the script not to delete the .cov trace files generated while running the coverage. These trace files are used for debugging purposes and will not be needed for a normal user.

For example: To generate a coverage report of hello.exe for leon3 on SIS, the following command is used:

```

1 rtems-test \
2 --rtems-tools=$HOME/development/rtems/5 \
3 --log=coverage_analysis.log \
4 --no-clean \
5 --coverage \
6 --rtems-bsp=leon3-sis-cov \
7 $HOME/development/rtems/kernel/leon3/sparc-rtems5/c/leon3/testsuites/samples/hello.exe

```

The command will create the coverage report in the following tree structure:

```

1 — coverage_analysis.log
2 — leon3-sis-coverage
3   └─ score
4       └─ annotated.html
5       └─ annotated.txt
6       └─ branch.html
7       └─ branch.txt
8       └─ covoar.css
9       └─ ExplanationsNotFound.txt
10      └─ index.html
11      └─ no_range_uncovered.html
12      └─ no_range_uncovered.txt
13      └─ NotReferenced.html
14      └─ sizes.html
15      └─ sizes.txt
16      └─ summary.txt
17      └─ symbolSummary.html
18      └─ symbolSummary.txt
19      └─ table.js
20      └─ uncovered.html
21      └─ uncovered.txt
22 — leon3-sis-report.html

```

The html on top of the directory, i.e., leon3-sis-report.html is the top level navigation for the coverage analysis report and will let the user browse through all the generated reports from different subsystems.

9.6 Consoles

The RTEMS Tester uses the target's console output to determine the state of a test. Console interfaces vary depending on the testing mode, the BSP, and the target hardware.

Consoles for simulator work best if mapped to the simulator's stdout interface. The RTEMS Tester can capture and process the stdout data from a simulator while it is running.

Target hardware console interfaces can vary. The most universal and stable interface target hardware is a UART interface. There are a number of physical interfaces for UART data these days. They are:

1. RS232
2. TTL
3. USB

RS232 is still present on a number of targets. The best solution is to use a RS232 to USB pod and convert the port to USB.

TTL is common on a number of boards where cost is important. A console interface is typically a development tool and removing the extra devices need to convert the signal to RS232 or directly to USB is not needed on production builds of the target. There is a standard header pin out for TTL UART consoles and you can purchase low cost cables with the header and a built in UART to USB converter. The cables come in different voltage levels so make sure you check and use the correct voltage level.

The USB interface on a target is typically a slave or OTG interface and all you need is a standard USB cable.

We recommend a low cost and low power device to be a terminal server. A Raspberry Pi or similar low cost computer running Linux can be set up quickly and with a powered USB hub and can support a number of USB UART ports. A USB hub with a high power port is recommended that can supply the Raspberry Pi.

The open source daemon `ser2net` is easy to configure to map the USB UART ports to the Telnet protocol. There is no need for security because a typical test environment is part of a lab network that should be partitioned off from an engineering or corporate network and not directly connected to the internet.

A test set up like this lets you place a terminal server close to your target hardware providing you with the flexibility to select where you run the RTEMS Tester. It could be your desktop or an expensive fast host machine in a server rack. None of this equipment needs to directly interface to the target hardware.

The RTEMS Tester directly supports the telnet protocol as a console and can interface to the `ser1net` server. The telnet console will poll the server waiting for the remote port to connect. If the terminal server `ser2net` does not have a `tty` device it will not listen on the port assigned to that `tty`. A USB `tty` can come and go depending on the power state of the hardware and the target hardware's design and this can cause timing issues if the target hardware is power cycled as part of a reset process.

9.7 Simulation

Simulation is a important regression and development tool for RTEMS. Developers use simulation to work on core parts of RTEMS as it provides excellent debugging supporting. Simulation run via the RTEMS Tester allows a test to run on each core of your testing host machine lower the time to run all tests.

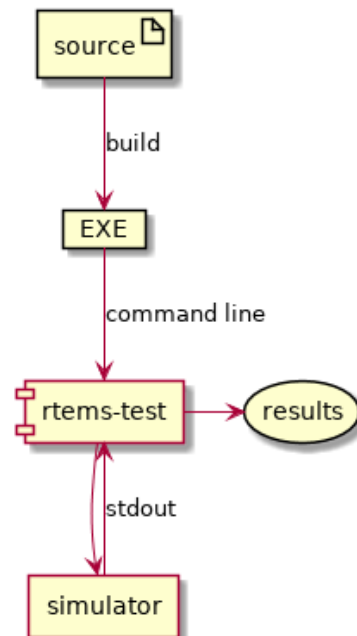


Fig. 3: RTEMS Tester Simulation

The *RTEMS Tester Simulation* (page 171) figure shows the structure of RTEMS Testing using simulation. The executables are built and the `rtems-test` command is run from the top of the build directory. The RTEMS Tester executes the BSP specific simulator for each test capturing the output

9.8 GDB and JTAG

GDB with JTAG provides a low level way to runs tests on hardware with limited resources. The RTEMS Tester runs and controls an instance of GDB per test and GDB connects via the GDB remote protocol to a GDB server that interfaces to the JTAG port of a target.

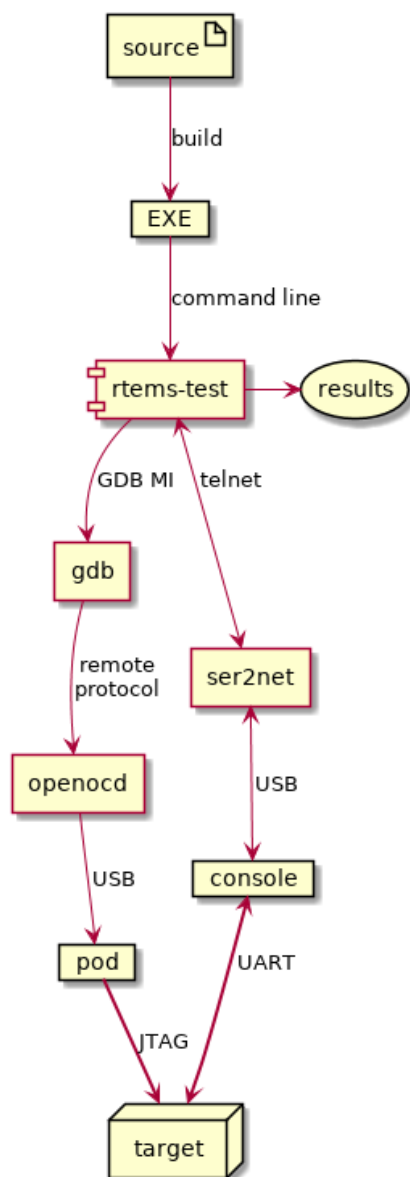


Fig. 4: RTEMS Tester using GDB and JTAG

The Figure *RTEMS Tester using GDB and JTAG* (page 172) shows the structure of RTEMS Testing using GDB and JTAG. The executables are built and the `rtems-test` command is run from the top of the build directory. The RTEMS Tester executes the BSP architecture's GDB and expects the user to provide a `gdb-script` to connect t the JTAG GDB server.

9.9 TFTP and U-Boot

TFTP and U-Boot provides a simple way to test RTEMS on a network capable target. The RTEMS Tester starts a TFTP server for each test and the target's boot monitor, in this case U-Boot request a file, any file, which the TFTP server supplies. U-Boot loads the executable and boots it using a standard U-Boot script.

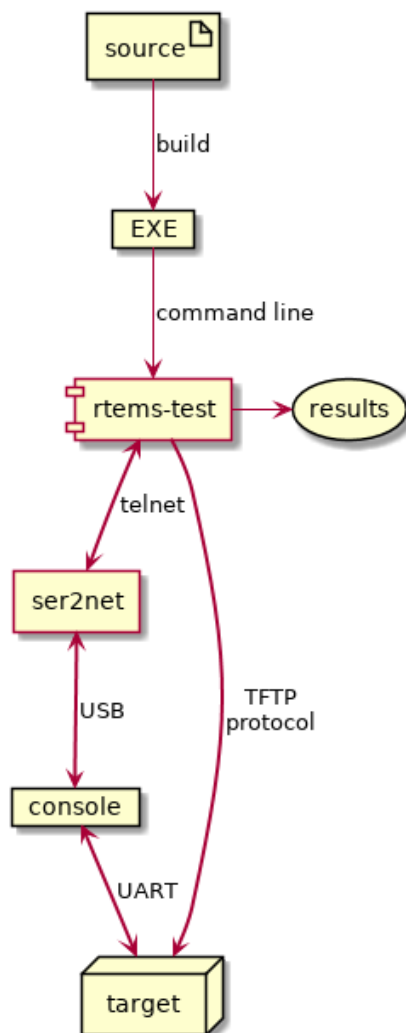


Fig. 5: RTEMS Tester using TFTP and U-Boot.

The Figure *RTEMS Tester using TFTP and U-Boot*. (page 173) figure shows the structure and control flow of the RTEMS Tester using TFTP and U-boot. The executables are built and the `rtems-test` command is run from the top of the build directory.

This test mode can only support a single test job running at once. You cannot add more test target hardware and run the tests in parallel.

9.9.1 Target Hardware

The RTEMS Tester TFTP and U-Boot method of testing requires:

1. A target with network interface.

2. U-Boot, iPXE or similar boot loader with network driver support for your target hardware and support for the TFTP protocol.
3. Network power of IO switch.
4. Network DHCP server.
5. Console interface cable that matches your target's console UART interface.
6. Telnet terminal server. See *Consoles* (page 170).

The network power or IO switch is a device that can control power or an IO pin over a network connection using a script-able protocol such as Telnet or curl. This device can be used with the target control commands.

9.9.1.1 U-Boot Set Up

Obtain a working image of the U-Boot boot loader for your target. We suggest you follow the instructions for you target.

Configure U-Boot to network boot using the TFTP protocol. This is U-Boot script for a Zedboard:

```
1 loadaddr=0x02000000
2 uenvcmd=echo Booting RTEMS Zed from net; set autoload no; dhcp; set serverip 10.10.5.2;
  ↪ tftpboot zed/rtems.img; bootm; reset;
```

The load address variable `loadaddr` is specific to the Zedboard and can be found in the various examples scripts on the internet. The script then sets U-Boot environment variable `autoload` to `no` causing DHCP to only request a DHCP lease from the DHCP server. The script sets the `serverip` to the host that will be running the RTEMS Tester then issues a TFTP request. The file name can be anything because the RTEMS Tester ignores it sending the executable image under test. Finally the script boots the download executable and if that fails the catch all `reset` resets the board and starts the boot process over.

Test the target boots and U-Boot runs and obtains a valid DHCP lease. Manually connect the console's telnet port.

9.9.2 BSP Configuration

The BSP's configuration file must contain the standard fields:

- `bsp`
- `arch`
- `jobs` - Must be set to 1.
- `tester` - Set to `%{_rtscripts}/tftp.cfg`

For example the Zedboard's configuration is:

```
1 [xilinx_zynq_zedboard]
2 bsp    = xilinx_zynq_zedboard
3 arch   = arm
4 jobs   = 1
5 tester = %{_rtscripts}/tftp.cfg
```

The TFTP configuration supports the following field's:

bsp_tty_dev

The target's tty console. For telnet this is a host and port pair written in the standard networking format, for example `server:12345`.

test_restarts

The number of restarts before the test is considered invalid.

target_reset_regex

The target reset regular expression. This is a **Python regular expression** used to filter the console input. If a match is made something has happened during the boot process that requires a reset. The `target_reset_command` is issued to perform the reset. This field is typically looks for boot loader error messages that indicate the boot process as failed.

target_start_regex

The target start regular expression. This also a Python regular expression to filter the console input to detect if a target has reset. If a board crashes running a test or at any point in time and reset this filter detects this as happened and end the test with a suitable result.

target_on_command

The target on command is a host shell command that is called before the first test. This command powers on a target. Targets should be left powered off when not running tests or the target may request TFTP downloads that are for another target interfering with those test results. We recommend you implement this command as a target off command, a pause, then a target on command.

target_off_command

The target off command is a host shell command that is called after the last test powering off the target.

target_reset_command

The target reset command is a host shell command that is called when the target needs to be reset. This command can power cycle the target or toggle a reset signal connected to the target. If you are power cycling a target make sure you have a suitable pause to let the target completely power down.

target_pretest_command

The target pretest command is a host shell comment that is called before the test is run

The commands in the listed fields can include parameters that are substituted. The parameters are:

@ARCH@

The BSP architecture

@BSP@

The BSP's name

@EXE@

The executable name.

@FEXE@

The

. The

@ARCH is the

substituted

Some of these field are normally provided by a user's configuration. To do this use:

```
1 requires = bsp_tty_dev, target_on_command, target_off_command, target_reset_command
```

The requires value requires the user provide these settings in their configuration file.

The Zedboard's configuration file is:

```
1 [xilinx_zynq_zedboard]
2 bsp                = xilinx_zynq_zedboard
3 arch               = arm
4 jobs               = 1
5 tester             = %[_rtscripts]/tftp.cfg
6 test_restarts      = 3
7 target_reset_regex = ^No ethernet found.*|^BOOTP broadcast 6.*|^.+complete\..+ TIMEOUT.*
8 target_start_regex = ^U-Boot SPL .*
9 requires           = target_on_command, target_off_command, target_reset_command, bsp_tty_
  ↪ dev
```

The target_start_regex searches for U-Boot's first console message. This indicate the board can restarted.

The target_reset_regex checks if no ethernet interface is found. This can happen if U-Boot cannot detect the PHY device. It also checks if too many DHCP requests happen and finally a check is made for any timeouts reported by U-Boot.

An example of a user configuration for the Zedboard is:

```
1 [xilinx_zynq_zedboard]
2 bsp_tty_dev        = selserv:12345
3 target_pretest_command = zynq-mking @EXE@
4 target_exe_filter   = /\.exe/.exe.img/
5 target_on_command    = power-ctl toggle-on 1 4
6 target_off_command   = power-ctl off 1
7 target_reset_command = power-ctl toggle-on 1 3
```

9.9.3 TFTP Sequences

Running a large number of tests on real hardware exposes a range of issues and RTEMS Tester is designed to be tolerant of failures in booting or loading that can happen, for example a hardware design. These sequence diagrams document some of the sequences that can occur when errors happen.

The simplest sequence is running a test. The target is powered on, the test is loaded and executed and a pass or fail is determined:

The target start filter triggers if a start condition is detected. This can happen if the board crashes or resets with no output. If this happens repeatedly the test result is invalid:

The reset filter triggers if an error condition is found such as the bootloader not being able to load the test executable. If the filter triggers the target_reset_command is run:

If the RTEMS Tester does not detect a test has started it can restart the test by resetting the target. The reset command can toggle an IO pin connected to reset, request a JTAG pod issue a reset or turn the power off and on:

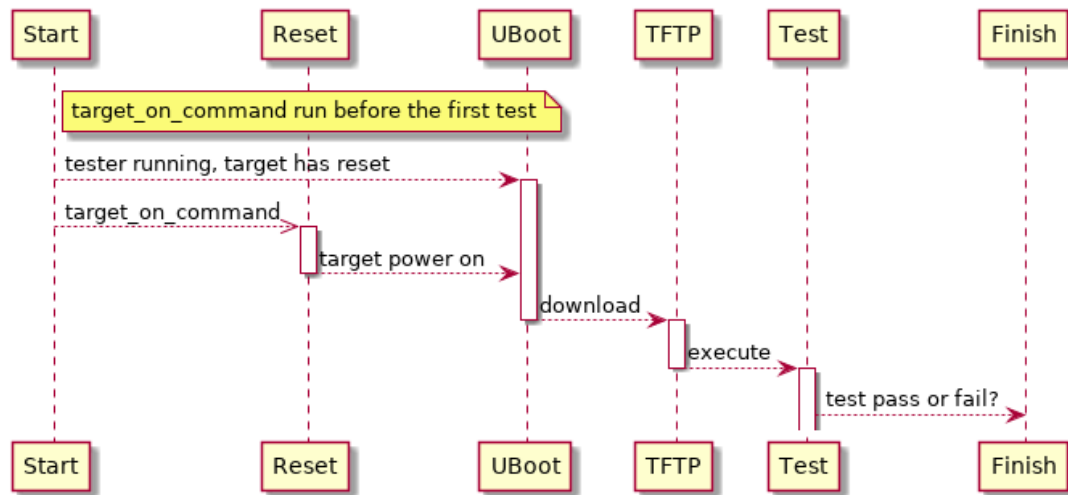


Fig. 6: Test Pass and Fail Sequences

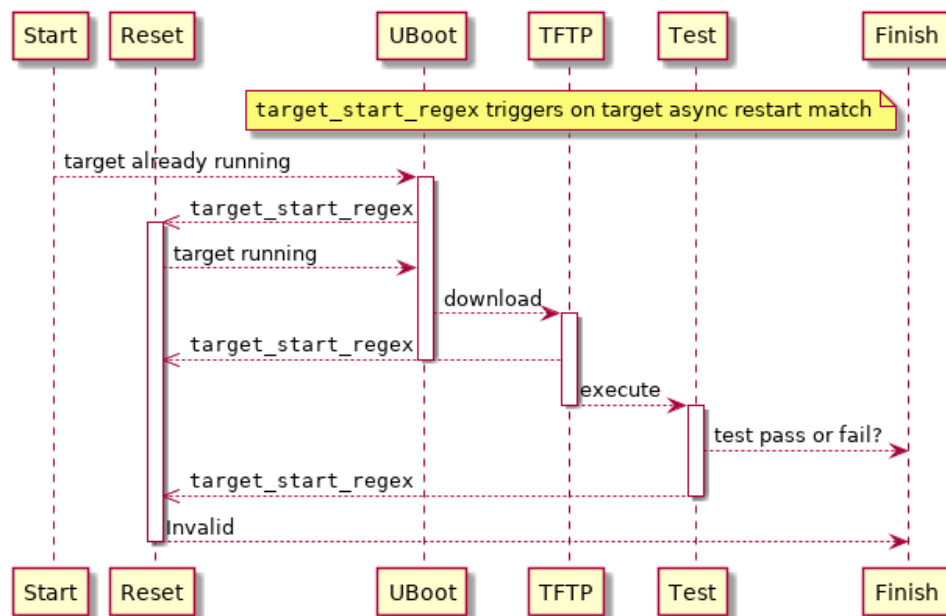


Fig. 7: Target Start Filter Trigger

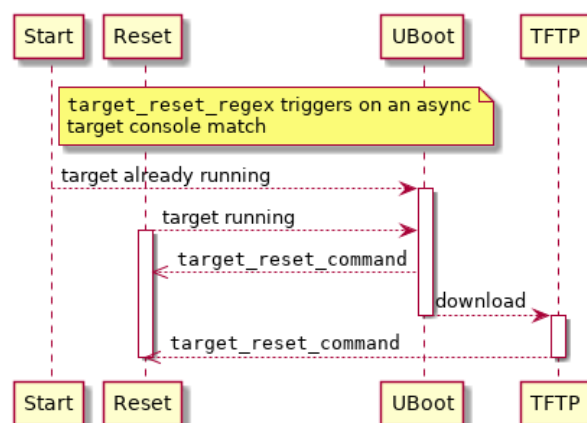


Fig. 8: Target Reset Filter Trigger

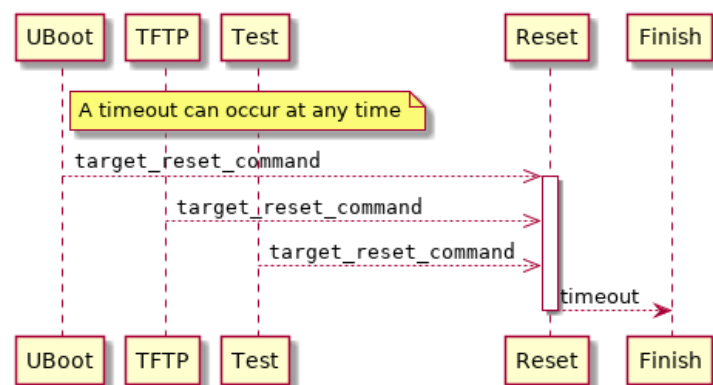


Fig. 9: Target Timeout

TRACING

RTEMS Tracing Framework is an on-target software based system which helps track the ongoings inside the operation of applications, third-party packages, and the kernel in real time.

Software based tracing is a complex process which requires components on both the target and the host to work together. However its portability across all architectures and board support packages makes it a useful asset. A key requirement in RTEMS trace process is to take existing code in compiled format (ELF) and instrument it in order to log various events and records in real time. However instrumenting of the code for tracing should happen without rebuilding the code from the source and without annotating the source with trace code.

10.1 Introduction to Tracing

Tracing is an important function which has several applications including identification of complex threading, detection of deadlocks, tracing functions along with their argument values, and return values through progression of several function calls and audit the performance of an application according to required specifications.

RTEMS tracing framework is under development and welcomes contribution by users.

RTEMS has the following trace components:

- *RTEMS Trace Linker* (page 189)
- *RTEMS Capture Engine* (page 186)
- *RTEMS Event Recording* (page 198)
- Common Trace Format Integration

RTEMS trace framework can currently function using the following methods. Both of the methods make use of the *Trace Linker* (page 189) :

10.1.1 RTEMS Trace Using Trace Buffering

This scheme of tracing goes through the flow of events described in a subsequent flowchart:

Step 1: The user creates an application and user configuration file. The configuration file specifies the use of the trace buffer generator and other standard initializations. The user then configures their BSP and invokes the trace linker using a command to link the application executable. The trace linker uses the application files in compiled format (ELF) and the libraries used to build the application for performing this link.

Step 2: The RTEMS Trace Linker reads the user's configuration file and that results in it reading the standard Trace Buffering Configuration files installed with the RTEMS Trace Linker. The trace linker uses the target compiler and linker to create the trace enabled application executable. It wraps the functions defined in the user's configuration with code that captures trace records into the statically allocated buffer. The trace wrapper code is compiled with the target compiler and the resulting ELF object file is added to the standard link command line used to link the application and the application is re-linked using the wrapping option of the GNU linker.

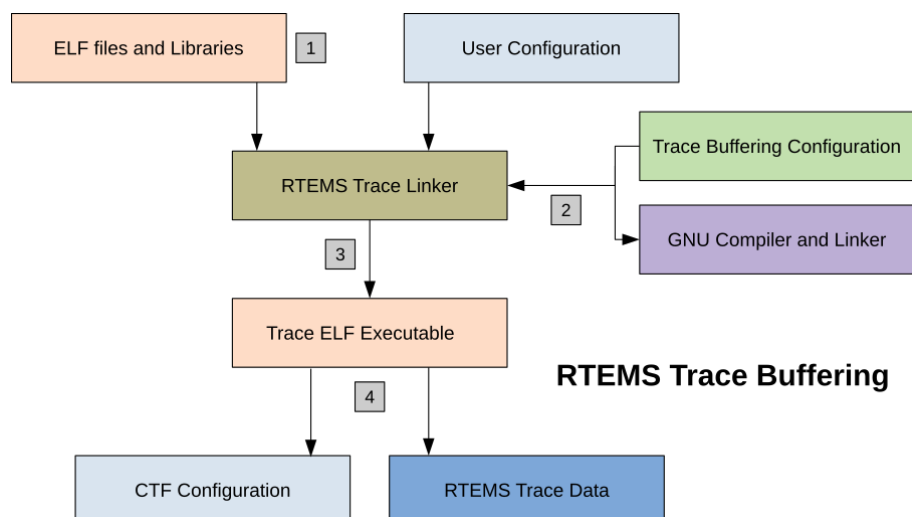
Step 3: The trace linker creates an executable which is capable of running on the target hardware or simulator.

Step 4: RTEMS shell provides the "rtrace" command to display and save trace buffers.

10.1.2 RTEMS Trace Using Printk

This scheme of tracing goes through the flow of events described in a subsequent flowchart:

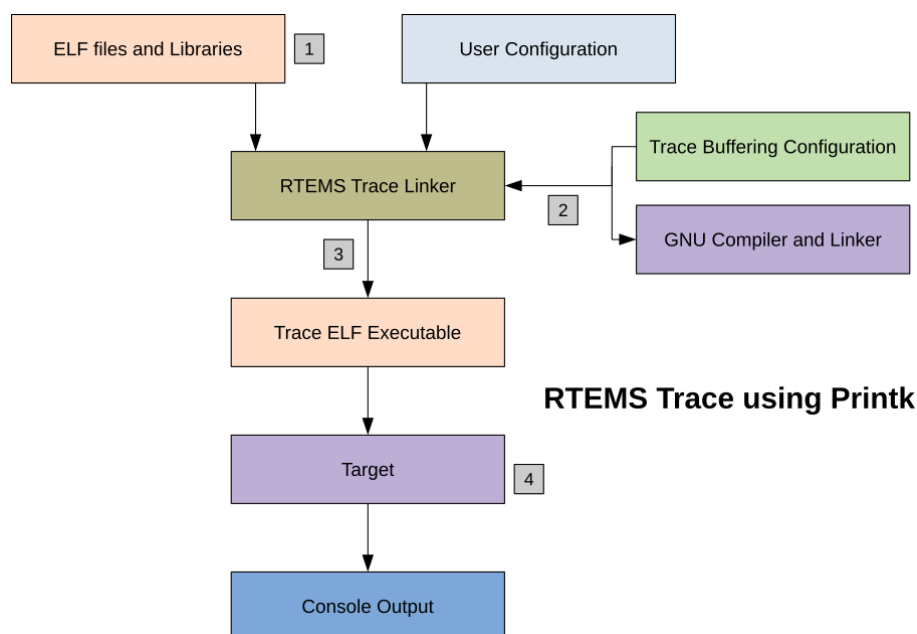
Step 1: The user creates an RTEMS application in the normal manner as well as a Trace Linker configuration file. The configuration file specifies using the Printk trace mode and the functions to trace. The user invokes the Trace Linker with the configuration and the normal link command line used to link the application executable. The application ELF object files and libraries, including the RTEMS libraries are standard and do not need to be built specially.



Step 2: The RTEMS Trace Linker reads the user's configuration file and that results in it reading the standard Printk Trace Configuration files installed with the RTEMS Trace Linker. The trace linker uses the target compiler and linker to create the trace enabled application executable. It wraps the functions defined in the user's configuration with code that prints the entry with arguments and exit and return value if any. The trace wrapper code is compiled with the target compiler and the resulting ELF object file is added to the standard link command line used to link the application and the application is relinked using the wrapping option of the GNU linker.

Step 3: The trace linker creates and RTEMS ELF executable that can be run on the target hardware or simulator.

Step 4: The application is run in the hardware directly or using a debugger. The printk() output appears on the target console and the user can save that to a file.



The *Tracing Examples* (page 182) section describes generation of traces using Trace Buffering technique for the *fileio* testsuite available with RTEMS installation.

10.2 Tracing Examples

The following example executes RTEMS trace using trace buffering for the *fileio* sample testcase.

10.2.1 Features

Tracing using trace buffering consists of the following sets of features:

- Individual entry and exit records.
- Task details such as CPU, current priority, real priority, task state and interrupt state.
- Nano-second timestamp.
- Interrupt safe buffer management.
- Function argument capture.
- Return value capture.
- Shell command support to report to the console, save a buffer, assess status of tracing, or view buffers between specified index ranges.

10.2.2 Prerequisites

1. Setup RTEMS for the *sparc/erc32* architecture-bsp pair to run the following example.
2. Download the fileio **configuration file** and store it on the top of the installed BSP's directory.
3. Change the value of the keys: *rtems-path* and *prefix* according to your rtems installation. The *rtems-path* is the path to the bsp installation and *prefix* is the path to the tools used to build rtems. Also set the value of the *rtems-bsp* key to *sparc/erc32*.

