BlueCat Linux User's Guide

BlueCat Linux Release 5.4.x

DOC-0906-00

Product names mentioned in the *BlueCat Linux User's Guide* are trademarks of their respective manufacturers and are used here for identification purposes only.

Printed in the United States of America.

Contents

Preface

For More Information

For more information on the features of BlueCat Linux, refer to the following online documentation:

- The complete BlueCat Linux documentation set is available on the BlueCat Linux Documentation CD-ROM. Books are provided in both HTML and PDF formats.

  Updates to these documents are available online at the LynuxWorks Website: `http://www.lynuxworks.com`.

- Additional information about commands and utilities is provided online with the `man` command. For example, to find information about the GNU GCC compiler, use the following syntax:

  ```
  man gcc
  ```

Typographical Conventions

The typefaces used in this manual, summarized below, emphasize important concepts. All references to filenames and commands are case-sensitive and should be typed accurately.

Kind of Text	Examples
Body text; *italicized* for emphasis, new terms, and book titles	Refer to the *BlueCat Linux User's Guide*.

Environment variables, filenames, functions, methods, options, parameter names, path names, commands, and computer data	`ls`
	`-l`
	`myprog.c`
	`/dev/null`

Commands that need to be highlighted within body text, or commands that must be typed as is by the user are **`bolded`**.	`login:` **`myname`** `#` **`cd /usr/home`**
Text that represents a variable, such as a filename or a value that must be entered by the user, is *italicized*.	`cat` *`<filename>`* `mv` *`<file1>`* *`<file2>`*
Blocks of text that appear on the display screen after entering instructions or commands	`Linux version 2.4.10-1 (bin@build1) (gcc version 2.95.3 20010315 (release)) #5 Tue Dec 18 13:33:08 MSK 2001 Processor: Intel StrongARM-IXP1200 revision 3 Architecture: Intel IXP1200 On node 0 totalpages: 32768 zone(0): 32768 pages. zone(1): 0 pages. zone(2): 0 pages.`
Keyboard options, button names, and menu sequences	**Enter**, **Ctrl-C**

Special Notes

The following notations highlight any key points and cautionary notes that may appear in this manual.

NOTE: These callouts note important or useful points in the text.

CAUTION! Used for situations that present minor hazards that may interfere with or threaten equipment/performance.

Technical Support

LynuxWorks Support handles support requests from current support subscribers.

For questions regarding LynuxWorks products or evaluation CDs, or to become a support subscriber, our knowledgeable sales staff will be pleased to help you (`http://www.lynuxworks.com/corporate/contact/sales.php3`).

How to Submit a Support Request

When you are ready to submit a support request, please include *all* the following information:

- First name
- Last name
- Your job title
- Phone number
- Fax number
- E-mail address
- Company name
- Address
- City, state, ZIP

- Country
- LynxOS or BlueCat Linux version you are using
- Target platform (for example, PowerPC or x86)
- Board Support Package (BSP)
- Current patch revision level
- Development host OS version
- Description of problem you are experiencing

Where to Submit a Support Request

By E-mail:

Support, Europe	`tech_europe@lnxw.com`
Support, worldwide except Europe	`support@lnxw.com`
Training and courses	USA: `training-usa@lnxw.com` Europe: `training-europe@lnxw.com`

By Phone:

Training and courses	USA: +1 408-979-4353 Europe: +33 1 30 85 06 00
Support, Europe (from our Paris, France office)	+33 1 30 85 93 96
Support, worldwide except Europe and Japan (from our San José, CA, USA headquarters)	+1 800-327-5969 or +1 408-979-3940
Support, Japan	+81 33 449 3131

By Fax:

Support, Europe (from our Paris, France office)	+33 1 30 85 06 06
Support, worldwide except Europe and Japan (from our San José, CA, USA headquarters)	+1 408-979-3945
Support, Japan	+81 22 449 3803

CHAPTER 1 *Introduction and Installation*

This chapter introduces BlueCat Linux and describes the installation procedure.

BlueCat Linux is the LynuxWorks distribution of open source Linux (based on Linux kernel 2.6.13) tailored for *embedded systems* in a cross-development environment. BlueCat Linux supports the creation of embedded applications, kernels, and device drivers meant for deployment on target systems such as telecommunication and networking products, military equipment, medical equipment, and high-volume printers and copiers.

The subset of Linux and LynuxWorks *development and embedding tools* included in BlueCat Linux includes a nonproprietary open-source kernel, a complete suite of GNU development tools, C and C++ compilers, debuggers, profilers, libraries, and tools to build software images for the embedded system. A kernel debugger supports the development of custom drivers on the target system. Users can also create a bootable disk or ROM with the LynuxWorks utilities included.

BlueCat Linux also comes with a set of *demonstration systems* that include specially configured kernels and sample applications that demonstrate networking, debugging, and other functionalities.

