# Getting Started with RTEMS for C/C++ Users

Edition 1, for 4.5.0

26 September 2000

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 1999-04-25.15



Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to either:

```
On-Line Applications Research Corporation
4910-L Corporate Drive
Huntsville, AL 35805
VOICE: (256) 722-9985
FAX:   (256) 722-0985
EMAIL: rtems@OARcorp.com
```

# 1 Introduction

The purpose of this document is to guide you through the process of installing a GNU cross development environment to use with RTEMS.

If you are already familiar with the concepts behind a cross compiler and have a background in Unix, these instructions should provide the bare essentials for performing a setup of the following items:

- GNU C/C++ Cross Compilation Tools for RTEMS on your build-host system
- RTEMS OS for the target
- GDB Debugger

The remainder of this chapter provides background information on real-time embedded systems and cross development and an overview of other resources of interest on the Internet. If you are not familiar with real-time embedded systems or the other areas, please read those sections. These sections will help familiarize you with the types of systems RTEMS is designed to be used in and the cross development process used when developing RTEMS applications.

## 1.1 Real-Time Embedded Systems

Real-time embedded systems are found in practically every facet of our everyday lives. Today's systems range from the common telephone, automobile control systems, and kitchen appliances to complex air traffic control systems, military weapon systems, and production line control including robotics and automation. However, in the current climate of rapidly changing technology, it is difficult to reach a consensus on the definition of a real-time embedded system. Hardware costs are continuing to rapidly decline while at the same time the hardware is increasing in power and functionality. As a result, embedded systems that were not considered viable two years ago are suddenly a cost effective solution. In this domain, it is not uncommon for a single hardware configuration to employ a variety of architectures and technologies. Therefore, we shall define an embedded system as any computer system that is built into a larger system consisting of multiple technologies such as digital and analog electronics, mechanical devices, and sensors.

Even as hardware platforms become more powerful, most embedded systems are critically dependent on the real-time software embedded in the systems themselves. Regardless of how efficiently the hardware operates, the performance of the embedded real-time software determines the success of the system. As the complexity of the embedded hardware platform grows, so does the size and complexity of the embedded software. Software systems must routinely perform activities which were only dreamed of a short time ago. These large, complex, real-time embedded applications now commonly contain one million lines of code or more.

Real-time embedded systems have a complex set of characteristics that distinguish them from other software applications. Real-time embedded systems are driven by and must respond to real world events while adhering to rigorous requirements imposed by the environment with which they interact. The correctness of the system depends not only on the results of computations, but also on the time at which the results are produced. The most important and complex characteristic of real-time application systems is that they must receive and respond to a set of external stimuli within rigid and critical time constraints.

A single real-time application can be composed of both soft and hard real-time components. A typical example of a hard real-time system is a nuclear reactor control system that must not only detect failures, but must also respond quickly enough to prevent a meltdown. This application also has soft real-time requirements because it may involve a man-machine interface. Providing an interactive input to the control system is not as critical as setting off an alarm to indicate a failure condition. However, the interactive system component must respond within an acceptable time limit to allow the operator to interact efficiently with the control system.

## 1.2 Cross Development

Today almost all real-time embedded software systems are developed in a **cross development** environment using cross development tools. In the cross development environment, software development activities are typically performed on one computer system, the **build-host** system, while the result of the development effort (produced by the cross tools) is a software system that executes on the **target** platform. The requirements for the target platform are usually incompatible and quite often in direct conflict with the requirements for the build-host. Moreover, the target hardware is often custom designed for a particular project. This means that the cross development toolset must allow the developer to customize the tools to address target specific run-time issues. The toolset must have provisions for board dependent initialization code, device drivers, and error handling code.

The build-host computer is optimized to support the code development cycle with support for code editors, compilers, and linkers requiring large disk drives, user development windows, and multiple developer connections. Thus the build-host computer is typically a traditional UNIX workstation such as those available from SUN or Silicon Graphics, or a PC running either a version of MS-Windows or UNIX. The build-host system may also be required to execute office productivity applications to allow the software developer to write documentation, make presentations, or track the project's progress using a project management tool. This necessitates that the build-host computer be general purpose with resources such as a thirty-two or sixty-four bit processor, large amounts of RAM, a monitor, mouse, keyboard, hard and floppy disk drives, CD-ROM drive, and a graphics card. It is likely that the system will be multimedia capable and have some networking capability.

Conversely, the target platform generally has limited traditional computer resources. The hardware is designed for the particular functionality and requirements of the embedded system and optimized to perform those tasks effectively. Instead of hard drives and keyboards, it is composed of sensors, relays, and stepper motors. The per-unit cost of the target platform is typically a critical concern. No hardware component is included without being cost justified. As a result, the processor of the target system is often from a different processor family than that of the build-host system and usually has lower performance. In addition to the processor families designed only for use in embedded systems, there are versions of nearly every general-purpose processor specifically tailored for real-time embedded systems. For example, many of the processors targeting the embedded market do not include hardware floating point units, but do include peripherals such as timers, serial controllers, or network interfaces.

## 1.3 Resources on the Internet

This section describes various resources on the Internet which are of use to RTEMS users.

### 1.3.1 Online Tool Documentation

Each of the tools in the GNU development suite comes with documentation. It is in the reader's and tool maintainers' interest that one read the documentation before posting a problem to a mailing list or news group. The RTEMS Project provides formatted documentation for the primary tools in the cross development toolset including BINUTILS, GCC, NEWLIB, and GDB at http://www.oarcorp.com/rtemsdoc-4.5.0.

Much of the documentation is available at other sites on the Internet. The following is a list of URLs where one can find HTML versions of the GNU manuals:

**Free Software Foundation**
: http://www.gnu.org/manual/manual.html

**Delorie Software**
: http://www.delorie.com/gnu/docs

### 1.3.2 RTEMS Mailing List

rtems-users@OARcorp.com

This mailing list is dedicated to the discussion of issues related to RTEMS, including GNAT/RTEMS. If you have questions about RTEMS, wish to make suggestions, or just want to pick up hints, this is a good list to monitor. Subscribe by sending an empty mail message to rtems-users-subscribe@OARcorp.com. Messages sent to rtems-users@OARcorp.com are posted to the list.

### 1.3.3 CrossGCC Mailing List

crossgcc@sources.redhat.com

This mailing list is dedicated to the use of the GNU tools in cross development environments. Most of the discussions focus on embedded issues. Information on subscribing to this mailing list is included in the CrossGCC FAQ.

The CrossGCC FAQ as well as a number of patches and utilities of interest to cross development system users are available at `ftp://ftp.cygnus.com/pub/embedded/crossgcc`.

### 1.3.4 GCC Mailing Lists

The GCC Project is hosted at http://gcc.gnu.org. They maintain multiple mailing lists that are described at the web site along with subscription information.

# 2 Requirements

This chapter describes the build-host system requirements and initial steps in installing the GNU C/C++ Cross Compiler Tools and RTEMS on a build-host.

## 2.1 Disk Space

A fairly large amount of disk space is required to perform the build of the GNU C/C++ Cross Compiler Tools for RTEMS. The following table may help in assessing the amount of disk space required for your installation:

| Component | Disk Space Required |
| --- | --- |
| archive directory | 35 Mbytes |
| tools src unarchived | 150 Mbytes |
| each individual build directory | up to 500 Mbytes |
| each installation directory | 20-200 Mbytes |

It is important to understand that the above requirements only address the GNU C/C++ Cross Compiler Tools themselves. Adding additional languages such as Fortran or Objective-C can increase the size of the build and installation directories. Also, the unarchived source and build directories can be removed after the tools are installed.

After the tools themselves are installed, RTEMS must be built and installed for each Board Support Package that you wish to use. Thus the precise amount of disk space required for each installation directory depends highly on the number of RTEMS BSPs which are to be installed. If a single BSP is installed, then the additional size of each install directory will tend to be in the 40-60 Mbyte range.

There are a number of factors which must be taken into account in order to estimate the amount of disk space required to build RTEMS itself. Attempting to build multiple BSPs in a single step increases the disk space requirements. Similarly enabling optional features increases the build and install space requirements. In particular, enabling and building the RTEMS tests results in a significant increase in build space requirements but since the tests are not installed has, enabling them has no impact on installation requirements.

## 2.2 General Host Software Requirements

The instructions in this manual should work on any computer running a UNIX variant. Some native GNU tools are used by this procedure including:

- GCC
- GNU make
- GNU makeinfo

In addition, some native utilities may be deficient for building the GNU tools.