10.2.3 Demonstration

Inside the RTEMS build directory (the directory where the fileio configuration has been stored) run the following commands to generate traces:

BSP is configured with the following command -

```
1 ../rtems/configure --target=sparc-rtems5 --prefix=/development/rtems/5 \
2 --enable-networking --enable-tests --enable-rtemsbsp=erc32 --enable-cxx
```

The next two commands are used to link the fileio executable. The *-B* option signifies the use of the complete path to the required directory or file. Write the full path instead of the path file: *sparc-rtems5/erc32/lib/* in the following commands according to your installation. Also confirm the path of the fileio's executable and object files in the last line of the command according to your installation.

```
1 sparc-rtems5-gcc -Bsparc-rtems5/erc32/lib/ \
2 -specs bsp_specs -qrtems -mcpu=cypress -O2 -g -ffunction-sections \
3 -fdata-sections -Wall -Wmissing-prototypes -Wimplicit-function-declaration \
4 -Wstrict-prototypes -Wnested-externs -Wl,--gc-sections -mcpu=cypress \
```

(continues on next page)

(continued from previous page)

```

5 -o sparc-rtems5/c/erc32/testsuites/samples/fileio.exe sparc-rtems5/c/erc32/\
6 testsuites/samples/fileio/fileio-init.o

```

This is the trace linker command to generate and compile the wrapper c file for the application. The link command follows the escape sequence “-”. “-C” option denotes the name of the user configuration file and “-W” specifies the name of the wrapper c file.

```

1 rtems-tld -C fileio-trace.ini -W fileio-wrapper -- -Bsparc-rtems5/erc32/lib/ \
2 -specs bsp_specs -qrtems -mcpu=cypress -O2 -g -ffunction-sections \
3 -fdata-sections -Wall -Wmissing-prototypes -Wimplicit-function-declaration \
4 -Wstrict-prototypes -Wnested-externs -Wl,--gc-sections -mcpu=cypress \
5 -o sparc-rtems5/c/erc32/testsuites/samples/fileio.exe sparc-rtems5/c/erc32/\
6 testsuites/samples/fileio/fileio-init.o

```

The following command is used to run the application. Hit enter key quickly and type “s” and “root” and “pwd” to run the rtems shell. Use the *rtrace status*, *rtrace trace* and *rtrace save* commands to know the status of the tracing, display the contents of the trace buffer and save the buffer to disk in the form of binary files. Use *rtrace -l* to list the available options for commands with *rtrace*.

```

1 sparc-rtems5-run sparc-rtems5/c/erc32/testsuites/samples/fileio.exe

```

The output from the above commands will be as follows:

```

1 *** BEGIN OF TEST FILE I/O ***
2 *** TEST VERSION: 5.0.0.de9b7d712bf5da6593386fd4fbca0d5f8b8431d8
3 *** TEST STATE: USER_INPUT
4 *** TEST BUILD: RTEMS_NETWORKING RTEMS_POSIX_API
5 *** TEST TOOLS: 7.3.0 20180125 (RTEMS 5, RSB a3a6c34c150a357e57769a26a460c475e188438f,
↳Newlib 3.0.0)
6 Press any key to start file I/O sample (20s remaining)
7 Press any key to start file I/O sample (19s remaining)
8 Press any key to start file I/O sample (18s remaining)
9 Press any key to start file I/O sample (17s remaining)
10 Press any key to start file I/O sample (16s remaining)
11 Press any key to start file I/O sample (15s remaining)
12 Press any key to start file I/O sample (14s remaining)
13 =====
14 RTEMS FILE I/O Test Menu
15 =====
16 p -> part_table_initialize
17 f -> mount all disks in fs_table
18 l -> list file
19 r -> read file
20 w -> write file
21 s -> start shell
22 Enter your selection ==>s
23 Creating /etc/passwd and group with four useable accounts:
24 root/pwd
25 test/pwd
26 rtems/NO PASSWORD
27 chroot/NO PASSWORD
28 Only the root user has access to all available commands.
29 =====
30 starting shell

```

(continues on next page)

(continued from previous page)

```

31 =====
32
33 Welcome to rtems-5.0.0 (SPARC/w/FPU/erc32)
34 COPYRIGHT (c) 1989-2008.
35 On-Line Applications Research Corporation (OAR).
36
37 Login into RTEMS
38 /dev/foobar login: root
39 Password:
40
41 RTEMS Shell on /dev/foobar. Use 'help' to list commands.
42 SHLL [/] # rtrace status
43 RTEMS Trace Bufferring: status
44   Running: yes
45   Triggered: yes
46   Level: 0%
47   Traces: 25
48 SHLL [/] # rtrace stop
49 RTEMS Trace Bufferring: stop
50 SHLL [/] # rtrace trace
51 RTEMS Trace Bufferring: trace
52   Trace buffer: 0x20921d8
53   Words traced: 1487
54   Traces: 25
55   0:00:40.983197010 2081910 0a010002 [ 2/ 2] > malloc((size_t) 00000130)
56   0:00:40.983333119 136109 0a010002 [ 2/ 2] < malloc => (void*) 0x219bb88
57   0:00:40.983471669 138550 0a010002 [ 2/ 2] > malloc((size_t) 00000006)
58   0:00:40.983606557 134888 0a010002 [ 2/ 2] < malloc => (void*) 0x219bcc0
59   0:00:40.983684682 78125 0a010002 [ 2/ 2] > malloc((size_t) 00000007)
60   0:00:40.983819569 134887 0a010002 [ 2/ 2] < malloc => (void*) 0x219bcd0
61   0:00:40.983909901 90332 0a010002 [ 2/ 2] > malloc((size_t) 000003fc)
62   0:00:40.984046620 136719 0a010002 [ 2/ 2] < malloc => (void*) 0x219bce0
63   0:00:40.986624137 2577517 0a010003 [200/200] > malloc((size_t) 00000080)
64   0:00:40.986767569 143432 0a010003 [200/200] < malloc => (void*) 0x219bce0
65   0:00:40.987531119 763550 0a010003 [200/200] > calloc((size_t) 00000001, (size_t)
    ↳ 0000005d)
66   0:00:40.987603751 72632 0a010003 [200/200] > malloc((size_t) 0000005d)
67   0:00:40.987744743 140992 0a010003 [200/200] < malloc => (void*) 0x219bce0
68   0:00:40.987824699 79956 0a010003 [200/200] < calloc => (void*) 0x219bce0
69   0:00:40.988302604 477905 0a010003 [200/200] > malloc((size_t) 00000080)
70   0:00:40.988446647 144043 0a010003 [200/200] < malloc => (void*) 0x219bd48
71   0:00:40.988667595 220948 0a010003 [200/200] > calloc((size_t) 00000001, (size_t)
    ↳ 00000080)
72   0:00:40.988740837 73242 0a010003 [200/200] > malloc((size_t) 00000080)
73   0:00:40.988884880 144043 0a010003 [200/200] < malloc => (void*) 0x219bdd0
74   0:00:40.988964836 79956 0a010003 [200/200] < calloc => (void*) 0x219bdd0
75   0:00:40.989042961 78125 0a010003 [200/200] > calloc((size_t) 00000001, (size_t)
    ↳ 00000080)
76   0:00:40.989110100 67139 0a010003 [200/200] > malloc((size_t) 00000080)
77   0:00:40.989254143 144043 0a010003 [200/200] < malloc => (void*) 0x219be58
78   0:00:40.989334099 79956 0a010003 [200/200] < calloc => (void*) 0x219be58
79   0:00:40.990118401 784302 0a010003 [200/200] > calloc((size_t) 00000001, (size_t)
    ↳ 00000061)
80   0:00:40.990176995 58594 0a010003 [200/200] > malloc((size_t) 00000061)
81   0:00:40.990309441 132446 0a010003 [200/200] < malloc => (void*) 0x219bd48
82   0:00:40.990384515 75074 0a010003 [200/200] < calloc => (void*) 0x219bd48

```

(continues on next page)

(continued from previous page)

```

83 0:00:40.990870355 485840 0a010003 [200/200] > malloc((size_t) 00000080)
84 0:00:40.991011346 140991 0a010003 [200/200] < malloc => (void*) 0x219bee0
85 0:00:40.991227411 216065 0a010003 [200/200] > calloc((size_t) 00000001, (size_t)
↪ 00000080)
86 0:00:40.991296380 68969 0a010003 [200/200] > malloc((size_t) 00000080)
87 0:00:40.991438593 142213 0a010003 [200/200] < malloc => (void*) 0x219bf68
88 0:00:40.991514276 75683 0a010003 [200/200] < calloc => (void*) 0x219bf68
89 0:00:40.991589349 75073 0a010003 [200/200] > calloc((size_t) 00000001, (size_t)
↪ 00000080)
90 0:00:40.991653437 64088 0a010003 [200/200] > malloc((size_t) 00000080)
91 0:00:40.991794428 140991 0a010003 [200/200] < malloc => (void*) 0x219bff0
92 0:00:40.991871332 76904 0a010003 [200/200] < calloc => (void*) 0x219bff0
93 0:00:40.992283320 411988 0a010003 [200/200] > malloc((size_t) 00000008)
94 SHLL [/] # rtrace save fileio-trace.bin
95 RTEMS Trace Bufferring: trace
96   Trace File: fileio-trace.bin
97   Trace buffer: 0x20921d8
98   Words traced: 1487
99   Traces: 25
100 SHLL [/] #

```

10.3 Capture Engine

Capture Engine is a trace tool built inside the RTEMS operating system. Capture Engine is designed to cause the lowest load on the system when operating. Hence it does not effect RTEMS when operating or when disabled. It binds to RTEMS at runtime and does not require RTEMS or your application to be rebuilt in order to use it.

The Capture Engine's sample testcase for the *sparc/erc32* is available in build directory created when building RTEMS in the path file: *sparc-rtems5/c/erc32/testsuites/samples*. In order to access the capture testcase perform the following set of operations inside the RTEMS build directory.

```

1 $ cd /sparc-rtems5/c/erc32/testsuites/samples
2 $ sparc-rtems5-run ./capture.exe
3
4
5 *** BEGIN OF TEST CAPTURE ENGINE ***
6 *** TEST VERSION: 5.0.0.de9b7d712bf5da6593386fd4fbca0d5f8b8431d8
7 *** TEST STATE: USER_INPUT
8 *** TEST BUILD: RTEMS_NETWORKING RTEMS_POSIX_API
9 *** TEST TOOLS: 7.3.0 20180125 (RTEMS 5, RSB a3a6c34c150a357e57769a26a460c475e188438f, ↵
↵Newlib 3.0.0)
10 Press any key to start capture engine (20s remaining)
11 Press any key to start capture engine (19s remaining)
12 Press any key to start capture engine (18s remaining)
13
14 Monitor ready, press enter to login.
15
16 1-rtems $

```

Capture Engine comes with a set of commands to perform various actions.

10.3.1 Capture Engine Commands

- 1) `copen <buffer-size>`: Used to initialize the Capture Engine with the trace buffer size in bytes. By default the Capture Engine is not initialized and not running.
- 2) `cwceil <priority-value>`: Capture Engine filter used to put an upper limit on the event priority to be captured.
- 3) `cwfloor <priority-value>`: Capture Engine filter used to put a lower limit on the event priority to be captured.
- 4) `cwglob <on/off>`: Enable or disable the global watch.
- 5) `cenable`: Enables the Capture Engine. Capture Engine is by default disabled after being opened.
- 6) `cdisable`: Disables the Capture Engine.
- 7) `ctlist`: Lists the watch and trigger configurations.
- 8) `ctrace`: Dumps the recorded traces. By default this command displays 24 trace records. Repeated use of this command will display all the recorded traces.
- 9) `cwadd <task-name>`: Add watch on a particular task.

10) `cwtctl <task-name> <on/off>`: Enable or disable watch on a particular task.

11) `ctset`: Used to set a trigger. The general form of the command is:

```
ctset [-?] type [to name/id] [from] [from name/id]
```

type in the above command refers to the type of trigger needed. The types of triggers that currently exist are:

- `switch` : a context switch from one task to another task
- `create` : the executing task creates a task
- `start` : the executing task starts a task
- `restart` : the executing task restarts a task
- `delete` : the executing task deletes a task
- `begin` : a task is beginning
- `exitted` : a task is exiting

10.3.2 Example

The following is a sample run of the capture testsuite. The `test1` command on the Capture Engine Command Line Interface (CLI) makes the *RMON* task invoke a call to the `capture_test_1()` command. This function (in the `test1.c` source code) creates and starts three tasks : *CT1a*, *CT1b* and *CT1c*. These tasks are passed the object id of a semaphore as a task argument. This run through traces the context switches between these tasks. `cwceil` and `cwfloor` are set to a narrow range of task priorities to avoid creating noise from a large number of context switches between tasks we are not interested in.

```

1 *** BEGIN OF TEST CAPTURE ENGINE ***
2 *** TEST VERSION: 5.0.0.de9b7d712bf5da6593386fd4fbca0d5f8b8431d8
3 *** TEST STATE: USER_INPUT
4 *** TEST BUILD: RTEMS_NETWORKING RTEMS_POSIX_API
5 *** TEST TOOLS: 7.3.0 20180125 (RTEMS 5, RSB a3a6c34c150a357e57769a26a460c475e188438f, ↵
  ↵Newlib 3.0.0)
6 Press any key to start capture engine (20s remaining)
7 Press any key to start capture engine (19s remaining)
8 Press any key to start capture engine (18s remaining)
9 Press any key to start capture engine (17s remaining)
10
11 Monitor ready, press enter to login.
12
13 1-rtems $ copen 50000
14 capture engine opened.
15 1-rtems $ cwceil 100
16 watch ceiling is 100.
17 1-rtems $ cwfloor 102
18 watch floor is 102.
19 1-rtems $ cwglob on
20 global watch enabled.
21 1-rtems $ ctset RMON
22 trigger set.
23 1-rtems $ cenable
24 capture engine enabled.
```

(continues on next page)

(continued from previous page)

```

25 1-rtems $ test1
26 1-rtems $ cdisable
27 capture engine disabled.
28 1-rtems $ ctrace
29 0 0:18:17.462314124      0a010003 CT1a 102 102 102  4096 TASK_RECORD
30 0 0:18:17.462398963      0 0a010003 CT1a 102 102      CREATED
31 0 0:18:17.462647987      249024 0a010003 CT1a 102 102      STARTED
32 0 0:18:17.462904334      256347 0a010003 CT1a 102 102      SWITCHED_IN
33 0 0:18:17.463069129      164795 0a010003 CT1a 102 102      BEGIN
34 0 0:18:17.463335853      266724 0a010003 CT1a 102 102      SWITCHED_OUT
35 0 0:18:18.461348547      0a010004 CT1b 101 101 101  4096 TASK_RECORD
36 0 0:18:18.461433997 998098144 0a010004 CT1b 101 101      CREATED
37 0 0:18:18.461683631      249634 0a010004 CT1b 101 101      STARTED
38 0 0:18:18.461934485      250854 0a010004 CT1b 101 101      SWITCHED_IN
39 0 0:18:18.462099891      165406 0a010004 CT1b 101 101      BEGIN
40 0 0:18:19.460935339 998835448 0a010004 CT1b 101 101      SWITCHED_OUT
41 0 0:18:19.461431555      0a010005 CT1c 100 100 100  4096 TASK_RECORD
42 0 0:18:19.461516394      581055 0a010005 CT1c 100 100      CREATED
43 0 0:18:19.461765418      249024 0a010005 CT1c 100 100      STARTED
44 0 0:18:19.462019324      253906 0a010005 CT1c 100 100      SWITCHED_IN
45 0 0:18:19.462184119      164795 0a010005 CT1c 100 100      BEGIN
46 0 0:18:19.462475257      291138 0a010005 CT1c 100 100      SWITCHED_OUT
47 0 0:18:19.462551551      76294 0a010004 CT1b 101 101      SWITCHED_IN
48 0 0:18:19.960935645 498384094 0a010004 CT1b 101 101      SWITCHED_OUT
49 0 0:18:19.961012549      76904 0a010003 CT1a 102 100      SWITCHED_IN
50 0 0:18:19.961341528      328979 0a010003 CT1a 102 102      SWITCHED_OUT
51 1-rtems $ ctrace
52 0 0:18:19.961418433      0 0a010005 CT1c 100 100      SWITCHED_IN
53 0 0:18:19.961672339      253906 0a010005 CT1c 100 100      SWITCHED_OUT
54 0 0:18:19.961749854      77515 0a010004 CT1b 101 101      SWITCHED_IN
55 0 0:18:20.460967077 499217223 0a010004 CT1b 101 101      SWITCHED_OUT
56 0 0:18:20.461219763      252686 0a010005 CT1c 100 100      SWITCHED_IN
57 0 0:18:20.461424231      204468 0a010005 CT1c 100 100      TERMINATED
58 0 0:18:20.461747107      322876 0a010005 CT1c 100 100      SWITCHED_OUT
59 0 0:18:20.461824011      76904 0a010004 CT1b 101 101      SWITCHED_IN
60 0 0:18:20.462015052      191041 0a010004 CT1b 101 101      TERMINATED
61 0 0:18:20.462336707      321655 0a010004 CT1b 101 101      SWITCHED_OUT
62 0 0:18:20.462414222      77515 0a010003 CT1a 102 102      SWITCHED_IN
63 0 0:18:20.462608924      194702 0a010003 CT1a 102 102      TERMINATED
64 0 0:18:20.462933021      324097 0a010003 CT1a 102 102      SWITCHED_OUT
65 1-rtems $ ctrace
66 1-rtems $

```


10.4 Trace Linker

RTEMS trace linker is a post link tool central to the RTEMS trace framework. It is installed as a part of the RTEMS Tool Project. The RTEMS Trace Linker is a post link tool that performs a re-link of your application to produce a trace executable. A trace executable has been instrumented by the RTEMS Trace Linker with additional code that implements software tracing. A key requirement of the trace process in RTEMS is to take existing code in a compiled format (ELF) and instrument it without rebuilding that code from source and without annotating that source with trace code.

10.4.1 Command Line

A typical command to invoke the trace linker consists of two parts separated by `--`. The first part controls the trace linker and provides the various options it needs and the second part is a standard linker command line you would use to link an RTEMS application. The current command line for trace linker consists of:

```

1 $ rtems-tld -h
2 rtems-trace-ld [options] objects
3 Options and arguments:
4 -h          : help (also --help)
5 -V          : print linker version number and exit (also --version)
6 -v          : verbose (trace import parts), can supply multiple times
7              to increase verbosity (also --verbose)
8 -w          : generate warnings (also --warn)
9 -k          : keep temporary files (also --keep)
10 -c compiler : target compiler is not standard (also --compiler)
11 -l linker   : target linker is not standard (also --linker)
12 -E prefix   : the RTEMS tool prefix (also --exec-prefix)
13 -f cflags   : C compiler flags (also --cflags)
14 -r path     : RTEMS path (also --rtems)
15 -B bsp      : RTEMS arch/bsp (also --rtems-bsp)
16 -W wrapper   : wrapper file name without ext (also --wrapper)
17 -C ini      : user configuration INI file (also --config)
18 -P path     : user configuration INI file search path (also --path)

```

The trace linker generates code that needs to be compiled and linked to the application executable so it needs to know the target compiler and *CFLAGS*. There are a couple of ways to do this. The simplest is to provide the path to RTEMS using the *-r* option and the architecture and BSP name in the standard RTEMS format of arch/bsp. The trace linker will extract the compiler and flags used to build RTEMS and will use them. If you require specific options you can use the *-f*, *-c*, *-l*, and *-E* options to provide them. If the functions you are tracing use types from your code then add the include path to the *CFLAGS*.

The trace linker requires you to provide a user configuration file using the *-C* or *--config* option. This is an INI format file detailed in the Configuration section. You can also provide an INI file search path using the *-P* option.

If you are working with new configuration files and you want to view the files the trace linker generates, add the *-k* option to keep the temporary files, and *-W* to specify an explicit wrapper C file name. If you set the dump-on-error option in the configuration options section you will get a dump of the configuration on an error.

10.4.2 Configuration (INI) files

The Trace Linker is controlled using configuration files. Configuration files are categorized into 3 types:

- **User Configuration:** These are specific to the user application to be traced. This file initializes the values of the trace generator, triggers, enables, and traces.
- **Tracer Configuration:** These are like a library of common or base trace functions that can be referenced by an application. These files tend to hold the details needed to wrap a specific set of functions. Examples provided with the RTEMS Linker are the RTEMS API and Libc.
- **Generator Configuration:** This is used to encapsulate a specific method of tracing. RTEMS currently provides generators for trace buffering, printk, and printf.

The configuration files are in the *INI file format* which is composed of *sections*. Each section has a section name and set of *keys* which consist of *names* and *values*. A typical key is of the form *name=value*. Keys can be used to include other INI files using the *include* key name. This is shown in the following example where the values indicate *rtems* and *rtld-base* configuration files:

```
1 include = rtems.ini, rtld-base.ini
```

The trace linker also uses values in keys to specify other sections. In this example the functions name lists *test-trace-funcs* and that section contains a headers key that further references a section called *test-headers*:

```
1 functions = test-trace-funcs, rtems-api
2
3 [test-trace-funcs]
4 ; Parsed via the 'function-set', not parse as a 'trace'.
5 headers = test-headers
6
7 [test-headers]
8 header = '#include "test-trace-1.h"'
```

The format of a configuration file is explained next. Snippets of the file: *test-trace.ini* have been used for explicit understanding. This file can be found in the *rtems-tools* directory of the *rtems* installation.

10.4.2.1 Tracer Section

The topmost level section is the tracer section. It can contains the following keys:

- **name:** The name of trace being linked.
- **options:** A list of option sections.
- **defines:** A list of sections containing defines or define record.
- **define:** A list of define string that are single or double quoted.
- **enables:** The list of sections containing enabled functions to trace.
- **triggers:** The list of sections containing enabled functions to trigger trace on.
- **traces:** The list of sections containing function lists to trace.

- **functions:** The list of sections containing function details.
- **include:** The list of files to include.

The tracer section of the file:*test-trace.ini* is shown below with explanatory comments.

```

1 ;
2 ; RTEMS Trace Linker Test Configuration.
3 ;
4 ; We must provide a top level trace section.
5 ;
6 [tracer]
7 ;
8 ; Name of the trace.
9 ;
10 name = RTEMS Trace Linker Test
11 ;
12 ; The BSP.
13 ;
14 bsp = sparc/sis
15 ;
16 ; Functions to trace.
17 ;
18 traces = test-trace, test-trace-funcs, rtems-api-task
19 ;
20 ; Specify the options.
21 ;
22 options = test-options
23 ;
24 ; Define the function sets. These are the function's that can be
25 ; added to the trace lists.
26 ;
27 functions = test-trace-funcs, rtems-api
28 ;
29 ; Include RTEMS Trace support.
30 ;
31 include = rtems.ini, rtld-base.ini

```

10.4.2.2 Options section

The options section in the file:*io-trace.ini* is called the *fileio-options*. A general options section can contain following sets of keys:

- **dump-on-error:** Dump the parsed configuration data on error. The value can be true or false.
- **verbose:** Set the verbose level. The value can be true or a number value.
- **prefix:** The prefix for the tools and an install RTEMS if rtems-path is not set.
- **cc:** The compiler used to compile the generated wrapper code. Overrides the BSP configuration value if a BSP is specified.
- **ld:** The linker used to link the application. The default is the cc value as read from the BSP configuration if specified. If your application contains C++ code use this setting to change the linker to g++.

- `cflags`: Set the CFLAGS used to compile the wrapper. These flags are pre-pended to the BSP read flags if a BSP is specified. This option is used to provide extra include paths to header files in your application that contain types referenced by functions being traced.
- `rtems-path`: The path to an install RTEMS if not installed under the prefix.
- `rtems-bsp`: The BSP we are building the trace executable for. There is an arch and bsp pair. For example `sparc/erc32`.

The options section of the file: *test-trace.ini* uses two of the aforementioned keys as shown below:

```

1 ;
2 ; Options can be defined here or on the command line.
3 ;
4 [test-options]
5 prefix = /development/rtems/5
6 verbose = true

```

10.4.2.3 Trace Section

A trace section defines how trace wrapper functions are built. To build a trace function that wraps an existing function in an ELF object file or library archive we need to have the function's signature. A signature is the function's declaration with any types used. The signature has specific types and we need access to those types which means the wrapper code needs to include header files that define those types. There may also be specific defines needed to access those types. A trace section can contain the following keys:

- `generator`: The generator defines the type of tracing being used.
- `headers`: List of sections that contain header file's keys.
- `header`: A header key. Typically the include code.
- `defines`: List of sections that contain defines.
- `define`: A define key. Typically the define code.
- `signatures`: List of function signature sections.
- `trace`: Functions that are instrumented with trace code.

The trace section of the file: *test-trace.ini* is shown below. A trace section can reference other trace sections of a specific type. This allows a trace sections to build on other trace sections.

```

1 ; User application trace example.
2 ;
3 [test-trace]
4 generator = printf-generator
5 ; Just here for testing.
6 trace = test_trace_3
7
8 [test-trace-funcs]
9 ; Parsed via the 'function-set', not parse as a 'trace'.
10 headers = test-headers
11 header = '#include "test-trace-2.h"'
12 defines = test-defines

```

(continues on next page)

(continued from previous page)

```

13 define = "#define TEST_TRACE_2 2"
14 signatures = test-signatures
15 ; Parsed via the 'trace', not parsed as a function-set
16 trace = test_trace_1, test_trace_2
17
18 [test-headers]
19 header = '#include "test-trace-1.h"'
20
21 [test-defines]
22 define = "#define TEST_TRACE_1 1"
23
24 [test-signatures]
25 test_trace_1 = void, int
26 test_trace_2 = test_type_2, test_type_1
27 test_trace_3 = float, float*

```

10.4.2.4 Function Section

Function sections define functions that can be traced. Defining a function so it can be traced does not mean it is traced. The function must be added to a trace list to be traced. Function sections provide any required defines, header files, and the function signatures.

A function signature is the function's declaration. It is the name of the function, the return value, and the arguments. Tracing using function wrappers requires that we have accurate function signatures and ideally we would like to determine the function signature from the data held in ELF files. ELF files can contain DWARF data, the ELF debugging data format. In time the trace project would like to support libdwwarf so the DWARF data can be accessed and used to determine a function's signature. This work is planned but not scheduled to be done and so in the meantime we explicitly define the function signatures in the configuration files.

A function section can consist of the following keys:

- headers: A list of sections containing headers or header records.
- header: A list of include string that are single or double quoted.
- defines: A list of sections containing defines or define record.
- defines: A list of define string that are single or double quoted.
- signatures: A list of section names of function signatures.
- includes: A list of files to include.

Function signatures are specified with the function name being the key's name and the key's value being the return value and a list of function arguments. You need to provide void if the function uses void. Variable argument list are currently not supported. There is no way to determine statically a variable argument list. The function section in the file: *test-trace.ini* has been labeled as *test-trace-funcs*. This can be seen in the file snippet of the previous section.

10.4.2.5 Generators

The trace linker's major role is to wrap functions in the existing executable with trace code. The directions on how to wrap application functions is provided by the generator configuration. The wrapping function uses a GNU linker option called `-wrap=symbol`. The GNU Ld manual states:

“Use a wrapper function for symbol. Any undefined reference to symbol will be resolved to `__wrap_symbol`. Any undefined reference to `__real_symbol` will be resolved to symbol.”

Generator sections specify how to generate trace wrapping code. The trace linker and generator section must match to work. The trace linker expects a some things to be present when wrapping functions. The section’s name specifies the generator and can be listed in a generator key in a tracer or trace section. If the generator is not interested in a specific phase it does not need to define it. Nothing will be generated in regard to this phase. For example code to profile specific functions may only provide the entry-trace and exit-trace code where a nano-second time stamp is taken.

The generate code will create an entry and exit call and the generator code block can be used to allocate buffer space for each with the lock held. The entry call and argument copy is performed with the lock released. The buffer space having been allocated will cause the trace events to be in order. The same goes for the exit call. Space is allocated in separate buffer allocate calls so the blocking calls will have the exit event appear in the correct location in the buffer.

The following keys can be a part of the generator configuration:

- `headers`: A list of sections containing headers or header records.
- `header`: A list of include string that are single or double quoted.
- `defines`: A list of sections containing defines or define record.
- `define`: A list of define string that are single or double quoted.
- `entry-trace`: The wrapper call made on a function’s entry. Returns bool where true is the function is being traced. This call is made without the lock being held if a lock is defined.
- `arg-trace`: The wrapper call made for each argument to the trace function if the function is being traced. This call is made without the lock being held if a lock is defined.
- `exit-trace`: The wrapper call made after a function’s exit. Returns bool where true is the function is being traced. This call is made without the lock being held if a lock is defined.
- `ret-trace`: The wrapper call made to log the return value if the function is being traced. This call is made without the lock being held if a lock is defined.
- `lock-local`: The wrapper code to declare a local lock variable.
- `lock-acquire`: The wrapper code to acquire the lock.
- `lock-release`: The wrapper code to release the lock.
- `buffer-local`: The wrapper code to declare a buffer index local variable.
- `buffer-alloc`: The wrapper call made with a lock held if defined to allocate buffer space to hold the trace data. A suitable 32bit buffer index is returned. If there is no space an invalid index is returned. The generator must handle any overhead space needed. The generator needs to make sure the space is available before making the alloc all.
- `code-blocks`: A list of code block section names.
- `code`: A code block in `<<CODE — CODE` (without the single quote).
- `includes`: A list of files to include.

The following macros can be used in wrapper calls:

- `@FUNC_NAME@`: The trace function name as a quote C string.

- @FUNC_INDEX@: The trace function index as a held in the sorted list of trace functions by the trace linker. It can be used to index the names, enables, and triggers data.
- @FUNC_LABEL@: The trace function name as a C label that can be referenced. You can take the address of the label.
- @FUNC_DATA_SIZE@: The size of the data in bytes.
- @FUNC_DATA_ENTRY_SIZE@: The size of the entry data in bytes.
- @FUNC_DATA_RET_SIZE@: The size of the return data in bytes.
- @ARG_NUM@: The argument number to the trace function.
- @ARG_TYPE@: The type of the argument as a C string.
- @ARG_SIZE@: The size of the type of the argument in bytes.
- @ARG_LABEL@: The argument as a C label that can be referenced.
- @RET_TYPE@: The type of the return value as a C string.
- @RET_SIZE@: The size of the type of the return value in bytes.
- @RET_LABEL@: The return value as a C label that can be referenced.

The *buffer-alloc*, *entry-trace*, and *exit-trace* can be transformed using the following macros:

- @FUNC_NAME@
- @FUNC_INDEX@
- @FUNC_LABEL@
- @FUNC_DATA_SIZE@
- @FUNC_DATA_ENTRY_SIZE@
- @FUNC_DATA_EXIT_SIZE@

The *arg-trace* can be transformed using the following macros:

- @ARG_NUM@
- @ARG_TYPE@
- @ARG_SIZE@
- @ARG_LABEL@

The *ret-trace* can be transformed using the following macros:

- @RET_TYPE@
- @RET_SIZE@
- @RET_LABEL@

The file: *test-trace.ini* specifies printf-generator as its generator. This section can be found in the file: *rtld-print.ini* in the rtems-tools directory and is shown below:

```

1 ;
2 ; A printf generator prints to stdout the trace functions.
3 ;
4 [printf-generator]
```

(continues on next page)

(continued from previous page)

```

5 headers = printf-generator-headers
6 entry-trace = "rtld_pg_printf_entry(@FUNC_NAME@, (void*) &@FUNC_LABEL@);"
7 arg-trace = "rtld_pg_printf_arg(@ARG_NUM@, @ARG_TYPE@, @ARG_SIZE@, (void*) &@ARG_LABEL@);"
8 exit-trace = "rtld_pg_printf_exit(@FUNC_NAME@, (void*) &@FUNC_LABEL@);"
9 ret-trace = "rtld_pg_printf_ret(@RET_TYPE@, @RET_SIZE@, (void*) &@RET_LABEL@);"
10 code = <<<CODE
11 static inline void rtld_pg_printf_entry(const char* func_name,
12                                       void*      func_addr)
13 {
14     printf(">>> %s (0x%08x)\n", func_name, func_addr);
15 }
16 static inline void rtld_pg_printf_arg(int      arg_num,
17                                       const char* arg_type,
18                                       int      arg_size,
19                                       void*      arg)
20 {
21     const unsigned char* p = arg;
22     int i;
23     printf(" %2d] %s(%d) = ", arg_num, arg_type, arg_size);
24     for (i = 0; i < arg_size; ++i, ++p) printf("%02x", (unsigned int) *p);
25     printf("\n");
26 }
27 static inline void rtld_pg_printf_exit(const char* func_name,
28                                       void*      func_addr)
29 {
30     printf("<<< %s (0x%08x)\n", func_name, func_addr);
31 }
32 static inline void rtld_pg_printf_ret(const char* ret_type,
33                                       int      ret_size,
34                                       void*      ret)
35 {
36     const unsigned char* p = ret;
37     int i;
38     printf(" rt] %s(%d) = ", ret_type, ret_size);
39     for (i = 0; i < ret_size; ++i, ++p) printf("%02x", (unsigned int) *p);
40     printf("\n");
41 }
42 CODE
43
44 [printf-generator-headers]
45 header = "#include <stdio.h>"

```

The trace linker generates C code with a wrapper for each function to be instrumented. The trace code generated is driven by the configuration INI files.

10.4.3 Development

The Trace Linker is part of the RTEMS tools git repository available at : <https://git.rtems.org/rtems-tools> The RTEMS tools project utilizes the waf build system. Use the following commands in the topmost build directory to build the tools project:

First we configure using:

```
1 $./waf configure --prefix=$HOME/development/rtems/5
```


Then we build and install using:

```
1 $./waf build install
```

10.5 Event Recording

The *event recording* support focuses on the recording of high frequency events such as

- thread switches,
- thread queue enqueue and surrender,
- interrupt entry and exit,
- heap/workspace memory allocate/free,
- UMA zone allocate/free,
- Ethernet packet input/output, and
- etc.

