
# RTEMS Source Builder

*Release 4.11.3*

# CONTENTS

Chris Johns <chrisj@rtems.org>



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

**RTEMS Online Resources**

| | |
|---|---|
| Home | https://www.rtems.org/ |
| Developers | https://devel.rtems.org/ |
| Documentation | https://docs.rtems.org/ |
| Bug Reporting | https://devel.rtems.org/query |
| Mailing Lists | https://lists.rtems.org/ |
| Git Repositories | https://git.rtems.org/ |

# Part I

# Table of Contents

CHAPTER
**ONE**

# RTEMS SOURCE BUILDER

The RTEMS Source Builder or RSB is a tool to build packages from source. It is used by the RTEMS project to build it's compilers and OS. The RSB helps consolidate the details you need to build a package from source in a controlled and verifiable way. The tool is aimed at developers of software who use tool sets for embedded development. The RSB is not limited to building tools just for RTEMS, you can build bare metal development environments.

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 tools or a **build set**. A **build set** is a collection of packages and a package is a specific tool, for example gcc or gdb, or library. 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.

The RTEMS Source Builder has been tested on:

- ArchLinux
- CentOS
- Fedora
- Raspbian
- Ubuntu (includes XUbuntu)
- Linux Mint
- openSUSE
- FreeBSD
- NetBSD
- MacOS
- Windows

**Setting up your Host**

*Chapter 3 - Hosts* (page 19) details setting up hosts.

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 is 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 `pkgconfg` type files so you can check and use standard libraries if present. If you have a problem please ask on our Developers Mailing List.

This documentation caters for a range of users from new to experienced RTEMS developers. New users can follow the Quick Start section up to the "Installing and Tar Files" to get a working tools and RTEMS. 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.

**Bug Reporting**

If you think you have found a problem please see *Chapter 9 - Bugs, Crashes, and Build Failures* (page 75).

## 1.1 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 rebuilt 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 to run on them 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 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 C compiler (gcc) currently uses a number of 3rd party libraries internally (gmp, 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.

## 1.2 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 where 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.

CHAPTER
**TWO**

# QUICK START

The quick start will show you how to build a set of RTEMS tools for a supported architecture. The tools are installed into a build *prefix*. The *prefix* is the top of a group of directories containing all the files needed to develop RTEMS applications. Building an RTEMS build set will build all that you need. This includes autoconf, automake, assemblers, linkers, compilers, debuggers, standard libraries and RTEMS itself.

There is no need to become root or the administrator and we recommend you avoid doing this. You can build and install the tools anywhere on the host's file system you, as a standard user, have read and write access too. I recommend you use your home directory and work under the directory `~/development/rtems`. The examples shown here will do this.

You can use the build *prefix* to install and maintain different versions of the tools. Doing this lets you try a new set of tools while not touching your proven working production set of tools. Once you have proven the new tools are working rebuild with the *production* prefix switching your development to them.

We 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 *Chapter 3 - Hosts* (page 19) and your host's section for 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:

```
1 $ 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.

**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.

## 2.1 Setup

Setup a development work space:

```
$ cd
$ mkdir -p development/rtems/src
$ cd development/rtems/src
```

The RTEMS Source Builder is distributed as source. It is Python code so all you need to do is download the release's RSB tarball or clone the code directly from the RTEMS GIT repository:

```
$ git clone git://git.rtems.org/rtems-source-builder.git
$ cd rtems-source-builder
```

**Workspaces**

The examples in the *Quick Start Guide* build and install tools in your *home* directory. Please refer to the RTEMS User Manual for more detail about *Sandboxing* and the *prefix*.

## 2.2 Checking

The next step is to check if your host is set up correctly. The RTEMS Source Builder provides a tool to help:

```
$ source-builder/sb-check
warning: exe: absolute exe found in path: (_
 ↪_objcopy) /usr/local/bin/objcopy <1>
warning: exe: absolute exe found in path: (_
 ↪_objdump) /usr/local/bin/objdump
error: exe: not found: (_xz) /usr/local/bin/
 ↪xz    <2>
RTEMS Source Builder environment is not␣
 ↪correctly set up
$ source-builder/sb-check
RTEMS Source Builder environment is ok   <3>
```

**Items:**

1. A tool is in the environment path but it does not match the expected path.
2. The executable `xz` is not found.
3. The host's environment is set up correct.

The checking tool will output a list of executable files not found if problems are detected. Locate those executable files and install them. You may also be given a list of warnings about executable files not in the expected location however the executable was located somewhere in your environment's path. You will need to check each tool to determine if this is an issue. An executable not in the standard location may indicate it is not the host operating system's standard tool. It maybe ok or it could be buggy, only you can determine this.

The *Chapter 3 - Hosts* (page 19) section lists packages you should install for common host operating systems. It maybe worth checking if you have those installed.

## 2.3 Build Sets

The RTEMS tools can be built within the RTEMS Source Builder's source tree. We recommend you do this so lets change into the RTEMS directory in the RTEMS Source Builder package:

```
$ cd rtems
```

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

```
$ ../source-builder/sb-set-builder --list-bsets   <1>
RTEMS Source Builder - Set Builder, v4.11.0
Examining: config
Examining: ../source-builder/config    <2>
    4.10/rtems-all.bset      <3>
    4.10/rtems-arm.bset      <4>
    4.10/rtems-autotools.bset
    4.10/rtems-avr.bset
    4.10/rtems-bfin.bset
    4.10/rtems-h8300.bset
    4.10/rtems-i386.bset
    4.10/rtems-lm32.bset
    4.10/rtems-m32c.bset
    4.10/rtems-m32r.bset
    4.10/rtems-m68k.bset
    4.10/rtems-mips.bset
    4.10/rtems-nios2.bset
    4.10/rtems-powerpc.bset
    4.10/rtems-sh.bset
    4.10/rtems-sparc.bset
    4.11/rtems-all.bset
    4.11/rtems-arm.bset
    4.11/rtems-autotools.bset
    4.11/rtems-avr.bset
    4.11/rtems-bfin.bset
    4.11/rtems-h8300.bset
    4.11/rtems-i386.bset
    4.11/rtems-lm32.bset
    4.11/rtems-m32c.bset
    4.11/rtems-m32r.bset
    4.11/rtems-m68k.bset
    4.11/rtems-microblaze.bset
    4.11/rtems-mips.bset
    4.11/rtems-moxie.bset
    4.11/rtems-nios2.bset
    4.11/rtems-powerpc.bset
    4.11/rtems-sh.bset
    4.11/rtems-sparc.bset
    4.11/rtems-sparc64.bset
    4.11/rtems-v850.bset
    4.9/rtems-all.bset
    4.9/rtems-arm.bset
    4.9/rtems-autotools.bset
    4.9/rtems-i386.bset
    4.9/rtems-m68k.bset
    4.9/rtems-mips.bset
    4.9/rtems-powerpc.bset
    4.9/rtems-sparc.bset
    gnu-tools-4.6.bset
    rtems-4.10-base.bset    <5>
    rtems-4.11-base.bset
    rtems-4.9-base.bset
    rtems-base.bset         <5>
```

**Items:**

1. Only option required is `--list-bsets`
2. The paths inspected. See *Chapter 7 - Configuration* (page 45).
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.

## 2.4 Building

The quick start builds a SPARC tool set:

```
$ ../source-builder/sb-set-builder --log=l-sparc.txt \   <1>
      --prefix=$HOME/development/rtems/4.11 \       <2>
      4.11/rtems-sparc            <3>
Source Builder - Set Builder, v0.2.0
Build Set: 4.11/rtems-sparc
config: expat-2.1.0-1.cfg         <4>
package: expat-2.1.0-x86_64-freebsd9.1-1
building: expat-2.1.0-x86_64-freebsd9.1-1
config: tools/rtems-binutils-2.22-1.cfg    <5>
package: sparc-rtems4.11-binutils-2.22-1
building: sparc-rtems4.11-binutils-2.22-1
config: tools/rtems-gcc-4.7.2-newlib-1.20.0-1.cfg   <6>
package: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
building: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
config: tools/rtems-gdb-7.5.1-1.cfg       <7>
package: sparc-rtems4.11-gdb-7.5.1-1
building: sparc-rtems4.11-gdb-7.5.1-1
installing: rtems-4.11-sparc-rtems4.11-1 ->/home/chris/development/rtems/4.11 <8>
installing: rtems-4.11-sparc-rtems4.11-1 ->/home/chris/development/rtems/4.11
installing: rtems-4.11-sparc-rtems4.11-1 ->/home/chris/development/rtems/4.11
installing: rtems-4.11-sparc-rtems4.11-1 ->/home/chris/development/rtems/4.11
cleaning: expat-2.1.0-x86_64-freebsd9.1-1   <9>
cleaning: sparc-rtems4.11-binutils-2.22-1
cleaning: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
cleaning: sparc-rtems4.11-gdb-7.5.1-1
Build Set: Time 0:13:43.616383        <10>
```

**Items**

1. Providing a log file redirects the build output into a file. Logging the build output provides a simple way to report problems.
2. The prefix is the location on your file system the tools are installed into. Provide a prefix to a location you have read and write access. You can use the prefix to install different versions or builds of tools. Just use the path to the tools you want to use when building RTEMS.
3. The build set. This is the SPARC build set. First a specifically referenced file is checked for and if not found the `%{_configdir}` path is searched. The set builder will first look for files with a `.bset` extension and then for a configuration file with a `.cfg` extension.
4. The SPARC build set first builds the expat library as it is used in GDB. This is the expat configuration used.
5. The binutils build configuration.
6. The GCC and Newlib build configuration.
7. The GDB build configuration.
8. Installing the built packages to the install prefix.
9. All the packages built are cleaned at the end. If the build fails all the needed files are present. You may have to clean up by deleting the build directory if the build fails.
10. The time to build the package. This lets you see how different host hardware or configurations perform.

Your SPARC RTEMS 4.11 tool set will be installed and ready to build RTEMS and RTEMS applications. When the build runs you will notice the tool fetch the source code from the internet. These files are cached in directies called `source` and `patches`. If you run the build again the cached files are used. This is what happened in the shown example. Archiving these directories archives the source you need to recreate the build.

**RTEMS Releases**

The RSB found in a release will automatically build and install RTEMS. If you do not want a released version of the RSB to build RTEMS add `--without-rtems` to the command line. The development ver-

sion requires adding `--with-rtems` to build RTEMS. Use this option to create a single command to build the tools and RTEMS.

The source used in release builds is downloaded from the RTEMS FTP server. This ensures the source is always available for a release.

The installed tools:

```
$ ls $HOME/development/rtems/4.11
bin         include     lib         libexec    share       sparc-rtems4.11
$ ls $HOME/development/rtems/4.11/bin
sparc-rtems4.11-addr2line      sparc-rtems4.11-cpp
sparc-rtems4.11-gcc-ar         sparc-rtems4.11-gprof
sparc-rtems4.11-objdump        sparc-rtems4.11-size
sparc-rtems4.11-ar             sparc-rtems4.11-elfedit
sparc-rtems4.11-gcc-nm         sparc-rtems4.11-ld
sparc-rtems4.11-ranlib         sparc-rtems4.11-strings
sparc-rtems4.11-as             sparc-rtems4.11-g++
sparc-rtems4.11-gcc-ranlib     sparc-rtems4.11-ld.bfd
sparc-rtems4.11-readelf        sparc-rtems4.11-strip
sparc-rtems4.11-c++            sparc-rtems4.11-gcc
sparc-rtems4.11-gcov           sparc-rtems4.11-nm
sparc-rtems4.11-run             xmlwf
sparc-rtems4.11-c++filt        sparc-rtems4.11-gcc-4.7.2
sparc-rtems4.11-gdb            sparc-rtems4.11-objcopy
sparc-rtems4.11-sis
$    $HOME/development/rtems/4.11/bin/sparc-rtems4.11-gcc -v
Using built-in specs.
COLLECT_GCC=/home/chris/development/rtems/4.11/bin/sparc-rtems4.11-gcc
COLLECT_LTO_WRAPPER=/usr/home/chris/development/rtems/4.11/bin/../ \
libexec/gcc/sparc-rtems4.11/4.7.2/lto-wrapper
Target: sparc-rtems4.11                                          <1>
Configured with: ../gcc-4.7.2/configure                          <2>
--prefix=/home/chris/development/rtems/4.11
--bindir=/home/chris/development/rtems/4.11/bin
--exec_prefix=/home/chris/development/rtems/4.11
--includedir=/home/chris/development/rtems/4.11/include
--libdir=/home/chris/development/rtems/4.11/lib
--libexecdir=/home/chris/development/rtems/4.11/libexec
--mandir=/home/chris/development/rtems/4.11/share/man
--infodir=/home/chris/development/rtems/4.11/share/info
--datadir=/home/chris/development/rtems/4.11/share
--build=x86_64-freebsd9.1      --host=x86_64-freebsd9.1 --target=sparc-rtems4.11
--disable-libstdcxx-pch    --with-gnu-as    --with-gnu-ld --verbose --with-newlib
--with-system-zlib  --disable-nls  --without-included-gettext
--disable-win32-registry    --enable-version-specific-runtime-libs --disable-lto
--enable-threads   --enable-plugin   --enable-newlib-io-c99-formats
--enable-newlib-iconv   --enable-languages=c,c++
Thread model: rtems                                              <3>
gcc version 4.7.2 20120920                                       <4>
        (RTEMS        4.11        RSB        cb12e4875c203f794a5cd4b3de36101ff9a88403)-1 newlib 2.0.0) (GCC)
```

**Items**

1. The target the compiler is built for.
2. The configure options used to build GCC.
3. The threading model is always RTEMS. This makes using the RTEMS tools for bare metal development more difficult.
4. The version string. It contains the Git hash of the RTEMS Source Builder you are using. If your local clone has been modifed that state is also recorded in the version string. The hash allows you to track from a GCC executable back to the original source used to build it.

**Note:** The RTEMS thread model enables specific hooks in GCC so applications built with RTEMS tools need the RTEMS runtime to operate correctly. You can use RTEMS tools to build bare metal component but it is more difficult than with a bare metal tool chain and you need to know what you are doing at a low level. The RTEMS Source Builder can build bare metal tool chains as well. Look in the top level `bare` directory.

## 2.5 Deployment

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:

```
$ cd /somewhere
$ tar --strip-components=3 -xjf \
          $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:

```
$ ../source-builder/sb-set-builder --log=l-
 ↪sparc.txt \
        --prefix=$HOME/development/rtems/4.
 ↪11 \
        --bset-tar-file \     <1>
        4.11/rtems-sparc
Source Builder - Set Builder, v0.2.0
Build Set: 4.11/rtems-sparc
config: expat-2.1.0-1.cfg
package: expat-2.1.0-x86_64-freebsd9.1-1
building: expat-2.1.0-x86_64-freebsd9.1-1
config: tools/rtems-binutils-2.22-1.cfg
package: sparc-rtems4.11-binutils-2.22-1
building: sparc-rtems4.11-binutils-2.22-1
config: tools/rtems-gcc-4.7.2-newlib-1.20.0-
 ↪1.cfg
package: sparc-rtems4.11-gcc-4.7.2-newlib-1.
 ↪20.0-1
building:  sparc-rtems4.11-gcc-4.7.2-newlib-
 ↪1.20.0-1
config: tools/rtems-gdb-7.5.1-1.cfg
package: sparc-rtems4.11-gdb-7.5.1-1
building: sparc-rtems4.11-gdb-7.5.1-1
installing: rtems-4.11-sparc-rtems4.11-1 ->␣
 ↪/home/chris/development/rtems/4.11 <2>
installing: rtems-4.11-sparc-rtems4.11-1 ->␣
 ↪/home/chris/development/rtems/4.11
installing: rtems-4.11-sparc-rtems4.11-1 ->␣
 ↪/home/chris/development/rtems/4.11
installing: rtems-4.11-sparc-rtems4.11-1 ->␣
 ↪/home/chris/development/rtems/4.11
tarball:   tar/rtems-4.11-sparc-rtems4.11-1.
 ↪tar.bz2      <3>
cleaning: expat-2.1.0-x86_64-freebsd9.1-1
cleaning: sparc-rtems4.11-binutils-2.22-1
cleaning:  sparc-rtems4.11-gcc-4.7.2-newlib-
 ↪1.20.0-1
cleaning: sparc-rtems4.11-gdb-7.5.1-1
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 accessiable, for example when building Canadian cross compiled tool sets:

```
$ ../source-builder/sb-set-builder --log=l-
 ↪sparc.txt \
          --prefix=$HOME/development/rtems/
 ↪4.11 \
```

```
            --bset-tar-file \
            --no-install \      <1>
            4.11/rtems-sparc
Source Builder - Set Builder, v0.2.0
Build Set: 4.11/rtems-sparc
config: expat-2.1.0-1.cfg
package: expat-2.1.0-x86_64-freebsd9.1-1
building: expat-2.1.0-x86_64-freebsd9.1-1
config: tools/rtems-binutils-2.22-1.cfg
package: sparc-rtems4.11-binutils-2.22-1
building: sparc-rtems4.11-binutils-2.22-1
config: tools/rtems-gcc-4.7.2-newlib-1.20.0-1.cfg
package: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
building:  sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
config: tools/rtems-gdb-7.5.1-1.cfg
package: sparc-rtems4.11-gdb-7.5.1-1
building: sparc-rtems4.11-gdb-7.5.1-1
tarball:   tar/rtems-4.11-sparc-rtems4.11-1.tar.bz2    <2>
cleaning: expat-2.1.0-x86_64-freebsd9.1-1
cleaning: sparc-rtems4.11-binutils-2.22-1
cleaning:  sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
cleaning: sparc-rtems4.11-gdb-7.5.1-1
Build Set: Time 0:14:11.721274
$ ls tar
rtems-4.11-sparc-rtems4.11-1.tar.bz2
```

**Items**

1. The option to supressing 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:

```
$ ../source-builder/sb-set-builder --log=l-sparc.txt \
         --prefix=$HOME/development/rtems/4.11 \
        --bset-tar-file \
        --pkg-tar-files \        <1>
        --no-install 4.11/rtems-sparc
Source Builder - Set Builder, v0.2.0
Build Set: 4.11/rtems-sparc
config: expat-2.1.0-1.cfg
package: expat-2.1.0-x86_64-freebsd9.1-1
building: expat-2.1.0-x86_64-freebsd9.1-1
config: tools/rtems-binutils-2.22-1.cfg
package: sparc-rtems4.11-binutils-2.22-1
building: sparc-rtems4.11-binutils-2.22-1
config: tools/rtems-gcc-4.7.2-newlib-1.20.0-1.cfg
package: sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
building:  sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
config: tools/rtems-gdb-7.5.1-1.cfg
package: sparc-rtems4.11-gdb-7.5.1-1
building: sparc-rtems4.11-gdb-7.5.1-1
tarball:   tar/rtems-4.11-sparc-rtems4.11-1.tar.bz2
cleaning: expat-2.1.0-x86_64-freebsd9.1-1
cleaning: sparc-rtems4.11-binutils-2.22-1
cleaning:  sparc-rtems4.11-gcc-4.7.2-newlib-1.20.0-1
cleaning: sparc-rtems4.11-gdb-7.5.1-1
Build Set: Time 0:14:37.658460
$ ls tar
expat-2.1.0-x86_64-freebsd9.1-1.tar.bz2      sparc-rtems4.11-binutils-2.22-1.tar.bz2
sparc-rtems4.11-gdb-7.5.1-1.tar.bz2 <2>      rtems-4.11-sparc-rtems4.11-1.tar.bz2 <3>
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.

## 2.6 Controlling the Build

Build sets can be controlled via the command line to enable and disable 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`.

Following are currentlt available:

**`--without-rtems`**
Do not build RTEMS when building an RTEMS build set.

**`--without-cxx`**
Do not build a C++ compiler.

**`--with-objc`**
Attempt to build a C++ compiler.

**`--with-fortran`**
Attempt to build a Fortran compiler.

CHAPTER
**THREE**

# HOSTS

The known supported hosts are listed in the following sections. If a host or a new version of a host is known to work and it not listed please lets us know.

## 3.1 Linux

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

### 3.1.1 ArchLinux

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

```
# 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:

```
# pacman -R texinfo
$ yaourt -S texinfo-legacy
#  ln  -s  /usr/bin/makeinfo-4.13a  /usr/bin/␣
↪makeinfo
```

### 3.1.2 CentOS

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

```
# yum install autoconf automake binutils gcc␣
↪gcc-c++ gdb make patch \
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.

### 3.1.3 Fedora

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

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

### 3.1.4 Raspbian

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

```
$ sudo apt-get install autoconf automake␣
↪bison flex binutils gcc g++ gdb \
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.

### 3.1.5 Ubuntu

The latest version is Ubuntu 16.04.1 LTS 64bit. This section also includes Xubuntu. A minimal installation was used and the following packages installed:

```
$ sudo apt-get build-dep binutils gcc g++␣
↪gdb unzip git
$ sudo apt-get install python2.7-dev
```

### 3.1.6 Linux Mint

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

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

### 3.1.7 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.

## 3.2 FreeBSD

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

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

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

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

The +zlip+ and +pthreads+ ports for MinGW32 are used for builiding a Windows QEMU.

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

## 3.3 NetBSD

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

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

## 3.4 MacOS

The RTEMS Source Builder has been tested on Mountain Lion. You will need to install the Xcode app using the *App Store* tool, run Xcode and install the Developers Tools package within Xcode.

### 3.4.1 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 3rd party packaging on MacOS and prefer to build the packages from source using a prefix of `/usr/local`. There are good 3rd 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:

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

### 3.4.2 Serria

The RSB works on Serria with the latest Xcode.

## 3.5 Windows

Windows tool sets are supported. The tools are native Windows executable which means they do not need an emulation layer to run once built. The tools understand and use standard Windows paths and integrate easily into Windows IDE environments because they understand and use standard Windows paths. Native Windows tools have proven over time to be stable and reliable with good performance. If you are a Windows user or you are required to use Windows you can still develop RTEMS application as easily as a Unix operating system. Some debugging experiences may vary and if this is an issue please raised the topic on the RTEMS Users mailing list.

Building the tools or some other packages may require a Unix or POSIX type shell. There are a few options, Cygwin and MSYS2. I recommend MSYS2.

### 3.5.1 MSYS2

This is a new version of the MinGW project's original MSYS. MSYS2 is based around the Arch Linux pacman packager. MSYS and MSYS2 are a specific fork of the Cygwin project with some fundamental changes in the handling of paths and mounts that allow easy interaction between the emulated POSIX environment and the native Windows environment.

Install MSYS2 using the installer you can download from https://msys2.github.io/. Follow the instructions on the install page and make sure you remove any global path entries to any other Cygwin, MinGW, MSYS or packages that may uses a Cygwin DLL, for example some ports of Git.

To build the tools you need install the following packages using pacman:

```
$ pacman -S git cvs bison make texinfo patch␣
↪unzip diffutils tar \
      mingw64/mingw-w64-x86_64-gcc mingw64/
↪mingw-w64-x86_64-binutils
```

To build make sure you add '–without-python –jobs=none' to the standard RSB command line. MSYS2 has a temp file name issue and so the GNU AR steps on itself when running in parallel on SMP hardware which means we have to set the jobs option to none.

Install a suitable version of Python from http://www.python.org/ and add it to the start of your path. The MSYS2 python does not work with waf.

### 3.5.2 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 3.1: 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 3rd 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 Cygiwn 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 Cygiwn. The build output for an RTEMS 4.10 ARM tool set is:

```
chris@cygthing ~/development/rtems/src/rtems-source-builder/rtems
$ ../source-builder/sb-set-builder --log=l-arm.txt --prefix=$HOME/development/rtems/4.10 4.10/rtems-arm
RTEMS Source Builder - Set Builder, v0.2
Build Set: 4.10/rtems-arm
config: expat-2.1.0-1.cfg
package: expat-2.1.0-x86_64-w64-mingw32-1
building: expat-2.1.0-x86_64-w64-mingw32-1
reporting: expat-2.1.0-1.cfg -> expat-2.1.0-x86_64-w64-mingw32-1.html
config: tools/rtems-binutils-2.20.1-1.cfg
package: arm-rtems4.10-binutils-2.20.1-1 <1>
building: arm-rtems4.10-binutils-2.20.1-1
package: (Cxc) arm-rtems4.10-binutils-2.20.1-1 <2>
building: (Cxc) arm-rtems4.10-binutils-2.20.1-1
reporting: tools/rtems-binutils-2.20.1-1.cfg -> arm-rtems4.10-binutils-2.20.1-1.html
config: tools/rtems-gcc-4.4.7-newlib-1.18.0-1.cfg
package: arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1
building: arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1
package: (Cxc) arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1
building: (Cxc) arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1
reporting: tools/rtems-gcc-4.4.7-newlib-1.18.0-1.cfg -> arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1.html
config: tools/rtems-gdb-7.3.1-1.cfg
package: arm-rtems4.10-gdb-7.3.1-1
building: arm-rtems4.10-gdb-7.3.1-1
reporting: tools/rtems-gdb-7.3.1-1.cfg -> arm-rtems4.10-gdb-7.3.1-1.html
config: tools/rtems-kernel-4.10.2.cfg
package: arm-rtems4.10-kernel-4.10.2-1
building: arm-rtems4.10-kernel-4.10.2-1
reporting: tools/rtems-kernel-4.10.2.cfg -> arm-rtems4.10-kernel-4.10.2-1.html
installing: expat-2.1.0-x86_64-w64-mingw32-1 -> /cygdrive/c/Users/chris/development/rtems/4.10
installing: arm-rtems4.10-binutils-2.20.1-1 -> /cygdrive/c/Users/chris/development/rtems/4.10 <3>
installing: arm-rtems4.10-gcc-4.4.7-newlib-1.18.0-1 -> /cygdrive/c/Users/chris/development/rtems/4.10
installing: arm-rtems4.10-gdb-7.3.1-1 -> /cygdrive/c/Users/chris/development/rtems/4.10
```

```
35 installing:  arm-rtems4.10-kernel-4.10.2-1 ␣
    ↪->   /cygdrive/c/Users/chris/development/
    ↪rtems/4.10
36 cleaning: expat-2.1.0-x86_64-w64-mingw32-1
37 cleaning: arm-rtems4.10-binutils-2.20.1-1
38 cleaning:  arm-rtems4.10-gcc-4.4.7-newlib-1.
    ↪18.0-1
39 cleaning: arm-rtems4.10-gdb-7.3.1-1
40 cleaning: arm-rtems4.10-kernel-4.10.2-1
41 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.

CHAPTER
**FOUR**

# 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, other projects they use the RTEMS Source Builder or privately on your local file system.

The configuration file loader searches the macro `_configdir` and by default this is set to `%{_topdir}/config:%{_sbdir}/config` where `_topdir` is the your current working direct, in other words the directory you invoke the RTEMS Source Builder command in, and `_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 also 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

## 4.1 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*:

```
$ cd rtems-source-builder/bare
$     ../source-builder/sb-set-builder     --log=log_arm_eabihf \
         --prefix=$HOME/development/bare  --target=arm-eabihf lang/gcc491
RTEMS Source Builder - Set Builder, v0.3.0
Build Set: lang/gcc491
config: devel/expat-2.1.0-1.cfg
package:  expat-2.1.0-x86_64-apple-darwin13.2.0-1
building: expat-2.1.0-x86_64-apple-darwin13.2.0-1
config: devel/binutils-2.24-1.cfg
package: arm-eabihf-binutils-2.24-1
building: arm-eabihf-binutils-2.24-1
config: devel/gcc-4.9.1-newlib-2.1.0-1.cfg
package: arm-eabihf-gcc-4.9.1-newlib-2.1.0-1
building: arm-eabihf-gcc-4.9.1-newlib-2.1.0-1
config: devel/gdb-7.7-1.cfg
package: arm-eabihf-gdb-7.7-1
building: arm-eabihf-gdb-7.7-1
installing:        expat-2.1.0-x86_64-apple-darwin13.2.0-1      ->      /Users/chris/development/bare
installing: arm-eabihf-binutils-2.24-1 -> /Users/chris/development/bare
installing: arm-eabihf-gcc-4.9.1-newlib-2.1.0-1 -> /Users/chris/development/bare
installing: arm-eabihf-gdb-7.7-1 -> /Users/chris/development/bare
cleaning: expat-2.1.0-x86_64-apple-darwin13.2.0-1
cleaning: arm-eabihf-binutils-2.24-1
cleaning: arm-eabihf-gcc-4.9.1-newlib-2.1.0-1
cleaning: arm-eabihf-gdb-7.7-1
```

## 4.2 RTEMS

The RTEMS Configurations found in the `rtems` directory. The configurations are grouped by RTEMS version. 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 be limited to the architecture it relates to. The patch may fix a problem on the effect architecture however it could introduce a problem in another architecture. Limit exposure limits any possible crosstalk between architectures.

If you are building a released version of RTEMS the release RTEMS tar file will be downloaded and built as part of the build process. If you are building a tool set for use with the development branch of RTEMS, the development branch will be cloned directly from the RTEMS GIT repository and built.

When building RTEMS within the RTEMS Source Builder it needs a suitable working `autoconf` and `automake`. These packages need to built and installed in their prefix in order for them to work. The RTEMS Source Builder installs all packages only after they have been built so if you host does not have a recent enough version of `autoconf` and `automake` you first need to build them and install them then build your tool set. The commands are:

```
$ ../source-builder/sb-set-builder --log=l-4.11-at.txt \
    --prefix=$HOME/development/rtems/4.11 4.11/rtems-autotools
$ export PATH=~/development/rtems/4.11/bin:$PATH    <1>
$ ../source-builder/sb-set-builder --log=l-4.11-sparc.txt \
    --prefix=$HOME/development/rtems/4.11 4.11/rtems-sparc
```

**Items:**

1. Setting the path.

If this is your first time building the tools and RTEMS it pays to add 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.

To build snapshots for testing purposes you use the available macro maps passing them on the command line using the `--macros` option. For RTEMS these are held in `config/snapshots` directory. The following builds *newlib* from CVS:

```
$ ../source-builder/sb-set-builder --log=l-4.11-sparc.txt \
  --prefix=$HOME/development/rtems/4.11 \
  --macros=snapshots/newlib-head.mc \
  4.11/rtems-sparc
```

and the following uses the version control heads for `binutils`, `gcc`, `newlib`, `gdb` and *RTEMS*:

```
$ ../source-builder/sb-set-builder --log=l-heads-sparc.txt \
     --prefix=$HOME/development/rtems/4.11-head \
    --macros=snapshots/binutils-gcc-newlib-gdb-head.mc \
  4.11/rtems-sparc
```

## 4.3 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 seperate directory called `patches` relative to the configuration directory you are building. For example `%{_topdir}/patches:%{_sbdir}/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.