### 2.2.1 GCC

Although RTEMS itself is intended to execute on an embedded target, there is source code for some native programs included with the RTEMS distribution. Some of these programs are used to assist in the building of RTEMS itself, while others are BSP specific tools. Regardless, no attempt has been made to compile these programs with a non-GNU compiler.

### 2.2.2 GNU Make

Both NEWLIB and RTEMS use GNU make specific features and can only be built using GNU make. Many systems include a make utility that is not GNU make. The safest way to meet this requirement is to ensure that when you invoke the command `make`, it is GNU make. This can be verified by attempting to print the GNU make version information:

```
make --version
```

If you have GNU make and another make on your system, it is common to put the directory containing GNU make before the directory containing other implementations of make.

### 2.2.3 GNU makeinfo Version Requirements

In order to build gcc 2.9.x or newer versions, the GNU `makeinfo` program installed on your system must be at least version 1.68. The appropriate version of `makeinfo` is distributed with `gcc`.

The following demonstrates how to determine the version of `makeinfo` on your machine:

```
makeinfo --version
```

## 2.3 Host Specific Notes

### 2.3.1 Solaris 2.x

The following problems have been reported by Solaris 2.x users:

- The build scripts are written in "shell". The program `/bin/sh` on Solaris 2.x is not robust enough to execute these scripts. If you are on a Solaris 2.x host, then change the first line of the files `bit`, `bit_gdb`, and `bit_rtems` to use the `/bin/ksh` shell instead.
- The native `patch` program is broken. Install the GNU version.
- The native `m4` program is deficient. Install the GNU version.

### 2.3.2 Linux

The following problems have been reported by Linux users:

- Certain versions of GNU fileutils include a version of `install` which does not work properly. Please perform the following test to see if you need to upgrade:

```
install -c -d /tmp/foo/bar
```

If this does not create the specified directories your install program will not install RTEMS properly. You will need to upgrade to at least GNU fileutils version 3.16 to resolve this problem.

## 2.4 Archive and Build Directories

If you are using RPM or another packaging format that supports building a package from source, then there is probably a directory structure assumed by that packaging format. Otherwise, you are free to use whatever organization you like. However, this document will use the directory organization described in Section 2.4.2 [Archive and Build Directory Format], page 7.

### 2.4.1 RPM Archive and Build Directory Format

For RPM, it is assumed that the following subdirectories are under a root directory such as `/usr/src/redhat`:

```
BUILD
RPMS
SOURCES
SPECS
SRPMS
```

For the purposes of this document, the RPM `SOURCES` directory is the directory into which all tool source and patches are assumed to reside. The `BUILD` directory is where the actual build is performed when building binaries from a source RPM. The `SOURCES` and `BUILD` are logically equivalent to the `archive` and `tools` directory discussed in the next section.

### 2.4.2 Archive and Build Directory Format

When no packaging format requirements are present, the root directory for the storage of source archives and patches as well as for building the tools is up to the user. The only concern is that there be enough disk space to complete the build.

Make an `archive` directory to contain the downloaded source code and a `tools` directory to be used as a build directory. The command sequence to do this is shown below:

```
mkdir archive
mkdir tools
```

This will result in an initial directory structure similar to the one shown in the following figure:

```
/whatever/prefix/you/choose/
        archive/
        tools/
```

# 3 Prebuilt Toolset Executables

Precompiled toolsets are available for Linux, Cygwin, FreeBSD, and Solaris. These are packaged in the following formats:

- Linux - RPM and Debian
- Cygwin - RPM and zipped tar
- FreeBSD - native package
- Solaris - RPM and zipped tar

RPM is an acronym for the RedHat Package Manager. RPM is the native package installer for many Linux distributions including RedHat and SuSE. RPM supports other operating systems including Cygwin. David Fiddes <D.J@fiddes.surfaid.org> did the initial groundwork that lead to Cygwin RPMs being available.

The prebuilt binaries are intended to be easy to install and the instructions are similar regardless of the host environment. There are a few structural issues with the packaging of the RTEMS Cross Toolset binaries that you need to be aware of.

1. There are dependencies between the various packages. This requires that certain packages be installed before others may be. Some packaging formats enforce this dependency.
2. Some packages are target CPU family independent and shared across all target architectures. These are referred to as "base" packages.
3. If buildable for a particular CPU, RPMs are provided for Chill, Java (gcj), Fortran (g77), and Objective-C (objc). These binaries are strictly optional.

NOTE: Installing toolset binaries does not install RTEMS itself, only the tools required to build RTEMS. See Chapter 5 [Building RTEMS], page 27 for the next step in the process.

## 3.1 RPMs

This section provides information on installing and removing RPMs.

### 3.1.1 Installing RPMs

The following is a sample session illustrating the installation of a C/C++ toolset targeting the SPARC architecture.

```
rpm -i rtems-base-binutils-2.9.5.0.24-1.i386.rpm
rpm -i sparc-rtems-binutils-2.9.5.0.24-1.i386.rpm
rpm -i rtems-base-gcc-gcc2.95.2newlib1.8.2-7.i386.rpm
rpm -i sparc-rtems-gcc-gcc2.95.2newlib1.8.2-7.i386.rpm
rpm -i rtems-base-gdb-4.18-4.i386.rpm
rpm -i sparc-rtems-gdb-4.18-4.i386.rpm
```

Upon successful completion of the above command sequence, a C/C++ cross development toolset targeting the SPARC is installed in `/opt/rtems`. In order to use this toolset, the directory `/opt/rtems/bin` must be included in your PATH.

Once you have successfully installed the RPMs for BINUTILS, GCC, NEWLIB, and GDB, then you may proceed directly to Chapter 5 [Building RTEMS], page 27.

### 3.1.2 Determining Which RTEMS RPMs are Installed

The following command will report which RTEMS RPMs are currently installed:

```
rpm -q -g rtems
```

### 3.1.3 Removing RPMs

The following is a sample session illustrating the removal of a C/C++ toolset targeting the SPARC architecture.

```
rpm -e sparc-rtems-gdb-4.18-2.i386.rpm
rpm -e rtems-base-gdb-4.18-2.i386.rpm
rpm -e sparc-rtems-gcc-gcc2.95.2newlib1.8.2-4.i386.rpm
rpm -e rtems-base-gcc-gcc2.95.2newlib1.8.2-4.i386.rpm
rpm -e sparc-rtems-binutils-2.9.5.0.24-1.i386.rpm
rpm -e rtems-base-binutils-2.9.5.0.24-1.i386.rpm
```

NOTE: If you have installed any RTEMS BSPs, then it is likely that RPM will complain about not being able to remove everything.

## 3.2 Zipped Tar Files

This section provides information on installing and removing Zipped Tar Files (.tgz).

### 3.2.1 Installing Zipped Tar Files

The following is a sample session illustrating the installation of a C/C++ toolset targeting the SPARC architecture assuming that GNU tar is installed as `tar`:

```
cd /
tar xzf rtems-base-binutils-2.9.5.0.24-1.tgz
tar xzf sparc-rtems-binutils-2.9.5.0.24-1.tgz
tar xzf rtems-base-gcc-gcc2.95.2newlib1.8.2-4.tgz
tar xzf sparc-rtems-gcc-gcc2.95.2newlib1.8.2-4.tgz
tar xzf rtems-base-gdb-4.18-2.tgz
tar xzf sparc-rtems-gdb-4.18-2.tgz
```

Upon successful completion of the above command sequence, a C/C++ cross development toolset targeting the SPARC is installed in `/opt/rtems`. In order to use this toolset, the directory `/opt/rtems/bin` must be included in your PATH.

### 3.2.2 Removing Zipped Tar Files

There is no automatic way to remove the contents of a `tgz` once it is installed. The contents of the directory `/opt/rtems` can be removed but this will likely result in other packages being removed as well.

# 4 Building the GNU C/C++ Cross Compiler Toolset

NOTE: This chapter does **NOT** apply if you installed prebuilt toolset executables for BINUTILS, GCC, NEWLIB, and GDB. If you installed prebuilt executables for all of those, proceed to Chapter 5 [Building RTEMS], page 27. If you require a GDB with a special configuration to connect to your target board, then proceed to Section 4.2 [Building the GNU Debugger GDB], page 20 for some advice.

This chapter describes the steps required to acquire the source code for a GNU cross compiler toolset, apply any required RTEMS specific patches, compile that toolset and install it.

It is recommended that when toolset binaries are available for your particular host, that they be used. Prebuilt binaries are much easier to install.

## 4.1 Building BINUTILS GCC and NEWLIB

NOTE: This step is NOT required if prebuilt executables for BINUTILS, GCC, and NEWLIB were installed.