There is a fixed set of 512 system reserved and 512 user defined events which are identified by an event number (`rtems_record_event`).

The event recording support allows post-mortem analysis in fatal error handlers, e.g. the last events are in the record buffers, the newest event overwrites the oldest event. It is possible to detect record buffer overflows for consumers that expect a continuous stream of events, e.g. to display the system state changes in real-time.

The implementation supports high-end SMP machines (more than 1GHz processor frequency, more than four processors). It uses per-processor ring buffers to record the events. Synchronization is done without atomic read-modify-write operations. The CPU counter is used to get the time of events. It is combined with periodic uptime events to synchronize it with the monotonic system clock (`CLOCK_MONOTONIC`).

The application must configure the event recording via the configuration options `CONFIGURE_RECORD_PER_PROCESSOR_ITEMS` and `CONFIGURE_RECORD_EXTENSIONS_ENABLED`.

Events can be recorded for example with the `rtems_record_produce()` function.

```
1 #include <rtems/record.h>
2
3 void f( void )
4 {
5     rtems_record_produce( RTEMS_RECORD_USER( 0 ), 123 );
6 }
```

Recorded events can be sent to a host computer with a very simple record server started by `rtems_record_start_server()` via a TCP connection.

On the host computer you may use the command line tool `rtems-record` to get recorded events from the record server running on the target system.

HOST TOOLS

The RTEMS kernel is developed on host computers cross-compiled and linking the kernel, language runtime libraries, third-party packages and application source code so it can run on target hardware. RTEMS and some of the hardware it support cannot self-host so we need a range of tools to support the wide range of available host computers users wish to develop on. This section details the tools available on the host computers to help support RTEMS users and developers.

11.1 RTEMS Linker

The RTEMS Linker is an RTEMS tool to help build RTEMS application that dynamically loader code into a running RTEMS system.

[NOT COMPLETE]

11.2 RTEMS Symbols

The RTEMS Symbols (**rtems-syms**) command is an RTEMS tool to generate symbol tables used by the RTEMS Runtime Loader (RTL). The symbol table contains the exported base kernel symbols user code dynamically loaded can reference.

The RTEMS Runtime Loader supports two methods of loading a symbol table, embedded and runtime loading. Embedding the table requires linking the symbol table with the base image and runtime loading loads the table using the dynamic loader when RTEMS is running.

Filtering Symbols

Currently there is no filtering of symbols in the symbol table. This means all base kernel image symbols are present in the symbol table when only a sub-set of the symbols are referenced.

Embedding the symbol table creates self contained images. A target may not have any external media, for example RTEMS tests, or there is a requirement to avoid the management need to match the symbol table with the kernel base image. Embedding the symbol table requires a 2-pass link process making the application's build system more complicated.

A dynamically loadable symbol table is simpler to create however the symbol table and the kernel base image must match or the behaviour is undefined. There is currently no mechanism to ensure the symbol table and the kernel image match. The **rtems-syms** command is run against the base kernel image and the generated symbol table is installed on to the target hardware and loaded before any other modules.

11.2.1 Symbol Table

The symbol table is an ELF object file in the target's ELF format and is built using the target's RTEMS C compiler. The **rtems-syms** command searches for the C compiler under the prefix this command is installed under or the system path. If the target's C compiler is not located in either of these paths use the option **-c** or **--cc** to specify the path to the compiler.

The **rtems-syms** command loads the base kernel image's ELF file and reads the global or public symbols, creates a temporary C file and then compiles it using the target's RTEMS C compiler. The command automatically detects the architecture from the base kernel image's ELF file and uses it to create the C compiler's name. The option **-E** or **--exec-prefix** can be used to override the executable prefix used.

It is important to supply suitable C compiler flags (**cflags**) that match the kernel image's so the symbol table can be linked or loaded.

11.2.2 2-Pass Linking

2-Pass linking is used to embed a symbol table in a base kernel image. The first link pass is a normal RTEMS kernel link process. The link output is passed to the **rtems-syms** command and the **-e** or **--embed** option is used. The symbol table object file created by **rtems-syms** is added to the linker command used in the first pass to create the second pass. The address map will change between the first pass and second pass without causing a problem, the symbol table embedded in the second link pass will adjust the symbol addresses to match.

11.2.3 Command

rtems-syms [options] kernel

- V, --version**
Display the version information and then exit.
- v, --verbose**
Increase the verbose level by 1. The option can be used more than once to get more detailed trace and debug information.
- w, --warn**
Enable build warnings. This is useful when debugging symbol table generation.
- k, --keep**
Do not delete temporary files on exit, keep them.
- e, --embed**
Create a symbol table that can be embedded in the base kernel image using a 2-pass link process.
- S, --symc**
Specify the symbol's C source file. The default is to use a temporary file name.
- o, --output**
Specify the ELF output file name.
- m, --map**
Create a map file using the provided file name.
- C, --cc**
Specify the C compile executable file name. The file can be absolute and no path is search or relative and the environment's path is searched.
- E, --exec-prefix**
Specify the RTEMS tool prefix. For example for RTEMS 5 and the SPARC architecture the prefix is sparc-rtems5.
- c, --cflags**
Specify the C compiler flags used to build the symbol table with. These should be the same or compatible with the flags used to build the RTEMS kernel.
- , -h**
Reort the usage help.

11.2.4 Examples

Create a dynamlically loaded symbol table for the minimum.exe sample program for the i386/pc686 BSP:

```
1 $ rtems-syms -o ms.o i386-rtems5/c/pc686/testsuites/samples/minimum/minimum.exe
2 $ file ms.o
3 ms.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

Run the same command, this time create a map file:

```

1 $ rtems-syms -o ms.o -m ms.map i386-rtems5/c/pc686/testsuites/samples/minimum/minimum.exe
2 $ head -10 ms.map
3 RTEMS Kernel Symbols Map
4 kernel: i386-rtems5/c/pc686/testsuites/samples/minimum/minimum.exe
5
6 Globals:
7 No.  Index Scope      Type      SHNDX  Address      Size      Name
8   0    931 STB_GLOBAL STT_OBJECT    11 0x0012df08      4 BSPBaseBaud (minimum.exe)
9   1   1124 STB_GLOBAL STT_OBJECT    11 0x0012d894      4 BSPPrintkPort (minimum.exe)
10  2    836 STB_GLOBAL STT_FUNC      1 0x00104b00     302 BSP_dispatch_isr (minimum.
11 ↪exe)
12  3   1156 STB_GLOBAL STT_FUNC      1 0x001082d0      92 BSP_install_rtems_shared_irq_
13 ↪handler (minimum.exe)
14  4    876 STB_GLOBAL STT_FUNC      1 0x00106500     138 BSP_outch (minimum.exe)

```

Run the same command with a raise verbose level to observe the stages the command performs:

```

1 $ rtems-syms -vvv -o ms.o i386-rtems5/c/pc686/testsuites/samples/minimum/minimum.exe
2 RTEMS Kernel Symbols 5.a72a462adc18
3 kernel: i386-rtems5/c/pc686/testsuites/samples/minimum/minimum.exe
4 cache:load-sym: object files: 1
5 cache:load-sym: symbols: 1043
6 symbol C file: /tmp/rld--X7paaa.c
7 symbol O file: ms.o
8 execute: i386-rtems5-gcc -O2 -c -o ms.o /tmp/rld--X7paaa.c
9 execute: status: 0

```

11.3 RTEMS Executable Infomation

The RTEMS Executable Information (**rtems-exeinfo**) command is an RTEMS tool to display some important parts of an RTEMS executable. RTEMS uses ELF as the final linker output and this tool displays useful RTEMS specific information held in the ELF executable. The tool does not replace tools like *readelf*, rather it focuses on reporting specific information RTEMS builds into the executable.

11.3.1 System Initialisation

Linker based system initialisation automatically lets RTEMS only link into the executable the system initialisation code referenced by the user's application or indirectly by RTEMS. The technique is a variation of the system initialisation process used in the FreeBSD kernel. It is also similar to the process used by the C++ language to run static constructors before entering *main* and destructors after *exit* is called.

Linker based system initialisation collects the address of referenced system initialisation functions in specially named sections. The system initialisation function's address is placed in a variable and the section attribute of the variable is set to a special section name. The linker is instructed via a special linker command file to collect these variables together to create a table. The start-up code in RTEMS loops over the table of addresses and calling each address or system initialisation function. Special section names given to the variables sorts the table placing the functions in a specific order.

A user places a call to an API function in their application and the linker pulls the API code from the RTEMS kernel library adding it to the executing. The API code the linker loads references the variable containing the address of the that API's system initialisation function. The linker loads the API system initialisation code into the executable to resolve the external reference created by the variable. If the user does not reference the API the variable is loaded into the executable and no reference to the API system initialisation code is made so it is not linked into the executable.

The design automatically creates a unique system initialisation table for each executable and the code in RTEMS does not change, there is no special build system tricks, or stub libraries.

The RTEMS Executable Information reports the tables created and you can use this information to debug any initialisation issues.

11.3.2 Command

The **rtems-exeinfo** tool reports RTEMS specific information about the executable. The *init* and *fini* tables print the symbol referenced for each table entry and if the symbol is from the C++ language it is demangled.

rtems-exeinfo

-V

Display the version information and then exit.

-v

Increase the verbose level by 1. The option can be used more than once to get more detailed trace and debug information.

- a**
Report all types of output data.
- I**
Report the `init` or initialisation table.
- F**
Report the `fini` or finalisation table.
- S**
Report the sections.
- , -h**
Reort the usage help.

11.3.3 Examples

Prints all reports for the `hello.exe` for the `i386/pc686` BSP:

```

1 $ rtems-exeinfo -a i386-rtems5/c/pc686/testsuites/samples/hello/hello.exe
2 RTEMS Executable Info 5.6f5cfada964c
3 rtems-exeinfo -a i386-rtems5/c/pc686/testsuites/samples/hello/hello.exe
4 exe: i386-rtems5/c/pc686/testsuites/samples/hello/hello.exe
5 Sections: 22
6 ----- addr: 0x00000000 0x00000000 size:      0 align:  0_
↳relocs:      0
7 .bss      WA----- addr: 0x00135760 0x0013b300 size:    23456 align: 32_
↳relocs:      0
8 .comment  ---MS----- addr: 0x00000000 0x00000083 size:     131 align:  1_
↳relocs:      0
9 .ctors    WA----- addr: 0x0013322c 0x00133234 size:       8 align:  4_
↳relocs:      0
10 .data     WA----- addr: 0x00133240 0x0013574c size:    9484 align: 32_
↳relocs:      0
11 .debug_abbrev ----- addr: 0x00000000 0x0003c5ce size:   247246 align:  1_
↳relocs:      0
12 .debug_aranges ----- addr: 0x00000000 0x00003a18 size:    14872 align:  8_
↳relocs:      0
13 .debug_info  ----- addr: 0x00000000 0x0032496d size:  3295597 align:  1_
↳relocs:      0
14 .debug_line  ----- addr: 0x00000000 0x0006606b size:    417899 align:  1_
↳relocs:      0
15 .debug_loc   ----- addr: 0x00000000 0x0003b704 size:    243460 align:  1_
↳relocs:      0
16 .debug_ranges ----- addr: 0x00000000 0x00008128 size:    33064 align:  1_
↳relocs:      0
17 .debug_str   ---MS----- addr: 0x00000000 0x0001a9d7 size:   109015 align:  1_
↳relocs:      0
18 .dtors      WA----- addr: 0x00133234 0x0013323c size:       8 align:  4_
↳relocs:      0
19 .eh_frame   -A----- addr: 0x0012b884 0x0013222c size:    27048 align:  4_
↳relocs:      0
20 .fini       -AE----- addr: 0x00127fdd 0x00127fe5 size:       8 align:  1_
↳relocs:      0
21 .init       -AE----- addr: 0x00127fd0 0x00127fdd size:     13 align:  1_
↳relocs:      0

```

(continues on next page)

(continued from previous page)

```

22  .rodata      -A----- addr: 0x00128000 0x0012b884 size:      14468 align: 32_
↪relocs:      0
23  .rtemsroset WA----- addr: 0x00127f94 0x00127fd0 size:         60 align:  4_
↪relocs:      0
24  .shstrtab   ----- addr: 0x00000000 0x000000c6 size:         198 align:  1_
↪relocs:      0
25  .strtab     ----- addr: 0x00000000 0x000068ca size:      26826 align:  1_
↪relocs:      0
26  .symtab     ----- addr: 0x00000000 0x00006290 size:      25232 align:  4_
↪relocs:      0
27  .text      WAE----- addr: 0x00100000 0x00127f91 size:     163729 align: 16_
↪relocs:      0
28
29 Init sections: 3
30 .ctors
31 0xffffffff RamSize
32 0x00000000 REG_EFLAGS
33 .init
34 0xfd81ebe8 no symbol
35 0xff86e8ff no symbol
36 0x00c2ffff no symbol
37 .rtemsroset
38 0x00100310 bsp_work_area_initialize
39 0x00100440 bsp_start_default
40 0x001160e0 _User_extensions_Handler_initialization
41 0x0010fe60 rtems_initialize_data_structures
42 0x0010fcf0 _RTEMS_tasks_Manager_initialization
43 0x0010f310 _Semaphore_Manager_initialization
44 0x0010ed90 _POSIX_Keys_Manager_initialization
45 0x00113af0 _Thread_Create_idle
46 0x0010c100 rtems_libio_init
47 0x0010bec0 rtems_filesystem_initialize
48 0x00100420 bsp_predriver_hook
49 0x0010bfb0 _Console_simple_Initialize
50 0x0010ff30 _IO_Initialize_all_drivers
51 0x0010fc10 _RTEMS_tasks_Initialize_user_tasks_body
52 0x0010ccb0 rtems_libio_post_driver
53
54 Fini sections: 2
55 .dtors
56 0xffffffff RamSize
57 0x00000000 REG_EFLAGS
58 .fini
59 0xfd815ee8 no symbol
60 0x0000c2ff no symbol

```

The Init section `.rtemsroset` shows the initialisation call order for the `hello.exe` sample application. The order is initialise the BSP work area, call the BSP start up, initialise the User extensions, initialise the RTEMS data structures, then call the various Classic API managers that have been linked into the application. Next any POSIX managers are initialisations, in this case the POSIX Keys manager which is used by the thread local storage (TLS) support. Finally the IO and file system is initialise followed by the drivers.

Print the Init section data for the `cdtest.exe` for the i386/pc686 BSP:

```

1 $ rtems-exeinfo -I i386-rtems5/c/pc686/testsuites/samples/cdtest/cdtest.exe
2 RTEMS Executable Info 5.6f5cfada964c
3 rtems-exeinfo -I i386-rtems5/c/pc686/testsuites/samples/cdtest/cdtest.exe
4 exe: i386-rtems5/c/pc686/testsuites/samples/cdtest/cdtest.exe
5 Init sections: 3
6 .ctors
7 0xffffffff RamSize
8 0x00100ea0 _GLOBAL__sub_I_rtems_test_name
9 0x001015d0 __gnu_cxx:.__freeres()
10 0x00101df0 __cxxabiv1:.__terminate(void (*)(*))
11 0x00102ac0 _GLOBAL__sub_I___cxa_get_globals_fast
12 0x00103260 std::nothrow
13 0x00000000 REG_EFLAGS
14 .init
15 0xfcb3dbe8 no symbol
16 0xff86e8ff no symbol
17 0x00c2ffff no symbol
18 .rtemsroset
19 0x001112c0 bsp_work_area_initialize
20 0x001113f0 bsp_start_default
21 0x001276c0 _User_extensions_Handler_initialization
22 0x00121260 rtems_initialize_data_structures
23 0x001210f0 _RTEMS_tasks_Manager_initialization
24 0x00120710 _Semaphore_Manager_initialization
25 0x0011ff70 _POSIX_Keys_Manager_initialization
26 0x001250d0 _Thread_Create_idle
27 0x0011d220 rtems_libio_init
28 0x0011cfe0 rtems_filesystem_initialize
29 0x001113d0 bsp_predriver_hook
30 0x0011d0d0 _Console_simple_Initialize
31 0x00121310 _IO_Initialize_all_drivers
32 0x00121010 _RTEMS_tasks_Initialize_user_tasks_body
33 0x0011ddd0 rtems_libio_post_driver

```

The C++ constructor section `.ctors` shows you the C++ static objects the RTEMS kernel will construct before calling `main`.

11.4 RTEMS BSP Builder

The RTEMS BSP Builder is an RTEMS developer tool to build RTEMS in ways users do not to test changes to RTEMS. RTEMS has large number of architectures, board support packages and configuration options. This tool provides a standard way to test a change.

11.4.1 Developer Workflows

There are a number of RTEMS developers each with a different view or expertise in RTEMS. Developers can work in the generic areas such as scheduling, file systems or the shell, or users can become developers adding a new BSP, or even a new port to a new architecture. A common approach for all these developers is to select a BSP and to work with that BSP. Developers working in a generic areas of RTEMS tend to select a BSP that has good simulator support with good debugging such as QEMU, while developers of a new BSP or a new port tend to work on target hardware. This type of development does not check the other architectures, BSP, and build options and a change may change the number of warnings or introduce build errors. It is important for the RTEMS project to have developers fix these issues before pushing the changes to the master repository to avoid breaking the code for other developers. It is best for a developer to resolve as many issues as they work on changes because coming back to a problem often proves difficult.

The RTEMS BSP Builder forms part of a developers workflow where patches are tested before being pushed to the repository.

11.4.2 Build Characteristics

Build characteristic are the various parts of a build that can varied changing what is built. RTEMS can vary builds based on:

1. Architecture
2. Board Support Package (BSP)
3. Build options
4. BSP Options

The BSP Builder provides a template of builds to try and reduce the possible combinations to something manageable. It is not realistic to build all possible combinations on a single machine in reasonable time.

The RTEMS BSP Builder specifies it builds in terms of:

1. Profiles
2. Architectures
3. BSPs
4. Builds

The RTEMS BSP Builder builds are created by user options that vary these parameters.

11.4.2.1 Profiles

A profile is named collection of architectures and board support packages. When the RTEMS BSP Builder is asked to build a specific profile it builds the BSPs in the specified architectures.

The default configuration provides standard profiles for *Tiers* (page 89). They are:

1. tier-1 (default)
2. tier-2
3. tier-3
4. tier-4

The everythings profile allows all BSPs to be built.

11.4.2.2 Builds

A build is a list of builds or a build set and each BSP in a profile, architecture of BSP is built with.

The default configuration provides standard builds based around the commonly varied configure options.

The builds are:

1. all (default)
2. tests
3. standard, also no-tests
4. debug
5. profiling
6. smp
7. smp-debug
8. posix
9. no-posix
10. posix-debug
11. posix-profiling
12. network
13. no-network
14. network-debug
15. smp-network
16. smp-network-debug

All Build

The all build is:

- debug
- profiling
- smp
- smp-debug
- posix
- no-posix
- posix-debug
- posix-profiling
- network
- no-network
- network-debug
- smp-network
- smp-network-debug

A build maps to specific configuration options. The mappings are:

debug	config:base, config:debug
profiling	config:base, config:profiling
smp	config:base, config:smp
smp-debug	config:base, config:smp, config:debug
posix	config:base, config:posix
no-posix	config:base, config:no-posix
posix-debug	config:base, config:posix, config:debug
posix-profiling	config:base, config:posix, config:profiling
network	config:base, config:network
no-network	config:base, config:no-network
network-debug	config:base, config:network, config:debug
smp-network	config:base, config:smp, config:network
smp-network-debug	config:base, config:smp, config:network, config:debug

11.4.3 Build Configurations

Build configurations are configure options. These are mapped to the various builds. The configurations are:

base	--target=@ARCH@-rtems@RTEMS_VERSION@ --enable-rtemsbsp=@BSP@ --prefix=@PREFIX@
tests	--enable-tests
debug	--enable-debug
no-debug	--disable-debug
profiling	--enable-profiling
no-profiling	--disable-profiling
smp	--enable-smp
no-smp	--disable-smp
posix	--enable-posix
no-posix	--disable-posix
network	--enable-networking
no-network	--disable-networking

11.4.4 Performance

The RTEMS BSP Builder is designed to extract the maximum performance from your hardware when building RTEMS. The RTEMS build system is based on autoconf, automake and GNU make. Building consists of two phases:

1. Configuring
2. Building

The Configuring phase and the start of the Build phase runs autoconf's configure scripts. These execute as a single linear process and are not run in parallel even if you specify more than one job to make. The configure part of a build is approximately 30% of the total time and higher if building the tests. Performing a single build at a time will not fully utilize a multi-core machine because of the large amount of time the system is idle.

The RTEMS BSP Builder can run more than one build in parallel. A build can also request make run its build with more than one job. The `--jobs` option lets a user specify the number of build jobs to run at once and the number of make jobs each build runs with. Together these options can fully load a system and can overload a machine.

Tuning the best ratio of build jobs to make jobs requires running some builds and observing the system's performance. If the build job count is too low the system will show idle periods and if you have too many build jobs with too many make jobs the system will have too many processing running and the operating system's overheads in administering too processes at once lowers the overall performance.

A fast eight core machine where the operating system shows sixteen cores can support a build option of `--jobs=5/10`. The machine will be fully loaded the average build time is around 18 seconds.

The type of build selected effects the optimum jobs option. For example building the tests changes the percentage of time spent configuring compared to building so the make jobs parameter becomes a dominant factor. Lowering the make jobs value avoids having too many active processes running at once.

11.4.5 Command

rtems-bsp-builder [options]

-?

Display a compact help.

-h, --help

Display the full help.

--prefix

Prefix to pass to configure when building a BSP.

--rtems-tools

The path the RTEMS tools such as the C compiler. This option avoid polluting your path. This path is to the tool's prefix used to build and install the tools and not exact path to an executable.

--rtems

The path the RTEMS source tree to build.

--build-path

The path to build the BSP and place the build output. This can be any path and away from your current directory or the RTEMS source code. The storage does not need to be fast like an SSD.

--log

The log file.

--config-report

Print a configuration report and exit.

--warnings-report

Create a warnings report once all builds have finished.

--stop-on-error

Stop the build on an error. The default is to build all the builds for a profile.

--no-clean

Do not remove the build once finished. This option lets you inspect the built output. The amount of output can be large and disks can fill with this option.

--profiles

Build the comma separated list of profiles. The default is tier-1.

--arch

A comma separated list of architectures to build using the selected build.

--bsp

A comma separated list of BSPs to build where a BSP is of the format arch/bsp using the selected build.

--build

The build to be used. The default is all. See --config-report for a list of valid builds.

--jobs

The jobs options where the format is build-jobs/make-jobs. The default is 1/num-cores where num-cores is the operating system reported number of cores.

--dry-run

Do not do the actual builds just show what would be built.

11.4.5.1 Examples

The following is a *tier-1* profile build of *all* on a machine where all the source and tools are located on fast SSD disks and the build happens on a spinning disk mounted under *build*. The build uses a development source tree that is bootstrapped and ready to build. The source can have local patches that need to be regression tested:

```

1 $ /opt/rtems/5/bin/rtems-bsp-builder --build-path=/build/rtems \
2     --rtems-tools=/opt/work/rtems/5 \
3     --rtems=/opt/work/chris/rtems/kernel/rtems.git \
4     --profiles=tier-1 \
5     --jobs=5/10
6 RTEMS Tools Project - RTEMS Kernel BSP Builder, 5.not_released
7 Profile(s): tier-1
8 Cleaning: bsp-builds
9 [ 1/655] arm/altcy cv_devkit (debug)           Start
10 [ 1/655] arm/altcy cv_devkit (debug)           Creating: bsp-builds/arm/
    ↳ altcy cv_devkit.debug
11 [ 2/655] arm/altcy cv_devkit (no-posix)        Start
12 [ 2/655] arm/altcy cv_devkit (no-posix)        Creating: bsp-builds/arm/
    ↳ altcy cv_devkit.no-posix
13 [ 3/655] arm/altcy cv_devkit (posix)           Start
14 [ 1/655] arm/altcy cv_devkit (debug)           Configuring
15 [ 3/655] arm/altcy cv_devkit (posix)           Creating: bsp-builds/arm/
    ↳ altcy cv_devkit.posix
16 [ 2/655] arm/altcy cv_devkit (no-posix)        Configuring
17 [ 4/655] arm/altcy cv_devkit (posix-debug)     Start
18 [ 1/655] arm/altcy cv_devkit (debug)           Building
19 [ 3/655] arm/altcy cv_devkit (posix)           Configuring
20 [ 4/655] arm/altcy cv_devkit (posix-debug)     Creating: bsp-builds/arm/
    ↳ altcy cv_devkit.posix-debug
21 [ 2/655] arm/altcy cv_devkit (no-posix)        Building
22 [ 5/655] arm/altcy cv_devkit (posix-profiling) Start
23 [ 4/655] arm/altcy cv_devkit (posix-debug)     Configuring
24 [ 3/655] arm/altcy cv_devkit (posix)           Building
25 ....
26 [654/655] sparc/ngmp (posix-profiling)         PASS
27 [654/655] sparc/ngmp (posix-profiling)         Warnings:0 exes:0 objs:0
    ↳ libs:0
28 [654/655] sparc/ngmp (posix-profiling)         Finished (duration:0:01:49.
    ↳ 002189)
29 [654/655] sparc/ngmp (posix-profiling)         Status: Pass: 655 Fail:
    ↳ 0 (configure:0 build:0)
30 [655/655] sparc/ngmp (profiling)              PASS
31 [655/655] sparc/ngmp (profiling)              Warnings:0 exes:0 objs:0
    ↳ libs:0
32 [655/655] sparc/ngmp (profiling)              Finished (duration:0:01:260.
    ↳ 002098)
33 [655/655] sparc/ngmp (profiling)              Status: Pass: 655 Fail:
    ↳ 0 (configure:0 build:0)
34 [651/655] sparc/ngmp (no-posix)               Cleaning: bsp-builds/sparc/
    ↳ ngmp.no-posix

```

(continues on next page)

(continued from previous page)

```

35 [652/655] sparc/ngmp (posix) Cleaning: bsp-builds/sparc/
    ↪ngmp.posix
36 [653/655] sparc/ngmp (posix-debug) Cleaning: bsp-builds/sparc/
    ↪ngmp.posix-debug
37 [654/655] sparc/ngmp (posix-profiling) Cleaning: bsp-builds/sparc/
    ↪ngmp.posix-profiling
38 [655/655] sparc/ngmp (profiling) Cleaning: bsp-builds/sparc/
    ↪ngmp.profiling
39 Total: Warnings:31689 exes:6291 objs:793839 libs:37897
40 Failures:
41 No failure(s)
42 Average BSP Build Time: 0:00:18.165000
43 Total Time 3:41:48.075006
44 Passes: 655 Failures: 0

```

To build a couple of BSPs you are interested in with tests:

```

1 $ /opt/rtems/5/bin/rtems-bsp-builder --build-path=/build/rtems \
2   --rtems-tools=/opt/work/rtems/5 \
3   --rtems=/opt/work/chris/rtems/kernel/rtems.git \
4   ----log=lpc-log \
5   --bsp=arm/lpc2362,arm/lpc23xx_tli800 \
6   --build=tests \
7   --jobs=5/12
8 RTEMS Tools Project - RTEMS Kernel BSP Builder, 5.not_released
9 BSPS(s): arm/lpc2362, arm/lpc23xx_tli800
10 Cleaning: bsp-builds
11 [1/2] arm/lpc2362 (tests) Start
12 [1/2] arm/lpc2362 (tests) Creating: bsp-builds/arm/lpc2362.tests
13 [2/2] arm/lpc23xx_tli800 (tests) Start
14 [2/2] arm/lpc23xx_tli800 (tests) Creating: bsp-builds/arm/lpc23xx_tli800.tests
15 [1/2] arm/lpc2362 (tests) Configuring
16 [2/2] arm/lpc23xx_tli800 (tests) Configuring
17 [1/2] arm/lpc2362 (tests) Building
18 [2/2] arm/lpc23xx_tli800 (tests) Building
19 [1/2] arm/lpc2362 (tests) FAIL
20 [1/2] arm/lpc2362 (tests) Warnings:74 exes:58 objs:1645 libs:74
21 [1/2] arm/lpc2362 (tests) Finished (duration:0:01:31.708252)
22 [1/2] arm/lpc2362 (tests) Status: Pass: 0 Fail: 2 (configure:0 build:2)
23 [2/2] arm/lpc23xx_tli800 (tests) FAIL
24 [2/2] arm/lpc23xx_tli800 (tests) Warnings:74 exes:51 objs:1632 libs:74
25 [2/2] arm/lpc23xx_tli800 (tests) Finished (duration:0:01:31.747582)
26 [2/2] arm/lpc23xx_tli800 (tests) Status: Pass: 0 Fail: 2 (configure:0 build:2)
27 [1/2] arm/lpc2362 (tests) Cleaning: bsp-builds/arm/lpc2362.tests
28 [2/2] arm/lpc23xx_tli800 (tests) Cleaning: bsp-builds/arm/lpc23xx_tli800.tests
29 Total: Warnings:74 exes:109 objs:3277 libs:148
30 Failures:
31 1 tests arm/lpc2362 build:
32   configure: /opt/work/chris/rtems/kernel/rtems.git/configure --target\
33   =arm-rtems5 --enable-rtemsbsp=lpc2362 --prefix=/opt/rtems/5\
34   --enable-tests
35   error: ld/collect2:0 error: math.exe section '.rodata' will not fit
36   in region 'ROM_INT'; region 'ROM_INT' overflowed by 7284 bytes
37
38 2 tests arm/lpc23xx_tli800 build:
39   configure: /opt/work/chris/rtems/kernel/rtems.git/configure --target\

```

(continues on next page)

(continued from previous page)

```
40     =arm-rtems5 --enable-rtemsbsp=lpc23xx_tli800\  
41     --prefix=/opt/rtems/5 --enable-tests  
42     error: ld/collect2:0 error: math.exe section '.text' will not fit in  
43         region 'ROM_INT'; region 'ROM_INT' overflowed by 13972 bytes  
44  
45 Average BSP Build Time: 0:00:46.658257  
46 Total Time 0:01:33.316514  
47 Passes: 0   Failures: 2
```

The summary report printed shows both BSP builds failed with the error detail shown. In this case both are linker related errors where the test do not fit into the target's available resources.

11.5 RTEMS Tester and Run

The RTEMS Tester is a test tool that provides a command line interface to run test executable on supported targets. The tool provides back-end support for common simulators, debuggers and boot loaders. Board support package (BSP) configurations for RTEMS are provided and can be used to run all the tests in the RTEMS test suite. The tool and it's framework is not specific to RTEMS and can be configured to run any suitable application.

RTEMS is an embedded operating system and is cross-compiled on a range of host machines. The executables run on the target hardware and this can vary widely from open source simulators, commercial simulators, debuggers with simulators, debuggers with hardware specific pods and devices to target boot loaders. Testing RTEMS requires the cross-compiled test executable is transferred to the target hardware, executed and the output captured and returned to the test host where it is analyzed to determine the test result.

Running the RTEMS tests on your target is very important. It provides you with a traceable record showing that your RTEMS version and its tools are working at the level the RTEMS development team expect when releasing RTEMS. Being able to easily run the tests and verify the results is critical in maintaining a high standard.

11.5.1 Available BSP testers

You can list the available BSP testers with:

```

1 $ rtems-test --list-bsps
2 arm920
3 beagleboardxm
4 beagleboneblack
5 jmr3904-run
6 jmr3904
7 mcf5235
8 pc
9 psim-run
10 psim
11 realview_pbx_a9_qemu
12 sis-run
13 sis
14 xilinx_zynq_a9_qemu
15 xilinx_zynq_a9_qemu_smp
16 xilinx_zynq_zc706
17 xilinx_zynq_zc706_qemu
18 xilinx_zynq_zedboard

```

Note: The list is growing all the time and if your BSP is not supported we encourage you to add it and submit the configuration back to the project.

Some of the BSPs may appear more than once in the list. These are aliased BSP configurations that may use a different back-end. An example is the erc32 BSP. There is the erc32 tester which uses the GDB back-end and the erc32-run tester which uses the run command for erc32. We will show how to use **rtems-test** command with the erc32 BSP because it is easy to build an use.

11.5.2 Building RTEMS Tests

Build the RTEMS Kernel (See *RTEMS Kernel* (page 75)) by cloning the repository, running the bootstrap procedure, building and finally installing the kernel. Be sure to enable tests by using `--enable-tests` option with `configure` after running bootstrap.