Cross-development is defined as a development paradigm where applications are developed and debugged on a system (*host*), other than the board/platform (*target*) where they are deployed.

Cross-development is required and optimal in situations where targets do not have the necessary development resources. The building and debugging of applications is conducted mainly on a host, connected to a target via a network or serial port.

A target is a combination of hardware (computer and input/output devices) and software (operating system and applications) that is intended to perform a specific function.

BlueCat Linux supports cross-development on Windows and Linux hosts.

Overview

The important features of the BlueCat Linux development platform are described below:

- BlueCat Linux is a cross-development product. It allows for software development on a host and provides the necessary tools for transferring software to the target board.
- A set of standard CD-ROMs is the distribution medium for installing BlueCat Linux.
- Installation is easy. The installation program installs the minimal product configuration onto the host, utilizing the Red Hat Linux Package Manager (RPM). This simplifies installing and uninstalling packages.
- BlueCat Linux can be installed and used by anyone logged in on the cross-development host with permission to mount a CD-ROM.
- BlueCat Linux supports multiple independent installations on a single host system.
- BlueCat Linux coexists with native host tools and features. The installed BlueCat Linux is activated by executing a shell script that sets up the execution environment.

System Requirements

Installation is carried out on a preinstalled, fully operational cross-development host. The host system can be either Linux (Intel IA-32 or x86 PC compatible running Red Hat Enterprise Linux 4.0) or Windows (Intel IA-32 or x86 PC compatible running Windows XP Professional).

System requirements for the cross-development host are as follows:

- Standard set of Linux or Windows utilities. Please note that the `bash` shell must be used on the cross-development host with BlueCat Linux.
- Free disk space needed for installing BlueCat Linux on Linux/Windows hosts for supported target boards.
- CD-ROM drive.

NOTE: BlueCat Linux installation instructions and examples in this guide are based on x86 target boards. For detailed information regarding non-x86 target boards, please consult the appropriate *Board Support Guide*.

Distribution CD-ROMs

The BlueCat Linux distribution is distributed on the following types of CD-ROMs:

1. The *Installation CD-ROMs*—Installation of BlueCat Linux support for a target board requires two Installation CD-ROMs: A *Binary Architecture CD-ROM* (for a specific microprocessor family), and a *Board Support Package CD-ROM* (binary and source for a specific target).

   - The *Binary Architecture CD-ROM* contains the common binary files for all supported boards necessary for development on a specific microprocessor family (for example, the x86 microprocessor family). There is one *Binary Architecture CD-ROM* per family.
   - The *Board Support Package (BSP) CD-ROM* contains both BlueCat Linux binary as well as source files used to support development on a specific target board. A BSP CD-ROM is installed after installing the core components.

   Both these CD-ROMs install BlueCat Linux binaries on the host.

2. The *Source Architecture CD-ROM*—This CD-ROM is installed on the cross-development host and contains the source files required to rebuild the binaries, in case retrieval is needed. Source files also allow the developer the flexibility to customize the RPM packages.

3. *Documentation CD-ROM*—This CD-ROM contains the *BlueCat Linux User's Guide* and *Board Support Guides* (each Board Support Package, or *BSP*, is accompanied by its own Guide) for supported targets in PDF and HTML formats.

Binary Architecture CD-ROM Structure

The Binary Architecture CD-ROM directories are organized as shown in the following figure:

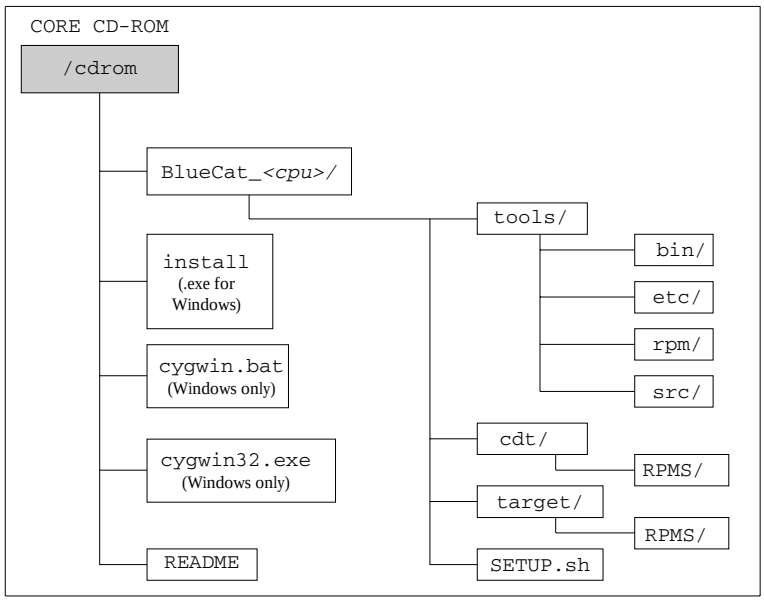

Figure 1-1: Binary Architecture CD-ROM Structure

NOTE: In the `BlueCat_<cpu>` directory, `<cpu>` implies a target architecture family, for example, `i386` for x86 boards.