This section describes the process of building BINUTILS, GCC, and NEWLIB using a variety of methods. Included is information on obtaining the source code and patches, applying patches, and building and installing the tools using multiple methods.

### 4.1.1 Obtain Source and Patches for BINUTILS GCC and NEWLIB

NOTE: This step is required for all methods of building BINUTILS, GCC, and NEWLIB.

This section lists the components required to build BINUTILS, GCC, and NEWLIB from source to target RTEMS. These files should be placed in your `archive` directory. Included are the locations of each component as well as any required RTEMS specific patches.

#### gcc 2.95.2

```
FTP Site:    gcc.gnu.org
Directory:   /pub/gcc/gcc-2.95.2
File:        gcc-2.95.2.tar.gz
```

#### binutils 2.9.5.0.24

```
FTP Site:    ftp.varesearch.com
Directory:   /pub/support/hjl/binutils
File:        binutils-2.9.5.0.24.tar.gz
```

#### newlib 1.8.2

```
FTP Site:    sources.redhat.com
Directory:   /pub/newlib
File:        newlib-1.8.2.tar.gz
```

### RTEMS Specific Tool Patches and Scripts

```
FTP Site:    ftp.OARcorp.com
Directory:   /pub/rtems/releases/4.5.0/c_tools/source
File:        c_build_scripts-4.5.0.tgz
File:        binutils-2.9.5.0.24-rtems-20000207.diff.gz
File:        newlib-1.8.2-rtems-20000606.diff.gz
File:        gcc-2.95.2-rtems-20000606.diff.gz
```

### 4.1.2 Unarchiving the Tools

NOTE: This step is required if building BINUTILS, GCC, and NEWLIB using the procedures described in Section 4.1.4.2 [Using configure and make], page 16 or Section 4.1.4.3 [Using the bit Script], page 17. It is **NOT** required if using the procedure described in Section 4.1.4.1 [Using RPM to Build BINUTILS GCC and NEWLIB], page 14.

While in the `tools` directory, unpack the compressed tar files using the following command sequence:

```
cd tools
tar xzf ../archive/gcc-2.95.2.tar.gz
tar xzf ../archive/binutils-2.9.5.0.24.tar.gz
tar xzf ../archive/newlib-1.8.2.tar.gz
```

After the compressed tar files have been unpacked, the following directories will have been created under tools.

- binutils-2.9.5.0.24
- gcc-2.95.2
- newlib-1.8.2

The tree should look something like the following figure:

```
/whatever/prefix/you/choose/
        archive/
            gcc-2.95.2.tar.gz
            binutils-2.9.5.0.24.tar.gz
            newlib-1.8.2.tar.gz
            gcc-2.95.2-rtems-20000606.diff.gz
            binutils-2.9.5.0.24-rtems-20000207.diff.gz
            newlib-1.8.2-rtems-20000606.diff.gz
        tools/
            binutils-2.9.5.0.24/
            gcc-2.95.2/
            newlib-1.8.2/
```

### 4.1.3 Applying RTEMS Patches

NOTE: This step is required if building BINUTILS, GCC, and NEWLIB using the procedures described in Section 4.1.4.2 [Using configure and make], page 16 or Section 4.1.4.3 [Using the bit Script], page 17. It is **NOT** required if using the procedure described in Section 4.1.4.1 [Using RPM to Build BINUTILS GCC and NEWLIB], page 14.

This section describes the process of applying the RTEMS patches to GCC, NEWLIB, and BINUTILS.

#### Apply RTEMS Patch to GCC

Apply the patch using the following command sequence:

```
cd tools/gcc-2.95.2
zcat ../../archive/gcc-2.95.2-rtems-20000606.diff.gz | \
    patch -p1
```

Check to see if any of these patches have been rejected using the following sequence:

```
cd tools/gcc-2.95.2
find . -name "*.rej" -print
```

If any files are found with the .rej extension, a patch has been rejected. This should not happen with a good patch file which is properly applied.

#### Apply RTEMS Patch to binutils

Apply the patch using the following command sequence:

```
cd tools/binutils-2.9.5.0.24
zcat ../../archive/binutils-2.9.5.0.24-rtems-20000207.diff.gz | \
    patch -p1
```

Check to see if any of these patches have been rejected using the following sequence:

```
cd tools/binutils-2.9.5.0.24
find . -name "*.rej" -print
```

If any files are found with the .rej extension, a patch has been rejected. This should not happen with a good patch file which is properly applied.

#### Apply RTEMS Patch to newlib

Apply the patch using the following command sequence:

```
cd tools/newlib-1.8.2
zcat ../../archive/newlib-1.8.2-rtems-20000606.diff.gz | \
    patch -p1
```

Check to see if any of these patches have been rejected using the following sequence:

```
cd tools/newlib-1.8.2
find . -name "*.rej" -print
```

If any files are found with the .rej extension, a patch has been rejected. This should not happen with a good patch file which is properly applied.

### 4.1.4 Compiling and Installing BINUTILS GCC and NEWLIB

There are three methods to compile and install BINUTILS, GCC, and NEWLIB:

- RPM
- direct invocation of `configure` and `make`
- using the `bit` script

Direct invocation of `configure` and `make` provides more control and easier recovery from problems when building.

#### 4.1.4.1 Using RPM to Build BINUTILS GCC and NEWLIB

NOTE: The procedures described in the following sections must be completed before this step:

- Section 4.1.1 [Obtain Source and Patches for BINUTILS GCC and NEWLIB], page 11

RPM automatically unarchives the source and applies any needed patches so you do **NOT** have to manually perform the procedures described Section 4.1.2 [Unarchiving the Tools], page 12 and Section 4.1.3 [Applying RTEMS Patches], page 13.

This section describes the process of building binutils, gcc, and newlib using RPM. RPM is a packaging format which can be used to distribute binary files as well as to capture the procedure and source code used to produce those binary files. Before attempting to build any RPM from source, it is necessary to ensure that all required source and patches are in the `SOURCES` directory under the RPM root (probably `/usr/src/redhat` or `/usr/local/src/redhat`) on your machine. This procedure starts by installing the source RPMs as shown in the following example:

```
rpm -i i386-rtems-binutils-collection-2.9.5.0.24-1.nosrc.rpm
rpm -i i386-rtems-gcc-newlib-gcc2.95.2newlib1.8.2-7.nosrc.rpm
```

Because RTEMS tool RPMS are called "nosrc" to indicate that one or more source files required to produce the RPMs are not present. The RTEMS source RPMs do not include the large `.tar.gz` or `.tgz` files for each component such as BINUTILS, GCC, or NEWLIB. These are shared by all RTEMS RPMs regardless of target CPU and there was no reason to duplicate them. You will have to get the required source archive files by hand and place them in the `SOURCES` directory before attempting to build. If you forget to do this, RPM is smart – it will tell you what is missing. To determine what is included or referenced by a particular RPM, use a command like the following:

```
$ rpm -q -l -p i386-rtems-binutils-collection-2.9.5.0.24-1.nosrc.rpm
binutils-2.9.5.0.24-rtems-20000207.diff
binutils-2.9.5.0.24.tar.gz
i386-rtems-binutils-2.9.5.0.24.spec
```

Notice that there is a patch file (the `.diff` file), a source archive file (the `.tar.gz`), and a file describing the build procedure and files produced (the `.spec` file). The `.spec` file is placed in the `SPECS` directory under the RPM root directory.

## Configuring and Building BINUTILS using RPM

The following example illustrates the invocation of RPM to build a new, locally compiled, binutils binary RPM that matches the installed source RPM. This example assumes that all of the required source is installed.

```
cd <RPM_ROOT_DIRECTORY>/SPECS
rpm -bb i386-rtems-binutils-2.9.5.0.24.spec
```

If the build completes successfully, RPMS like the following will be generated in a build-host architecture specific subdirectory of the RPMS directory under the RPM root directory.

```
rtems-base-binutils-2.9.5.0.24-1.i386.rpm
i386-rtems-binutils-2.9.5.0.24-1.i386.rpm
```

NOTE: It may be necessary to remove the build tree in the `BUILD` directory under the RPM root directory.