```
1 $ ../../rtems.git/configure --target=sparc-rtems5 \
2                               --enable-tests --enable-rtemsbsp=erc32
3 $ make
```

Add the `-j` option to the `make` command with the number of cores to run a parallel build.

Building all the tests takes time and it uses more disk so be patient. When finished all the tests will have been built. Some BSPs may require a post-build process to be run on the RTEMS ELF executable to create an image suitable for execution. This can be built into the configuration script and the tester will perform a pre-test command to convert the executable to a suitable format for your target.

Before running all the tests it is a good idea to run the hello test. The hello test is an RTEMS version of the classic “Hello World” example and running it shows you have a working tool chain and build of RTEMS ready to run the tests. Using the run with the ERC32 BSP the command is:

```
1 $ sparc-rtems5-run sparc-rtems5/c/erc32/testsuites/samples/hello/hello.exe
2
3 *** BEGIN OF TEST HELLO WORLD ***
4 Hello World
5 *** END OF TEST HELLO WORLD ***
```

The `run` command is the GDB simulator without the GDB part.

Running the example using SIS:

```
1 $ sparc-rtems5-sis sparc-rtems5/c/erc32/testsuites/samples/hello/hello.exe
2 SIS - SPARC/RISCV instruction simulator 2.20, copyright Jiri Gaisler 2019
3 Bug-reports to jiri@gaisler.se
4 ERC32 emulation enabled
5
6 Loaded sparc-rtems5/c/erc32/testsuites/samples/hello.exe, entry 0x02000000
7
8 sis> run
9
10
11 *** BEGIN OF TEST HELLO WORLD ***
12 *** TEST VERSION: 5.0.0.c6d8589bb00a9d2a5a094c68c90290df1dc44807
13 *** TEST STATE: EXPECTED-PASS
14 *** TEST BUILD: RTEMS_POSIX_API
15 *** TEST TOOLS: 7.5.0 20191114 (RTEMS 5, RSB 83fa79314dd87c0a8c78fd642b2cea3138be8dd6, ↵
   ↵Newlib 3e24fbf6f)
16 Hello World
17
18 *** END OF TEST HELLO WORLD ***
19
20
21 *** FATAL ***
22 fatal source: 0 (INTERNAL_ERROR_CORE)
23 fatal code: 5 (INTERNAL_ERROR_THREAD_EXITED)
```

(continues on next page)

(continued from previous page)

```

24 RTEMS version: 5.0.0.c6d8589bb00a9d2a5a094c68c90290df1dc44807
25 RTEMS tools: 7.5.0 20191114 (RTEMS 5, RSB 83fa79314dd87c0a8c78fd642b2cea3138be8dd6, ↵
    ↵Newlib 3e24fbf6f)
26 executing thread ID: 0x08a010001
27 executing thread name: UI1
28 cpu 0 in error mode (tt = 0x101)
29     116401 02009ae0: 91d02000    ta 0x0
30
31 sis> q

```

The examples can also be run using GDB with SIS as the backend. SIS can be connected to gdb through a network socket using the gdb remote interface.

Either start SIS with `-gdb`, or issue the `gdb` command inside SIS, and connect gdb with target remote:1234. The default port is 1234, the port can be changed using the `-port` option.

Open a terminal and issue the command:

```

1 $ sparc-rtems5-sis -gdb
2 SIS - SPARC/RISC-V instruction simulator 2.20, copyright Jiri Gaisler 2019
3 Bug-reports to jiri@gaisler.se
4 ERC32 emulation enabled
5
6 gdb: listening on port 1234

```

Now open another terminal and issue the command:

```

1 $ sparc-rtems5-gdb sparc-rtems5/c/erc32/testsuites/samples/hello/hello.exe
2 GNU gdb (GDB) 8.3
3 Copyright (C) 2019 Free Software Foundation, Inc.
4 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5 This is free software: you are free to change and redistribute it.
6 There is NO WARRANTY, to the extent permitted by law.
7 Type "show copying" and "show warranty" for details.
8 This GDB was configured as "--host=x86_64-linux-gnu --target=sparc-rtems5".
9 Type "show configuration" for configuration details.
10 For bug reporting instructions, please see:
11 <http://www.gnu.org/software/gdb/bugs/>.
12 Find the GDB manual and other documentation resources online at:
13     <http://www.gnu.org/software/gdb/documentation/>.
14
15 For help, type "help".
16 Type "apropos word" to search for commands related to "word"...
17 Reading symbols from sparc-rtems5/c/erc32/testsuites/samples/hello.exe...
18 (gdb) target remote:1234

```

The target `remote:1234` will tell gdb to connect to the sis simulator. After this command the output of the first terminal will change to

```

1 $ sparc-rtems5-sis -gdb
2 SIS - SPARC/RISC-V instruction simulator 2.20, copyright Jiri Gaisler 2019
3 Bug-reports to jiri@gaisler.se
4 ERC32 emulation enabled
5
6 gdb: listening on port 1234 connected

```

Before running the executable, it must be loaded, this is done using the load command in gdb, and to run, issue continue command.

```

1 $ sparc-rtems5-gdb sparc-rtems5/c/erc32/testsuites/samples/hello/hello.exe
2 GNU gdb (GDB) 8.3
3 Copyright (C) 2019 Free Software Foundation, Inc.
4 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5 This is free software: you are free to change and redistribute it.
6 There is NO WARRANTY, to the extent permitted by law.
7 Type "show copying" and "show warranty" for details.
8 This GDB was configured as "--host=x86_64-linux-gnu --target=sparc-rtems5".
9 Type "show configuration" for configuration details.
10 For bug reporting instructions, please see:
11 <http://www.gnu.org/software/gdb/bugs/>.
12 Find the GDB manual and other documentation resources online at:
13   <http://www.gnu.org/software/gdb/documentation/>.
14
15 For help, type "help".
16 Type "apropos word" to search for commands related to "word"...
17 Reading symbols from sparc-rtems5/c/erc32/testsuites/samples/hello.exe...
18 (gdb) target remote:1234
19 Remote debugging using :1234
20 0x00000000 in ?? ()
21 (gdb) load
22 Loading section .text, size 0x17170 lma 0x2000000
23 Loading section .rtemsroset, size 0x40 lma 0x2017170
24 Loading section .data, size 0x600 lma 0x20181c0
25 Start address 0x2000000, load size 96176
26 Transfer rate: 4696 KB/sec, 270 bytes/write.
27 (gdb) continue
28 Continuing.
```

You can see your executable running in the first terminal.

```

1 SIS - SPARC/RISCV instruction simulator 2.20, copyright Jiri Gaisler 2019
2 Bug-reports to jiri@gaisler.se
3
4 ERC32 emulation enabled
5
6 gdb: listening on port 1235 connected
7 X2000000,0:#40
8
9
10 *** BEGIN OF TEST HELLO WORLD ***
11 *** TEST VERSION: 5.0.0.c6d8589bb00a9d2a5a094c68c90290df1dc44807
12 *** TEST STATE: EXPECTED-PASS
13 *** TEST BUILD: RTEMS_POSIX_API
14 *** TEST TOOLS: 7.5.0 20191114 (RTEMS 5, RSB 83fa79314dd87c0a8c78fd642b2cea3138be8dd6, ↵
   ↵Newlib 3e24fbf6f)
15 Hello World
16
17 *** END OF TEST HELLO WORLD ***
18
19 ^Csis> q
```

For more information on the sis simulator refer to this doc: <https://gaisler.se/sis/sis.pdf>

The command r is used to debug set break points before issuing the GDB run command.

There are currently close to 500 separate tests and you can run them all with a single RTEMS Tester command.

11.5.3 Running the Tests

The **rtems-test** command line accepts a range of options. These are discussed later in the manual. Any command line argument without a **-** prefix is a test executable. You can pass more than one executable on the command line. If the executable is a path to a directory the directories under that path are searched for any file with a **.exe** extension. This is the default extension for RTEMS executables built within RTEMS.

To run the erc32 tests enter the following command from the top of the erc32 BSP build tree:

```

1 $ ~/development/rtems/test/rtems-tools.git/tester/rtems-test \
2     --log=log_erc32_run \
3     --rtems-bsp=erc32-run \
4     --rtems-tools=$HOME/development/rtems/5 \
5         sparc-rtems5/c/erc32/testsuites/samples
6 RTEMS Testing - Tester, 5.not_released
7 [ 1/13] p:0 f:0 u:0 e:0 I:0 B:0 t:0 i:0 | sparc/erc32: base_sp.exe
8 [ 2/13] p:0 f:0 u:0 e:0 I:0 B:0 t:0 i:0 | sparc/erc32: capture.exe
9 [ 3/13] p:0 f:0 u:0 e:0 I:0 B:0 t:0 i:0 | sparc/erc32: cdtest.exe
10 [ 4/13] p:0 f:0 u:0 e:0 I:0 B:0 t:0 i:0 | sparc/erc32: fileio.exe
11 [ 5/13] p:2 f:0 u:0 e:0 I:0 B:0 t:0 i:0 | sparc/erc32: hello.exe
12 [ 6/13] p:2 f:0 u:0 e:0 I:0 B:0 t:0 i:0 | sparc/erc32: cxx_iostream.exe
13 [ 8/13] p:2 f:0 u:0 e:0 I:0 B:0 t:2 i:0 | sparc/erc32: minimum.exe
14 [ 7/13] p:2 f:0 u:0 e:0 I:0 B:0 t:2 i:0 | sparc/erc32: loopback.exe
15 [ 9/13] p:3 f:0 u:0 e:0 I:0 B:0 t:3 i:0 | sparc/erc32: nsecs.exe
16 [10/13] p:3 f:0 u:0 e:0 I:0 B:0 t:3 i:0 | sparc/erc32: paranoia.exe
17 [11/13] p:4 f:0 u:0 e:0 I:0 B:0 t:3 i:0 | sparc/erc32: pppd.exe
18 [12/13] p:6 f:0 u:0 e:0 I:0 B:0 t:3 i:0 | sparc/erc32: ticker.exe
19 [13/13] p:6 f:0 u:0 e:0 I:0 B:0 t:3 i:0 | sparc/erc32: unlimited.exe
20 Passed:          7
21 Failed:          0
22 User Input:      0
23 Expected Fail:   0
24 Indeterminate:   0
25 Benchmark:       0
26 Timeout:         5
27 Invalid:         1
28 Total:          13
29 Average test time: 0:00:27.963000
30 Testing time      : 0:06:03.519012

```

- The RTEMS Tester's test command. In this example we are using an absolute path.
- The **--log** option sends the output to a log file. By default only failed tests log the complete output.
- Select the erc32 BSP and use GDB.
- Path to the RTEMS tools so GDB can be found.
- Path to the erc32 BSP built with all tests to run. If you add subdirectories to the path specific tests can be run.
- The output has been shortened so it fits nicely here.

- The test results shows passes, fails, timeouts, and invalid results. In this run 13 tests passed and 5 tests timed out and 1 is invalid. The timeouts are probably due to the tests not having enough execute time to complete. The default timeout is 180 seconds and some of the interrupt tests need longer. The amount of time depends on the performance of your host CPU running the simulations.
- The output shows the average time per test and the total time taken to run all the tests.
- If the path to the testsuites was put to `sparc-rtems5/c/erc32/testsuites` instead of `sparc-rtems5/c/erc32/testsuites/samples` then all the executables would have been tested and not just those in samples.

This BSP requires the `--rtems-tools` option because the SPARC GDB is the `sparc-rtems4.11-gdb` command that is part of the RTEMS tools. Not every BSP will require this option so you will need to check the specifics of the BSP configuration to determine if it is needed.

The output you see is each test starting to run. The `rtems-test` command by default runs multiple tests in parallel so you will see a number start quickly and then new tests start as others finish. The output shown here is from an 8 core processor so the first 8 are started in parallel and the status shows the order in which they actually started, which is not 1 to 8.

The test start line shows the current status of the tests. The status reported is when the test starts and not the result of that test. A fail, timeout or invalid count changing means a test running before this test started failed, not the starting test. The status here has 7 tests passed, no failures, 5 timeouts and 1 invalid test.

```
1 [ 5/13] p:2 f:0 u:0 e:0 I:0 B:0 t:0 i:0 | sparc/erc32: hello.exe
```

- [5/13] indicates the test number, in this case test 5 of 13 tests.
- p is the passed test count (2 in this case)
- f is the failed test count (0 in this case)
- u is the count for test marked as “user-input” as they expect input from user
- e is the expected-fail count (tests that are expected to fail)
- I is the count for tests the results of which are indeterminate
- B is the count for benchmarked tests
- t is the timeout test count
- i is the invalid test count.
- sparc/erc32 is the architecture and BSP names.
- hello.exe is the executable name.

The test log records all the tests and results. The logging mode by default only provides the output history if a test fails, times out, or is invalid. The time taken by each test is also recorded.

The tests must complete in a specified time or the test is marked as timed out. The default timeout is 3 minutes and can be globally changed using the `--timeout` command line option. The time required to complete a test can vary. When simulators are run in parallel the time taken depends on the specifics of the host machine being used. A test per core is the most stable method even though more tests can be run than available cores. If your machine needs longer or you are using a VM you may need to lengthen the timeout.

11.5.4 Test Status

Tests can be marked with one of the following:

- Pass
- Fail
- User-input
- Expected-fail
- Indeterminate
- Benchmark
- Timeout
- Invalid

The RTEMS console or stdout output from the test is needed to determine the result of the test.

11.5.4.1 Pass

A test passes if the start and end markers are seen in the test output. The start marker is *** and the end mark is *** END OF TEST. All tests in the RTEMS test suite have these markers.

11.5.4.2 Fail

A test fails if the start marker is seen and there is no end marker.

11.5.4.3 User-input

A test marked as “user-input” as it expects input from user

11.5.4.4 Expected-fail

A test that is expected to fail.

11.5.4.5 Indeterminate

A test the results of which are indeterminate.

11.5.4.6 Benchmark

A benchmarked test.

11.5.4.7 Timeout

If the test does not complete within the timeout setting the test is marked as having timed out.

11.5.4.8 Invalid

If no start marker is seen the test is marked as invalid. If you are testing on real target hardware things can sometimes go wrong and the target may not initialize or respond to the debugger in an expected way.

11.5.5 Logging

The following modes of logging are available:

- All (all)
- Failures (failures)
- None (none)

The mode is controlled using the command line option `--log-mode` using the values listed above.

11.5.5.1 All

The output of all tests is written to the log.

11.5.5.2 Failures

The output of the all tests that do not pass is written to the log.

11.5.5.3 None

No output is written to the log.

The output is tagged so you can determine where it comes from. The following is the complete output for the In Memory File System test `imfs_fslink.exe` running on a Coldfire MCF5235 using GDB and a BDM pod:

```

1 [ 11/472] p:9  f:0  t:0  i:1  | m68k/mcf5235: imfs_fslink.exe
2 > gdb: ....bin/m68k-rtems4.11-gdb -i=mi --nx --quiet ../imfs_fslink.exe
3 > Reading symbols from ....fstests/imfs_fslink/imfs_fslink.exe...
4 > done.
5 > target remote | m68k-bdm-gdbserver pipe 003-005
6 > Remote debugging using | m68k-bdm-gdbserver pipe 003-005
7 > m68k-bdm: debug module version 0
8 > m68k-bdm: detected MCF5235
9 > m68k-bdm: architecture CF5235 connected to 003-005
10 > m68k-bdm: Coldfire debug module version is 0 (5206(e)/5235/5272/5282)
11 > Process 003-005 created; pid = 0
12 > 0x00006200 in ?? ()
13 > thb *0xffe254c0
14 > Hardware assisted breakpoint 1 at 0xffe254c0
15 > continue
16 > Continuing.
17 ]
18 ]
19 ] External Reset

```

(continues on next page)

(continued from previous page)

```

20 ]
21 ] ColdFire MCF5235 on the BCC
22 ] Firmware v3b.1a.1a (Built on Jul 21 2004 17:31:28)
23 ] Copyright 1995-2004 Freescale Semiconductor, Inc. All Rights Reserved.
24 ]
25 ] Enter 'help' for help.
26 ]
27 > Temporary breakpoint
28 > 1, 0xffe254c0 in ?? ()
29 > load
30 > Loading section .text, size 0x147e0 lma 0x40000
31 > Loading section .data, size 0x5d0 lma 0x547e0
32 > Start address 0x40414, load size 85424
33 > Transfer rate: 10 KB/sec, 1898 bytes/write.
34 > b bsp_reset
35 > Breakpoint 2 at 0x41274: file ../shared/bspreset_loop.c, line 14.
36 > continue
37 > Continuing.
38 ] dBUG>
39 ]
40 ] *** FILE SYSTEM TEST ( IMFS ) ***
41 ] Initializing filesystem IMFS
42 ]
43 ]
44 ] *** LINK TEST ***
45 ] link creates hardlinks
46 ] test if the stat is the same
47 ] chmod and chown
48 ] unlink then stat the file
49 ] *** END OF LINK TEST ***
50 ]
51 ]
52 ] Shutting down filesystem IMFS
53 ] *** END OF FILE SYSTEM TEST ( IMFS ) ***
54 > Breakpoint
55 > 2, bsp_reset () at ../m68k/mcf5235/../../shared/bspreset_loop.c:14
56 > 14 {
57 Result: passed      Time: 0:00:10.045447

```

- GDB command line (Note: paths with ‘...’ have been shortened)
- Lines starting with > are from GDB’s console.
- Line starting with] are from the target’s console.
- The result with the test time.

11.5.6 Reporting

The RTEMS Tester supports output in a machine parsable format. This can be enabled using the options “--report-path” and “--report-format”. Currently, JSON output is supported using these options like so: ‘--report-path=“report” --report-format=json’

This will produce a file “report.json” that contains output equivalent to the “failure” logging mode.

11.5.7 Running Tests in Parallel

The RTEMS Tester supports parallel execution of tests by default. This only makes sense if the test back-end can run in parallel without resulting in resource contention. Simulators are an example of back-ends that can run in parallel. A hardware debug tool like a BDM or JTAG pod can manage only a single test at once so the tests need to be run one at a time.

The test framework manages the test jobs and orders the output in the log in test order. Output is held for completed tests until the next test to be reported has finished.

11.5.8 Command Line Help

The **rtems-test** command line accepts a range of options. You can review the available option by the **--help** option:

```

1 RTEMS Tools Project (c) 2012-2014 Chris Johns
2 Options and arguments:
3 --always-clean           : Always clean the build tree, even with an error
4 --debug-trace            : Debug trace based on specific flags
5 --dry-run                : Do everything but actually run the build
6 --force                  : Force the build to proceed
7 --jobs=[0..n,none,half,full] : Run with specified number of jobs, default: num CPUs.
8 --keep-going             : Do not stop on an error.
9 --list-bsps              : List the supported BSPs
10 --log file               : Log file where all build output is written to
11 --macros file[,file]    : Macro format files to load after the defaults
12 --no-clean              : Do not clean up the build tree
13 --quiet                  : Quiet output (not used)
14 --report-path            : Report output base path (file extension will be added)
15 --report-format          : Formats in which to report test results: json
16 --log-mode               : Log modes, failures (default),all,none
17 --rtems-bsp              : The RTEMS BSP to run the test on
18 --rtems-tools            : The path to the RTEMS tools
19 --target                 : Set the target triplet
20 --timeout                : Set the test timeout in seconds (default 180 seconds)
21 --trace                  : Trace the execution
22 --warn-all              : Generate warnings

```

11.6 RTEMS Boot Image

The RTEMS Boot Image (**rtems-boot-image**) command is an RTEMS tool to create disk images suitable for SD cards to boot RTEMS on a range of boards. The supported hosts are:

- FreeBSD
- Linux
- MacOS

The tool captures the specific details for a host operating system to create a bootable disk image as well as capturing the specific detail of the boards that are supported. The tool brings these detail together under a single command line interface that is portable across the supported hosts.

The boot image tool can:

- Create a disk image to boot an RTEMS executable
- Create a disk image to network boot an RTEMS executable
- Convert an RTEMS executable into the format a board's bootloader can load.

The disk images are suitable for booting a range of hardware that have media interfaces, such as an SD card. The default partition type is the Master Boot Record (MRB) and a single root DOS-FS partition is created.

11.6.1 Boot Loaders

The boot image tool supports the following boot loaders:

- U-boot

11.6.1.1 U-Boot

The U-Boot boards supported are:

- BeagleBone (*arm-ti-am335x_evm*)
- Zedboard (*arm-xilinx-zynq-common*)

These boards can be booted with executable and Flat Device Tree (FDT) blobs on disk or view a network if supported by the boards.

The boot image tool can create the following boot configurations for U-Boot:

- **Executable**

A kernel executable is copied in the disk image, loaded by U-Boot and control is passed to the kernel. A reset is performed if the load fails or the kernel returns control to U-Boot.

- **Executable and FDT**

A kernel executable and FDT blob are copied to the disk image, loaded by U-boot and control is passed to the kernel. A reset is performed if the load fails or the kernel returns control to U-Boot.

- **Network DHCP and Executable**

The board's network interface is initialised, a DHCP request made and a kernel image loaded using TFTP. The loaded kernel is passed control. A reset is performed if the load fails or the kernel returns control to U-Boot.

- **Network DHCP, Executable and FDT**

The board's network interface is initialised, a DHCP request made and a kernel image loaded using TFTP. The loaded kernel is passed control. A reset is performed if the load fails or the kernel returns control to U-Boot.

The FDT can be installed in and disk image and loaded from it on each boot.

- **Network Static IP and Executable**

The board's network interface is initialised with a static IP address and a kernel image loaded using TFTP. The loaded kernel is passed control. A reset is performed if the load fails or the kernel returns control to U-Boot.

- **Network Static IP, Executable and FDT**

The board's network interface is initialised with a static IP address and a kernel image loaded using TFTP. The loaded kernel is passed control. A reset is performed if the load fails or the kernel returns control to U-Boot.

The FDT can be installed in and disk image and loaded from it on each boot.

11.6.2 Hosts

The hosts each require specific set up to run the boot image builder. The tool creates special devices to access the image as a disk and runs file system partitioning and formatting tools. These tools typically require super user or root access. It is not good practice to run commands like this one as root and so the tool dispatches any specific command that needs higher privileges via sudo. If you see a password prompt please enter your password, not a root password if you have one configured.

11.6.2.1 FreeBSD

Install the sudo package. All commands used are in standard operating system paths and should not require any specific configurations.

11.6.2.2 Linux

The loop back kernel module needs to be loaded.

11.6.2.3 MacOS

All command used are part of the base OS. No external packages are required.

11.6.3 Configuration

The boot image tool is configured by the file `rtems-boot.ini` that is installed in the

11.6.4 Command

The **rtems-boot-image** tool creates a boot disk image for a specified board. The command line options are:

rtems-boot-image

-h, --help

Display the command line help.

-l, --log

Set the log file name. The default is `rtems-log-boot-image.txt`.

-v, --trace

Enable trace or debug logging.

-s IMAGE_SIZE, --image-size IMAGE_SIZE

Set the image size. The size can be in SI units of k, m, or g. The size needs to be something the host's partition and format tools will accept and it must be large enough to fit the root partition plus any alignments. The default is 64m.

-F FS_FORMAT, --fs-format FS_FORMAT

Specify type type of format. The supported formats are fat16 and fat32. The default format is fat16.

-S FS_SIZE, --fs-size FS_SIZE

Set the size of the first partition in the disk image. The partition need to be less than the size of the image plus the alignment. The default size is auto which will fill the image with the partition.

-A FS_ALIGN, --fs-align FS_ALIGN

Set the alignment of the first partition. The default is 1m.

-k KERNEL, --kernel KERNEL

Optionally provide a kernel image that is copied into the root partition of the disk image and loaded and run when the board boots. The file is an RTEMS executable in the ELF format which is converted to a format the boot loader can load.

-d FDT, --fdt FDT

Optionally provide a FDT blob that is copied into the root partition of the disk image and loaded when the board boots. If a kernel is provided or a kernel is loaded via a net boot a kernel boot with FDT is executabled. The file is an FDT blob created by the FDT compiler.

-f FILE, --file FILE

Optionally provide a file to be copied to the root partition of the disk image. This option can be provided more than once if more than one file needs to be installed.

--net-boot

Not used and will be removed.

--net-boot-dhcp

Configure a network boot using DHCP. The kernel will be loaded using TFTP and the file request can be specific by the `--net-boot-file` option.

--net-boot-ip NET_BOOT_IP

Configure a network boot using a static IP address. The kernel will be loaded using TFTP and the file request can be specific by the `--net-boot-file` option. A server IP needs to be specified using the `--net-boot-server`.

--net-boot-file NET_BOOT_FILE

Specify the kernel image file name requested using the TFTP protocol. The default is `rtems.img`.

--net-boot-fdt NET_BOOT_FDT

Optionally specify the file name of a FDT blob loaded using the TFTP protocol. If a net boot FDT file is provide the kernel will be executable with a suitable kernel and FDT boot command.

-U CUSTOM_UENV, **--custom-uenv** CUSTOM_UENV

Optionally provide a custom U-boot `uEnv.txt` file that is copied to into the root directory of the root partition of the disk image.

-b BOARD, **--board** BOARD

Specify the board the disk image is built for. The default board is `list` which lists the available board configurations.

--convert-kernel

Convert an RTEMS ELF executable into an image file the selected board's bootloader can load. This option does not create a disk image. The option can be used to create images that can be loaded when network booting.

--no-clean

If provided the build directory will not be removed after the disk image has been created.

-o OUTPUT, **--output** OUTPUT

The output file name for the image. If the **--convert-kernel** option is used the conversion is written as this file name and if it is not provided the output file is the built disk image.

paths [paths ...]

The required paths depend on the mix of other options.

If the **--convert-kernel** option is provided a single path to an RTEMS executable file is required. If this option is not provided the number of paths provided determine how they are processed.

If a single path a built U-boot directory is provided the board configuration will automatically find and pick up the first and second stage boot loader executables.

If two paths are provided they are paths to the first and second stage boot loader executables. This can be used with loader images they you have not built.

11.6.5 Examples

The examples show the output for FreeBSD. It may vary depending on your type of host, how it is configured and what is running.

If the board option is not provided a list of boards is displayed:

```

1 $ rtems-boot-image -o sd-card.img u-boot
2 RTEMS Tools - Boot Image, 5.0.not_released
3 Board list: bootloaders (1)
4   u-boot: 2
5     u-boot-beaglebone
6     u-boot-zedboard

```

Create a disk image from a built U-Boot sandbox:

```

1 $ rtems-boot-image -o sd-card.img -b u-boot-beaglebone u-boot
2 RTEMS Tools - Boot Image, 5.0.not_released
3 Create image: sd-card.img size 64m
4 Attach image to device: sd-card.img
5 Password:
6 Partition device: md0 as MBR
7 Format: /dev/md0s1 as fat16
8 Mount: /dev/md0s1
9 Install: MLO
10 Install: u-boot.img
11 Finished
12 Cleaning up

```

Create a 32M byte SD card image with the testsuite's hello world executable (hello.exe):

```

1 $ rtems-boot-image -o sd-card.img -b u-boot-beaglebone -s 32m -k hello.exe u-boot
2 RTEMS Tools - Boot Image, 5.0.not_released
3 Create image: sd-card.img size 32m
4 Attach image to device: sd-card.img
5 Password:
6 Partition device: md0 as MBR
7 Format: /dev/md0s1 as fat16
8 Mount: /dev/md0s1
9 Install: MLO
10 Install: u-boot.img
11 Install: hello.exe.img
12 Uenv template: uenv_exe
13 Install: uEnv.txt
14 Finished
15 Cleaning up

```

Build the same image using the first and second stage boot loaders:

```

1 $ rtems-boot-image -o sd-card.img -b u-boot-beaglebone -s 32m -k hello.exe MLO u-boot.img
2 RTEMS Tools - Boot Image, 5.0.not_released
3 Create image: sd-card.img size 32m
4 Attach image to device: sd-card.img
5 Password:
6 Partition device: md0 as MBR
7 Format: /dev/md0s1 as fat16
8 Mount: /dev/md0s1
9 Install: MLO
10 Install: u-boot.img
11 Install: hello.exe.img
12 Uenv template: uenv_exe
13 Install: uEnv.txt
14 Finished
15 Cleaning up

```

Install and load the TI standard FDT for the Beaglebone Black board with the LibBSD DHCP 01 test application:

```

1 $ rtems-boot-image -o sd-card.img -b u-boot-beaglebone -s 32m \
2   -k dhcpcd01.exe -d am335x-boneblack.dtb MLO u-boot.img
3 RTEMS Tools - Boot Image, 5.0.not_released
4 Create image: sd-card.img size 32m

```

(continues on next page)

(continued from previous page)