Patches for packages developed by the RTEMS project can be placed in the RTEMS Tools Git repository. The `tools` directory in the repository has various places a patch can live. The tree is broken down in RTEMS releases and then tools within that release. If the package is not specific to any release the patch can be added closer to the top under the package's name. Patches to fix specific tool related issues for a specific architecture should be grouped under the specific architecture and only applied when building that architecture avoiding a patch breaking an uneffected architecture.

Patches in the RTEMS Tools repository need to be submitted to the upstream project. It should not be a clearing house for patches that will not be accepted upstream.

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 efficently.

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:

```
%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:

```
%hash md5 <1> gdb-sim-lm32uart.diff <2> ␣
 ↪77d07087112783292461bd6e7db17fb <3>
```

**Items:**

1. The type of checksum, in the case an MD5 hash.
2. The patch file the checksum is for.
3. The MD5 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:

```
%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. For example, to test a local patch for newlib, add the following two lines to the .cfg file in `rtems/config/tools/` that is included by the bset you use:

```
%patch add newlib file://0001-this-is-a-newlib-patch.patch <1>
%hash md5 0001-this-is-a-newlib-patch.diff 77d070878112783292461bd6e7db17fb <2>
```

**Items:**

1. The diff file prepended with `file://` to tell RSB this is a local file.
2. The output from md5sum on the diff file.

CHAPTER
**FIVE**

# 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.

## 5.1 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 NTP for RTEMS on your development machine.

To build the NTP package for RTEMS you enter the RSB command:

```
$ ../source-builder/sb-set-builder \
   --log=log_ntp_arm.txt \
   --prefix=$HOME/development/rtems/4.11 \
  ↪<1>
   --host=arm-rtems4.11 \  <2>
   --with-rtems-bsp=xilinx_zynq_zc706 \  <3>
   4.11/net/ntp
```

**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 arhcitecture must match the `--host` architecture.

## 5.2 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`:

```
$ ../source-builder/sb-set-builder --
↪host=mingw32 \
  --log=l-mingw32-4.11-sparc.txt \
  --prefix=$HOME/development/rtems/4.11 \
  4.11/rtems-sparc
```

If you are on a Linux Fedora build host with the MinGW packages installed the command line is:

```
$ ../source-builder/sb-set-builder --
↪host=i686-w64-mingw32 \
  --log=l-mingw32-4.11-sparc.txt \
  --prefix=$HOME/development/rtems/4.11 \
  4.11/rtems-sparc
```

CHAPTER
**SIX**

# RTEMS 3RD PARTY PACKAGES

This section describes how to build and add an RTEMS 3rd party package to the RSB.

A 3rd party package is a library or software package built to run on RTEMS, examples are NTP, Net-Snmp, libjpeg or Python. 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 and an installed RTEMS Board Support Package (BSP).

The RSB support for building 3rd 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. If you have any issues with this support please ask on the RTEMS developers mailing list.

## 6.1 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 3rd party libraries, you can crate build sets that stack library dependences vertically to create a *stack*.

## 6.2 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 host, and the prefix path to the installed RTEMS BSP. The prefix needs to be the same as the prefix used to build RTEMS.

To build Net-SNMP the command is:

```
$ cd rtems-source-builder/rtems
$     ../source-builder/sb-set-builder     --log=log_sis_net_snmp \
     --prefix=$HOME/development/rtems/bsps/4.11 \
      --with-tools=$HOME/development/rtems/4.11 \
        --host=sparc-rtems4.11  --with-rtems-bsp=sis 4.11/net-mgmt/net-snmp
RTEMS Source Builder - Set Builder, v0.3.0
Build Set: 4.11/net-mgmt/net-snmp
config: net-mgmt/net-snmp-5.7.2.1-1.cfg
package: net-snmp-5.7.2.1-sparc-rtems4.11-1
building: net-snmp-5.7.2.1-sparc-rtems4.11-1
installing:    net-snmp-5.7.2.1-sparc-rtems4.11-1   ->   /Users/chris/development/rtems/bsps/4.11
cleaning: net-snmp-5.7.2.1-sparc-rtems4.11-1
Build Set: Time 0:01:10.651926
```

## 6.3 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 you 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 so it can be added to the RTEMS Tools git repository. A patch in the RTEMS Tools git reposiitory can then be referenced by an RSB configuration file.

A package may create executables, for example NTP normally creates executables such as `ntdp`, `ntpupdate`, or `ntpdc`. These executables can be useful when testing the package however they are of limited use by RTEMS users because they cannot be directly linked into a user application. Users need to link to the functions in these executables or even the executable as a function placed in libraries. If the package does not export the code in a suitable manner please contact the project's commuinity and see if you can work 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 3 files to be created:

- The first file is the RTEMS build set file and it resides in the `rtems/config/%{rtems_version}` path in a directory tree based on the FreeBSD ports collection. For the NTP package and RTEMS 4.11 this is `rtems/config/4.11/net/ntp.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.
- 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 again under the FreeBSD port's path name. For example the NTP package is found in the `net` directory of the FreeBSD ports tree so the NTP configuration path is `rtems/config/net/ntp-4.2.6p5-1.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.
- 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 *Chapter 7 - Configuration* (page 45) for more details.

## 6.4 BSP Support

The RSB provides support to help 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 meaning compiler ABI flags cannot be mixed when building code. The 3rd party package need to 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 3rd party packages and you need to include this file in your RTEMS version specific configuration file. For example the Net-SNMP configuration file `rtems/config/net-mgmt/net-snmp-5.7.2.1-1.cfg`:

```
#
# NetSNMP 5.7.2.1
#
%if %{release} == %{nil}
 %define release 1    <1>
%endif

%include %{_configdir}/rtems-bsp.cfg    <2>

#
# NetSNMP Version
#
%define net_snmp_version 5.7.2.1    <3>

#
# We need some special flags to build this version.
#
%define net_snmp_cflags <4> -DNETSNMP_CAN_USE_SYSCTL=1 -DARP_SCAN_FOUR_ARGUMENTS=1 -DINP_IPV6=0

#
# Patch for RTEMS support.
#
%patch add net-snmp %{rtems_git_tools}/net-snmp/rtems-net-snmp-5.7.2.1-20140623.patch <5>

#
# NetSNMP Build configuration
#
%include %{_configdir}/net-snmp-5-1.cfg   <6>
```

**Items:**

1. The release number.
2. Include the RSB RTEMS BSP support.
3. The Net-SNMP package's version.
4. Add specific CFLAGS to the build process. See the

   `net-snmp-5.7.2.1-1.cfg` for details.
5. The RTEMS Net-SNMP patch downloaded from the RTEMS Tools git repo.
6. The Net-SNMP standard build configuration.

The RSB RTEMS BSP support file `rtems/config/rtems-bsp.cfg` checks to make sure standard 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:

```
%if %{_host} == %{nil} <1>
  %error No RTEMS target specified: --host=host
%endif

%ifn %{defined with_rtems_bsp} <2>
 %error No RTEMS BSP specified: --with-rtems-bsp=bsp
%endif

%ifn %{defined with_tools} <3>
 %define with_tools %{_prefix}
%endif

#
```

```
# Set the path to the tools.
#
%{path prepend %{with_tools}/bin} <4>
```

**Items:**

1. Check the host has been set.
2. Check a BSP has been specified.
3. If no tools path has been provided assume they are under the `%{_prefix}`.
4. Add the tools `bin` path to the system path.