## Configuring and Building GCC and NEWLIB using RPM

The following example illustrates the invocation of RPM to build a new, locally compiled, set of GCC and NEWLIB binary RPMs that match the installed source RPM. It is also necessary to install the BINUTILS RPMs and place them in your PATH. This example assumes that all of the required source is installed.

```
cd <RPM_ROOT_DIRECTORY>/RPMS/i386
rpm -i rtems-base-binutils-2.9.5.0.24-1.i386.rpm
rpm -i i386-rtems-binutils-2.9.5.0.24-1.i386.rpm
export PATH=/opt/rtems/bin:$PATH
cd <RPM_ROOT_DIRECTORY>/SPECS
rpm -bb i386-rtems-gcc-2.95.2-newlib-1.8.2.spec
```

If the build completes successfully, a set of RPMS like the following will be generated in a build-host architecture specific subdirectory of the RPMS directory under the RPM root directory.

```
rtems-base-gcc-gcc2.95.2newlib1.8.2-7.i386.rpm
rtems-base-chill-gcc2.95.2newlib1.8.2-7.i386.rpm
rtems-base-g77-gcc2.95.2newlib1.8.2-7.i386.rpm
rtems-base-gcj-gcc2.95.2newlib1.8.2-7.i386.rpm
i386-rtems-gcc-gcc2.95.2newlib1.8.2-7.i386.rpm
i386-rtems-chill-gcc2.95.2newlib1.8.2-7.i386.rpm
i386-rtems-g77-gcc2.95.2newlib1.8.2-7.i386.rpm
i386-rtems-gcj-gcc2.95.2newlib1.8.2-7.i386.rpm
i386-rtems-objc-gcc2.95.2newlib1.8.2-7.i386.rpm
```

NOTE: Some targets do not support building all languages.

NOTE: It may be necessary to remove the build tree in the `BUILD` directory under the RPM root directory.

#### 4.1.4.2 Using configure and make

NOTE: The procedures described in the following sections must be completed before this step:

- Section 4.1.1 [Obtain Source and Patches for BINUTILS GCC and NEWLIB], page 11
- Section 4.1.2 [Unarchiving the Tools], page 12
- Section 4.1.3 [Applying RTEMS Patches], page 13

This section describes the process of building binutils, gcc, and newlib manually using `configure` and `make` directly.

### Configuring and Building BINUTILS

The following example illustrates the invocation of `configure` and `make` to build and install binutils-2.9.5.0.24 for the sparc-rtems target:

```
mkdir b-binutils
cd b-binutils
../binutils-2.9.5.0.24/configure --target=sparc-rtems \
  --prefix=/opt/rtems
make all
make info
make install
```

After binutils-2.9.5.0.24 is built and installed the build directory `b-binutils` may be removed.

For more information on the invocation of `configure`, please refer to the documentation for binutils-2.9.5.0.24 or invoke the binutils-2.9.5.0.24 `configure` command with the `--help` option.

### Configuring and Building GCC and NEWLIB

Before building gcc-2.95.2 and newlib-1.8.2, binutils-2.9.5.0.24 must be installed and the directory containing those executables must be in your PATH.

The C Library is built as a subordinate component of gcc-2.95.2. Because of this, the newlib-1.8.2 directory source must be available inside the gcc-2.95.2 source tree. This is normally accomplished using a symbolic link as shown in this example:

```
cd gcc-2.95.2
ln -s ../newlib-1.8.2/newlib .
```

The following example illustrates the invocation of `configure` and `make` to build and install binutils-2.9.5.0.24 for the sparc-rtems target:

```
mkdir b-gcc
cd b-gcc
../gcc-2.95.2/configure --target=sparc-rtems \
   --with-gnu-as --with-gnu-ld --with-newlib --verbose \
   --enable-threads --prefix=/opt/rtems
make all
make info
make install
```

After gcc-2.95.2 is built and installed the build directory `b-gcc` may be removed.

For more information on the invocation of `configure`, please refer to the documentation for gcc-2.95.2 or invoke the gcc-2.95.2 `configure` command with the `--help` option.

#### 4.1.4.3 Using the bit Script

NOTE: The procedures described in the following sections must be completed before this step:

- Section 4.1.1 [Obtain Source and Patches for BINUTILS GCC and NEWLIB], page 11
- Section 4.1.2 [Unarchiving the Tools], page 12
- Section 4.1.3 [Applying RTEMS Patches], page 13

This section describes the process of building using the `bit` script. This script hides many of the details of building the tools but can be a hindrance if you encounter problems building the tools.

### Unarchiving the Build Scripts

While in the `tools` directory, unpack the compressed tar file for the build tools using the following command sequence:

```
cd tools
tar xzf ../archive/c_build_scripts-4.5.0.tgz
```

After the compressed tar file c˙build˙scripts-4.5.0.tgz has been unpacked, there will be a set of scripts in the tools directory along with any source code you have previously unarchived. These scripts are intended to aid in building the tools and RTEMS. These scripts may be used to automate the tool building process and hide the invocation of `configure` and `make` from you. They are:

- bit
- bit_gdb
- bit_rtems
- common.sh
- user.cfg

If `bit` is executed later in this process, it will automatically create this subdirectory:

- build-${CPU}-tools

At this point, the tree should look something like the following figure:

```
/whatever/prefix/you/choose/
        archive/
            gcc-2.95.2.tar.gz
            binutils-2.9.5.0.24.tar.gz
            newlib-1.8.2.tar.gz
            c_build_scripts-4.5.0.tgz
            gcc-2.95.2-rtems-20000606.diff.gz
            binutils-2.9.5.0.24-rtems-20000207.diff.gz
            newlib-1.8.2-rtems-20000606.diff.gz
        tools/
            binutils-2.9.5.0.24/
            gcc-2.95.2/
            newlib-1.8.2/
            bit
            bit_gdb
            bit_rtems
            common.sh
            user.cfg
```

## Localizing the Configuration

Edit the `user.cfg` file to alter the settings of various variables which are used to tailor the build process. Each of the variables set in `user.cfg` may be modified as described below:

`INSTALL_POINT`
: is the location where you wish the GNU C/C++ cross compilation tools for RTEMS to be built. It is recommended that the directory chosen to receive these tools be named so that it is clear from which gcc distribution it was generated and for which target system the tools are to produce code for.

  **WARNING**: The `INSTALL_POINT` should not be a subdirectory under the build directory. The build directory will be removed automatically upon successful completion of the build procedure.

`BINUTILS`
: is the directory under tools that contains binutils-2.9.5.0.24. For example:

```
BINUTILS=binutils-2.9.5.0.24
```

`GCC`
: is the directory under tools that contains gcc-2.95.2. For example,

```
GCC=gcc-2.95.2
```

`GDB`
: is the directory under tools that contains gdb-4.18. For example,

```
GDB=gdb-4.18
```

`NEWLIB`
: is the directory under tools that contains newlib-1.8.2. For example:

```
NEWLIB=newlib-1.8.2
```

`BUILD_DOCS`
: is set to "yes" if you want to install documentation. This requires that tools supporting documentation production be installed. This currently is limited to the GNU texinfo package. For example:

```
BUILD_DOCS=yes
```

`BUILD_OTHER_LANGUAGES`
: is set to "yes" if you want to build languages other than C and C++. At the current time, the set of alternative languages includes Java, Fortran, and Objective-C. These alternative languages do not always build cross. Hence this option defaults to "no".

For example:

```
BUILD_OTHER_LANGUAGES=yes
```

**NOTE:** Based upon the version of the compiler being used, it may not be possible to build languages other than C and C++ cross. In many cases, the language run-time support libraries are not "multilib'ed". Thus the executable code in these libraries will be for the default compiler settings and not necessarily be correct for your CPU model.

The other variables in `user.cfg` are RTEMS specific and are not technically required to be set unless you build RTEMS using the `bit_rtems` script as described in Section 5.5.2 [Using the bit_rtems Script], page 29. They are described in detail in that section.

## Running the bit Script

After the `bit` script has been modified to reflect the local installation, the modified `bit` script is run using the following sequence:

```
cd tools
./bit <target configuration>
```

Where <target configuration> is one of the following:

- hppa1.1
- i386
- i386-coff
- i386-elf
- i960
- m68k
- m68k-coff
- mips64orion
- powerpc
- sh
- sh-elf
- sparc

The build process can take a while to complete. Many users find it handy to run the build process in the background, capture the output in a file, and monitor the output. This can be done as follows:

```
./bit <target configuration> >bit.log 2>&1 &
tail -f bit.log
```

If no errors are encountered, the `bit` script will conclude by printing messages similar to the following:

```
The build-i386-tools subdirectory may now be removed.

Started:  Fri Apr 10 10:14:07 CDT 1998
Finished: Fri Apr 10 12:01:33 CDT 1998
```

If the `bit` script successfully completes, then the GNU C/C++ cross compilation tools are installed.

If the `bit` script does not successfully complete, then investigation will be required to determine the source of the error.

## 4.2 Building the GNU Debugger GDB

NOTE: This step is NOT required if prebuilt executables for the GNU Debugger GDB were installed.