```

5 Attach image to device: sd-card.img
6 Password:
7 Partition device: md0 as MBR
8 Format: /dev/md0s1 as fat16
9 Mount: /dev/md0s1
10 Install: MLO
11 Install: u-boot.img
12 Install: dhcpcd01.exe.img
13 Install: am335x-boneblack.dtb
14 Uenv template: uenv_exe_fdt
15 Install: uEnv.txt
16 Finished
17 Cleaning up

```

Create a DHCP network boot image where the TFTP client requests `rtems.img`:

```

1 $ rtems-boot-image -o sd-card.img -b u-boot-beaglebone -s 32m \
2   --net-boot-dhcp MLO u-boot.img
3 RTEMS Tools - Boot Image, 5.0.not_released
4 Create image: sd-card.img size 32m
5 Attach image to device: sd-card.img
6 Password:
7 Partition device: md0 as MBR
8 Format: /dev/md0s1 as fat16
9 Mount: /dev/md0s1
10 Install: MLO
11 Install: u-boot.img
12 Uenv template: uenv_net_dhcp
13 Install: uEnv.txt
14 Finished
15 Cleaning up

```

Select a specific kernel image to load using TFTP and load a FDT blob from the SD card:

```

1 $ rtems-boot-image -o sd-card.img -b u-boot-beaglebone -s 32m \
2   --net-boot-dhcp --net-boot-file bbb1a.img \
3   -d am335x-boneblack.dtb MLO u-boot.img
4 RTEMS Tools - Boot Image, 5.0.not_released
5 Create image: sd-card.img size 32m
6 Attach image to device: sd-card.img
7 Password:
8 Partition device: md0 as MBR
9 Format: /dev/md0s1 as fat16
10 Mount: /dev/md0s1
11 Install: MLO
12 Install: u-boot.img
13 Install: am335x-boneblack.dtb
14 Uenv template: uenv_net_dhcp
15 Install: uEnv.txt
16 Finished
17 Cleaning up

```

Create an image where a specific kernel image and FDT blob is loaded using the TFTP protocol:

```

1 $ rtems-boot-image -o sd-card.img -b u-boot-beaglebone -s 32m \

```

(continues on next page)

(continued from previous page)

```
2  --net-boot-dhcp --net-boot-file bbb1a.img \  
3  --net-boot-fdt bbb/am335x-boneblack.dtb MLO u-boot.img  
4  RTEMS Tools - Boot Image, 5.0.not_released  
5  Create image: sd-card.img size 32m  
6  Attach image to device: sd-card.img  
7  Password:  
8  Partition device: md0 as MBR  
9  Format: /dev/md0s1 as fat16  
10 Mount: /dev/md0s1  
11 Install: MLO  
12 Install: u-boot.img  
13 Uenv template: uenv_net_dhcp_net_fdt  
14 Install: uEnv.txt  
15 Finished  
16 Cleaning up
```

11.7 RTEMS TFTP Proxy

The RTEMS TFTP Proxy (**rtems-tftp-proxy**) command is an RTEMS tool to simplify hardware testing using the RTEMS Test and Run commands. This command lets a test set up support a number of similarly configured boards running tests at the same time by proxying the TFTP session requests. The *TFTP and U-Boot* (page 173) section details the process to run a test executable on a network connected board.

The TFTP Proxy approach does not require any special modifications in a boot loader to work and works with any compliant TFTP boot client.

An identical SD card boot configuration can be used in similar board when a test set up has a number of similar boards. There is no need to specialize boot configurations. The TFTP proxy server identifies each board by MAC address.

A configuration file maps a board's MAC address to a TFTP server's IP address and port number. This provides a centralized means to partition hardware in a test rack between members of a team, continuous integration services or any other project demands.

The TFTP port number a proxied service runs with does not need to be the privileged TFTP port number removing the need to be root to run the RTEMS Test or Run commands. Only the TFTP Proxy needs to running as a privileged user. The RTEMS Test and Run commands lets you specified the TFTP port to bind too.

11.7.1 Operation

A network connected board with a suitable boot loader such as U-Boot is configured to boot using TFTP. The boot loader's configured TFTP server IP address is the address of the host computer running the TFTP Proxy server or the proxy. The TFTP Proxy runs as root or an administrator as it binds by default to the default TFTP port of 69.

A reset board sends a TFTP read request (RRQ) packet to the host machine running the TFTP proxy on the standard TFTP port (69). The proxy server searches the configuration data for a matching MAC address. A configuration match creates a session, forwarding the read request to the proxied IP and port.

The response from the proxied server identifies the remote session port number and the proxy server knows the board's client port number from the initial request. The proxy transfers the TFTP data transparently between the session's ports until the transfer finishes.

An example configuration is three different types of boards used for RTEMS kernel regression testing and application development.

The project has a continuous integration (CI) server on address 10.0.0.100 and two boards, a BeagleBone Black and Xilinx MicroZed board, are configured for testing. A developer on another host machine is using a RaspberryPi to develop an application. The configuration file is:

```

1 ;
2 ; Project Foo Test network.
3 ;
4 [default]
5 clients = bbb, uzed, rpi2
6
7 [bbb]
```

(continues on next page)

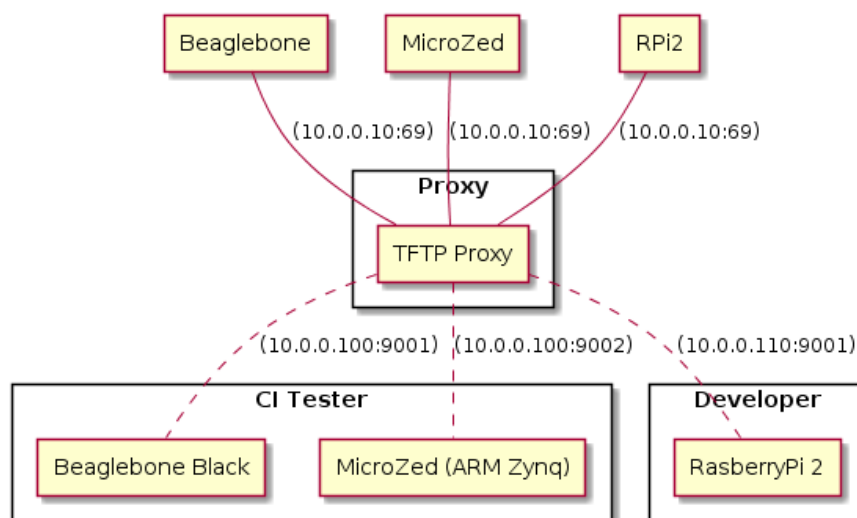


Fig. 1: RTEMS TFTP Proxy Test Lab

(continued from previous page)

```

8 mac = 1c:ba:8c:96:20:bc
9 host = 10.0.0.100:9001
10
11 [uzed]
12 mac = 6e:3a:1c:22:aa:5f, 8a:3d:5f:67:55:cb
13 host = 10.0.0.100:9002
14
15 [rpi2]
16 mac = b8:27:eb:29:6b:bc
17 host = 10.0.0.110:9001

```

The MicroZed board can be seen by different MAC addresses depending on how U-Boot starts. As a result both are listed.

11.7.2 Configuration

The boot image tool is configured by an INI file that is passed to the TFTP proxy on the command line when it starts.

The [default] section has to contain a clients entry that lists the clients. There needs to be a client section for each listed client.

A client section header is a client name listed in the clients record in the defaults section. A client section has to contain a mac record and a host record. The MAC record is a comma separated list of MAC addresses in the standard 6 octet hex format separated by :. A list of MAC addresses will match for any address listed. The host record is the IP address and port number of the proxied TFTP server.

11.7.3 Command

The **rtems-tftp-proxy** tool runs a TFTP proxy server using a user provided configuration file. The command line options are:

rtems-tftp-proxy**-h, --help**

Display the command line help.

-l, --log

Set the log file name.

-v, --trace

Enable trace or debug logging.

-c CONFIG, --config CONFIG

The INI format configuration file.

-B BIND, --bind BIND

The interface address the proxy binds too. The default is all which means the proxy binds to all interfaces.

-P PORT, --port PORT

The port the proxy server binds too. The default is the TFTP standard port of 69. This is a privileged port so if using this port number run the TFTP proxy with root or administrator privileges.

11.7.4 Examples

The examples show running the TFTP Proxy as a privileged user:

```

1 $ sudo rtems-tftp-proxy -c foo-test-lab.ini
2 Password:
3 RTEMS Tools - TFTP Proxy, 5.1.0
4 Command Line: rtems-tftp-proxy -c foo-test-lab.ini
5 Host: FreeBSD ruru 12.0-RELEASE-p3 FreeBSD 12.0-RELEASE-p3 GENERIC amd64
6 Python: 3.6.9 (default, Nov 14 2019, 01:16:50) [GCC 4.2.1 Compatible FreeBSD Clang 6.0.1_
  ↪(tags/RELEASE_601/final 335540)]
7 Proxy: all:6

```


SOURCE BUILDER

The RTEMS Source Builder or RSB is a tool to build packages from source. The RTEMS project uses it to build its compilers, tools, kernel and 3rd party packages. The tool is aimed at developers of software for RTEMS who use tool sets for embedded development.

The RSB consolidates the details you need to build a package from source in a controlled and verifiable way. The RSB is not limited to building tools for RTEMS, you can build bare metal development environments.

The RSB section of this manual caters for a range of users from new to experienced RTEMS developers who want to understand the RTEMS Source Builder. New users who just want to build tools should follow *Quick Start* (page 13) in this manual. Users building a binary tool set for release can read the “Installing and Tar Files”. Users wanting to run and test bleeding edge tools or packages, or wanting update or extend the RSB’s configuration can read the remaining sections.

Embedded development typically uses cross-compiling tool chains, debuggers, and debugging aids. Together we call these a **tool set**. The RTEMS Source Builder is designed to fit this specific niche but is not limited to it. The RSB can be used outside of the RTEMS project and we welcome this.

The RTEMS Source Builder is typically used to build a set of packages or a **build set**. A **build set** is a collection of packages and a package is a specific tool, for example GCC, GDB, or library of code and a single **build set** can build them all in a single command. The RTEMS Source Builder attempts to support any host environment that runs Python and you can build the package on. The RSB is not some sort of magic that can take any piece of source code and make it build. Someone at some point in time has figured out how to build that package from source and taught this tool.

Setting up your Host

See *Prepare Your Host Computer* (page 14) for details on setting up hosts.

The RTEMS Source Builder is known to work on:

- ArchLinux
- CentOS
- Fedora
- Raspbian

- Ubuntu (includes XUbuntu)
- Linux Mint
- openSUSE
- FreeBSD
- NetBSD
- Solaris
- MacOS
- Windows

The RTEMS Source Builder has two types of configuration data. The first is the *build set*. A *build set* describes a collection of packages that define a set of tools you would use when developing software for RTEMS. For example the basic GNU tool set is Binutils, GCC, and GDB and is the typical base suite of tools you need for an embedded cross-development type project. The second type of configuration data are the configuration files and they define how a package is built. Configuration files are scripts loosely based on the RPM spec file format and they detail the steps needed to build a package. The steps are *preparation*, *building*, and *installing*. Scripts support macros, shell expansion, logic, includes plus many more features useful when build packages.

The RTEMS Source Builder does not interact with any host package management systems. There is no automatic dependence checking between various packages you build or packages and software your host system you may have installed. We assume the build sets and configuration files you are using have been created by developers who do. Support is provided for package config or pkgconfig type files so you can check and use standard libraries if present. If you have a problem please ask on our [Developers Mailing List](#).

Bug Reporting

If you think you have found a problem please see *Bugs, Crashes, and Build Failures* (page 291).

12.1 Why Build from Source?

The RTEMS Source Builder is not a replacement for the binary install systems you have with commercial operating systems or open source operating system distributions. Those products and distributions are critically important and are the base that allows the RSB to work. The RTEMS Source Builder sits somewhere between you manually entering the commands to build a tool set and a tool such as yum or apt-get to install binary packages made specifically for your host operating system. Building manually or installing a binary package from a remote repository are valid and real alternatives. The RSB provides the specific service of repeatably being able to build tool sets from source code. The process leaves you with the source code used to build the tools and the ability to rebuild it.

If you are developing a system or product that has a long shelf life or is used in a critical piece of infrastructure that has a long life cycle being able to build from source is important. It insulates the project from the fast ever changing world of the host development machines. If your tool set is binary and you have lost the ability to build it you have lost a degree of control and flexibility open source gives you. Fast moving host environments are fantastic. We have powerful multi-core computers with huge amounts of memory and state of the art operating systems your development uses however the product or project you are part of may need to be maintained well past the life time of these host. Being able to build from source is an important and critical part of this process because you can move to a newer host and create an equivalent tool set.

Building from source provides you with control over the configuration of the package you are building. If all or the most important dependent parts are built from source you limit the exposure to host variations. For example the GNU Compiler Collection (GCC) currently uses a number of third-party libraries internally (GMP, ISL, MPC, MPFR, etc.). If your validated compiler generating code for your target processor is dynamically linked against the host's version of these libraries any change in the host's configuration may effect you. The changes the host's package management system makes may be perfectly reasonable in relation to the distribution being managed however this may not extend to you and your tools. Building your tools from source and controlling the specific version of these dependent parts means you are not exposing yourself to unexpected and often difficult to resolve problems. On the other side you need to make sure your tools build and work with newer versions of the host operating system. Given the stability of standards based libraries like libc and ever improving support for standard header file locations this task is becoming easier.

The RTEMS Source Builder is designed to be audited and incorporated into a project's verification and validation process. If your project is developing critical applications that needs to be traced from source to executable code in the target, you need to also consider the tools and how to track them.

If your IT department maintains all your computers and you do not have suitable rights to install binary packages, building from source lets you create your own tool set that you install under your home directory. Avoiding installing any extra packages as a super user is always helpful in maintaining a secure computing environment.

12.2 Project Sets

The RTEMS Source Builder supports project configurations. Project configurations can be public or private and can be contained in the RTEMS Source Builder project if suitable.

The configuration file loader searches the macro `_configdir` and by default this is set to `%{_topdir}/config:%{_sbdir}/config` where `_topdir` is your current working directory, or the directory you invoke the RTEMS Source Builder command in. The macro `_sbdir` is the directory where the RTEMS Source Builder command resides. Therefore the config directory under each of these is searched so all you need to do is create a config in your project and add your configuration files. They do not need to be under the RTEMS Source Builder source tree. Public projects are included in the main RTEMS Source Builder such as RTEMS.

You can add your own patches directory next to your config directory as the `%patch` command searches the `_patchdir` macro variable and it is by default set to `%{_topdir}/patches:%{_sbdir}/patches`.

The source-builder/config directory provides generic scripts for building various tools. You can specialise these in your private configurations to make use of them. If you add new generic configurations please contribute them back to the project

Build sets can be controlled via the command line to enable (`--with-<feature>`) and disable (`--without-<feature>`) various features. There is no definitive list of build options that can be listed because they are implemented with the configuration scripts. The best way to find what is available is to grep the configuration files for with and without.

12.2.1 Bare Metal

The RSB contains a bare configuration tree and you can use this to add packages you use on the hosts. For example 'qemu' is supported on a range of hosts. RTEMS tools live in the `rtems/config` directory tree. RTEMS packages include tools for use on your host computer as well as packages you can build and run on RTEMS.

The **bare metal** support for GNU Tool chains. An example is the `lang/gcc491` build set. You need to provide a target via the command line `--target` option and this is in the standard 2 or 3 tuple form. For example for an ARM compiler you would use `arm-eabi` or `arm-eabihf`, and for SPARC you would use `sparc-elf`:

```

1 $ cd rtems-source-builder/bare
2 $ ../source-builder/sb-set-builder --log=log_arm_eabihf \
3   --prefix=$HOME/development/bare --target=arm-eabihf lang/gcc491
4 RTEMS Source Builder - Set Builder, v0.3.0
5 Build Set: lang/gcc491
6 config: devel/expat-2.1.0-1.cfg
7 package: expat-2.1.0-x86_64-apple-darwin13.2.0-1
8 building: expat-2.1.0-x86_64-apple-darwin13.2.0-1
9 config: devel/binutils-2.24-1.cfg
10 package: arm-eabihf-binutils-2.24-1
11 building: arm-eabihf-binutils-2.24-1
12 config: devel/gcc-4.9.1-newlib-2.1.0-1.cfg
13 package: arm-eabihf-gcc-4.9.1-newlib-2.1.0-1
14 building: arm-eabihf-gcc-4.9.1-newlib-2.1.0-1
15 config: devel/gdb-7.7-1.cfg
16 package: arm-eabihf-gdb-7.7-1

```

(continues on next page)

(continued from previous page)

```

17 building: arm-eabi-hf-gdb-7.7-1
18 installing: expat-2.1.0-x86_64-apple-darwin13.2.0-1 -> /Users/chris/development/bare
19 installing: arm-eabi-hf-binutils-2.24-1 -> /Users/chris/development/bare
20 installing: arm-eabi-hf-gcc-4.9.1-newlib-2.1.0-1 -> /Users/chris/development/bare
21 installing: arm-eabi-hf-gdb-7.7-1 -> /Users/chris/development/bare
22 cleaning: expat-2.1.0-x86_64-apple-darwin13.2.0-1
23 cleaning: arm-eabi-hf-binutils-2.24-1
24 cleaning: arm-eabi-hf-gcc-4.9.1-newlib-2.1.0-1
25 cleaning: arm-eabi-hf-gdb-7.7-1

```

12.2.2 RTEMS

The RTEMS Configurations are found in the `rtems` directory. The configurations are grouped by RTEMS version and a release normally only contains the configurations for that release.. In RTEMS the tools are specific to a specific version because of variations between Newlib and RTEMS. Restructuring in RTEMS and Newlib sometimes moves *libc* functionality between these two parts and this makes existing tools incompatible with RTEMS.

RTEMS allows architectures to have different tool versions and patches. The large number of architectures RTEMS supports can make it difficult to get a common stable version of all the packages. An architecture may require a recent GCC because an existing bug has been fixed, however the more recent version may have a bug in other architecture. Architecture specific patches should only be applied when build the related architecture. A patch may fix a problem on one architecture however it could introduce a problem in another architecture. Limiting exposure limits any possible crosstalk between architectures.

If you have a configuration issue try adding the `--dry-run` option. This will run through all the configuration files and if any checks fail you will see this quickly rather than waiting for until the build fails a check.

Following features can be enabled/disabled via the command line for the RTEMS build sets:

--without-cxx

Do not build a C++ compiler.

--with-ada

Attempt to build an Ada compiler. You need a native GNAT installed.

--with-fortran

Attempt to build a Fortran compiler.

--with-objc

Attempt to build a C++ compiler.

The RSB provides build sets for some BSPs. These build sets will build:

- Compiler, linker, debugger and RTEMS Tools.
- RTEMS Kernel for the BSP
- Optionally LibBSD if supported by the BSP.
- Third party packages if supported by the BSP.

12.2.3 Patches

Packages being built by the RSB need patches from time to time and the RSB supports patching upstream packages. The patches are held in a separate directory called patches relative to the configuration directory you are building. For example `%{_topdir}/patches:%{_smdir}/patches`. Patches are declared in the configuration files in a similar manner to the package's source so please refer to the %source documentation. Patches, like the source, are to be made publically available for configurations that live in the RSB package and are downloaded on demand.

If a package has a patch management tool it is recommended you reference the package's patch management tools directly. If the RSB does not support the specific patch manage tool please contact the mailing list to see if support can be added.

Referenced patches should be placed in a location that is easy to access and download with a stable URL. We recommend attaching a patch to an RTEMS ticket in its bug reporting system or posting to a mailing list with online archives.

RTEMS's former practice of placing patches in the RTEMS Tools Git repository has been stopped.

Patches are added to a component's name and in the %prep: section the patches can be set up, meaning they are applied to source. The patches are applied in the order they are added. If there is a dependency make sure you order the patches correctly when you add them. You can add any number of patches and the RSB will handle them efficiently.

Patches can have options. These are added before the patch URL. If no options are provided the patch's setup default options are used.

Patches can be declared in build set up files.

This examples shows how to declare a patch for gdb in the lm32 architecture:

```
1 %patch add <1> gdb <2> %{rtems_gdb_patches}/lm32/gdb-sim-lm32uart.diff <3>
```

Items:

1. The patch's add command.
2. The group of patches this patch belongs too.
3. The patch's URL. It is downloaded from here.

Patches require a checksum to avoid a warning. The %hash directive can be used to add a checksum for a patch that is used to verify the patch:

```
1 %hash sha512 <1> gdb-sim-lm32uart.diff <2> 77d07087 ... e7db17fb <3>
```

Items:

1. The type of checksum, in the case an SHA512 hash.
2. The patch file the checksum is for.
3. The SHA512 hash.

The patches are applied when a patch setup command is issued in the %prep: section. All patches in the group are applied. To apply the GDB patch above use:

```
1 %patch setup <1> gdb <2> -p1 <3>
```

Items:

1. The patch's setup command.
2. The group of patches to apply.
3. The patch group's default options. If no option is given with the patch these options are used.

Architecture specific patches live in the architecture build set file isolating the patch to that specific architecture. If a patch is common to a tool it resides in the RTEMS tools configuration file. Do not place patches for tools in the source-builder/config template configuration files.

To test a patch simply copy it to your local patches directory. The RSB will see the patch is present and will not attempt to download it. Once you are happy with the patch submit it to the project and a core developer will review it and add it to the RTEMS Tools git repository.

12.2.3.1 Testing a Newlib Patch

To test a local patch for newlib, you need to add the following two lines to the .cfg file in rsb/rtems/config/tools/ that is included by the bset you use:

Steps:

1. Create patches for the changes you want to test. (Note: For RSB, before creating Newlib patch, you must run autoreconf -fvi in the required directory after you make changes to the code. This is not required when you create patch to send to newlib-devel. But if you want RSB to address your changes, your patch should also include regenerated files.)
2. Calculate sha512 of your patch.
3. Place the patches in rsb/rtems/patches directory.
4. Open the .bset file used by your BSP in rsb/rtems/config. For example, for rtems5, SPARC, the file will be rsb/rtems/config/5/rtems-sparc.bset.
5. Inside it you will find the name of .cfg file for Newlib, used by your BSP. For example, I found tools/rtems-gcc-7.4.0-newlib-1d35a003f.
6. Edit your .cfg file. In my case it will be, rsb/rtems/config/tools/rtems-gcc-7.4.0-newlib-1d35a003f.cfg. And add the information about your patch as mentioned below.

```
1 %patch add newlib -p1 file://0001-Port-ndbm.patch <1>
2 %hash sha512 0001-Port-ndbm.patch_
↪ 7d999ceeea4f3dc82e8e0aad09d983a7a68b44470da8a3d61ab6fc558fdb6f2c2de3acc2f32c0b0b97fcc9ab799c27e87afe0
↪ <2>
```

Items:

1. The diff file prepended with `file://` to tell RSB this is a local file.
2. The output from `sha512sum` on the patch file.

12.3 Cross and Canadian Cross Building

Cross building and Canadian Cross building is the process of building on one machine an executable that runs on another machine. An example is building a set of RTEMS tools on Linux to run on Windows. The RSB supports cross building and Canadian cross building.

This sections details how to the RSB to cross and Canadian cross build.

12.3.1 Cross Building

Cross building is where the *build* machine and *host* are different. The *build* machine runs the RSB and the *host* machine is where the output from the build runs. An example is building a package such as Curl for RTEMS on your development machine.

To build the Curl package for RTEMS you enter the RSB command:

```

1 $ ../source-builder/sb-set-builder \
2   --log=log_curl_arm.txt \
3   --prefix=$HOME/development/rtems/5 \ <1>
4   --host=arm-rtems5 \ <2>
5   --with-rtems-bsp=xilinx_zynq_zc706 \ <3>
6   5/ftp/curl

```

Items:

1. The tools and the RTEMS BSP are installed under the same prefix.
2. The `--host` command is the RTEMS architecture and version.
3. The BSP is built and installed in the prefix. The architecture must match the `--host` architecture.

12.3.2 Canadian Cross Building

A Canadian cross builds are where the **build**, **host** and **target** machines all differ. For example building an RTEMS compiler for an ARM processor that runs on Windows is built using a Linux machine. The process is controlled by setting the build triplet to the host you are building, the host triplet to the host the tools will run on and the target to the RTEMS architecture you require. The tools needed by the RSB are:

- Build host C and C++ compiler
- Host C and C++ cross compiler

The RTEMS Source Builder requires you provide the build host C and C++ compiler and the final host C and C++ cross-compiler. The RSB will build the build host RTEMS compiler and the final host RTEMS C and C++ compiler, the output of this process.

The Host C and C++ compiler is a cross-compiler that builds executables for the host you want the tools for. You need to provide these tools. For Windows a number of Unix operating systems provide MinGW tool sets as packages.

The RSB will build an RTEMS tool set for the build host. This is needed when building the final host's RTEMS compiler as it needs to build RTEMS runtime code such as *libc* on the build host.

TIP: Make sure the host's cross-compiler tools are in your path before run the RSB build command.

TIP: Canadian Cross built tools will not run on the machine being used to build them so you should provide the `--bset-tar-files` and `--no-install` options. The option to not install the files lets you provide a prefix that does not exist or you cannot access.

To perform a cross build add `--host=` to the command line. For example to build a MinGW tool set on FreeBSD for Windows add `--host=mingw32` if the cross compiler is `mingw32-gcc`:

```
1 $ ../source-builder/sb-set-builder --host=mingw32 \  
2   --log=l-mingw32-4.11-sparc.txt \  
3   --prefix=$HOME/development/rtems/5 \  
4   5/rtems-sparc
```

If you are on a Linux Fedora build host with the MinGW packages installed the command line is:

```
1 $ ../source-builder/sb-set-builder --host=i686-w64-mingw32 \  
2   --log=l-mingw32-4.11-sparc.txt \  
3   --prefix=$HOME/development/rtems/5 \  
4   5/rtems-sparc
```

12.4 Third-Party Packages

This section describes how to build and add an RTEMS third-party package to the RSB.

A third-party package is a library or software package built to run on RTEMS, examples are Curl, NTP, Net-Snmp, libjpeg and more. These pieces of software can be used to help build RTEMS applications. The package is built for a specific BSP and so requires a working RTEMS tool chain, an installed RTEMS Board Support Package (BSP), and a network stack if the package uses networking resources.

Help

If you have any issues using, building or adding third party packages please ask on the RTEMS users mailing list.

The RSB support for building third-party packages is based around the *pkconfig* files (PC) installed with the BSP. The *pkgconfig* support in RTEMS is considered experimental and can have some issues for some BSPs. This issue is rooted deep in the RTEMS build system.

12.4.1 Vertical Integration

The RSB supports horizontal integration with support for multiple architectures. Adding packages to the RSB as libraries is vertical integration. Building the GCC tool chain requires you build an assembler before you build a compiler. The same can be done for third-party libraries, you can create build sets that stack library dependences vertically to create a *stack*.

12.4.2 Building

To build a package you need to have a suitable RTEMS tool chain and RTEMS BSP installed. The set builder command line requires you provide the tools path, the RTEMS architecture (host), the BSP, and the prefix path used to the install RTEMS BSP.

The RSB prefix option (`--prefix`) provided when building a package is the path to:

1. The tools, RTEMS kernel and any dependent libraries such as LibBSD. The package will be installed into the prefix path. This build configuration can be used to make a complete set of development tools and libraries for a project or product under a single path.
2. The RTEMS kernel and any dependent libraries such as LibBSD. The tools path needs to be in the environment path (not recommended) or provided to the set builder command by the `--with-tools` option. The package will be installed into the prefix path. This build configuration can be used when you have a set of tools used with a number of RTEMS BSPs. The tools can be shared between the different BSPs.
3. The path the package is installed into. The tools path needs to be in the environment path (not recommended) or provided to the set builder command using the `--with-tools` option. The path to the RTEMS kernel and any dependent libraries such as LibBSD needs to be supplied to the set builder command using the `--with-rtems` option. This build configuration can be used when you have a set of libraries you are testing with a changing RTEMS kernel. Be careful using this configuration as changes in RTEMS interfaces may require rebuilding these packages.

The set builder command option `--host` is used to provide the RTEMS architecture the package is being built for. For example `--host=arm-rtems5` is used for any ARM BSP.

The set builder command option `--with-rtems-bsp` is the RTEMS BSP the package is being built for. The BSP is searched for under the path provided by the command option `--with-rtems` and if this option is not provided the provided prefix is searched.

The following example builds and installs the Curl networking package for the ARM BeagleBone Black BSP installing it into the same path the tools, RTEMS kernel and LibBSD are installed in.