RTEMS exports the build flags used in *pkgconfig* (.pc) files and the RSB can read and manage them even when there is no pkgconfig support installed on your build machine. Using this support we can obtain a BSP's configuration and set some standard macros variables (`rtems/config/rtems-bsp.cfg`):

```
%{pkgconfig prefix %{_prefix}/lib/pkgconfig}
 ↪ <1>
%{pkgconfig crosscompile yes} <2>
%{pkgconfig filter-flags yes} <3>

#
# The RTEMS BSP Flags
#
%define rtems_bsp           %{with_rtems_bsp}
%define  rtems_bsp_ccflags      %{pkgconfig␣
 ↪ccflags %{_host}-%{rtems_bsp}} <4>
%define  rtems_bsp_cflags       %{pkgconfig␣
 ↪cflags  %{_host}-%{rtems_bsp}}
%define  rtems_bsp_ldflags      %{pkgconfig␣
 ↪ldflags %{_host}-%{rtems_bsp}}
%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.
4. Ask pkgconfig for the various items we require.

The flags obtain by pkgconfig and given a `rtems_bsp_` prefix and we uses these to set the RSB host support CFLAGS, LDFLAGS and LIBS flags. When we build a 3rd party library your host computer is the _build_ machine and RTEMS is the _host_ machine therefore we set the `host` variables (`rtems/config/rtems-bsp.cfg`):

```
%define host_cflags  %{rtems_bsp_cflags}
%define host_ldflags %{rtems_bsp_ldflags}
%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 so needs to install into the specific BSP path under the `_prefix`. This allows more than BSP build of this package to be install under the same `_prefix` at the same time (`rtems/config/rtems-bsp.cfg`):

```
%define  rtems_bsp_prefix     %{_prefix}/%{_
 ↪host}/%{rtems_bsp} <1>
%define _exec_prefix      %{rtems_bsp_prefix}
%define _bindir           %{_exec_prefix}/bin
%define _sbindir             %{_exec_prefix}/
 ↪sbin
%define _libexecdir          %{_exec_prefix}/
 ↪libexec
%define _datarootdir         %{_exec_prefix}/
 ↪share
%define _datadir           %{_datarootdir}
%define _sysconfdir       %{_exec_prefix}/etc
%define _sharedstatedir   %{_exec_prefix}/com
%define _localstatedir    %{_exec_prefix}/var
%define _includedir        %{_libdir}/include
%define _lib               lib
%define _libdir             %{_exec_prefix}/%
 ↪{_lib}
%define _libexecdir          %{_exec_prefix}/
 ↪libexec
%define _mandir           %{_datarootdir}/man
%define _infodir             %{_datarootdir}/
 ↪info
%define _localedir           %{_datarootdir}/
 ↪locale
%define _localedir         %{_datadir}/locale
%define _localstatedir    %{_exec_prefix}/var
```

**Items:**

1. The path to the 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 (`source-builder/config/ntp-4-1.cfg`):

```
../${source_dir_ntp}/configure \ <1>
  --host=%{_host} \
  --prefix=%{_prefix} \
  --bindir=%{_bindir} \
  --exec_prefix=%{_exec_prefix} \
  --includedir=%{_includedir} \
  --libdir=%{_libdir} \
  --libexecdir=%{_libexecdir} \
  --mandir=%{_mandir} \
  --infodir=%{_infodir} \
  --datadir=%{_datadir} \
  --disable-ipv6 \
  --disable-HOPFPCI
```

**Items:**

1. The configure command for NTP.

## 6.5 RTEMS BSP Configuration

To build a package for RTEMS you need to build it with the matching BSP configuration. A BSP can be built with specific flags that require all code being used needs to be built with the same flags.

CHAPTER
**SEVEN**

# 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 (`_sbdir`). 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<2>  %{_topdir}/
  ↪config:%{_sbdir}/config <1>
```

**Items:**

1. The `_topdir` is the directory you run the command from and `_sbdir` 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.

## 7.1 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 *Chapter 7 Section 8.2 - %prep* (page 59) 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.

### 7.1.1 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<1> gcc<2> 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 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 feed 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 encoutner 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 reide 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:

```
%define stlink_version 3494c11
%source set stlink https://api.github.
↪com/repos/texane/stlink/texane-stlink-%
↪{stlink_version}.tar.gz
```

### 7.1.2 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:

```
%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 protcols.

### 7.1.3 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:

```
%source set newlib cvs://pserver:
↪anoncvs@sourceware.org/cvs/src?
↪module=newlib?src-prefix=src
```

## 7.2 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:

```
%define release 1
```

Once defined if can be accessed in a build set or package configuration file with:

```
%{release}
```

The `sb-defaults` command lists the defaults for your host. I will not include the output of this command because of its size:

```
$ ../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.

### 7.2.1 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:

```
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:

```
# Comments
[my-special-map] <1>
_host:  none, override, 'abc-xyz'
multiline: none, override, '''First line,
second line,
and finally the last line'''
```

**Items:**

1. The map is set to `my-special-map`.

Any macro defintions 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:

```
%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 specificed 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>
# Build all tool parts from version control␣
 ↪head.
#
%include   %{_configdir}/snapshots/binutils-
 ↪head.mc
%include %{_configdir}/snapshots/gcc-head.mc
%include     %{_configdir}/snapshots/newlib-
 ↪head.mc
%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.

### 7.2.2 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.

## 7.3 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 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 check. 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`.

## 7.4 Build Set Files

Build set files lets 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.

## 7.5 Configuration Control

The RTEMS Souce 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:

```
%define gcc_version    4.7.2
%define newlib_version 2.0.0
%define mpfr_version   3.0.1
%define mpc_version    0.8.2
%define gmp_version    5.0.5
```

The package build options, if there are any are also defined:

```
%define with_threads 1
%define with_plugin  0
%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:

```
Patch0: gcc-4.7.2-rtems4.11-20121026.diff
```

Finally including the GCC 4.7 configuration script:

```
%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.

## 7.6 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:

```
$ cd ~/work
$ mkdir test
$ cd test
$ 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 confguration.

## 7.7 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.

### 7.7.1 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.

### 7.7.2 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.

### 7.7.3 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.

### 7.7.4 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:

```
#
# DTC 1.x.x Version 1.
#
# This configuration file configure's, make
 ↪'s and install's DTC.
#

%if %{release} == %{nil}
%define release 1
%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:

```
Name:          dtc-%{dtc_version}-%{_host}-%{release}
Summary:       Device Tree Compiler v%{dtc_version} for target %{_target} on host %{_host}
Version:    %{dtc_version}
Release:    %{release}
URL:           http://www.jdl.com/software/
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 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:

```
#
# Source
#
%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:

```
#
# Prepare the source code.
#
%prep
  build_top=$(pwd)

  %source setup dtc -q -n dtc-v%{dtc_version}
  %patch setup dtc -p1

  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 GNU 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:

```
%build
  build_top=$(pwd)

  cd dtc-v%{dtc_version}
```

```
%{build_build_flags}

%{__make} PREFIX=%{_prefix}

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:

```
%install
  build_top=$(pwd)

  %{__rmdir} -rf $SB_BUILD_ROOT

  cd dtc-v%{dtc_version}
   %{__make} DESTDIR=$SB_BUILD_ROOT PREFIX=%
  ↪{_prefix} install

  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:

```
$    ../source-builder/sb-builder  --prefix=/
 ↪usr/local \
     --log=log_dtc devel/dtc-1.2.0
RTEMS Source Builder, Package Builder v0.2.0
config: devel/dtc-1.2.0
package: dtc-1.2.0-x86_64-freebsd9.1-1
download:    http://www.jdl.com/software/dtc-
 ↪v1.2.0.tgz -> sources/dtc-v1.2.0.tgz
building: dtc-1.2.0-x86_64-freebsd9.1-1
$ ls tar
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:

```
#
# Build the DTC.
#

%define release 1

devel/dtc-1.2.0.cfg
```

To build this you can use something similar to:

```
$     ../source-builder/sb-set-builder     --
 ↪prefix=/usr/local --log=log_dtc \
   --trace --bset-tar-file --no-install dtc
```

```
RTEMS Source Builder - Set Builder, v0.2.0
Build Set: dtc
config: devel/dtc-1.2.0.cfg
package: dtc-1.2.0-x86_64-freebsd9.1-1
building: dtc-1.2.0-x86_64-freebsd9.1-1
tarball:  tar/x86_64-freebsd9.1-dtc-set.tar.bz2
cleaning: dtc-1.2.0-x86_64-freebsd9.1-1
Build Set: Time 0:00:02.865758
$ ls tar
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 priviledges so a build set or bset tar file is created. This can then be installed using root privildges.