The GNU Debugger GDB supports many configurations but requires some means of communicating between the host computer and target board. This communication can be via a serial port, Ethernet, BDM, or ROM emulator. The communication protocol can be the GDB remote protocol or GDB can talk directly to a ROM monitor. This setup is target board specific. The following configurations have been successfully used with RTEMS applications:

- Sparc Instruction Simulator (SIS)
- PowerPC Instruction Simulator (PSIM)
- DINK32
- BDM with 68360 and MPC860 CPUs
- Motorola Mxxxbug found on M68xxx VME boards
- Motorola PPCbug found on PowerPC VME and CompactPCI boards

GDB is currently RTEMS thread/task aware only if you are using the remote debugging support via Ethernet. These are configured using gdb targets of the form CPU-RTEMS. Note the capital RTEMS.

It is recommended that when toolset binaries are available for your particular host, that they be used. Prebuilt binaries are much easier to install but in the case of gdb may or may not include support for your particular target board.

### 4.2.1 Obtain Source and Patches for GDB

NOTE: This step is required for all methods of building GDB.

This section lists the components required to build GDB from source to target RTEMS. These files should be placed in your `archive` directory. Included are the locations of each component as well as any required RTEMS specific patches.

### gdb 4.18

```
FTP Site:    ftp.gnu.org
Directory:   /pub/gnu/gdb
File:        gdb-4.18.tar.gz
```

### RTEMS Specific Tool Patches and Scripts

```
FTP Site:    ftp.OARcorp.com
Directory:   /pub/rtems/releases/4.5.0/c_tools/source
File:        c_build_scripts-4.5.0.tgz
File:        gdb-4.18-rtems-20000524.diff.gz
```

## 4.2.2 Unarchiving the GDB Distribution

Use the following commands to unarchive the GDB distribution:

```
cd tools
tar xzf ../archive/gdb-4.18.tar.gz
```

The directory gdb-4.18 is created under the tools directory.

## 4.2.3 Applying RTEMS Patch to GDB

Apply the patch using the following command sequence:

```
cd tools/gdb-4.18
zcat archive/gdb-4.18-rtems-20000524.diff.gz | \
    patch -p1
```

Check to see if any of these patches have been rejected using the following sequence:

```
cd tools/gdb-4.18
find . -name "*.rej" -print
```

If any files are found with the .rej extension, a patch has been rejected. This should not happen with a good patch file.

## 4.2.4 Compiling and Installing the GNU Debugger GDB

There are three methods of building the GNU Debugger:

- RPM
- direct invocation of `configure` and `make`
- using the `bit_gdb` script

Direct invocation of `configure` and `make` provides more control and easier recovery from problems when building.

#### 4.2.4.1 Using RPM to Build GDB

This section describes the process of building binutils, gcc, and newlib using RPM. RPM is a packaging format which can be used to distribute binary files as well as to capture the procedure and source code used to produce those binary files. Before attempting to build any RPM from source, it is necessary to ensure that all required source and patches are in the `SOURCES` directory under the RPM root (probably `/usr/src/redhat` or `/usr/local/src/redhat`) on your machine. This procedure starts by installing the source RPMs as shown in the following example:

```
rpm -i i386-rtems-gdb-collection-4.18-4.nosrc.rpm
```

Because RTEMS tool RPMS are called "nosrc" to indicate that one or more source files required to produce the RPMs are not present. The RTEMS source GDB RPM does not include the large `.tar.gz` or `.tgz` files for GDB. This is shared by all RTEMS RPMs regardless of target CPU and there was no reason to duplicate them. You will have to get the required source archive files by hand and place them in the `SOURCES` directory before attempting to build. If you forget to do this, RPM is smart – it will tell you what is missing. To determine what is included or referenced by a particular RPM, use a command like the following:

```
$ rpm -q -l -p i386-rtems-gdb-collection-4.18-4.nosrc.rpm
gdb-4.18-rtems-20000524.diff
gdb-4.18.tar.gz
i386-rtems-gdb-4.18.spec
```

Notice that there is a patch file (the `.diff` file), a source archive file (the `.tar.gz`), and a file describing the build procedure and files produced (the `.spec` file). The `.spec` file is placed in the `SPECS` directory under the RPM root directory.

### Configuring and Building GDB using RPM

The following example illustrates the invocation of RPM to build a new, locally compiled, binutils binary RPM that matches the installed source RPM. This example assumes that all of the required source is installed.

```
cd <RPM_ROOT_DIRECTORY>/SPECS
rpm -bb i386-rtems-gdb-4.18.spec
```

If the build completes successfully, RPMS like the following will be generated in a build-host architecture specific subdirectory of the RPMS directory under the RPM root directory.

```
rtems-base-gdb-4.18-4.i386.rpm
i386-rtems-gdb-4.18-4.i386.rpm
```

NOTE: It may be necessary to remove the build tree in the `BUILD` directory under the RPM root directory.

#### 4.2.4.2 Using the GDB configure Script Directly

This section describes how to configure the GNU debugger for RTEMS targets using `configure` and `make` directly. The following example illustrates the invocation of `configure` and `make` to build and install gdb-4.18 for the m68k-rtems target:

```
mkdir b-gdb
cd b-gdb
../gdb-4.18/configure --target=m68k-rtems \
  --prefix=/opt/rtems
make all
make info
make install
```

For some configurations, it is necessary to specify extra options to `configure` to enable and configure option components such as a processor simulator. The following is a list of configurations for which there are extra options:

**i960-rtems**
: `--enable-sim`

**powerpc-rtems**
: `--enable-sim --enable-sim-powerpc --enable-sim-timebase --enable-sim-hardware`

**sparc-rtems**
: `--enable-sim`

After gdb-4.18 is built and installed the build directory `b-gdb` may be removed.

For more information on the invocation of `configure`, please refer to the documentation for gdb-4.18 or invoke the gdb-4.18 `configure` command with the `--help` option.

#### 4.2.4.3 Using the bit_gdb Script

The simplest way to build gdb for RTEMS is to use the `bit_gdb` script. This script interprets the settings in the `user.cfg` file to produce the GDB configuration most appropriate for the target CPU. The variables in `user.cfg` were described in Section 4.1.4.3 [Using the bit Script], page 17 but only the `GDB` variable setting is used by `bit_gdb`.

The `bit_gdb` script is invoked as follows:

```
./bit_gdb CPU
```

Where CPU is one of the RTEMS supported CPU families from the following list:

- hppa1.1
- i386
- i386-coff
- i386-elf
- i960
- m68k
- m68k-coff
- mips64orion

- powerpc
- sh
- sh-elf
- sparc

If gdb supports a CPU instruction simulator for this configuration, then it is included in the build.

## 4.3 Common Problems

### 4.3.1 Error Message Indicates Invalid Option to Assembler

If a message like this is printed then the new cross compiler is most likely using the native assembler instead of the cross assembler or vice-versa (native compiler using new cross assembler). This can occur for one of the following reasons:

- Binutils Patch Improperly Applied
- Binutils Not Built
- Current Directory is in Your PATH

If you are using binutils 2.9.1 or newer with certain older versions of gcc, they do not agree on what the name of the newly generated cross assembler is. Older binutils called it `as.new` which became `as.new.exe` under Windows. This is not a valid file name, so `as.new` is now called `as-new`. By using the latest released tool versions and RTEMS patches, this problem will be avoided.

If binutils did not successfully build the cross assembler, then the new cross gcc (`xgcc`) used to build the libraries can not find it. Make sure the build of the binutils succeeded.

If you include the current directory in your PATH, then there is a chance that the native compiler will accidentally use the new cross assembler instead of the native one. This usually indicates that "." is before the standard system directories in your PATH. As a general rule, including "." in your PATH is a security risk and should be avoided. Remove "." from your PATH.

NOTE: In some environments, it may be difficult to remove "." completely from your PATH. In this case, make sure that "." is after the system directories containing "as" and "ld".

### 4.3.2 Error Messages Indicating Configuration Problems

If you see error messages like the following,

- cannot configure libiberty
- coff-emulation not found
- etc.

Then it is likely that one or more of your gnu tools is already configured locally in its source tree. You can check for this by searching for the `config.status` file in the various tool source trees. The following command does this for the binutils source:

```
find binutils-2.9.5.0.24 -name config.status -print
```

The solution for this is to execute the command `make distclean` in each of the GNU tools root source directory. This should remove all generated files including Makefiles.

This situation usually occurs when you have previously built the tool source for some non-RTEMS target. The generated configuration specific files are still in the source tree and the include path specified during the RTEMS build accidentally picks up the previous configuration. The include path used is something like this:

```
-I../../binutils-2.9.5.0.24/gcc -I/binutils-2.9.5.0.24/gcc/include -I.
```

Note that the tool source directory is searched before the build directory.