```

1 $ ../source-builder/sb-set-builder --prefix=$HOME/development/cs/rtems/5 \
2   --log=curl.txt --host=arm-rtems5 --with-rtems-bsp=beagleboneblack ftp/curl
3 RTEMS Source Builder - Set Builder, 5 (2bdae1f169e4)
4 Build Set: ftp/curl
5 config: ftp/curl-7.65.1-1.cfg
6 package: curl-v7.65.1-arm-rtems5-1
7 download: https://curl.haxx.se/download/curl-7.65.1.tar.xz -> sources/curl-7.65.1.tar.xz
8 downloading: sources/curl-7.65.1.tar.xz - 2.3MB of 2.3MB (100%)
9 building: curl-v7.65.1-arm-rtems5-1
10 sizes: curl-v7.65.1-arm-rtems5-1: 87.055MB (installed: 2.238MB)
11 cleaning: curl-v7.65.1-arm-rtems5-1
12 reporting: ftp/curl-7.65.1-1.cfg -> curl-v7.65.1-arm-rtems5-1.txt
13 reporting: ftp/curl-7.65.1-1.cfg -> curl-v7.65.1-arm-rtems5-1.xml
14 installing: curl-v7.65.1-arm-rtems5-1 -> /Users/chris/development/cs/rtems/5
15 cleaning: curl-v7.65.1-arm-rtems5-1
16 Build Set: Time 0:01:10.006872

```

12.4.3 Adding

Adding a package requires you first build it manually by downloading the source for the package and building it for RTEMS using the command line of a standard shell. If the package has not been ported to RTEMS you will need to port it and this may require asking questions on the package's user or development support lists as well as RTEMS's developers list. Your porting effort may end up with a patch. RTEMS requires a patch be submitted upstream to the project's community as well as RTEMS. The RTEMS submission is best as a patch attached to ticket in Trac. A patch attached to a ticket can be referenced by an RSB configuration file and used in a build.

Patches in Trac

Attaching patches for packages to Trac tickets provides an easy to reference URL the RSB can fetch. The patch URL does not change across RTEMS versions and it does not depend on the state or layout of a git repo.

A package may create executables, for example Curl normally creates an executable called `curl` however it will probably not run because the needed RTEMS configuration is not suitable. If found the RSB automatically adds the RTEMS library `librtemsdefaultconfig.a` to the `LIBS` variable used to link executables. This library provides a limited configuration suitable for linking an executable however it is not a set up that allows the resulting executable to run correctly. As a result it is best not to install these executables.

A custom RTEMS patch to an executable's source code can turn it into a function that can be called by the RTEMS shell. Users can call the function in their executables simulating the

running of the package's command. If the package does not export the code in a suitable manner please contact the project's community and see if you can work with them to provide a way for the code to be exported. This may be difficult because exporting internal headers and functions opens the project up to API compatibility issues they did not have before. In the simplest case attempting to get the code into a static library with a single call entry point exported in a header would give RTEMS user's access to the package's main functionality.

A package requires at least three (3) files to be created:

Published Package Name:

The first file is the RTEMS build set file and it resides under the `rtems/config` path in a directory tree based on the FreeBSD ports collection. For the Curl package and RTEMS 5 this is `rtems/config/ftp/curl.bset`. If you do not know the FreeBSD port path for the package you are adding please ask. The build set file references a specific configuration file therefore linking the RTEMS version to a specific version of the package you are adding. Updating the package to a new version requires changing the build set to the new configuration file.

Package Version Configuration File:

The second file is an RTEMS version specific configuration file and it includes the RSB RTEMS BSP support. These configuration files reside in the `rtems/config` tree and under the FreeBSD port's path name. For example the Curl package is found in the `ftp` directory of the FreeBSD ports tree so the Curl configuration path is `rtems/config/ftp/curl-7.65.1-1.cfg` for that specific version. The configuration file name typically provides version specific references and the RTEMS build set file references a specific version. This configuration file references the build configuration file held in the common configuration file tree. An SHA512 hash is required to verify the source package that is downloaded.

Build Configuration File:

The build configuration. This is a common script that builds the package. It resides in the `source-builder/config` directory and typically has the packages's name with the major version number. If the build script does not change for each major version number a *common* base script can be created and included by each major version configuration script. The `gcc` compiler configuration is an example. This approach lets you branch a version if something changes that is not backwards compatible. It is important to keep existing versions building. The build configuration should be able to build a package for the build host as well as RTEMS as the RSB abstracts the RTEMS specific parts. See *Configuration* (page 257) for more details.

12.4.4 Host and Build Flags

A package's build is controlled by setting the compiler names and flags that are used when building. The RSB provides a macro called `%{host_build_flags}` to define these flags for you. Use this macro in the `%build` section of your config script file to define the set up needed to build a native package or to cross-compile to a specific host such as RTEMS. The typical `%build` section is:

```
1 %build
2   build_top=$(pwd)
3
```

(continues on next page)

(continued from previous page)

```

4  %{build_directory}
5
6  mkdir -p ${build_dir}
7  cd ${build_dir}
8
9  %{host_build_flags}
10
11  ../${source_dir_curl}/configure \
12  --host=%{_host} \
13  --prefix=%{_prefix} \
14  --bindir=%{_bindir} \
15  --exec_prefix=%{_exec_prefix} \
16  --includedir=%{_includedir} \
17  --libdir=%{_libdir} \
18  --libexecdir=%{_libexecdir} \
19  --mandir=%{_mandir} \
20  --infodir=%{_infodir} \
21  --datadir=%{_datadir}
22
23  %{{_make}} %{{?_smp_mflags}} all
24
25  cd ${build_top}

```

The `%{host_build_flags}` checks if the build is native for the development host or a cross-compile build.

For a cross-compilation build the flags are:

CC, CC_FOR_HOST:

The C compiler used to build the package. For an RTEMS build this is the RTEMS C compiler. For example the ARM architecture and RTEMS 5 the value is set to `arm-rtems5-gcc`.

CXX, CXX_FOR_HOST:

The C++ compiler used to build the package. For an RTEMS build this is the RTEMS C++ compiler. For example the ARM architecture and RTEMS 5 the value is set to `arm-rtems5-g++`.

CPPFLAGS, CPPFLAGS_FOR_HOST:

The C compiler preprocessor flags used to build the package. Set any include paths in this variable as some configure scripts will warn you if include paths are set in the `CFLAGS`.

CFLAGS, CFLAGS_FOR_HOST:

The C compiler flags used when running the C compiler. Set any include paths in the `CPPFLAGS` variable as some configure scripts will warn you if include paths in this variable.

CXXFLAGS, CXXFLAGS_FOR_HOST:

The C++ compiler flags used when running the C++ compiler. Set any include paths in the `CPPFLAGS` variable as some configure scripts will warn you if include paths in this variable.

LDFLAGS, LDFLAGS_FOR_HOST:

The linker flags used when link package executables. The C or C++ compiler is used to run the linker.

LIBS, LIBS_FOR_HOST:

A list of libraries passed to the linker when linking an executable.

CC_FOR_BUILD:

The native C compiler.

CXX_FOR_BUILD:

The native C++ compiler.

CPPFLAGS_FOR_BUILD:

The C preprocessor flags used when preprocessing a native C source file.

CFLAGS_FOR_BUILD:

The native C compiler flags used when running the native C compiler.

CXXFLAGS_FOR_BUILD:

The native C++ compiler flags used when running the native C++ compiler.

LDLFLAGS_FOR_BUILD:

The native linker flags used when linking a native executable.

LIBS_FOR_BUILD:

The native libraries used to when linking a native executable.

For a native build the flags are:

CC, CC_FOR_BUILD:

The native C compiler.

CXX, CXX_FOR_BUILD:

The native C++ compiler.

CPPFLAGS, CPPFLAGS_FOR_BUILD:

The C preprocessor flags used when preprocessing a native C source file.

CFLAGS, CFLAGS_FOR_BUILD:

The native C compiler flags used when running the native C compiler.

CXXFLAGS, CXXFLAGS_FOR_BUILD:

The native C++ compiler flags used when running the native C++ compiler.

LDLFLAGS, LDLFLAGS_FOR_BUILD:

The native linker flags used when linking a native executable.

LIBS, LIBS_FOR_BUILD:

The native libraries used to when linking a native executable.

12.4.5 BSP Support

The RSB provides support to build packages for RTEMS. RTEMS applications can be viewed as statically linked executables operating in a single address space. As a result only the static libraries a package builds are required and these libraries need to be ABI compatible with the RTEMS kernel and application code. This means the compiler ABI flags used to build all the code in the executable must be the same. A 3rd party package must use the same compiler flags as the BSP used to build RTEMS.

Note: RTEMS's dynamic loading support does not use the standard shared library support found in Unix and the ELF standard. RTEMS's loader uses static libraries and the runtime link editor performs a similar function to a host based static linker. RTEMS will only reference static libraries even if dynamic libraries are created and installed.

The RSB provides the configuration file `rtems/config/rtems-bsp.cfg` to support building third-party packages and you need to include this file in your RTEMS version specific configuration file. For example the Curl configuration file `rtems/config/curl/curl-7.65.1-1.cfg`:

```

1 #
2 # Curl 7.65.1
3 #
4
5 %if %{release} == %{nil}
6   %define release 1 <1>
7 %endif
8
9 %include %{_configdir}/rtems-bsp.cfg <2>
10
11 #
12 # Curl Version
13 #
14 %define curl_version 7.65.1 <3>
15
16 %hash sha512 curl-%{curl_version}.tar.xz aba2d979a...72b6ac55df4 <4>
17
18 #
19 # Curl Build configuration
20 #
21 %include %{_configdir}/curl-1.cfg <5>

```

Items:

1. The release number.
2. Include the RSB RTEMS BSP support.
3. The Curl package's version.
4. The SHA512 hash for the source file. The hash here has been shortened.
5. The Curl standard build configuration.

The RSB RTEMS BSP support file `rtems/config/rtems-bsp.cfg` checks to make sure the required RSB command line options are provided. These include `--host` and `--with-rtems-bsp`. If the `--with-tools` command line option is not given the `$_prefix` is used as the path to the tools. If the `--with-rtems` command line option is not given the `$_prefix` is used as the path to the installed RTEMS BSP.

Note: The RTEMS BSP and any dependent 3rd party packages must be installed to be seen as available. A path to the location the BSP has been built will not work.

The first check is to make sure a target is not specified. This is only used for Canadian cross-compilation builds and currently there is no support for RTEMS third party packages to build that way:

```

1 #
2 # The target is used by compilers or Cxc builds.

```

(continues on next page)

(continued from previous page)

```

3 #
4 %if %{_target} != %{nil}
5   %error RTEMS BSP builds use --host and not --target
6 %endif

```

A host is required using the --host option:

```

1 #
2 # We need a host from the user to specify the RTEMS architecture and major
3 # version.
4 #
5 %if %{_host} == %{nil} && %{rtems_bsp_error} <1>
6   %error No RTEMS host or BSP specified: --host=<arch>-rtems<ver>
7 %endif

```

An RTEMS BSP is required using the --with-bsp option:

```

1 #
2 # We need a BSP from the user.
3 #
4 %ifn %{defined with_rtems_bsp}
5   %if %{rtems_bsp_error}
6     %error No RTEMS BSP specified: --rtems-bsp=arch/bsp (or --with-rtems-bsp=bsp)
7   %endif
8   %define with_rtems_bsp sparc/erc32
9 %endif

```

Check if the --with-tools or --with-rtems options have been provided and if they are not provided use the --prefix path:

```

1 #
2 # If no tools or RTEMS provided use the prefix.
3 #
4 %ifn %{defined with_tools}
5   %define with_tools %{_prefix}
6 %endif
7
8 %ifn %{defined with_rtems}
9   %define with_rtems %{_prefix}
10 %endif

```

Add the tools path to the environment path:

```

1 #
2 # Set the path to the tools.
3 #
4 %{path prepend %{with_tools}/bin}

```

RTEMS exports the build configuration in *pkgconfig* (.pc) files. The RSB can read these files even when there is no pkgconfig support installed on your development machine. The *pkgconfig* support provides a BSP's configuration and the RSB uses it to set the following RSB macros variables:

```

1 %{pkgconfig prefix %{_prefix}/lib/pkgconfig} <1>
2 %{pkgconfig crosscompile yes} <2>

```

(continues on next page)

(continued from previous page)

```

3  %{pkgconfig filter-flags yes} <3>
4
5  #
6  # The RTEMS BSP Flags
7  #
8  %define rtems_bsp      %{with_rtems_bsp}
9  %define rtems_bsp_ccflags  %{pkgconfig ccflags  %{_host}-%{rtems_bsp}} <4>
10 %define rtems_bsp_cflags   %{pkgconfig cflags   %{_host}-%{rtems_bsp}}
11 %define rtems_bsp_ldflags  %{pkgconfig ldflags  %{_host}-%{rtems_bsp}}
12 %define rtems_bsp_libs     %{pkgconfig libs     %{_host}-%{rtems_bsp}}

```

Items:

1. Set the path to the BSP's pkgconfig file.
2. Let *pkgconfig* know this is a cross-compile build.
3. Filter flags such as warnings. Warning flags are specific to a package and RTEMS exports it's warnings flags in the BSP configuration settings.
4. Ask *pkgconfig* for the various settings we require.

The flags obtained by *pkgconfig* and given a *rtems_bsp* prefix are used to set the RTEMS host variables CFLAGS, LDFLAGS and LIBS. When we build a third party library your host computer is the **build** machine and RTEMS is the **host** machine therefore we set the host variables:

```

1 %define host_cflags  %{rtems_bsp_cflags}
2 %define host_ldflags %{rtems_bsp_ldflags}
3 %define host_libs    %{rtems_bsp_libs}

```

Finally we provide all the paths you may require when configuring a package. Packages by default consider the *_prefix* the base and install various files under this tree. The package you are building is specific to a BSP and needs to install it's files into the RTEMS specific BSP path under the *_prefix*. This allows more than BSP build of this package to be installed under the same *_prefix* at the same time:

```

1 %define rtems_bsp_prefix  %{_prefix}%{_host}%{rtems_bsp} <1>
2 %define _exec_prefix      %{rtems_bsp_prefix}
3 %define _bindir            %{_exec_prefix}/bin
4 %define _sbindir          %{_exec_prefix}/sbin
5 %define _libexecdir       %{_exec_prefix}/libexec
6 %define _datarootdir      %{_exec_prefix}/share
7 %define _datadir          %{_datarootdir}
8 %define _sysconfdir       %{_exec_prefix}/etc
9 %define _sharedstatedir   %{_exec_prefix}/com
10 %define _localstatedir    %{_exec_prefix}/var
11 %define _includedir       %{_libdir}/include
12 %define _lib              lib
13 %define _libdir           %{_exec_prefix}%{_lib}
14 %define _libexecdir       %{_exec_prefix}/libexec
15 %define _mandir           %{_datarootdir}/man
16 %define _infodir          %{_datarootdir}/info
17 %define _localedir        %{_datarootdir}/locale
18 %define _localedir        %{_datadir}/locale

```

(continues on next page)

(continued from previous page)

```
19 %define _localstatedir    ${_exec_prefix}/var
```

Items:

1. The path to the installed BSP.

When you configure a package you can reference these paths and the RSB will provide sensible default or in this case map them to the BSP:

```
1  ../${source_dir_curl}/configure \ <1>
2  --host=${_host} \
3  --prefix=${_prefix} \
4  --bindir=${_bindir} \
5  --exec_prefix=${_exec_prefix} \
6  --includedir=${_includedir} \
7  --libdir=${_libdir} \
8  --libexecdir=${_libexecdir} \
9  --mandir=${_mandir} \
10 --infodir=${_infodir} \
11 --datadir=${_datadir}
```

Items:

1. The configure command for Curl.

12.4.6 BSP Configuration

The following RSB macros are defined when building a package for RTEMS:

Note: A complete list can be obtained by building with the `--trace` flag. The log will contain a listing of all macros before and after the configuration is loaded.

%{rtems_bsp}:

The name of the RTEMS BSP.

%{rtems_bsp_cc}:

The C compiler name for the RTEMS BSP.

%{rtems_bsp_cflags}:

The C compiler flags for the RTEMS BSP.

%{rtems_bsp_ccflags}:

The C++ compiler flags for the RTEMS BSP.

%{rtems_bsp_incpath}:

The include path to the RTEMS BSP header files.

%{rtems_bsp_ldflags}:

The linker flags for the RTEMS BSP.

%{rtems_bsp_libs}:

The libraries used when linking an RTEMS BSP executable.

%{rtems_bsp_prefix}:

The prefix for the RTEMS BSP.

%{rtems-libbsd}:

The variable is set to found if LibBSD is available.

%{rtems-defaultconfig}:

The path of the RSB helper script to locate find header files or libraries.

%{_host}

The host triplet passed on the command line to the set builder using the `--host` options. This is the RTEMS architecture and version. For example `arm-rtems5`.

%{host_cflags}:

The BSP CFLAGS returned by `pkgconfig`.

%{host_cxxflags}:

The BSP CXXFLAGS returned by `pkgconfig`.

%{host_includes}:

The BSP include paths returned by `pkgconfig`.

%{host_ldflags}:

The BSP LDFLAGS returned by `pkgconfig`.

%{host_libs}:

The libraries needed to be linked to create an executable. If LibBSD is installed the library `-lbsd` is added. If the BSP has installed the RTEMS default configuration library (`-lrtemsdefaultconfig`) it is added to the list of libraries.

%{host_build_flags}:

This macro is defined in `defaults.mc` and is a series of shell commands that set up the environment to build an RTEMS package. If the host and the build triplets are the same it is a native build for your development host. If the host is not the build machine it is a cross-compilation build. For either case the following are defined.

%{_host_os}:

The host operating system extracted from the `--host` command line option. For example the operating system for the host of `arm-rtems5` is `rtems5`.

%{_host_arch}:

The host architecture extracted from the `--host` command line option. For example the architecture for the host of `arm-rtems5` is `arm`.

%{_host_cpu}:

The host cpu extracted from the `--host` command line option. For example the cpu for the host of `arm-rtems5` is `arm`.

12.5 Configuration

The RTEMS Source Builder has two types of configuration data:

- Build Sets
- Package Build Configurations

By default these files can be located in two separate directories and searched. The first directory is `config` in your current working directory (`_topdir`) and the second is `config` located in the base directory of the RTEMS Source Builder command you run (`_sbdire`). The RTEMS directory `rtems`` located at the top of the RTEMS Source Builder source code is an example of a specific build configuration directory. You can create custom or private build configurations and if you run the RTEMS Source Builder command from that directory your configurations will be used.

The configuration search path is a macro variable and is reference as `%{_configdir}`. It's default is defined as:

```
1 _configdir : dir optional %{_topdir}/config:%{_sbdire}/config
```

Items:

1. The `_topdir` is the directory you run the command from and `_sbdire` is the location of the RTEMS Source Builder command.
2. A macro definition in a macro file has 4 fields, the label, type, constraint and the definition.

Build set files have the file extension `.bset` and the package build configuration files have the file extension of `.cfg`. The `sb-set-builder` command will search for *build sets* and the `sb-builder` commands works with package build configuration files.

Both types of configuration files use the `#` character as a comment character. Anything after this character on the line is ignored. There is no block comment.

12.5.1 Source and Patches

The RTEMS Source Builder provides a flexible way to manage source. Source and patches are declare in configurations file using the `source` and `patch` directives. These are a single line containing a Universal Resource Location or URL and can contain macros and shell expansions. The *%prep* (page 271) section details the *source* and *patch* directives

The URL can reference remote and local source and patch resources. The following schemes are provided:

http:

Remote access using the HTTP protocol.

https:

Remote access using the Secure HTTP protocol.

ftp:

Remote access using the FTP protocol.

git:

Remote access to a GIT repository.

pm:

Remote access to a patch management repository.

file:

Local access to an existing source directory.

12.5.1.1 HTTP, HTTPS, and FTP

Remote access to TAR or ZIP files is provided using HTTP, HTTPS and FTP protocols. The full URL provided is used to access the remote file including any query components. The URL is parsed to extract the file component and the local source directory is checked for that file. If the file is located locally the remote file is not downloaded. Currently no other checks are made. If a download fails you need to manually remove the file from the source directory and start the build process again.

The URL can contain macros. These are expanded before issuing the request to download the file. The standard GNU GCC compiler source URL is:

```
1 %source set gcc ftp://ftp.gnu.org/gnu/gcc/gcc-%{gcc_version}/gcc-%{gcc_version}.tar.bz2
```

Items:

1. The %source command's set command sets the source. The first is set and following sets are ignored.
2. The source package is part of the gcc group.

The type of compression is automatically detected from the file extension. The supported compression formats are:

gz:

GNU ZIP

bzip2:

BZIP2

zip:

ZIP

xy:

XY

The output of the decompression tool is fed to the standard tar utility if not a ZIP file and unpacked into the build directory. ZIP files are unpacked by the decompression tool and all other files must be in the tar file format.

The %source directive typically supports a single source file tar or zip file. The set command is used to set the URL for a specific source group. The first set command encountered is registered and any further set commands are ignored. This allows you to define a base standard source location and override it in build and architecture specific files. You can also add extra source files to a group. This is typically done when a collection of source is broken down in a number

of smaller files and you require the full package. The source's setup command must reside in the `%prep:` section and it unpacks the source code ready to be built.

If the source URL references the GitHub API server <https://api.github.com/> a tarball of the specified version is download. For example the URL for the STLINK project on GitHub and version is:

```
1 %define stlink_version 3494c11
2 %source set stlink https://api.github.com/repos/texane/stlink/texane-stlink-%{stlink_
  ↪version}.tar.gz
```

12.5.1.2 GIT

A GIT repository can be cloned and used as source. The GIT repository resides in the 'source' directory under the `git` directory. You can edit, update and use the repository as you normally do and the results will used to build the tools. This allows you to prepare and test patches in the build environment the tools are built in. The GIT URL only supports the GIT protocol. You can control the repository via the URL by appending options and arguments to the GIT path. The options are delimited by `?` and option arguments are delimited from the options with `=`. The options are:

protocol:

Use a specific protocol. The supported values are `ssh`, `git`, `http`, `https`, `ftp`, `ftps`, `rsync`, and `none`.

branch:

Checkout the specified branch.

pull:

Perform a pull to update the repository.

fetch:

Perform a fetch to get any remote updates.

reset:

Reset the repository. Useful to remove any local changes. You can pass the `hard` argument to force a hard reset.

An example is:

```
1 %source set gcc git://gcc.gnu.org/git/gcc.git?branch=gcc-4_7-branch?reset=hard
```

This will clone the GCC git repository and checkout the 4.7-branch and perform a hard reset. You can select specific branches and apply patches. The repository is cleaned up before each build to avoid various version control errors that can arise.

The `protocol` option lets you set a specific protocol. The `git://` prefix used by the RSB to select a git repository can be removed using `none` or replaced with one of the standard git protocols.

12.5.1.3 CVS

A CVS repository can be checked out. CVS is more complex than GIT to handle because of the modules support. This can effect the paths the source ends up in. The CVS URL only supports the CVS protocol. You can control the repository via the URL by appending options

and arguments to the CVS path. The options are delimited by ? and option arguments are delimited from the options with =. The options are:

module:

The module to checkout.

src-prefix:

The path into the source where the module starts.

tag:

The CVS tag to checkout.

date:

The CVS date to checkout.

The following is an example of checking out from a CVS repository:

```
1 %source set newlib cvs://pserver:anoncvs@sourceware.org/cvs/src?module=newlib?src-
  ↪ prefix=src
```

12.5.2 Macros and Defaults

The RTEMS Source Builder uses tables of *macros* read in when the tool runs. The initial global set of macros is called the *defaults*. These values are read from a file called `defaults.mc` and modified to suite your host. This host specific adaption lets the Source Builder handle differences in the build hosts.

Build set and configuration files can define new values updating and extending the global macro table. For example builds are given a release number. This is typically a single number at the end of the package name. For example:

```
1 %define release 1
```

Once defined it can be accessed in a build set or package configuration file with:

```
1 %{release}
```

The `sb-defaults` command lists the defaults for your host. I will not include the output of this command because of its size:

```
1 $ ../source-builder/sb-defaults
```

A nested build set is given a separate copy of the global macro maps. Changes in one change set are not seen in other build sets. That same happens with configuration files unless inline includes are used. Inline includes are seen as part of the same build set and configuration and changes are global to that build set and configuration.

12.5.2.1 Macro Maps and Files

Macros are read in from files when the tool starts. The default settings are read from the defaults macro file called `defaults.mc` located in the top level RTEMS Source Builder command directory. User macros can be read in at start up by using the `--macros` command line option.

The format for a macro in macro files is:


```
1 Name Type Attribute String
```

where Name is a case insensitive macro name, the Type field is:

none:

Nothing, ignore.

dir:

A directory path.

exe:

An executable path.

triplet:

A GNU style architecture, platform, operating system string.

the Attribute field is:

none:

Nothing, ignore

required:

The host check must find the executable or path.

optional:

The host check generates a warning if not found.

override:

Only valid outside of the global map to indicate this macro overrides the same one in the global map when the map containing it is selected.

undefine:

Only valid outside of the global map to undefine the macro if it exists in the global map when the map containing it is selected. The global map's macro is not visible but still exists.

and the String field is a single or tripled multiline quoted string. The 'String' can contain references to other macros. Macro that loop are not currently detected and will cause the tool to lock up.

Maps are declared anywhere in the map using the map directive:

```
1 # Comments
2 [my-special-map] <1>
3 _host: none, override, 'abc-xyz'
4 multiline: none, override, '''First line,
5 second line,
6 and finally the last line'''
```

Items:

1. The map is set to my-special-map.

Any macro definitions following a map declaration are placed in that map and the default map is global when loading a file. Maps are selected in configuration files by using the %select directive:

```
1 %select my-special-map
```

Selecting a map means all requests for a macro first check the selected map and if present return that value else the global map is used. Any new macros or changes update only the global map. This may change in future releases so please make sure you use the override attribute.

The macro files specified on the command line are looked for in the `_configdir` paths. See `<<X1,“_configdir“>>` variable for details. Included files need to add the `%{_configdir}` macro to the start of the file.

Macro map files can include other macro map files using the `%include` directive. The macro map to build *binutils*, *gcc*, *newlib*, *gdb* and RTEMS from version control heads is:

```
1 #
2 # Build all tool parts from version control head.
3 #
4 %include %{_configdir}/snapshots/binutils-head.mc
5 %include %{_configdir}/snapshots/gcc-head.mc
6 %include %{_configdir}/snapshots/newlib-head.mc
7 %include %{_configdir}/snapshots/gdb-head.mc
```

Items:

1. The file is `config/snapshots/binutils-gcc-newlib-gdb-head.mc`.

The macro map defaults to global at the start of each included file and the map setting of the macro file including the other macro files does not change.

12.5.2.2 Personal Macros

When the tools start to run they will load personal macros. Personal macros are in the standard format for macros in a file. There are two places personal macros can be configured. The first is the environment variable `RSB_MACROS`. If present the macros from the file the environment variable points to are loaded. The second is a file called `.rsb_macros` in your home directory. You need to have the environment variable `HOME` defined for this work.

12.5.3 Report Mailing

The build reports can be mailed to a specific email address to logging and monitoring. Mailing requires a number of parameters to function. These are:

- To mail address
- From mail address
- SMTP host

The to mail address is taken from the macro `%{_mail_tools_to}` and the default is *rtems-tooltestresults at rtems.org*. You can override the default with a personal or user macro file or via the command line option `--mail-to`.

The from mail address is taken from:

- GIT configuration
- User .mailrc file
- Command line

If you have configured an email and name in git it will be used. If you do not a check is made for a .mailrc file. The environment variable MAILRC is used if present else your home directory is checked. If found the file is scanned for the from setting:

```
1 set from="Foo Bar <foo@bar>"
```

You can also support a from address on the command line with the `--mail-from` option.

The SMTP host is taken from the macro `%{_mail_smtp_host}` and the default is localhost. You can override the default with a personal or user macro file or via the command line option `--smtp-host`.

12.5.4 Build Set Files

Build set files let you list the packages in the build set you are defining and have a file extension of .bset. Build sets can define macro variables, inline include other files and reference other build set or package configuration files.

Defining macros is performed with the `%define` macro:

```
1 %define _target m32r-rtems4.11
```

Inline including another file with the `%include` macro continues processing with the specified file returning to carry on from just after the include point:

```
1 %include rtems-4.11-base.bset
```

This includes the RTEMS 4.11 base set of defines and checks. The configuration paths as defined by `_configdir` are scanned. The file extension is optional.

You reference build set or package configuration files by placing the file name on a single line:

```
1 tools/rtems-binutils-2.22-1
```

The `_configdir` path is scanned for `tools/rtems-binutils-2.22-1.bset` or `tools/rtems-binutils-2.22-1.cfg`. Build set files take precedent over package configuration files. If `tools/rtems-binutils-2.22-1` is a build set a new instance of the build set processor is created and if the file is a package configuration the package is built with the package builder. This all happens once the build set file has finished being scanned.

12.5.5 Configuration Control

The RTEMS Source Builder is designed to fit within most verification and validation processes. All of the RTEMS Source Builder is source code. The Python code is source and comes with a commercial friendly license. All configuration data is text and can be read or parsed with standard text based tools.

File naming provides configuration management. A specific version of a package is captured in a specific set of configuration files. The top level configuration file referenced in a *build set*

or passed to the sb-builder command relates to a specific configuration of the package being built. For example the RTEMS configuration file `rtems-gcc-4.7.2-newlib-2.0.0-1.cfg` creates an RTEMS GCC and Newlib package where the GCC version is 4.7.2, the Newlib version is 2.0.0, plus any RTEMS specific patches that related to this version. The configuration defines the version numbers of the various parts that make up this package:

```
1 %define gcc_version      4.7.2
2 %define newlib_version  2.0.0
3 %define mpfr_version     3.0.1
4 %define mpc_version      0.8.2
5 %define gmp_version      5.0.5
```

The package build options, if there are any are also defined:

```
1 %define with_threads 1
2 %define with_plugin  0
3 %define with_iconv    1
```

The generic configuration may provide defaults in case options are not specified. The patches this specific version of the package requires can be included:

```
1 Patch0: gcc-4.7.2-rtems4.11-20121026.diff
```

Finally including the GCC 4.7 configuration script:

```
1 %include %[_configdir]/gcc-4.7-1.cfg
```

The `gcc-4.7-1.cfg` file is a generic script to build a GCC 4.7 compiler with Newlib. It is not specific to RTEMS. A bare no operating system tool set can be built with this file.

The -1 part of the file names is a revision. The GCC 4.7 script maybe revised to fix a problem and if this fix effects an existing script the file is copied and given a -2 revision number. Any dependent scripts referencing the earlier revision number will not be effected by the change. This locks down a specific configuration over time.

12.5.6 Personal Configurations

The RSB supports personal configurations. You can view the RTEMS support in the `rtems` directory as a private configuration tree that resides within the RSB source. There is also the bare set of configurations. You can create your own configurations away from the RSB source tree yet use all that the RSB provides.

To create a private configuration change to a suitable directory:

```
1 $ cd ~/work
2 $ mkdir test
3 $ cd test
4 $ mkdir config
```

and create a config directory. Here you can add a new configuration or build set file. The section 'Adding New Configurations' details how to add a new configuration.

12.5.7 New Configurations

This section describes how to add a new configuration to the RSB. We will add a configuration to build the Device Tree Compiler. The Device Tree Compiler or DTC is part of the Flattened Device Tree project and compiles Device Tree Source (DTS) files into Device Tree Blobs (DTB). DTB files can be loaded by operating systems and used to locate the various resources such as base addresses of devices or interrupt numbers allocated to devices. The Device Tree Compiler source code can be downloaded from <http://www.jdl.com/software>. The DTC is supported in the RSB and you can find the configuration files under the bare/config tree. I suggest you have a brief look over these files.

12.5.7.1 Layering by Including

Configurations can be layered using the `%include` directive. The user invokes the outer layers which include inner layers until all the required configuration is present and the package can be built. The outer layers can provide high level details such as the version and the release and the inner layers provide generic configuration details that do not change from one release to another. Macro variables are used to provide the specific configuration details.

12.5.7.2 Configuration File Numbering

Configuration files have a number at the end. This is a release number for that configuration and it gives us the ability to track a specific configuration for a specific version. For example lets say the developers of the DTC package change the build system from a single makefile to autoconf and automake between version 1.3.0 and version 1.4.0. The configuration file used to build the package would change have to change. If we did not number the configuration files the ability to build 1.1.0, 1.2.0 or 1.3.0 would be lost if we update a common configuration file to build an autoconf and automake version. For version 1.2.0 the same build script can be used so we can share the same configuration file between version 1.1.0 and version 1.2.0. An update to any previous release lets us still build the package.

12.5.7.3 Common Configuration Scripts

Common configuration scripts that are independent of version, platform and architecture are useful to everyone. These live in the Source Builder's configuration directory. Currently there are scripts to build binutils, expat, DTC, GCC, GDB and libusb. These files contain the recipes to build these package without the specific details of the versions or patches being built. They expect to be wrapped by a configuration file that ties the package to a specific version and optionally specific patches.

12.5.7.4 DTC Example

We will be building the DTC for your host rather than a package for RTEMS. We will create a file called `source-builder/config/dtc-1-1.cfg`. This is a common script that can be used to build a specific version using a general recipe. The file name is `dtc-1-1.cfg` where the `cfg` extension indicates this is a configuration file. The first 1 says this is for the major release 1 of the package and the last 1 is the build configuration version.

The file starts with some comments that detail the configuration. If there is anything unusual about the configuration it is a good idea to add something in the comments here. The comments are followed by a check for the release. In this case if a release is not provided a default of 1 is used:

```

1 #
2 # DTC 1.x.x Version 1.
3 #
4 # This configuration file configure's, make's and install's DTC.
5 #
6
7 %if %{release} == %{nil}
8 %define release 1
9 %endif

```

The next section defines some information about the package. It does not effect the build and is used to annotate the reports. It is recommended this information is kept updated and accurate:

```

1 Name:      dtc-%{dtc_version}-%{_host}-%{release}
2 Summary:   Device Tree Compiler v%{dtc_version} for target %{_target} on host %{_host}
3 Version:  %{dtc_version}
4 Release:  %{release}
5 URL:      http://www.jdl.com/software/
6 BuildRoot: %{_tmppath}/%{name}-root-%(%{__id_u} -n)

```

The next section defines the source and any patches. In this case there is a single source package and it can be downloaded using the HTTP protocol. The RSB knows this is GZip'ped tar file. If more than one package is needed, add them increasing the index. The gcc-4.8-1.cfg configuration contains examples of more than one source package as well as conditionally including source packages based on the outer configuration options:

```

1 #
2 # Source
3 #
4 %source set dtc http://www.jdl.com/software/dtc-v%{dtc_version}.tgz

```

The remainder of the script is broken in to the various phases of a build. They are:

. Preperation . Bulding . Installing, and . Cleaning

Preparation is the unpacking of the source, applying any patches as well as any package specific set ups. This part of the script is a standard Unix shell script. Be careful with the use of % and \$. The RSB uses % while the shell scripts use \$.

A standard pattern you will observe is the saving of the build's top directory. This is used instead of changing into a subdirectory and then changing to the parent when finished. Some hosts will change in a subdirectory that is a link however changing to the parent does not change back to the parent of the link rather it changes to the parent of the target of the link and that is something the RSB nor you can track easily. The RSB configuration script's are a collection of various subtle issues so please ask if you are unsure why something is being done a particular way.

The preparation phase will often include source and patch setup commands. Outer layers can set the source package and add patches as needed while being able to use a common recipe for the build. Users can override the standard build and supply a custom patch for testing using the user macro command line interface:

```

1 #
2 # Prepare the source code.
3 #
4 %prep
5     build_top=$(pwd)
6
7     %source setup dtc -q -n dtc-v{%{dtc_version}}
8     %patch setup dtc -p1
9
10    cd ${build_top}

```

The configuration file `gcc-common-1.cfg` is a complex example of source preparation. It contains a number of source packages and patches and it combines these into a single source tree for building. It uses links to map source into the GCC source tree so GCC can be built using the *single source tree* method. It also shows how to fetch source code from version control. Newlib is taken directly from its CVS repository.

Next is the building phase and for the DTC example this is simply a matter of running `make`. Note the use of the RSB macros for commands. In the case of `%{__make}` it maps to the correct make for your host. In the case of BSD systems we need to use the BSD make and not the GNU make.

If your package requires a configuration stage you need to run this before the make stage. Again the GCC common configuration file provides a detailed example:

```

1 %build
2     build_top=$(pwd)
3
4     cd dtc-v{%{dtc_version}}
5
6     %{build_build_flags}
7
8     %{__make} PREFIX=%{_prefix}
9
10    cd ${build_top}

```

You can invoke `make` with the macro `%{?_smp_flags}` as a command line argument. This macro is controlled by the `--jobs` command line option and the host CPU detection support in the RSB. If you are on a multicore host you can increase the build speed using this macro. It also lets you disabled building on multicores to aid debugging when testing.

Next is the install phase. This phase is a little more complex because you may be building a tar file and the end result of the build is never actually installed into the prefix on the build host and you may not even have permissions to perform a real install. Most packages install to the prefix and the prefix is typically supplied via the command to the RSB or the package's default is used. The default can vary depending on the host's operating system. To install to a path that is not the prefix the `DESTDIR` make variable is used. Most packages should honour the `DISTDIR` make variables and you can typically specify it on the command line to make when invoking the install target. This results in the package being installed to a location that is not the prefix but one you can control. The RSB provides a shell variable called `SB_BUILD_ROOT` you can use. In a build set where you are building a number of packages you can collect all the built packages in a single tree that is captured in the tar file.

Also note the use of the macro `%{__rmdir}`. The use of these macros allow the RSB to vary specific commands based on the host. This can help on hosts like Windows where bugs can

effect the standard commands such as `rm`. There are many many macros to help you. You can find these listed in the `defaults.mc` file and in the trace output. If you are new to creating and editing configurations learning these can take a little time:

```

1 %install
2   build_top=$(pwd)
3
4   %[_rmdir] -rf $SB_BUILD_ROOT
5
6   cd dtc-v%{dtc_version}
7   %[_make] DESTDIR=$SB_BUILD_ROOT PREFIX=%[_prefix] install
8
9   cd ${build_top}

```

Finally there is an optional clean section. The RSB will run this section if `--no-clean` has not been provided on the command line. The RSB does clean up for you.

Once we have the configuration files we can execute the build using the `sb-builder` command. The command will perform the build and create a tar file in the tar directory:

```

1 $ ../source-builder/sb-builder --prefix=/usr/local \
2   --log=log_dtc devel/dtc-1.2.0
3 RTEMS Source Builder, Package Builder v0.2.0
4 config: devel/dtc-1.2.0
5 package: dtc-1.2.0-x86_64-freebsd9.1-1
6 download: http://www.jdl.com/software/dtc-v1.2.0.tgz -> sources/dtc-v1.2.0.tgz
7 building: dtc-1.2.0-x86_64-freebsd9.1-1
8 $ ls tar
9 dtc-1.2.0-x86_64-freebsd9.1-1.tar.bz2

```

If you want to have the package installed automatically you need to create a build set. A build set can build one or more packages from their configurations at once to create a single package. For example the GNU tools is typically seen as `binutils`, `GCC` and `GDB` and a build set will build each of these packages and create a single build set tar file or install the tools on the host into the prefix path.

The DTC build set file is called `dtc.bset` and contains:

```

1 #
2 # Build the DTC.
3 #
4
5 %define release 1
6
7 devel/dtc-1.2.0.cfg

```

To build this you can use something similar to:

```

1 $ ../source-builder/sb-set-builder --prefix=/usr/local --log=log_dtc \
2   --trace --bset-tar-file --no-install dtc
3 RTEMS Source Builder - Set Builder, v0.2.0
4 Build Set: dtc
5 config: devel/dtc-1.2.0.cfg
6 package: dtc-1.2.0-x86_64-freebsd9.1-1
7 building: dtc-1.2.0-x86_64-freebsd9.1-1
8 tarball: tar/x86_64-freebsd9.1-dtc-set.tar.bz2

```

(continues on next page)

(continued from previous page)

```

9 cleaning: dtc-1.2.0-x86_64-freebsd9.1-1
10 Build Set: Time 0:00:02.865758
11 $ ls tar
12 dtc-1.2.0-x86_64-freebsd9.1-1.tar.bz2  x86_64-freebsd9.1-dtc-set.tar.bz2

```

The build is for a FreeBSD host and the prefix is for user installed packages. In this example I cannot let the source builder perform the install because I never run the RSB with root privileges so a build set or bset tar file is created. This can then be installed using root privileges.

The command also supplies the `--trace` option. The output in the log file will contain all the macros.

12.5.7.5 Debugging

New configuration files require debugging. There are two types of debugging. The first is debugging RSB script bugs. The `--dry-run` option is used here. Supplying this option will result in most of the RSB processing to be performed and suitable output placed in the log file. This with the `--trace` option should help you resolve any issues.

The second type of bug to fix are related to the execution of one of phases. These are usually a mix of shell script bugs or package set up or configuration bugs. Here you can use any normal shell script type debug technique such as `set -x` to output the commands or `echo` statements. Debugging package related issues may require you start a build with the RSB and supply `--no-clean` option and then locate the build directories and change directory into them and manually run commands until to figure what the package requires.

12.5.8 Scripting

Configuration files specify how to build a package. Configuration files are scripts and have a `.cfg` file extension. The script format is based loosely on the RPM spec file format however the use and purpose in this tool does not compare with the functionality and therefore the important features of the spec format RPM needs and uses.

The script language is implemented in terms of macros. The built-in list is:

%{ }:

Macro expansion with conditional logic.

%():

Shell expansion.

%prep:

The source preparation shell commands.

%build:

The build shell commands.

%install:

The package install shell commands.

%clean:

The package clean shell commands.

%include:

Inline include another configuration file.

%name:

The name of the package.

%summary:

A brief package description. Useful when reporting about a build.

%release:

The package release. A number that is the release as built by this tool.

%version:

The package's version string.

%buildarch:

The build architecture.

%source:

Define a source code package. This macro has a number appended.

%patch:

Define a patch. This macro has a number appended.

%hash:

Define a checksum for a source or patch file.

%{echo message}:

Print the following string as a message.

%{warning message}:

Print the following string as a warning and continue.

%{error message}:

Print the following string as an error and exit.

%select:

Select the macro map. If there is no map nothing is reported.

%define:

Define a macro. Macros cannot be redefined, you must first undefine it.

%undefine:

Undefine a macro.

%if:

Start a conditional logic block that ends with a %endif.

%ifn:

Inverted start of a conditional logic block.

%ifarch:

Test the architecture against the following string.

%ifnarch:

Inverted test of the architecture

%ifos:

Test the host operating system.

%else:

Start the *else* conditional logic block.

%endfi:

End the conditional logic block.

%bconf_with:

Test the build condition *with* setting. This is the `--with-*` command line option.

%bconf_without:

Test the build condition *without* setting. This is the `--without-*` command line option.

12.5.8.1 Expanding

A macro can be `%{string}` or the equivalent of `%string`. The following macro expansions supported are:

%{string}:

Expand the 'string' replacing the entire macro text with the text in the table for the entry 'string'. For example if 'var' is 'foo' then `${var}` would become `foo`.

%{expand: string}:

Expand the 'string' and then use it as a string to the macro expanding the macro. For example if `foo` is set to `bar` and `bar` is set to `foobar` then `%{expand:foo}` would result in `foobar`. Shell expansion can also be used.

%{with string}:

Expand the macro to 1 if the macro `with_string` is defined else expand to 0. Macros with the name `with_string` can be define with command line arguments to the RTEMS Source Builder commands.

%{defined string}:

Expand the macro to 1 if a macro of name `string` is defined else expand to '0'.

%{?string: expression}:

Expand the macro to `expression` if a macro of name `string` is defined else expand to `%{nil}`.

%{!?string: expression}:

Expand the macro to `expression` if a macro of name `string` is not defined. If the macro is define expand to `%{nil}`.

%(expression):

Expand the macro to the result of running the expression in a host shell. It is assumed this is a Unix type shell. For example `%(whoami)` will return your user name and `%(date)` will return the current date string.

12.5.8.2 %prep

The `+%prep+` macro starts a block that continues until the next block macro. The *prep* or preparation block defines the setup of the package's source and is a mix of RTEMS Source Builder macros and shell scripting. The sequence is typically `+%source+` macros for source, `+%patch+` macros to patch the source mixed with some shell commands to correct any source issues:

```
1 %source setup gcc -q -c -T -n %{name}-${version}
```

Items:

1. The source group to set up is gcc.
2. The source's name is the macro %{name}.
3. The version of the source is the macro %{version}.

The source set up are declared with the source set and add commands. For example:

```
1 %source set gdb http://ftp.gnu.org/gnu/gdb/gdb-${gdb_version}.tar.bz2
```

This URL is the primary location of the GNU GDB source code and the RTEMS Source Builder can download the file from this location and by inspecting the file extension use bzip2 decompression with +tar+. When the %prep section is processed a check of the local source directory is made to see if the file has already been downloaded. If not found in the source cache directory the package is downloaded from the URL. You can append other base URLs via the command line option --url. This option accepts a comma delimited list of sites to try.

You could optionally have a few source files that make up the package. For example GNU's GCC was a few tar files for a while and it is now a single tar file. Support for multiple source files can be conditionally implemented with the following scripting:

```
1 %source set gcc ftp://ftp.gnu.org/gnu/gcc/gcc-${gcc_version}/gcc-code-${gcc_version}.tar.
  ↪bz2
2 %source add gcc ftp://ftp.gnu.org/gnu/gcc/gcc-${gcc_version}/gcc-g++-${gcc_version}.tar.
  ↪bz2
3 %source setup gcc -q -T -D -n gcc-${gcc_version}
```

Separate modules use separate source groups. The GNU GCC compiler for RTEMS uses Newlib, MPFR, MPC, and GMP source packages. You define the source with:

```
1 %source set gcc ftp://ftp.gnu.org/gnu/gcc/gcc-${gcc_version}/gcc-${gcc_version}.tar.bz2
2 %source set newlib ftp://sourceware.org/pub/newlib/newlib-${newlib_version}.tar.gz
3 %source set mpfr http://www.mpfr.org/mpfr-${mpfr_version}/mpfr-${mpfr_version}.tar.bz2
4 %source set mpc http://www.multiprecision.org/mpc/download/mpc-${mpc_version}.tar.gz
5 %source set gmp ftp://ftp.gnu.org/gnu/gmp/gmp-${gmp_version}.tar.bz2
```

and set up with:

```
1 %source setup gcc -q -n gcc-${gcc_version}
2 %source setup newlib -q -D -n newlib-${newlib_version}
3 %source setup mpfr -q -D -n mpfr-${mpfr_version}
4 %source setup mpc -q -D -n mpc-${mpc_version}
5 %source setup gmp -q -D -n gmp-${gmp_version}
```

Patching also occurs during the preparation stage. Patches are handled in a similar way to the source packages except you only add patches. Patches are applied using the +setup+ command. The +setup+ command takes the default patch option. You can provide options with each patch by adding them as arguments before the patch URL. Patches with no options uses the +setup+ default.

```

1 %patch add gdb %{rtems_gdb_patches}/gdb-sim-arange-inline.diff
2 %patch add gdb -p0 %{rtems_gdb_patches}/gdb-sim-cgen-inline.diff

```

Items:

1. This patch has the custom option of -p0.

To apply these patches:

```

1 %patch setup gdb -p1

```

Items:

1. The default options for gdb set up.

12.5.8.3 %build

The %build macro starts a block that continues until the next block macro. The build block is a series of shell commands that execute to build the package. It assumes all source code has been unpacked, patch and adjusted so the build will succeed.

The following is an example take from the GitHub STLink project. The STLink is a JTAG debugging device for the ST ARM family of processors:

```

1 %build
2   export PATH="%{_bindir}:${PATH}"
3
4   cd texane-stlink-%{stlink_version}
5
6   ./autogen.sh
7
8 %if "%{_build}" != "%{_host}"
9   CFLAGS_FOR_BUILD="-g -O2 -Wall" \
10 %endif
11   CPPFLAGS="-I $SB_TMPPREFIX/include/libusb-1.0" \
12   CFLAGS="$SB_OPT_FLAGS" \
13   LDFLAGS="-L $SB_TMPPREFIX/lib" \
14   ./configure \
15     --build=%{_build} --host=%{_host} \
16     --verbose \
17     --prefix=%{_prefix} --bindir=%{_bindir} \
18     --exec-prefix=%{_exec_prefix} \
19     --includedir=%{_includedir} --libdir=%{_libdir} \
20     --mandir=%{_mandir} --infodir=%{_infodir}
21
22   %{__make} %{?_smp_mflags} all
23
24   cd ..

```

Items:

1. Set up the PATH environment variable by setting the PATH environment variable. This is not always needed.
2. This package builds in the source tree `texane-stlink-#{stlink_version}` so enter it before building.
3. The package is actually checked directly out from the github project and so it needs its `autoconf` and `automake` files generated. Invoke the provided script `autogen.sh`
4. If the build machine and host are not the same the build is a cross-compile. Update the flags for a cross-compiled build.
5. The flags set in the environment before `configure` are various settings that need to be passed to customise the build. In this example an include path is being set to the install point of `libusb`. This package requires `libusb` is built before it.
6. The `configure` command. The RTEMS Source Builder provides all the needed paths as macro variables. You just need to provide them to `configure`.
7. Run `make`. Do not use `make` directly, use the RTEMS Source Builder's defined value. This value is specific to the host. A large number of packages need GNU `make` and on BSD systems this is `gmake`. You can optionally add the SMP flags if the packages build system can handle parallel building with multiple jobs. The `_smp_mflags` value is automatically setup for SMP hosts to match the number of cores the host has.

12.5.8.4 %install

The `%install` macro starts a block that continues until the next block macro. The `install` block is a series of shell commands that execute to install the package. You can assume the package has built correctly when this block starts executing.

Never install the package to the actual *prefix* the package was built with. Always install to the RTEMS Source Builder's temporary path defined in the macro variable `__tmpdir`. The RTEMS Source Builder sets up a shell environment variable called `SB_BUILD_ROOT` as the standard install point. Most packages support adding `DESTDIR=` to the `make install` command.

Looking at the same example as in `%build` (page 273):

```

1 %install
2   export PATH="%{_bindir}:${PATH}" <1>
3   rm -rf $SB_BUILD_ROOT <2>
4
5   cd texane-stlink-#{stlink_version} <3>
6   %{__make} DESTDIR=$SB_BUILD_ROOT install <4>
7
8   cd ..

```

Items:

1. Setup the PATH environment variable. This is not always needed.

2. Clean any installed files. This makes sure the install is just what the package installs and not any left over files from a broken build or install.
3. Enter the build directory. In this example it just happens to be the source directory.
4. Run `make install` to install the package overriding the `DESTDIR` make variable.

12.5.8.5 %clean

The `%clean` macro starts a block that continues until the next block macro. The clean block is a series of shell commands that execute to clean up after a package has been built and install. This macro is currently not been used because the RTEMS Source Builder automatically cleans up.

12.5.8.6 %include

The `%include` macro inline includes the specific file. The `__confdir` path is searched. Any relative path component of the include file is appended to each part of the `__configdir`. Adding an extension is optional as files with `.bset` and `.cfg` are automatically searched for.

Inline including means the file is processed as part of the configuration at the point it is included. Parsing continues from the next line in the configuration file that contains the `%include` macro.

Including files allow a kind of configuration file reuse. The outer configuration files provide specific information such as package version numbers and patches and then include a generic configuration script which builds the package:

```
1 %include %[_configdir]/gcc-4.7-1.cfg
```

12.5.8.7 %name

The name of the package being built. The name typically contains the components of the package and their version number plus a revision number. For the GCC with Newlib configuration the name is typically:

```
1 Name: %[_target]-gcc-%[gcc_version]-newlib-%[newlib_version]-%[release]
```

12.5.8.8 %summary

The `%summary` is a brief description of the package. It is useful when reporting. This information is not capture in the package anywhere. For the GCC with Newlib configuration the summary is typically:

```
1 Summary: GCC v%[gcc_version] and Newlib v%[newlib_version] for target %[_target] on host %
  ↳[_host]
```

12.5.8.9 %release

The %release is a packaging number that allows revisions of a package to happen where no package versions change. This value typically increases when the configuration building the package changes:

```
1 %define release 1
```

12.5.8.10 %version

The %version macro sets the version the package. If the package is a single component it tracks that component's version number. For example in the libusb configuration the %version is the same as %libusb_version, however in a GCC with Newlib configuration there is no single version number. In this case the GCC version is used:

```
1 Version: %{gcc_version}
```

12.5.8.11 %buildarch

The %buildarch macro is set to the architecture the package contains. This is currently not used in the RTEMS Source Builder and may go away. This macro is more important in a real packaging system where the package could end up on the wrong architecture.

12.5.8.12 %source

The %source macro has 3 commands that controls what it does. You can set the source files, add source files to a source group, and setup the source file group getting it ready to be used.

Source files are source code files in tar or zip files that are unpacked, copied or symbolically linked into the package's build tree. Building a package requires one or more dependent packages. These are typically the packages source code plus dependent libraries or modules. You can create any number of these source groups and set each of them up with a separate source group for each needed library or module. Each source group normally has a single tar, zip or repository and the set defines this. Some projects split the source code into separate tar or zip files and you install them by using the add command.

The first instance of a set command creates the source group and sets the source files to be set up. Subsequent set commands for the same source group are ignored. this lets you define the standard source files and override them for specific releases or snapshots. To set a source file group:

```
1 %source set gcc ftp://ftp.gnu.org/gnu/gcc/gcc-%{gcc_version}/gcc-%{gcc_version}.tar.bz2
```

Items:

1. The source group is gcc.

To add another source package to be installed into the same source tree you use the add command:


```
1 %source add gcc ftp://ftp.gnu.org/gnu/gcc/gcc-{gcc_version}/g++-{gcc_version}.tar.bz2
```

The source setup command can only be issued in the %prep: section. The setup is:

```
1 %source gcc setup -q -T -D -n {name}-{version}
```

Accepted options are:

-n:

The -n option is used to set the name of the software's build directory. This is necessary only when the source archive unpacks into a directory named other than <name>-<version>.

-c:

The -c option is used to direct %setup to create the top-level build directory before unpacking the sources.

-D:

The -D option is used to direct %setup to not delete the build directory prior to unpacking the sources. This option is used when more than one source archive is to be unpacked into the build directory, normally with the -b or -a options.

-T:

The -T option is used to direct %setup to not perform the default unpacking of the source archive specified by the first Source: macro. It is used with the -a or -b options.

-b <n>:

The -b option is used to direct %setup to unpack the source archive specified on the nth Source: macro line before changing directory into the build directory.

12.5.8.13 %patch

The %patch macro has the same 3 command as the %source command however the set commands is not really that useful with the %patch command. You add patches with the add command and setup applies the patches. Patch options can be added to each patch by placing them before the patch URL. If no patch option is provided the default options passed to the setup command are used. An option starts with a -. The setup command must reside inside the %prep section.

Patches are grouped in a similar way to the %source macro so you can control applying a group of patches to a specific source tree.

The __patchdir path is searched.

To add a patch:

```
1 %patch add gcc gcc-4.7.2-rtems4.11-20121026.diff
2 %patch add gcc -p0 gcc-4.7.2-rtems4.11-20121101.diff
```

Items:

1. The patch group is gcc.
2. Option -p0 is this specific to this patch.

Placing `%patch` setup in the `%prep` section will apply the groups patches:

```
1 .. code-block:: spec
```

```
    %patch setup gcc -p1
```

1. The patch group is `gcc`.
2. The default option used to apply the patch is `-p1`.

12.5.8.14 %hash

The `%hash` macro requires 3 arguments and defines a checksum for a specific file. The checksum is not applied until the file is checked before downloading and once downloaded. A patch or source file that does not have a hash defined generates a warning.

A file to be checksummed must be unique in the source and patch directories. The basename of the file is used as the key for the hash.

The hash algorithm can be `md5`, `sha1`, `sha224`, `sha256`, `sha384`, and `sha512` and we typically use `md5`.

To add a hash:

```
1 %hash md5 <1> net-snmp-%{net_snmp_version}.tar.gz <2> 7db683faba037249837b226f64d566d4 <3>
```

Items:

1. The type of checksum.
2. The file to checksum. It can contain macros that are expanded for you.
3. The MD5 hash for the Net-SNMP file `net-snmp-5.7.2.1.tar.gz`.

Do not include a path with the file name. Only the basename is required. Files can be searched for from a number of places and having a path component would create confusion. This does mean files with hashes must be unique.

Downloading off repositories such as `git` and `cvs` cannot be checksummed. It is assumed those protocols and tools manage the state of the files.

12.5.8.15 %echo

The `%echo` macro outputs the following string to `stdout`. This can also be used as `%{echo: message}`.

12.5.8.16 %warning

The `%warning` macro outputs the following string as a warning. This can also be used as `%{warning: message}`.

12.5.8.17 %error

The %error macro outputs the follow string as an error and exits the RTEMS Source Builder. This can also be used as %{error: message}.

12.5.8.18 %select

The %select macro selects the map specified. If there is no map no error or warning is generated. Macro maps provide a simple way for a user to override the settings in a configuration file without having to edit it. The changes are recorded in the build report so they can be traced.

Configurations use different maps so macro overrides can target a specific package.

The default map is global:

```
1 %select gcc-4.8-snapshot <1>
2 %define one_plus_one 2 <2>
```

Items:

1. The map switches to gcc-4.8-snapshot. Any overrides in this map will be used.
2. Defining macros only updates the global map and not the selected map.

12.5.8.19 %define

The %define macro defines a new macro or updates an existing one. If no value is given it is assumed to be 1:

```
1 %define foo bar
2 %define one_plus_one 2
3 %define one <1>
```

Items:

1. The macro `_one_` is set to 1.

12.5.8.20 %undefine

The %undefine macro removes a macro if it exists. Any further references to it will result in an undefine macro error.

12.5.8.21 %if

The %if macro starts a conditional logic block that can optionally have a *else* section. A test follows this macro and can have the following operators:

<code>%{}</code>	<p>Check the macro is set or <i>true</i>, ie non-zero:</p> <pre> 1 %if \${foo} 2 %warning The test passes, must not be ↳ empty or is non-zero 3 %else 4 %error The test fails, must be empty or ↳ zero 5 %endif </pre>
<code>!</code>	<p>The <i>not</i> operator inverts the test of the macro:</p> <pre> 1 %if ! \${foo} 2 %warning The test passes, must be empty ↳ or zero 3 %else 4 %error The test fails, must not be empty ↳ or is non-zero 5 %endif </pre>
<code>==</code>	<p>The left hand size must equal the right hand side. For example:</p> <pre> 1 %define one 1 2 %if \${one} == 1 3 %warning The test passes 4 %else 5 %error The test fails 6 %endif </pre> <p>You can also check to see if a macro is empty:</p> <pre> 1 %if \${nothing} == %{nil} 2 %warning The test passes 3 %else 4 %error The test fails </pre>
<code>!=</code>	<p>The left hand size does not equal the right hand side. For example:</p> <pre> 1 # 2 # Check a value not being equal. 3 # 4 %define one 1 5 %if \${one} != 2 6 %warning The test passes 7 %else 8 %error The test fails 9 %endif 10 # 11 # Check if a macro is set. 12 # 13 %if \${something} != %{nil} 14 %warning The test passes 15 %else 16 %error The test fails 17 %endif </pre>
<code>></code>	The left hand side is numerically greater than the right hand side.
<code>>=</code>	The left hand side is numerically greater than or equal to the right hand side.
<code><</code>	The left hand side is numerically less than the right hand side.
<code><=</code>	The left hand side is numerically less than or equal to the right hand side.

12.5.8.22 %ifn

The %ifn macro inverts the normal %if logic. It avoids needing to provide empty *if* blocks followed by *else* blocks. It is useful when checking if a macro is defined:

```
1 %ifn %{defined foo}  
2   %define foo bar  
3 %endif
```

12.5.8.23 %ifarch

The %ifarch is a short cut for %if %{_arch} == i386. Currently not used.

12.5.8.24 %ifnarch

The %ifnarch is a short cut for %if %{_arch} != i386. Currently not used.

12.5.8.25 %ifos

The %ifos is a short cut for %if %{_os} != mingw32. It allows conditional support for various operating system differences when building packages.

12.5.8.26 %else

The %else macro starts the conditional *else* block.

12.5.8.27 %endfi

The %endif macro ends a conditional logic block.

12.5.8.28 %bconf_with

The %bconf_with macro provides a way to test if the user has passed a specific option on the command line with the --with-<label> option. This option is only available with the sb-builder command.

12.5.8.29 %bconf_without

The %bconf_without macro provides a way to test if the user has passed a specific option on the command line with the --without-<label> option. This option is only available with the sb-builder command.

12.6 Commands

12.6.1 Checker (sb-check)

This commands checks your system is set up correctly. Most options are ignored:

```

1 $ ../source-builder/sb-check --help
2 sb-check: [options] [args]
3 RTEMS Source Builder, an RTEMS Tools Project (c) 2012-2013 Chris Johns
4 Options and arguments:
5 --force           : Force the build to proceed
6 --quiet           : Quiet output (not used)
7 --trace           : Trace the execution
8 --dry-run         : Do everything but actually run the build
9 --warn-all        : Generate warnings
10 --no-clean        : Do not clean up the build tree
11 --always-clean    : Always clean the build tree, even with an error
12 --jobs            : Run with specified number of jobs, default: num CPUs.
13 --host            : Set the host triplet
14 --build           : Set the build triplet
15 --target          : Set the target triplet
16 --prefix path     : Tools build prefix, ie where they are installed
17 --topdir path     : Top of the build tree, default is $PWD
18 --configdir path  : Path to the configuration directory, default: ./config
19 --builddir path   : Path to the build directory, default: ./build
20 --sourcedir path  : Path to the source directory, default: ./source
21 --patchdir path   : Path to the patches directory, default: ./patches
22 --tmppath path    : Path to the temp directory, default: ./tmp
23 --macros file[,file] : Macro format files to load after the defaults
24 --log file        : Log file where all build out is written too
25 --url url[,url]   : URL to look for source
26 --no-download     : Disable the source downloader
27 --targetcflags flags : List of C flags for the target code
28 --targetcxxflags flags : List of C++ flags for the target code
29 --libstdcxxflags flags : List of C++ flags to build the target libstdc++ code
30 --with-<label>     : Add the --with-<label> to the build
31 --without-<label>  : Add the --without-<label> to the build
32 --regression      : Set --no-install, --keep-going and --always-clean
33 $ ../source-builder/sb-check
34 RTEMS Source Builder - Check, v0.2.0
35 Environment is ok

```

12.6.2 Defaults (sb-defaults)

This commands outputs and the default macros for your when given no arguments. Most options are ignored:

```

1 $ ../source-builder/sb-defaults --help
2 sb-defaults: [options] [args]
3 RTEMS Source Builder, an RTEMS Tools Project (c) 2012-2013 Chris Johns
4 Options and arguments:
5 --force           : Force the build to proceed
6 --quiet           : Quiet output (not used)
7 --trace           : Trace the execution

```

(continues on next page)

(continued from previous page)