The command also supplies the `--trace` option. The output in the log file will contian all the macros.

### 7.7.5 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. Suppling 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 teh 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.

## 7.8 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 is number appended.

**`%hash`:**
Define a checksum for a source or patch file.

**`%echo`:**
Print the following string as a message.

**`%warning`:**
Print the following string as a warning and continue.

**`%error`:**
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.

### 7.8.1 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.

### 7.8.2 %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              <1>                  <2>       <3>
2 %source setup gcc -q -c -T -n %{name}-%
  ↪{version}
```

**Items:**

1. The source group to set up.
2. The source's name.
3. The version of the source.

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:

```
%source set gcc ftp://ftp.gnu.org/gnu/gcc/gcc-%{gcc_version}/gcc-%{gcc_version}.tar.bz2
%source set newlib ftp://sourceware.org/pub/newlib/newlib-%{newlib_version}.tar.gz
%source set mpfr http://www.mpfr.org/mpfr-%{mpfr_version}/mpfr-%{mpfr_version}.tar.bz2
%source set mpc http://www.multiprecision.org/mpc/download/mpc-%{mpc_version}.tar.gz
%source set gmp ftp://ftp.gnu.org/gnu/gmp/gmp-%{gmp_version}.tar.bz2
```

and set up with:

```
%source setup gcc -q -n gcc-%{gcc_version}
%source setup newlib -q -D -n newlib-%{newlib_version}
%source setup mpfr -q -D -n mpfr-%{mpfr_version}
%source setup mpc -q -D -n mpc-%{mpc_version}
%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.

```
%patch add gdb %{rtems_gdb_patches}/gdb-sim-arange-inline.diff
%patch add gdb -p0 <1> %{rtems_gdb_patches}/gdb-sim-cgen-inline.diff
```

**Items:**

1. This patch has a custom option.

To apply these patches:

```
%patch setup gdb -p1 <1>
```

**Items:**

1. The default options.

### 7.8.3 %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 GutHub STLink project. The STLink is a JTAG debugging device for the ST ARM family of processors:

```
%build
  export PATH="%{_bindir}:${PATH}" <1>

  cd texane-stlink-%{stlink_version} <2>

  ./autogen.sh <3>

%if "%{_build}" != "%{_host}"
  CFLAGS_FOR_BUILD="-g -O2 -Wall" \ <4>
%endif
  CPPFLAGS="-I $SB_TMPPREFIX/include/libusb-1.0" \ <5>
  CFLAGS="$SB_OPT_FLAGS" \
  LDFLAGS="-L $SB_TMPPREFIX/lib" \
  ./configure \ <6>
    --build=%{_build} --host=%{_host} \
    --verbose \
    --prefix=%{_prefix} --bindir=%{_bindir} \
    --exec-prefix=%{_exec_prefix} \
    --includedir=%{_includedir} --libdir=%{_libdir} \
   --mandir=%{_mandir} --infodir=%{_infodir}

  %{__make} %{?_smp_mflags} all <7>

  cd ..
```

**Items:**

1. Setup the PATH environment variable. This is not always needed.
2. This package builds in the source tree so enter it.

3. The package is actually checked directly out from the github project and so it needs its autoconf and automake files generated.
4. Flags for a cross-compiled build.
5. Various settings passed to configure 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. Running 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.

### 7.8.4 %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 build 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 *Chapter 7 Section 8.3 - %build* (page 60):

```
%install
  export PATH="%{_bindir}:${PATH}" <1>
  rm -rf $SB_BUILD_ROOT <2>

  cd texane-stlink-%{stlink_version} <3>
  %{__make} DESTDIR=$SB_BUILD_ROOT install␣
↪<4>

  cd ..
```

**Items:**

1. Setup the PATH environment variable. This is not always needed.
2. Clean any installed files. This make 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.

### 7.8.5 %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 currenly not been used because the RTEMS Source Builder automatically cleans up.

### 7.8.6 %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:

```
%include %{_configdir}/gcc-4.7-1.cfg
```

### 7.8.7 %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:

```
Name: %{_target}-gcc-%{gcc_version}-newlib-%
 ↪{newlib_version}-%{release}
```

### 7.8.8 %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:

```
Summary: GCC v%{gcc_version} and Newlib v%
 ↪{newlib_version} for target %{_target} on␣
 ↪host %{_host}
```

### 7.8.9 %release

The %release is packaging number that allows revisions of a package to happen where none package versions change. This value typically increases when the configuration building the package changes:

```
%define release 1
```

### 7.8.10 %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:

```
Version: %{gcc_version}
```

### 7.8.11 %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.

### 7.8.12 %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 separe 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. Subsequence 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:

```
%source set gcc <1> 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:

```
%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:

```
%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.

### 7.8.13 %patch

The %patch macro has the same 3 command as the %source command however the set commands is not really that useful with the with 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 search.

To add a patch:

```
%patch add gcc <1>   gcc-4.7.2-rtems4.11-
 ↪20121026.diff
%patch add gcc -p0 <2>  gcc-4.7.2-rtems4.11-
 ↪20121101.diff
```

**Items:**

1. The patch group is gcc.
2. Option for this specific patch.

Placing %patch setup in the %prep section will apply the groups patches:

```
 %patch setup gcc <1>  -p1 <2>

1. The patch group.

2. The default option used to apply the␣
 ↪patch.
```

### 7.8.14 %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 has a hash defined generates a warning.

A file to be checksum must be unqiue in the any of the source and patch directories. The basename of the file is used as the key for the hash.

The hash algorthim can be `md5`, `sha1`, `sha224`, `sha256`, `sha384`, and `sha512` and we typically use `md5`.

To add a hash:

```
%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 conponent would create confusion. This does mean files with hashes must be unique.

Downloading of repositories such as git and cvs cannot be checksumed. It is assumed those protocols and tools manage the state of the files.

### 7.8.15 %echo

The `%echo` macro outputs the following string to stdout. This can also be used as `%{echo: message}`.

### 7.8.16 %warning

The `%warning` macro outputs the following string as a warning. This can also be used as `%{warning: message}`.

### 7.8.17 %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}`.

### 7.8.18 %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 is a configuration file without having to edit it. The changes are recorded in the build report so can be traced.

Configuration use different maps so macro overrides can target a specific package.

The default map is `global`:

```
%select gcc-4.8-snapshot <1>
%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.

### 7.8.19 %define

The `%define` macro defines a new macro or updates an existing one. If no value is given it is assumed to be `1`:

```
%define foo bar
%define one_plus_one 2
%define one <1>
```

**Items:**

1. The macro _one_ is set to 1.

### 7.8.20 %undefine

The `%undefine` macro removes a macro if it exists. Any further references to it will result in an undefine macro error.

### 7.8.21 %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:

| | |
|---|---|
| **%{}** | Check the macro is set or *true*, ie non-zero:<br>`%if ${foo}`<br>`  %warning The test passes, must not be empty or is non-zero`<br>`%else`<br>`  %error The test fails, must be empty or zero`<br>`%endif` |
| **!** | The *not* operator inverts the test of the macro:<br>`%if ! ${foo}`<br>`  %warning The test passes, must be empty or zero`<br>`%else`<br>`  %error The test fails, must not be empty or is non-zero`<br>`%endif` |
| **==** | The left hand size must equal the right hand side. For example:<br>`%define one 1`<br>`%if ${one} == 1`<br>`  %warning The test passes`<br>`%else`<br>`  %error The test fails`<br>`%endif`<br>You can also check to see if a macro is empty:<br>`%if ${nothing} == %{nil}`<br>`  %warning The test passes`<br>`%else`<br>`  %error The test fails` |
| **!=** | The left hand size does not equal the right hand side. For |

### 7.8.22 %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:

```
%ifn %{defined foo}
 %define foo bar
%endif
```

### 7.8.23 %ifarch

The `%ifarch` is a short cut for `%if %{_arch} == i386`. Currently not used.

### 7.8.24 %ifnarch

The `%ifnarch` is a short cut for `%if %{_arch} != i386`. Currently not used.

### 7.8.25 %ifos

The `%ifos` is a short cut for `%if %{_os} != mingw32`. It allows conditional support for various operating system differences when building packages.

### 7.8.26 %else

The `%else` macro starts the conditional *else* block.

### 7.8.27 %endfi

The `%endif` macro ends a conditional logic block.

### 7.8.28 %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.

### 7.8.29 %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.

CHAPTER
**EIGHT**

# COMMANDS

## 8.1 Checker (sb-check)

This commands checks your system is set up correctly. Most options are ignored:

```
$ ../source-builder/sb-check --help
sb-check: [options] [args]
RTEMS Source Builder, an RTEMS Tools Project (c) 2012-2013 Chris Johns
Options and arguments:
--force                 : Force the build to proceed
--quiet                 : Quiet output (not used)
--trace                 : Trace the execution
--dry-run               : Do everything but actually run the build
--warn-all              : Generate warnings
--no-clean              : Do not clean up the build tree
--always-clean          : Always clean the build tree, even with an error
--jobs                  : Run with specified number of jobs, default: num CPUs.
--host                  : Set the host triplet
--build                 : Set the build triplet
--target                : Set the target triplet
--prefix path           : Tools build prefix, ie where they are installed
--topdir path           : Top of the build tree, default is $PWD
--configdir path        : Path to the configuration directory, default: ./config
--builddir path         : Path to the build directory, default: ./build
--sourcedir path        : Path to the source directory, default: ./source
--tmppath path          : Path to the temp directory, default: ./tmp
--macros file[,[file]   : Macro format files to load after the defaults
--log file              : Log file where all build out is written too
--url url[,url]         : URL to look for source
--no-download           : Disable the source downloader
--targetcflags flags    : List of C flags for the target code
--targetcxxflags flags  : List of C++ flags for the target code
--libstdcxxflags flags  : List of C++ flags to build the target libstdc++ code
--with-<label>          : Add the --with-<label> to the build
--without-<label>       : Add the --without-<label> to the build
--regression            : Set --no-install, --keep-going and --always-clean
$ ../source-builder/sb-check
RTEMS Source Builder - Check, v0.2.0
Environment is ok
```

## 8.2 Defaults (sb-defaults)

This commands outputs and the default macros for your when given no arguments. Most options are ignored:

```
$ ../source-builder/sb-defaults --help
sb-defaults: [options] [args]
RTEMS Source Builder, an RTEMS Tools Project (c) 2012-2013 Chris Johns
Options and arguments:
--force                : Force the build to proceed
--quiet                : Quiet output (not used)
--trace                : Trace the execution
--dry-run              : Do everything but actually run the build
--warn-all             : Generate warnings
--no-clean             : Do not clean up the build tree
--always-clean         : Always clean the build tree, even with an error
--jobs                 : Run with specified number of jobs, default: num CPUs.
--host                 : Set the host triplet
--build                : Set the build triplet
--target               : Set the target triplet
--prefix path          : Tools build prefix, ie where they are installed
--topdir path          : Top of the build tree, default is $PWD
--configdir path       : Path to the configuration directory, default: ./config
--builddir path        : Path to the build directory, default: ./build
--sourcedir path       : Path to the source directory, default: ./source
--tmppath path         : Path to the temp directory, default: ./tmp
--macros file[,[file]  : Macro format files to load after the defaults
--log file             : Log file where all build out is written too
--url url[,url]        : URL to look for source
--no-download          : Disable the source downloader
--targetcflags flags   : List of C flags for the target code
--targetcxxflags flags : List of C++ flags for the target code
--libstdcxxflags flags : List of C++ flags to build the target libstdc++ code
--with-<label>         : Add the --with-<label> to the build
--without-<label>      : Add the --without-<label> to the build
--regression           : Set --no-install, --keep-going and --always-clean
```

## 8.3 Set Builder (sb-set-builder)

This command builds a set:

```
$ ../source-builder/sb-set-builder --help
RTEMS Source Builder, an RTEMS Tools Project (c) 2012-2013 Chris Johns
Options and arguments:
--force                : Force the build to proceed
--quiet                : Quiet output (not used)
--trace                : Trace the execution
--dry-run              : Do everything but actually run the build
--warn-all             : Generate warnings
--no-clean             : Do not clean up the build tree
--always-clean         : Always clean the build tree, even with an error
--regression           : Set --no-install, --keep-going and --always-clean
---jobs                : Run with specified number of jobs, default: num CPUs.
--host                 : Set the host triplet
--build                : Set the build triplet
--target               : Set the target triplet
--prefix path          : Tools build prefix, ie where they are installed
--topdir path          : Top of the build tree, default is $PWD
--configdir path       : Path to the configuration directory, default: ./config
--builddir path        : Path to the build directory, default: ./build
--sourcedir path       : Path to the source directory, default: ./source
--tmppath path         : Path to the temp directory, default: ./tmp
--macros file[,[file]  : Macro format files to load after the defaults
--log file             : Log file where all build out is written too
--url url[,url]        : URL to look for source
--no-download          : Disable the source downloader
--no-install           : Do not install the packages to the prefix
--targetcflags flags   : List of C flags for the target code
--targetcxxflags flags : List of C++ flags for the target code
--libstdcxxflags flags : List of C++ flags to build the target libstdc++ code
--with-<label>         : Add the --with-<label> to the build
--without-<label>      : Add the --without-<label> to the build
--mail-from            : Email address the report is from.
--mail-to              : Email address to send the email too.
--mail                 : Send email report or results.
--smtp-host            : SMTP host to send via.
--no-report            : Do not create a package report.
--report-format        : The report format (text, html, asciidoc).
--bset-tar-file        : Create a build set tar file
--pkg-tar-files        : Create package tar files
--list-bsets           : List available build sets
--list-configs         : List available configurations
--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+.

**--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.

## 8.4 Set Builder (sb-builder)

This command builds a configuration as described in a configuration file. Configuration files have the extension of `.cfg`:

```
$ ./source-builder/sb-builder --help
sb-builder: [options] [args]
RTEMS Source Builder, an RTEMS Tools Project (c) 2012 Chris Johns
Options and arguments:
--force                : Force the build to proceed
--quiet                : Quiet output (not used)
--trace                : Trace the execution
--dry-run              : Do everything but actually run the build
--warn-all             : Generate warnings
--no-clean             : Do not clean up the build tree
--always-clean         : Always clean the build tree, even with an error
--jobs                 : Run with specified number of jobs, default: num CPUs.
--host                 : Set the host triplet
--build                : Set the build triplet
--target               : Set the target triplet
--prefix path          : Tools build prefix,  ie where they are installed
--topdir path          : Top of the build tree, default is $PWD
--configdir path       : Path to the configuration directory, default: ./config
--builddir path        : Path to the build directory, default: ./build
--sourcedir path       : Path to the source directory, default: ./source
--tmppath path         : Path to the temp directory, default: ./tmp
--macros file[,[file]  : Macro format files to load after the defaults
--log file             : Log file where all build out is written too
--url url[,url]        : URL to look for source
--targetcflags flags   : List of C flags for the target code
--targetcxxflags flags : List of C++ flags for the target code
--libstdcxxflags flags : List of C++ flags to build the target libstdc++ code
--with-<label>         : Add the --with-<label> to the build
--without-<label>      : Add the --without-<label> to the build
--list-configs         : List available configurations
```

CHAPTER
**NINE**

# 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 seach for `error:` and the error should be just above it.

# 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.

- genindex
- search