This situation can be avoided entirely by never using the source tree as the build directory – even for

# 5 Building RTEMS

## 5.1 Obtain the RTEMS Source Code

This section provides pointers to the RTEMS source code and Hello World example program. These files should be placed in your `archive` directory.

### RTEMS 4.5.0

```
FTP Site:    ftp.OARcorp.com
Directory:   /pub/rtems/releases/4.5.0
File:        rtems-4.5.0.tgz
```

### RTEMS Hello World

```
FTP Site:    ftp.OARcorp.com
Directory:   /pub/rtems/releases/4.5.0
File:        hello_world_c.tgz
```

## 5.2 Unarchive the RTEMS Source

Use the following command sequence to unpack the RTEMS source into the tools directory:

```
cd tools
tar xzf ../archive/rtems-4.5.0.tgz
```

This creates the directory rtems-4.5.0.

## 5.3 Add <INSTALL_POINT>/bin to Executable PATH

In order to compile RTEMS, you must have the cross compilation toolset in your search path. The following command appends the directory where the tools were installed prior to this point:

```
export PATH=$PATH:<INSTALL_POINT>/bin
```

NOTE: The above command is in Bourne shell (`sh`) syntax and should work with the Korn (`ksh`) and GNU Bourne Again Shell (`bash`). It will not work with the C Shell (`csh`) or derivatives of the C Shell.

## 5.4 Verifying the Operation of the Cross Toolset

In order to insure that the cross-compiler is invoking the correct subprograms (like `as` and `ld`), one can test assemble a small program. When in verbose mode, `gcc` prints out information showing where it found the subprograms it invokes. In a temporary working directory, place the following function in a file named `f.c`:

```
int f( int x )
{
  return x + 1;
}
```

Then assemble the file using a command similar to the following:

```
m68k-rtems-gcc -v -S f.c
```

Where `m68k-rtems-gcc` should be changed to match the installed name of your cross compiler. The result of this command will be a sequence of output showing where the cross-compiler searched for and found its subcomponents. Verify that these paths correspond to your <INSTALL_POINT>.

Look at the created file `f.s` and verify that it is in fact for your target processor.

Then try to compile the file `f.c` directly to object code using a command like the following:

```
m68k-rtems-gcc -v -c f.c
```

If this produces messages that indicate the assembly code is not valid, then it is likely that you have fallen victim to one of the problems described in Section 4.3.1 [Error Message Indicates Invalid Option to Assembler], page 24 Don't feel bad about this, one of the most common installation errors is for the cross-compiler not to be able to find the cross assembler and default to using the native `as`. This can result in very confusing error messages.

## 5.5 Building RTEMS for a Specific Target and BSP

This section describes how to configure and build RTEMS so that it is specifically tailored for your BSP and the CPU model it uses. There are two methods to compile and install RTEMS:

- direct invocation of `configure` and `make`
- using the `bit` script

Direct invocation of `configure` and `make` provides more control and easier recovery from problems when building.

This section describes how to build RTEMS.

### 5.5.1 Using the RTEMS configure Script Directly

Make a build directory under tools and build the RTEMS product in this directory. The ../rtems-4.5.0/configure command has numerous command line arguments. These arguments are discussed in detail in documentation that comes with the RTEMS distribution. If you followed the procedure described in the section Section 5.2 [Unarchive the RTEMS Source], page 27, these configuration options can be found in the file tools/rtems-4.5.0/README.configure.

**NOTE**: The GNAT/RTEMS run-time implementation is based on the POSIX API. Thus the RTEMS configuration for a GNAT/RTEMS environment MUST include the `--enable-posix` flag.

The following shows the command sequence required to configure, compile, and install RTEMS with the POSIX API, FreeBSD TCP/IP, and C++ support disabled. RTEMS will be built to target the `BOARD_SUPPORT_PACKAGE` board.

```
mkdir build-rtems
cd build-rtems
../rtems-4.5.0/configure --target=<TARGET_CONFIGURATION> \
    --disable-posix --disable-tcpip --disable-cxx \
    --enable-rtemsbsp=<BOARD_SUPPORT_PACKAGE>\
    --prefix=<INSTALL_POINT>
make all install
```

Where the list of currently supported <TARGET_CONFIGURATION>'s and <BOARD_SUPPORT_PACKAGE>'s can be found in tools/rtems-4.5.0/README.configure.

<INSTALL_POINT> is typically the installation point for the tools and is `/opt/rtems` when using prebuilt toolset executables.

NOTE: The `make` utility used should be GNU make.

### 5.5.2 Using the bit_rtems Script

If you have not previously unarchived the build tools, then you will need to unarchive the build scripts at this point if you plan to use `bit_rtems` to build RTEMS. If this is the case, you will have to execute the following additional command since you did not do it as part of building the tools.

```
cd tools
tar xzf ../archive/c_build_scripts-4.5.0.tgz
```

This script interprets the settings in the `user.cfg` file to enable or disable the various RTEMS options. The RTEMS specific entries described below must be set to tailor the RTEMS configuration to meet your application requirements:

`RTEMS`
: is the directory under tools that contains rtems-4.5.0.

`ENABLE_RTEMS_POSIX`
: is set to "yes" if you want to enable the RTEMS POSIX API support. At this time, this feature is not supported by the UNIX ports of RTEMS and is forced to "no" for those targets. This corresponds to the `configure` option `--enable-posix`.

`ENABLE_RTEMS_ITRON`
: is set to "yes" if you want to enable the RTEMS ITRON API support. At this time, this feature is not supported by the UNIX ports of RTEMS and is forced to "no" for those targets. This corresponds to the `configure` option `--enable-itron`.

`ENABLE_RTEMS_MP`
: is set to "yes" if you want to enable the RTEMS multiprocessing support. This feature is not supported by all RTEMS BSPs and is automatically forced to "no" for those BSPs. This corresponds to the `configure` option `--enable-multiprocessing`.

`ENABLE_RTEMS_CXX`
: is set to "yes" if you want to build the RTEMS C++ support including the C++ Wrapper for the Classic API. This corresponds to the `configure` option `--enable-cxx`.

`ENABLE_RTEMS_TESTS`
: is set to "yes" if you want to build the RTEMS Test Suite. If this is set to "no", then only the Sample Tests will be built. Setting this option to "yes" significantly increases the amount of disk space required to build RTEMS. This corresponds to the `configure` option `--enable-tests`.

`ENABLE_RTEMS_TCPIP`
: is set to "yes" if you want to build the RTEMS TCP/IP Stack. If a particular BSP does not support TCP/IP, then this feature is automatically disabled. This corresponds to the `configure` option `--enable-tcpip`.

`ENABLE_RTEMS_NONDEBUG`
: is set to "yes" if you want to build RTEMS in a fully optimized state. This corresponds to executing `make` after configuring the source tree.

`ENABLE_RTEMS_DEBUG`
: is set to "yes" if you want to build RTEMS in a debug version. When built for debug, RTEMS will include run-time code to perform consistency checks such as heap consistency checks. Although the precise compilation arguments are BSP dependent, the debug version of RTEMS is usually built at a lower optimization level. This is usually done to reduce inlining which can make tracing code execution difficult. This corresponds to executing `make VARIANT=debug` after configuring the source tree.

`INSTALL_RTEMS`
: is set to "yes" if you want to install RTEMS after building it. This corresponds to executing `make install` after configuring and building the source tree.

`ENABLE_RTEMS_MAINTAINER_MODE`
: is set to "yes" if you want to enabled maintainer mode functionality in the RTEMS Makefile. This is disabled by default and it is not expected that most users will want to enable this. When this option is enabled, the build process may attempt to regenerate files that require tools not required when this option is disabled. This corresponds to the `configure` option `--enable-maintainer-mode`.

After tailoring `user.cfg` for your application, the `bit_rtems` script may be invoked as follows:

```
./bit_rtems CPU [BSP]
```

Where CPU is one of the RTEMS supported CPU families from the following list:

- hppa1.1
- i386

- i386-coff
- i386-elf
- i960
- m68k
- m68k-coff
- mips64orion
- powerpc
- sh
- sh-elf
- sparc

BSP is a supported BSP for the selected CPU family. The list of supported BSPs may be found in the file tools/rtems-4.5.0/README.configure in the RTEMS source tree. If the BSP parameter is not specified, then all supported BSPs for the selected CPU family will be built.