```

8 --dry-run           : Do everything but actually run the build
9 --warn-all         : Generate warnings
10 --no-clean          : Do not clean up the build tree
11 --always-clean      : Always clean the build tree, even with an error
12 --jobs              : Run with specified number of jobs, default: num CPUs.
13 --host              : Set the host triplet
14 --build              : Set the build triplet
15 --target             : Set the target triplet
16 --prefix path       : Tools build prefix, ie where they are installed
17 --topdir path        : Top of the build tree, default is $PWD
18 --configdir path     : Path to the configuration directory, default: ./config
19 --builddir path      : Path to the build directory, default: ./build
20 --sourcedir path     : Path to the source directory, default: ./source
21 --patchdir path      : Path to the patches directory, default: ./patches
22 --tmppath path       : Path to the temp directory, default: ./tmp
23 --macros file[,file] : Macro format files to load after the defaults
24 --log file           : Log file where all build out is written too
25 --url url[,url]      : URL to look for source
26 --no-download        : Disable the source downloader
27 --targetcflags flags : List of C flags for the target code
28 --targetcxxflags flags : List of C++ flags for the target code
29 --libstdcxxflags flags : List of C++ flags to build the target libstdc++ code
30 --with-<label>        : Add the --with-<label> to the build
31 --without-<label>     : Add the --without-<label> to the build
32 --regression         : Set --no-install, --keep-going and --always-clean

```

12.6.3 Set Builder (sb-set-builder)

This command builds a set:

```

1 $ ../source-builder/sb-set-builder --help
2 RTEMS Source Builder, an RTEMS Tools Project (c) 2012-2013 Chris Johns
3 Options and arguments:
4 --force              : Force the build to proceed
5 --quiet              : Quiet output (not used)
6 --trace              : Trace the execution
7 --dry-run            : Do everything but actually run the build
8 --warn-all          : Generate warnings
9 --no-clean           : Do not clean up the build tree
10 --always-clean       : Always clean the build tree, even with an error
11 --regression         : Set --no-install, --keep-going and --always-clean
12 ---jobs              : Run with specified number of jobs, default: num CPUs.
13 --host               : Set the host triplet
14 --build               : Set the build triplet
15 --target              : Set the target triplet
16 --prefix path        : Tools build prefix, ie where they are installed
17 --topdir path         : Top of the build tree, default is $PWD
18 --configdir path     : Path to the configuration directory, default: ./config
19 --builddir path      : Path to the build directory, default: ./build
20 --sourcedir path     : Path to the source directory, default: ./source
21 --patchdir path      : Path to the patches directory, default: ./patches
22 --tmppath path       : Path to the temp directory, default: ./tmp
23 --macros file[,file] : Macro format files to load after the defaults
24 --log file           : Log file where all build out is written too

```

(continues on next page)

(continued from previous page)

```

25 --url url[,url]      : URL to look for source
26 --no-download       : Disable the source downloader
27 --no-install        : Do not install the packages to the prefix
28 --targetcflags flags : List of C flags for the target code
29 --targetcxxflags flags : List of C++ flags for the target code
30 --libstdcxxflags flags : List of C++ flags to build the target libstdc++ code
31 --with-<label>       : Add the --with-<label> to the build
32 --without-<label>    : Add the --without-<label> to the build
33 --mail-from          : Email address the report is from.
34 --mail-to            : Email address to send the email too.
35 --mail              : Send email report or results.
36 --smtp-host          : SMTP host to send via.
37 --no-report          : Do not create a package report.
38 --report-format      : The report format (text, html, asciidoc).
39 --bset-tar-file      : Create a build set tar file
40 --pkg-tar-files      : Create package tar files
41 --list-bsets         : List available build sets
42 --list-configs       : List available configurations
43 --list-deps          : List the dependent files.

```

The arguments are a list of build sets to build.

Options:

--force:

Force the build to proceed even if the host check fails. Typically this happens if executable files are found in the path at a different location to the host defaults.

--trace:

Trace enable printing of debug information to stdout. It is really only of use to RTEMS Source Builder's developers.

--dry-run:

Do everything but actually run the build commands. This is useful when checking a new configuration parses cleanly.

--warn-all:

Generate warnings.

--no-clean:

Do not clean up the build tree during the cleaning phase of the build. This leaves the source and the build output on disk so you can make changes, or amend or generate new patches. It also allows you to review configure type output such as config.log.

--always-clean:

Clean away the results of a build even if the build fails. This is normally used with --keep-going when regression testing to see which build sets fail to build. It keeps the disk usage down.

--jobs:

Control the number of jobs make is given. The jobs can be none for only 1 job, half so the number of jobs is half the number of detected cores, a fraction such as 0.25 so the number of jobs is a quarter of the number of detected cores and a number such as 25 which forces the number of jobs to that number.

--host:

Set the host triplet value. Be careful with this option.

--build:

Set the build triplet. Be careful with this option.

--target:

Set the target triplet. Be careful with this option. This is useful if you have a generic configuration script that can work for a range of architectures.

--prefix path:

Tools build prefix, ie where they are installed.

--topdir path:

Top of the build tree, that is the current directory you are in.

--configdir path:

Path to the configuration directory. This overrides the built in defaults.

--builddir path:

Path to the build directory. This overrides the default of +build+.

--sourcedir path:

Path to the source directory. This overrides the default of +source+.

--patchdir path:

Path to the patches directory. This overrides the default of +patches+.

--tmppath path:

Path to the temporary directory. This overrides the default of +tmp+.

--macros files:

Macro files to load. The configuration directory path is searched.

--log file:

Log all the output from the build process. The output is directed to +stdout+ if no log file is provided.

--url url:

URL to look for source when downloading. This is can be comma separate list.

--no-download:

Disable downloading of source and patches. If the source is not found an error is raised.

--targetcflags flags:

List of C flags for the target code. This allows for specific local customisation when testing new variations.

--targetcxxflags flags:

List of C++ flags for the target code. This allows for specific local customisation when testing new variations.

--libstdcxxflags flags:

List of C++ flags to build the target libstdc++ code. This allows for specific local customisation when testing new variations.

--with-<label>:

Add the --with-<label> to the build. This can be tested for in a script with the %bconf_with macro.

--without-<label>:

Add the --without-<label> to the build. This can be tested for in a script with the %bconf_without macro.

--mail-from:

Set the from mail address if report mailing is enabled.

--mail-to:

Set the to mail address if report mailing is enabled. The report is mailed to this address.

--mail:

Mail the build report to the mail to address.

--smtp-host:

The SMTP host to use to send the email. The default is localhost.

--no-report:

Do not create a report format.

--report-format format:

The report format can be text or html. The default is html.

--keep-going:

Do not stop on error. This is useful if your build sets performs a large number of testing related builds and there are errors.

--always-clean:

Always clean the build tree even with a failure.

--no-install:

Do not install the packages to the prefix. Use this if you are only after the tar files.

--regression:

A convenience option which is the same as --no-install, --keep-going and --always-clean.

--bset-tar-file:

Create a build set tar file. This is a single tar file of all the packages in the build set.

--pkg-tar-files:

Create package tar files. A tar file will be created for each package built in a build set.

--list-bsets:

List available build sets.

--list-configs:

List available configurations.

--list-deps:

Print a list of dependent files used by a build set. Dependent files have a dep[?]' prefix where '?' is a number. The files are listed alphabetically.

12.6.4 Set Builder (sb-builder)

This command builds a configuration as described in a configuration file. Configuration files have the extension of .cfg:

```

1 $ ./source-builder/sb-builder --help
2 sb-builder: [options] [args]
3 RTEMS Source Builder, an RTEMS Tools Project (c) 2012 Chris Johns
4 Options and arguments:
5 --force           : Force the build to proceed
6 --quiet           : Quiet output (not used)
```

(continues on next page)

(continued from previous page)

```

7 --trace           : Trace the execution
8 --dry-run        : Do everything but actually run the build
9 --warn-all       : Generate warnings
10 --no-clean       : Do not clean up the build tree
11 --always-clean   : Always clean the build tree, even with an error
12 --jobs           : Run with specified number of jobs, default: num CPUs.
13 --host           : Set the host triplet
14 --build          : Set the build triplet
15 --target         : Set the target triplet
16 --prefix path    : Tools build prefix, ie where they are installed
17 --topdir path    : Top of the build tree, default is $PWD
18 --configdir path : Path to the configuration directory, default: ./config
19 --builddir path  : Path to the build directory, default: ./build
20 --sourcedir path : Path to the source directory, default: ./source
21 --patchdir path  : Path to the patches directory, default: ./patches
22 --tmppath path   : Path to the temp directory, default: ./tmp
23 --macros file[,file] : Macro format files to load after the defaults
24 --log file       : Log file where all build out is written too
25 --url url[,url]  : URL to look for source
26 --targetcflags flags : List of C flags for the target code
27 --targetcxxflags flags : List of C++ flags for the target code
28 --libstdcxxflags flags : List of C++ flags to build the target libstdc++ code
29 --with-<label>    : Add the --with-<label> to the build
30 --without-<label> : Add the --without-<label> to the build
31 --list-configs    : List available configurations

```

12.7 Building and Deploying Tool Binaries

If you wish to create and distribute your build or you want to archive a build you can create a tar file. We term this deploying a build. This is a more advanced method for binary packaging and installing of tools.

By default the RTEMS Source Builder installs the built packages directly and optionally it can also create a *build set tar file* or a *package tar file* per package built. The normal and default behaviour is to let the RTEMS Source Builder install the tools. The source will be downloaded, built, installed and cleaned up.

The tar files are created with the full build prefix present and if you follow the examples given in this documentation the path is absolute. This can cause problems if you are installing on a host you do not have super user or administrator rights on because the prefix path may references part you do not have write access too and tar will not extract the files. You can use the `--strip-components` option in tar if your host tar application supports it to remove the parts you do not have write access too or you may need to unpack the tar file somewhere and copy the file tree from the level you have write access from. Embedding the full prefix path in the tar files lets you know what the prefix is and is recommended. For example if `/home/chris/development/rtems/4.11` is the prefix used you cannot change directory to the root (`/`) and untar the file because the `/home` is root access only. To install a tar file you have downloaded into your new machine's Downloads directory in your home directoty you would enter:

```
1 $ cd /somewhere
2 $ tar --strip-components=3 -xjf \
3     $HOME/Downloads/rtems-4.11-sparc-rtems4.11-1.tar.bz2
```

A build set tar file is created by adding `--bset-tar-file` option to the `sb-set-builder` command:

```
1 $ ../source-builder/sb-set-builder --log=l-sparc.txt \
2     --prefix=$HOME/development/rtems/4.11 \
3     --bset-tar-file \      <1>
4     4.11/rtems-sparc
5 Source Builder - Set Builder, v0.2.0
6 Build Set: 4.11/rtems-sparc
7 config: expat-2.1.0-1.cfg
8 package: expat-2.1.0-x86_64-freebsd9.1-1
9 building: expat-2.1.0-x86_64-freebsd9.1-1
10 config: tools/rtems-binutils-2.22-1.cfg
11 package: sparc-rtems4.11-binutils-2.22-1
12 building: sparc-rtems4.11-binutils-2.22-1
13 config: tools/rtems-gcc-4.7.2-newlib-1.20.0-1.cfg
14 package: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
15 building: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
16 config: tools/rtems-gdb-7.5.1-1.cfg
17 package: sparc-rtems4.11-gdb-7.5.1-1
18 building: sparc-rtems4.11-gdb-7.5.1-1
19 installing: rtems-4.11-sparc-rtems4.11-1 -> /home/chris/development/rtems/4.11 <2>
20 installing: rtems-4.11-sparc-rtems4.11-1 -> /home/chris/development/rtems/4.11
21 installing: rtems-4.11-sparc-rtems4.11-1 -> /home/chris/development/rtems/4.11
22 installing: rtems-4.11-sparc-rtems4.11-1 -> /home/chris/development/rtems/4.11
23 tarball: tar/rtems-4.11-sparc-rtems4.11-1.tar.bz2      <3>
24 cleaning: expat-2.1.0-x86_64-freebsd9.1-1
25 cleaning: sparc-rtems4.11-binutils-2.22-1
```

(continues on next page)

(continued from previous page)

```

26 cleaning: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
27 cleaning: sparc-rtems4.11-gdb-7.5.1-1
28 Build Set: Time 0:15:25.92873

```

Items

1. The option to create a build set tar file.
2. The installation still happens unless you specify `--no-install`.
3. Creating the build set tar file.

You can also suppress installing the files using the `--no-install` option. This is useful if your prefix is not accessible, for example when building Canadian cross compiled tool sets:

```

1 $ ../source-builder/sb-set-builder --log=l-sparc.txt \
2     --prefix=$HOME/development/rtems/4.11 \
3     --bset-tar-file \
4     --no-install \      <1>
5     4.11/rtems-sparc
6 Source Builder - Set Builder, v0.2.0
7 Build Set: 4.11/rtems-sparc
8 config: expat-2.1.0-1.cfg
9 package: expat-2.1.0-x86_64-freebsd9.1-1
10 building: expat-2.1.0-x86_64-freebsd9.1-1
11 config: tools/rtems-binutils-2.22-1.cfg
12 package: sparc-rtems4.11-binutils-2.22-1
13 building: sparc-rtems4.11-binutils-2.22-1
14 config: tools/rtems-gcc-4.7.2-newlib-1.20.0-1.cfg
15 package: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
16 building: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
17 config: tools/rtems-gdb-7.5.1-1.cfg
18 package: sparc-rtems4.11-gdb-7.5.1-1
19 building: sparc-rtems4.11-gdb-7.5.1-1
20 tarball: tar/rtems-4.11-sparc-rtems4.11-1.tar.bz2      <2>
21 cleaning: expat-2.1.0-x86_64-freebsd9.1-1
22 cleaning: sparc-rtems4.11-binutils-2.22-1
23 cleaning: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
24 cleaning: sparc-rtems4.11-gdb-7.5.1-1
25 Build Set: Time 0:14:11.721274
26 $ ls tar
27 rtems-4.11-sparc-rtems4.11-1.tar.bz2

```

Items

1. The option to suppressing installing the packages.
2. Create the build set tar.

A package tar file can be created by adding the `--pkg-tar-files` to the `sb-set-builder` command. This creates a tar file per package built in the build set:

```

1 $ ../source-builder/sb-set-builder --log=l-sparc.txt \
2     --prefix=$HOME/development/rtems/4.11 \
3     --bset-tar-file \
4     --pkg-tar-files \         <1>
5     --no-install 4.11/rtems-sparc
6 Source Builder - Set Builder, v0.2.0
7 Build Set: 4.11/rtems-sparc
8 config: expat-2.1.0-1.cfg
9 package: expat-2.1.0-x86_64-freebsd9.1-1
10 building: expat-2.1.0-x86_64-freebsd9.1-1
11 config: tools/rtems-binutils-2.22-1.cfg
12 package: sparc-rtems4.11-binutils-2.22-1
13 building: sparc-rtems4.11-binutils-2.22-1
14 config: tools/rtems-gcc-4.7.2-newlib-1.20.0-1.cfg
15 package: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
16 building: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
17 config: tools/rtems-gdb-7.5.1-1.cfg
18 package: sparc-rtems4.11-gdb-7.5.1-1
19 building: sparc-rtems4.11-gdb-7.5.1-1
20 tarball: tar/rtems-4.11-sparc-rtems4.11-1.tar.bz2
21 cleaning: expat-2.1.0-x86_64-freebsd9.1-1
22 cleaning: sparc-rtems4.11-binutils-2.22-1
23 cleaning: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
24 cleaning: sparc-rtems4.11-gdb-7.5.1-1
25 Build Set: Time 0:14:37.658460
26 $ ls tar
27 expat-2.1.0-x86_64-freebsd9.1-1.tar.bz2          sparc-rtems4.11-binutils-2.22-1.tar.bz2
28 sparc-rtems4.11-gdb-7.5.1-1.tar.bz2 <2>         rtems-4.11-sparc-rtems4.11-1.tar.bz2 <3>
29 sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1.tar.bz2

```

Items

1. The option to create packages tar files.
2. The GDB package tar file.
3. The build set tar file. All the others in a single tar file.

12.8 Bugs, Crashes, and Build Failures

The RTEMS Source Builder is a Python program and every care is taken to test the code however bugs, crashes, and build failures can and do happen. If you find a bug please report it via the [Developer Site](#) or email on the RTEMS Users list.

Please include the generated RSB report. If you see the following a report has been generated:

```

1 ...
2 ...
3 Build FAILED    <1>
4   See error report: rsb-report-4.11-rtems-lm32.txt    <2>

```

Items:

1. The build has failed.
2. The report's file name.

The generated report contains the command line, version of the RSB, your host's uname details, the version of Python and the last 200 lines of the log.

If for some reason there is no report please send please report the following:

- Command line,
- The git hash,
- Host details with the output of the `uname -a` command,
- If you have made any modifications.

If there is a Python crash please cut and paste the Python backtrace into the bug report. If the tools fail to build please locate the first error in the log file. This can be difficult to find on hosts with many cores so it sometimes pays to re-run the command with the `--jobs=None` option to get a log that is correctly sequenced. If searching the log file search for `error:` and the error should be just above it.

12.8.1 Contributing

We welcome all users adding, fixing, updating and upgrading packages and their configurations. The RSB is open source and open to contributions. These can be bug fixes, new features or new configurations. Please break patches down into changes to the core Python code, configuration changes or new configurations.

Please email patches generated using git so your commit messages and you are acknowledged as the contributor.

12.9 History

The RTEMS Source Builder is a stand alone tool based on another tool called the *SpecBuilder* written by Chris Johns. The *SpecBuilder* was written around 2010 for the RTEMS project to provide Chris with a way to build tools on hosts that did not support RPMs. At the time the RTEMS tools maintainer only supported *spec* files and these files held all the vital configuration data needed to create suitable tool sets. The available SRPM and *spec* files by themselves were of little use because a suitable rpm tool was needed to use them. At the time the available versions of rpm for a number of non-RPM hosts were broken and randomly maintained. The solution Chris settled on was to use the *spec* files and to write a Python based tool that parsed the *spec* file format creating a shell script that could be run to build the package. The approach proved successful and Chris was able to track the RPM version of the RTEMS tools on a non-RPM host for a number of years.

The *SpecBuilder* tool did not build tools or packages unrelated to the RTEMS Project where no suitable *spec* file was available so another tool was needed. Rather than start again Chris decided to take the parsing code for the *spec* file format and build a new tool called the RTEMS Source Builder.

GLOSSARY

ABI

Application Binary Interface

Architecture

Family or class of processor based around a common instruction set. RTEMS architectures follow the GCC architecture model as RTEMS needs an GCC architecture compiler for each support RTEMS architecture.

APA

Arbitrary Processor Affinity

API

Application Programming Interface

Binutils

GNU Binary Utilities such as the assembler `as`, linker `ld` and a range of other tools used in the development of software.

BSP

Board Support Package is a specific configuration RTEMS can be built for. An RTEMS install process installs specific library and headers files for a single BSP. A BSP optimises RTEMS to a specific target hardware.

Buildbot

A continuous integration build server.

C11

ISO/IEC 9899:2011

C++11

ISO/IEC 14882:2011

Crosscompiler

A compiler built to run on a Host that generate code for another architecture.

DLL

Dynamically Linker Library used on Windows.

EDF

Earliest Deadline First

EMB²

Embedded Multicore Building Blocks

FAT

File Allocation Table

Futex

Fast User-Space Locking

IMFS

In-Memory File System

JFFS2

Journalling Flash File System version 2

GCC

GNU Compiler Collection

GDB

GNU Debugger

GNU

GNU's Not Unix

Host

The computer and operating system that hosts the RTEMS development tools such as the compiler, linker and debugger.

MinGW

Minimal GNU system for Windows that lets GCC built programs use the standard Windows operating system DLLs. It lets you build native Windows programs with the GNU GCC compiler.

MinGW64

Minimal GNU system for 64bit Windows. MinGW64 is not the MinGW project.

MrsP

Multiprocessor Resource-Sharing Protocol

MSYS2

Minimal System 2 is a fork of the MinGW project's MSYS tool and the MinGW MSYS tool is a fork of Cygwin project. The Cygwin project provides a POSIX emulation layer for Windows so POSIX software can run on Windows. MSYS is a minimal version that is just enough to let configure scripts run. MSYS has a simplified path structure to make it easier to building native Windows programs.

NFSv2

Network File System version 2

OMIP

$O(m)$ Independence-Preserving Protocol

OpenMP

Open Multi-Processing

POSIX

Portable Operating System Interface is a standard that lets software be portable between compliant operating systems.

prefix

A path used when building a package so all parts of the package reside under that path.

RFS

RTEMS File System

RSB

RTEMS Source Builder is part of the RTEMS Tools Project. It builds packages such as the tools for the RTEMS operating system.

RTEMS

The Real-Time Executive for Multiprocessor Systems or RTEMS is an open source fully featured Real Time Operating System or RTOS that supports a variety of open standard application programming interfaces (API) and interface standards such as POSIX and BSD sockets.

SMP

Symmetric Multiprocessing

Target

A target is the hardware or simulator a BSP built executable runs on.

Test Suite

See Testsuite

Testsuite

RTEMS test suite located in the testsuites/ directory.

TLS

Thread-Local Storage

Waf

Waf build system. For more information see <http://www.waf.io/>

YAFFS2

[Yet Another Flash File System version 2](#)

INDEX

Symbols

- arch
 - command line option, 212
- bsp
 - command line option, 212
- build
 - command line option, 212
- build-path
 - command line option, 212
- config-report
 - command line option, 212
- convert-kernel
 - command line option, 229
- dry-run
 - command line option, 212
- jobs
 - command line option, 212
- log
 - command line option, 212
- net-boot
 - command line option, 228
- net-boot-dhcp
 - command line option, 228
- net-boot-fdt NET_BOOT_FDT
 - command line option, 229
- net-boot-file NET_BOOT_FILE
 - command line option, 228
- net-boot-ip NET_BOOT_IP
 - command line option, 228
- no-clean
 - command line option, 212, 229
- prefix
 - command line option, 212
- profiles
 - command line option, 212
- rtems
 - command line option, 212
- rtems-tools
 - command line option, 212
- stop-on-error
 - command line option, 212
- warnings-report
 - command line option, 212
- ?
 - command line option, 212
- ?, -h
 - command line option, 202, 205
- A FS_ALIGN, --fs-align FS_ALIGN
 - command line option, 228
- B BIND, --bind BIND
 - command line option, 235
- C, --cc
 - command line option, 202
- E, --exec-prefix
 - command line option, 202
- F
 - command line option, 205
- F FS_FORMAT, --fs-format FS_FORMAT
 - command line option, 228
- I
 - command line option, 205
- P PORT, --port PORT
 - command line option, 235
- S
 - command line option, 205
- S FS_SIZE, --fs-size FS_SIZE
 - command line option, 228
- S, --symc
 - command line option, 202
- U CUSTOM_UENV, --custom-uenv CUSTOM_UENV
 - command line option, 229
- V
 - command line option, 204
- V, --version
 - command line option, 202
- a
 - command line option, 204
- b BOARD, --board BOARD
 - command line option, 229
- c CONFIG, --config CONFIG

command line option, 235
 -c, --cflags
 command line option, 202
 -d FDT, --fdt FDT
 command line option, 228
 -e, --embed
 command line option, 202
 -f FILE, --file FILE
 command line option, 228
 -h, --help
 command line option, 212, 228, 235
 -k KERNEL, --kernel KERNEL
 command line option, 228
 -k, --keep
 command line option, 202
 -l, --log
 command line option, 228, 235
 -m, --map
 command line option, 202
 -o OUTPUT, --output OUTPUT
 command line option, 229
 -o, --output
 command line option, 202
 -s IMAGE_SIZE, --image-size IMAGE_SIZE
 command line option, 228
 -v
 command line option, 204
 -v, --trace
 command line option, 228, 235
 -v, --verbose
 command line option, 202
 -w, --warn
 command line option, 202
 @ARCH@, 164
 @BSP@, 164
 @EXE@, 164
 @FEXE@, 164
 %console, 166
 %execute, 167
 %gdb, 167
 %tftp, 167

A

ABI, **293**
 ABI, 132
 APA, **293**
 API, **293**
 Application Binary Interface, 132
 arch, 163
 Architecture, **293**
 Architectures, 87

B

base image symbols, 146
 Binutils, **293**
 Board Support Packages, 91
 BSP, **293**
 BSP, 91
 bsp, 163
 BSP configuration, 163
 BSP Initialization, 134
 bsp_tty_dev, 164
 bugs, 28
 build debug, 162
 build default, 162
 build mp, 162
 build paravirt, 162
 build posix, 162
 build profiling, 162
 build smp, 162
 Buildbot, **293**
 Building an Application, 131

C

C++11, **293**
 C11, **293**
 command line option
 --arch, 212
 --bsp, 212
 --build, 212
 --build-path, 212
 --config-report, 212
 --convert-kernel, 229
 --dry-run, 212
 --jobs, 212
 --log, 212
 --net-boot, 228
 --net-boot-dhcp, 228
 --net-boot-fdt NET_BOOT_FDT, 229
 --net-boot-file NET_BOOT_FILE, 228
 --net-boot-ip NET_BOOT_IP, 228
 --no-clean, 212, 229
 --prefix, 212
 --profiles, 212
 --rtems, 212
 --rtems-tools, 212
 --stop-on-error, 212
 --warnings-report, 212
 -?, 212
 -?, -h, 202, 205
 -A FS_ALIGN, --fs-align FS_ALIGN, 228
 -B BIND, --bind BIND, 235
 -C, --cc, 202

- E, --exec-prefix, 202
- F, 205
- F FS_FORMAT, --fs-format FS_FORMAT, 228
- I, 205
- P PORT, --port PORT, 235
- S, 205
- S FS_SIZE, --fs-size FS_SIZE, 228
- S, --symc, 202
- U CUSTOM_UENV, --custom-uenv CUSTOM_UENV, 229
- V, 204
- V, --version, 202
- a, 204
- b BOARD, --board BOARD, 229
- c CONFIG, --config CONFIG, 235
- c, --cflags, 202
- d FDT, --fdt FDT, 228
- e, --embed, 202
- f FILE, --file FILE, 228
- h, --help, 212, 228, 235
- k KERNEL, --kernel KERNEL, 228
- k, --keep, 202
- l, --log, 228, 235
- m, --map, 202
- o OUTPUT, --output OUTPUT, 229
- o, --output, 202
- s IMAGE_SIZE, --image-size IMAGE_SIZE, 228
- v, 204
- v, --trace, 228, 235
- v, --verbose, 202
- w, --warn, 202
- paths [paths ...], 229
- community
 - developers, 31
 - IRC, 28
 - users mailing list, 28
- Console, 166
- Crosscompiler, **293**
- D**
 - data object identifier, 145
 - Debugging, 137
 - Device Tree, 154
 - dlclose, 143
 - dLError, 144
 - dlfcn.h, 142
 - dlinfo, 144
 - DLL, **293**
 - dlopen, 142
 - dlsym, 144
 - documentation, 28
 - Dynamic Loader, 138, 140
- E**
 - Ecosystem, 8
 - EDF, **293**
 - EMB², **293**
 - Embedded executable, 129
 - embedded symbol table, 146
 - Executable, 129
 - Execute, 167
- F**
 - FAT, **294**
 - function identifier, 145
 - Futex, **294**
- G**
 - GCC, **294**
 - GDB, **294**
 - GDB, 167, 172
 - Git, 61
 - global symbol, 145
 - GNU, **294**
- H**
 - Hardware, 85
 - Host, **294**
 - Host Computer, 39
- I**
 - IMFS, **294**
 - Installation, 51
 - IRC, 28
- J**
 - JFFS2, **294**
 - jobs, 164
 - JTAG, 137, 172
- L**
 - libdebugger, 138
 - Libdl, 140
 - linker, 200
 - Loader Interface, 142
 - Loading object files, 142
- M**
 - Machine flags, 132
 - mailing lists
 - announce, 28

- bugs, 28
- build, 28
- devel, 28
- users, 28
- vc, 28
- Microsoft Windows Installation, 66
- MinGW, **294**
- MinGW64, **294**
- MrsP, **294**
- MSYS2, **294**

N

- NFSv2, **294**

O

- OMIP, **294**
- OpenMP, **294**
- OpenOCD, 137
- orphaned object file, 145

P

- paths [paths ...]
 - command line option, 229
- POSIX, **294**
- prefix, **294**
- prefix, 15

Q

- QEMU, 137
- Quick Start, 11

R

- release, 54
- reporting bugs, 28
- RFS, **295**
- RSB, **295**
- RTEMS, **295**
- RTEMS Executable, 130
- RTEMS Initialization, 135
- rtems-boot-image, 226
- rtems-bsp-builder, 208
- rtems-exeinfo, 204
- rtems-ld, 200
- rtems-run, 216
- rtems-syms, 201
- rtems-test, 216
- rtems-tftp-proxy, 233
- rtems_rtl_alloc_cmd, 151
- RTEMS_RTL_ALLOC_DEL, 151
- RTEMS_RTL_ALLOC_EXTERNAL, 151
- RTEMS_RTL_ALLOC_NEW, 151
- RTEMS_RTL_ALLOC_OBJECT, 151

- RTEMS_RTL_ALLOC_READ, 151
- RTEMS_RTL_ALLOC_READ_EXEC, 151
- RTEMS_RTL_ALLOC_READ_WRITE, 151
- RTEMS_RTL_ALLOC_SYMBOL, 151
- rtems_rtl_alloc_tags, 151
- RTEMS_RTL_ALLOC_WR_DISABLE, 151
- RTEMS_RTL_ALLOC_WR_ENABLE, 151
- rtems_rtl_allocator, 152
- RTL, 140
- Run-time Loader, 140

S

- Simulation, 171
- SMP, **295**
- strip library, 150
- support, 25
 - commercial, 36
 - RTEMS Project, 27
- symbol, 145

T

- tarball, 54
- Target, **295**
- Target Execution, 133
- Target Hardware, 85
- target_exe_filter, 165
- target_off_command, 165
- target_on_command, 165
- target_posttest_command, 165
- target_pretest_command, 165
- target_reset_command, 165
- target_reset_regex, 165
- target_start_regex, 165
- Targets, 86
- test banner version, 158
- TEST BEGIN, 158
- test begin, 158
- TEST BUILD, 158
- test build, 158
- TEST END, 158
- test end, 158
- TEST STATE, 158
- test state, 158
- test state benchmark, 159
- test state expected-fail, 159
- test state failure, 159
- test state indeterminate, 159
- test state invalid, 159
- test state passed, 159
- test state timeout, 159
- test state user-input, 159
- Test states, 158

Test Suite, **295**
TEST TOOLS, 158
test tools, 158
TEST VERSION, 158
test_restarts, 165
tester, 164
Testing, 171–173
Testsuite, **295**
TFTP, 167, 173
TLS, **295**
Tools, 53, 200, 201, 204, 208, 216, 226, 233
Tracing Framework, 179

U

U-Boot, 173
User configuration, 163

W

Waf, **295**

Y

YAFFS2, **295**