**NOTE:** The POSIX API must be enabled to use GNAT/RTEMS.

# 6 Building the Sample Application

## 6.1 Unarchive the Sample Application

Use the following command to unarchive the sample application:

```
cd tools
tar xzf ../archive/hello_world_c.tgz
```

## 6.2 Set the Environment Variable RTEMS_MAKEFILE_PATH

RTEMS_MAKEFILE_PATH must point to the appropriate directory containing RTEMS build for our target and board support package combination.

```
export RTEMS_MAKEFILE_PATH=<INSTALLATION_POINT>/<BOARD_SUPPORT_PACKAGE>
```

Where <INSTALLATION_POINT> and <BOARD_SUPPORT_PACKAGE> are those used when configuring and installing RTEMS.

NOTE: In release 4.0, BSPs were installed at `<INSTALLATION_POINT>/rtems/<BOARD_SUPPORT_PACKAGE>`. This was changed to be more in compliance with GNU standards.

## 6.3 Build the Sample Application

Use the following command to start the build of the sample hello world application:

```
cd hello_world_c
make
```

NOTE: GNU make is the preferred `make` utility. Other `make` implementations may work but all testing is done with GNU make.

If no errors are detected during the sample application build, it is reasonable to assume that the build of the GNU C/C++ Cross Compiler Tools for RTEMS and RTEMS itself for the selected host and target combination was done properly.

## 6.4 Application Executable

If the sample application has successfully been built, then the application executable is placed in the following directory:

```
hello_world_c/o-optimize/<filename>.exe
```

How this executable is downloaded to the target board is very dependent on the BOARD_SUPPORT_PACKAGE selected. The following is a list of commonly used BSPs classified by their RTEMS CPU family and pointers to instructions on how to use them. [NOTE: All file names should be prepended with rtems-4.5.0/c/src/lib/libbsp.]

**i386/pc386** See `i386/pc386/HOWTO`

**i386/pc486** The i386/pc386 BSP specially compiled for an i486-class CPU.

**i386/pc586** The i386/pc386 BSP specially compiled for a Pentium-class CPU.

**i386/pc686** The i386/pc386 BSP specially compiled for a Pentium II.

**i386/pck6** The i386/pc386 BSP specially compiled for an AMD K6.

**m68k/gen68360** This BSP is for a MC68360 CPU. See `m68k/gen68360/README` for details.

**m68k/mvme162** See `m68k/mvme162/README`.

**m68k/mvme167** See `m68k/mvme167/README`.

**powerpc/mcp750** See `powerpc/motorola_shared/README`.

**powerpc/mvme230x** See `powerpc/motorola_shared/README.MVME2300`.

**powerpc/psim** This is a BSP for the PowerPC simulator included with `powerpc-rtems-gdb`. The simulator is complicated to initialize by hand. The user is referred to the script `powerpc/psim/tools/psim`.

**sparc/erc32** The ERC32 is a radiation hardened SPARC V7. This BSP can be used with either real ERC32 hardware or with the simulator included with `sparc-rtems-gdb`. An application can be run on the simulator by executing the following commands upon entering `sparc-rtems-gdb`:

```
target sim
load
run
```

RTEMS has many more BSPs and new BSPs for commercial boards and CPUs with on-CPU peripherals are generally welcomed.

## 6.5 More Information on RTEMS Application Makefiles

The hello world sample application is a simple example of an RTEMS application that uses the RTEMS Application Makefile system. This Makefile system simplifies building RTEMS applications by providing Makefile templates and capturing the configuration information used to build RTEMS specific to your BSP. Building an RTEMS application for different BSPs is as simple as switching the setting of `RTEMS_MAKEFILE_PATH`. This Makefile system is described in the file `rtems-4.5.0/make/README`.

# 7 Where To Go From Here

At this point, you should have successfully installed a GNU C/C++ Cross Compilation Tools for RTEMS on your host system as well as the RTEMS OS for the target host. You should have successfully linked the "hello world" program. You may even have downloaded the executable to that target and run it. What do you do next?

The answer is that it depends. You may be interested in writing an application that uses one of the multiple APIs supported by RTEMS. You may need to investigate the network or filesystem support in RTEMS. The common thread is that you are largely finished with this manual and ready to move on to others.

Whether or not you decide to dive in now and write application code or read some documentation first, this chapter is for you. The first section provides a quick roadmap of some of the RTEMS documentation. The next section provides a brief overview of the RTEMS application structure.

## 7.1 Documentation Overview

When writing RTEMS applications, you should find the following manuals useful because they define the calling interface to many of the services provided by RTEMS:

- **RTEMS Applications C User's Guide** describes the Classic RTEMS API based on the RTEID specification.
- **RTEMS POSIX API User's Guide** describes the RTEMS POSIX API that is based on the POSIX 1003.1b API.
- **RTEMS ITRON 3.0 API User's Guide** describes the RTEMS implementation of the ITRON 3.0 API.
- **RTEMS Network Supplement** provides information on the network services provided by RTEMS.

In addition, the following manuals from the GNU C/C++ Cross Compilation Toolset include information on run-time services available.

- **Cygnus C Support Library** describes the Standard C Library functionality provided by Newlib's libc.
- **Cygnus C Math Library** describes the Standard C Math Library functionality provided by Newlib's libm.

Finally, the RTEMS FAQ and mailing list archives are available at `http://www.oarcorp.com`.

There is a wealth of documentation available for RTEMS and the GNU tools supporting it. If you run into something that is not clear or missing, bring it to our attention.

Also, some of the RTEMS documentation is still under construction. If you would like to contribute to this effort, please contact the RTEMS Team at rtems-users@OARcorp.com. If you are interested in sponsoring the development of a new feature, BSP, device driver, port of an existing library, etc., please contact sales@OARcorp.com.

## 7.2 Writing an Application

From an application author's perspective, RTEMS applications do not start at `main()`. They begin execution at one or more application initialization task or thread and `main()` is owned by the Board Support Package. Each API supported by RTEMS (Classic, POSIX, and ITRON) allows the user to configure a set of tasks that are created and started automatically during RTEMS initialization. The RTEMS Automatic Configuration Generation (`confdefs.h`) scheme can be used to easily generate the configuration information for an application that starts with a single initialization task. By convention, unless overridden, the default name of the initialization task varies based up API.

- `Init` - single Classic API Initialization Task
- `POSIX_Init` - single POSIX API Initialization Thread
- `ITRON_Init` - single ITRON API Initialization Task

See the Configuring a System chapter in the C User's Guide for more details.

Regardless of the API used, when the initialization task executes, all non-networking device drivers are normally initialized and processor interrupts are enabled. The initialization task then goes about its business of performing application specific initialization. This often involves creating tasks and other system resources such as semaphores or message queues and allocating memory. In the RTEMS examples and tests, the file `init.c` usually contains the initialization task. Although not required, in most of the examples, the initialization task completes by deleting itself.

As you begin to write RTEMS application code, you may be confused by the range of alternatives. Supporting multiple tasking APIs can make the choices confusing. Many application groups writing new code choose one of the APIs as their primary API and only use services from the others if nothing comparable is in their preferred one. However, the support for multiple APIs is a powerful feature when integrating code from multiple sources. You can write new code using POSIX services and still use services written in terms of the other APIs. Moreover, by adding support for yet another API, one could provide the infrastructure required to migrate from a legacy RTOS with a non-standard API to an API like POSIX.

# Appendix A Using MS-Windows as a Development Host

This chapter discusses the installation of the GNU tool chain on a computer running the Microsoft Windows NT operating system.

## A.1 Cygwin 1.0 or Newer

Recent versions of Cygwin are vastly improved over the beta versions. Most of the oddities, instabilities, and performance problems have been resolved. The installation procedure is much simpler. However, there are a handful of issues that remain to successfully use Cygwin as an RTEMS development environment.

- There is no `cc` program by default. The GNU configure scripts used by RTEMS require this to be present to work properly. The solution is to link `gcc.exe` to `cc.exe`.
- Make sure you unarchive and build in a binary mounted filesystem (e.g. mounted with the `-b` option). Otherwise, many confusing errors will result.
- If you want to use RPM, you will have to obtain that separately by following the links from the main Cygwin site.
- When using the RPMs, there may be warnings about `/etc/mtab` while installing the info files. This can be ignored.
- A user has reported that they needed to set CYGWIN=ntsec for chmod to work correctly, but had to set CYGWIN=nontsec for compile to work properly (otherwise there were complaints about permissions on a temporary file).
- If you want to build the tools from source, you have the same options as UNIX users – `bit` or `RPM`.

## A.2 Cygwin B19

This section is based on a draft provided by Geoffroy Montel <g_montel@yahoo.com>. Geoffroy's procedure was based on information from David Fiddes <D.J@fiddes.surfaid.org>. Their input and feedback is greatly appreciated.

**STATUS:** This chapter should be considered preliminary. Please be careful when following these instructions.

This installation process works well under Windows NT. Using Windows 95 or 98 is not recommended although it should be possible with version 3.77 of GNU make and an updated cygwinb19.dll.

This procedure should also work with newer versions of the tool versions listed in this chapter, but this has not been verified.

### A.2.1 MS-Windows Host Specific Requirements

This section details the components required to install and build a Windows hosted GNU cross development toolset.

#### A.2.1.1 Unzipping Archives

You will have to uncompress many archives during this process. You must **NOT** use `WinZip` or `PKZip`. Instead the un-archiving process uses the GNU `zip` and `tar` programs as shown below:

```
tar -xzvf archive.tgz
```

`tar` is provided with Cygwin32.

#### A.2.1.2 Text Editor

You absolutely have to use a text editor which can save files with Unix format (so don't use Notepad nor Wordpad). There are a number of editors freely available that can be used.

- **VIM** (**Vi IMproved**) is available from **http://www.vim.org/**. This editor has the very handy ability to easily read and write files in either DOS or UNIX style.
- **GNU Emacs** is available for many platforms including MS-Windows. The official homepage is **http://www.gnu.org/software/emacs/emacs.html**. The GNU Emacs on Windows NT and Windows 95/98 FAQ is at **http://www.gnu.org/software/emacs/windows/ntemacs.html**.
- **PFE** (**Programmers File Editor**) may be downloaded from **http://www.simtel.net/pub/simtelnet/win95/editor/pfe101i.zip**. Note this editor is no longer actively supported.

#### A.2.1.3 Bug in Patch Utility

There is a bug in the `patch` utility provided in Cygwin32 B19. The files modified end up having MS-DOS style line termination. They must have Unix format, so a `dos2unix`-like command must be used to put them back into Unix format as shown below:

```
$ dos2unix XYZ
Dos2Unix: Cleaning file XYZ ...
```

The dos2unix utility may be downloaded from:

**ftp://ftp.micros.hensa.ac.uk/platforms/ibm-pc/ms-dos/simtelnet/txtutl/dos2unix.zip**

You **must** change the format of every patched file for the toolset build to work correctly.

#### A.2.1.4 Files Needed

This section lists the files required to build and install a Windows hosted GNU cross development toolset and their home WWW site. In addition to the sources required for the RTEMS cross environment listed earlier in this manual, you may need to download the following files from their respective sites using your favorite Web browser or ftp client. [NOTE: This information was current when B19 was released and URLs may no longer be correct.]

**cdk.exe** **http://www.cygnus.com/misc/gnu-win32/**

**coolview.tar.gz** **http://www.lexa.ru/sos/**

#### A.2.1.5 System Requirements

Although the finished cross-compiler is fairly easy on resources, building it can take a significant amount of processing power and disk space. The recommended build system spec is:

1. An AMD K6-300, Pentium II-300 or better processor. GNU C and Cygwin32 are **very** CPU hungry.
2. At least 64MB of RAM.
3. At least 400MB of FAT16 disk space or 250MB if you have an NTFS partition.

Even with this spec of machine expect the full suite to take over 2 hours to build with a further half an hour for RTEMS itself.

### A.2.2 Installing Cygwin32 B19

This section describes the process of installing the version B19 of the Cygwin32 environment. It assumes that this toolset is installed in a directory referred to as `<RTOS>`.

1. Execute cdk.exe. These instructions assume that you install Cygwin32 under the <RTOS>\cygnus\b19 directory.
2. Execute Cygwin.bat (either on the start menu or under <RTOS>\cygnus\b19).
3. At this point, you are at the command line of `bash`, a Unix-like shell. You have to mount the "/" directory. Type:
   ```
   umount /
   mount -b <RTOS> /
   ```
   For example, the following sequence mounts the `E:\unix` as the root directory for the Cygwin32 environment. Note the use of two \s when specifying DOS paths in bash:
   ```
   umount /
   mount -b e:\\unix /
   ```
4. Create the /bin, /tmp, /source and /build directories.
   ```
   mkdir /bin
   mkdir /tmp
   mkdir /source
   mkdir /build
   mkdir /build/binutils
   mkdir /build/gcc
   ```
5. The light Bourne shell provided with Cygwin B19 is buggy. You should copy it to a fake name and copy `bash.exe` to `sh.exe`:
   ```
   cd <RTOS>/cygnus/b19/H-i386-cygwin32/bin
   mv sh.exe old_sh.exe
   cp bash.exe sh.exe
   ```
   The Bourne shell has to be present in /bin directory to run shell scripts properly:
   ```
   cp <RTOS>/cygnus/b19/H-i386-cygwin32/bin/sh.exe /bin
   cp <RTOS>/cygnus/b19/H-i386-cygwin32/bin/bash.exe /bin
   ```

6. Open the file `/cygnus/b19/H-i386-cygwin32/lib/gcc-lib/i386-cygwin32/2.7-b19/specs`, and append

   ```
   -ladvapi32
   ```

   to the following line:

   ```
   -lcygwin %{mwindows:-luser32 -lgdi32 -lcomdlg32} -lkernel32
   ```

At this point, you have a native installation of Cygwin32 and are ready to proceed to building a cross-compiler.

### A.2.3 Installing binutils

1. Unarchive binutils-2.9.5.0.24.tar.gz following the instructions in Section 4.1.2 [Unarchiving the Tools], page 12 into the /source directory. Apply the appropriate RTEMS specific patch as detailed in Section 4.1.3 [Applying RTEMS Patches], page 13.
2. In the `/build/binutils` directory, execute the following command to configure binutils 2.9.5.0.24:

   ```
   /source/binutils-2.9.5.0.24/configure \
       --verbose --target=m68k-rtems \
       --prefix=/gcc-m68k-rtems --with-gnu-as --with-gnu-ld
   ```

   Replace `m68k-rtems` with the target configuration of your choice. See Section 4.1.4.3 [Using the bit Script], page 17 for a list of the targets available.
3. Execute the following command to compile the toolset:

   ```
   make
   ```

4. Install the full package with the following command:

   ```
   make -k install
   ```

   There is a problem with the gnu info package which will cause an error during installation. Telling make to keep going with `-k` allows the install to complete.
5. In the `cygnus.bat` file, add the directory containing the cross-compiler executables to your search path by inserting the following line:

   ```
   PATH=E:\unix\gcc-m68k-rtems\bin;%PATH%
   ```

6. You can erase the /build/binutils directory content if disk space is tight.
7. Exit bash and run `cygnus.bat` to restart the Cygwin32 environment with the new path.

### A.2.4 Installing GCC AND NEWLIB

1. Unarchive and patch gcc-2.95.2.tar.gz and newlib-1.8.2.tar.gz following the instructions in Section 4.1.2 [Unarchiving the Tools], page 12. Apply the appropriate RTEMS specific patches as detailed in Section 4.1.3 [Applying RTEMS Patches], page 13.

   **NOTE**: See Section A.2.1.3 [Bug in Patch Utility], page 38.
2. Remove the following directories (we cannot use Fortran or Objective-C as Cygwin32 cross-compilers):

```
/source/gcc-2.95.2/libf2c
/source/gcc-2.95.2/gcc/objc
/source/gcc-2.95.2/gcc/f
```

**NOTE**: See Section A.2.1.3 [Bug in Patch Utility], page 38.

3. Link the following directories from Newlib to the main GCC directory, /source/gcc-2.95.2/ :
   - ln -s ../newlib-1.8.2/newlib newlib
   - ln -s ../newlib-1.8.2/libgloss libgloss
4. Change to the /build/gcc directory to configure the compiler:

```
/source/gcc-2.95.2/configure \
    --verbose --target=m68k-rtems \
    --prefix=/gcc-m68k --with-gnu-as --with-gnu-ld \
    --with-newlib
```

   Replace `m68k-rtems` with the target configuration of your choice. See Section 4.1.4.3 [Using the bit Script], page 17 for a list of the targets available.
5. Compile the toolset as follows:

```
make cross
```

   You must do a `make cross` (not a simple `make`) to insure that the different packages are built in the correct order. Making the compiler can take several hours even on fairly fast machines, beware.
6. Install with the following command:

```
make -k install
```

7. Just as with binutils package, a problem with the gnu info package not building correctly requires that you use -k to keep going.

With any luck, at this point you having a working cross-compiler. So as Geoffroy said:

**That's it! Celebrate!**

# Table of Contents