Binary Architecture CD-ROM Directories and Components

The BlueCat Linux Binary Architecture CD-ROM contains the following main directories under `BlueCat_<cpu>/`:

- `tools`—Directory containing installation tools needed during the installation process, including all required libraries and data files. The key utility used for installation is the BlueCat Linux `rpm` utility, which provides user-level package database management and file extraction relative to the installation point.

- `cdt`—Directory containing binary packages of the cross-development tools that run on the cross-development host
- `target`—Directory containing binary packages built to run on a target board

The table below briefly describes all the components of the BlueCat Linux Binary Architecture CD-ROM.

Table 1-1: Binary Architecture CD-ROM Components

Node	Description
`/mnt/cdrom`	Mount point (typical)
`- BlueCat_<cpu>/`	BlueCat Linux tools and packages for a specific CPU architecture (for example, `BlueCat_i386`)
`-- tools/`	Installation tools
`--- bin/`	Binary installation files
`--- etc/`	Configuration files needed for installation
`--- rpm/`	RPM-specific files needed for installation
`--- src/`	Source files of the installation tools
`-- cdt/`	Cross-development packages
`--- RPMS/`	Cross-development binary packages
`-- target/`	Directory in which Board Support Packages are installed
`--- RPMS/`	Target board binary packages directory
`-- SETUP.sh`	Shell script for setting up the environment
`- install (.exe)`	Installation program (`install.exe` for Windows host)
`- README`	README file
`- cygwin.bat`	Windows host installation script (for Windows host only)
`- cygwin32.exe`	Windows host installation script (for Windows host only)

Board Support Package (BSP) CD-ROM Structure

The Board Support Package CD-ROM directories are organized as shown in the following figure:

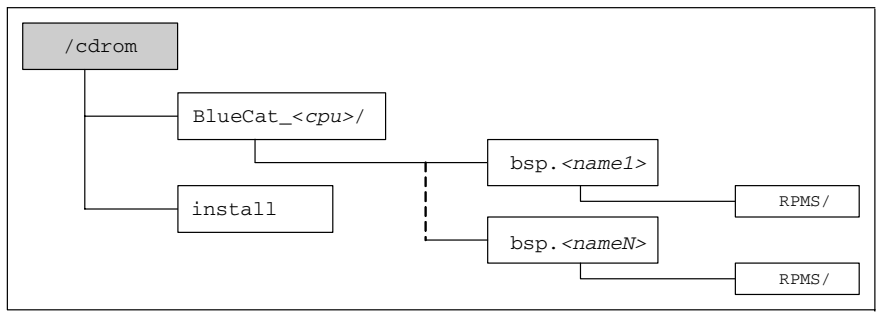

Figure 1-2: Board Support Package CD-ROM Tree

BSP CD-ROM Directories and Components

The BlueCat Linux Board Support Package CD-ROM has one or more directories `bsp.<name1>`, `...`, `bsp.<nameN>`, each for a specific board. Note that the accompanying *Board Support Guide* is on the Documentation CD-ROM.

- `bsp.<nameN>`—Directory containing target board-specific binary and source packages, where `<nameN>` is the name of the target board (for example, x86).

The table below briefly describes the components of the BlueCat Linux Board Support Package CD-ROM.

Table 1-2: Board Support Package CD-ROM Components

Node	Description
`/mnt/cdrom`	Mount point (typical)
`- BlueCat_<cpu>/`	BlueCat Linux tools and packages for a specific CPU architecture (for example, `BlueCat_i386` for x86)
`-- bsp.<nameN>/`	BlueCat Linux packages for a specific target board, where `<nameN>` is its Board Support Package

Table 1-2: Board Support Package CD-ROM Components (Continued)

Node	Description
`--- RPMS/`	Target board-specific packages
`- install`	Installation program

Source Architecture CD-ROM Structure

The Source Architecture CD-ROM includes directories organized as shown in the figure below:

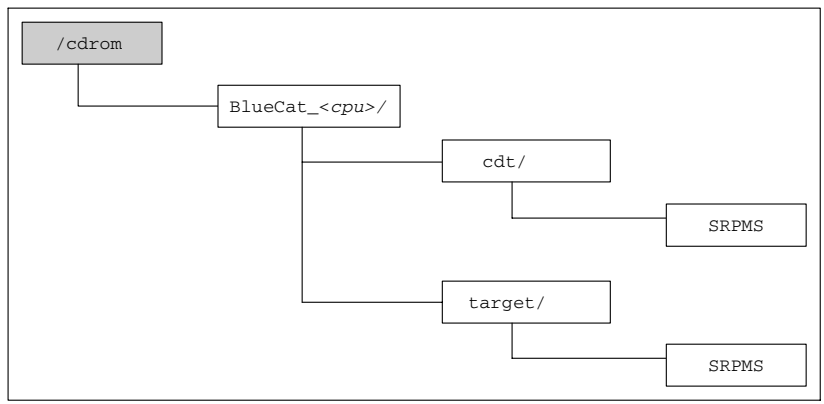

Figure 1-3: Source Architecture CD-ROM Tree

Source Architecture CD-ROM Directories and Components

The BlueCat Linux Source Architecture CD-ROM includes two main directories:

- `cdt`—Directory containing sources of RPM packages of the cross-development tools that run on the host
- `target`—Directory containing sources of the RPM packages configured to run on a target board

The following table briefly describes the components of the Source Architecture CD-ROM.

Table 1-3: Source Architecture CD-ROM Components

Node	Description
`/mnt/cdrom`	Mount point (typical)
`- BlueCat_<cpu>/`	BlueCat Linux tools and packages for a specific CPU architecture (for example, `BlueCat_i386` for x86)
`-- cdt/`	Cross-development packages directory
`--- SRPMS/`	Sources of cross-development packages
`-- target/`	Target board packages directory
`--- SRPMS/`	Sources of target board packages

Installation Procedure

Installing the Default Configuration

To install the BlueCat Linux core components on the cross-development host, perform the following steps:

1. Select a cross-development host type:

   - Linux

     Insert the *Binary Architecture CD-ROM* into the drive and mount it on the Linux host. Mounting the CD-ROM may require logging in as `root` (superuser). As all the remaining steps of the installation procedure can be performed under a regular user account, the user is advised to do so.

     ```
     # mount -r /dev/cdrom /mnt/cdrom
     ```

     Please note that some desktops such as Red Hat Linux KDE automount the CD-ROM. Please make sure that the CD-ROM is mounted with the options `exec` and `suid` to permit execution of binaries, and allow the `set-user-identifier` bit to take effect,

respectively. For instance, on a Red Hat Linux host, log in as `root` and modify `/etc/fstab` to contain the following line:

```
/dev/cdrom /mnt/cdrom iso9660 -noauto,user,ro,exec,suid 0 0
```

Alternatively, simply disable the automount feature and mount the CD-ROM manually.

- Windows

  Insert the *Binary Architecture* CD-ROM into the CD-ROM drive and run the `cygwin.bat` installation script from the CD-ROM directory. This script installs the Cygwin execution environment on drive `C:` of the Windows host. To install Cygwin on another drive, perform the following steps:

  - From the **Start** menu, run the MS-DOS prompt.
  - Change to the CD-ROM drive. For example, if the CD-ROM drive letter is `D`, use the following command:

    ```
    C:\WINDOWS> d:
    ```

  - Run the installation script using the letter of the local drive as a parameter. The following command runs `cygwin.bat` to install Cygwin on disk F:

    ```
    D:\> cygwin.bat f:
    ```

  Upon completion of this script a `bash` window appears. All of the following steps are to be performed in the `bash` window.

2. Go to the directory where BlueCat Linux is to be installed. This directory must be empty. For instance, enter:

   ```
   $ cd $HOME
   $ mkdir BlueCat
   $ cd BlueCat
   ```

3. Run the BlueCat Linux core components installation program by typing:

   ```
   $ /mnt/cdrom/install
   ```

   This program performs all the necessary installation steps:

   - Installs the minimal set of common components for all the boards supported by the family, including cross-development tools, target board tools and libraries, and the source tree of the BlueCat Linux kernel.
   - Sets correct permissions for the installed binaries.

The path of the installation directory to the `install` program may also be specified by typing:

```
$ cd $HOME
$ mkdir BlueCat
$ /mnt/cdrom/install BlueCat
```

When the `install` program has finished, the installation procedure is complete.

NOTE: If installing BlueCat Linux on an NFS-mounted disk, make sure to enable the NFS locking daemon on the NFS server.

Installing Target Board Support

Use the following procedure for installing support for a number (one or more) of target boards on a cross-development host. On a Windows host, all of the following steps are to be performed in the `bash` window. (Refer to the section entitled “Windows” under “Installing the Default Configuration” on page 8.)

1. Insert the Board Support Package CD-ROM containing BlueCat Linux for one or more boards in a microprocessor family and mount the CD-ROM. To mount the CD-ROM on a Linux host, refer to the section entitled “Linux” under “Installing the Default Configuration” on page 8.
2. From the top of the directory where the core components of BlueCat Linux have been installed, set up the core BlueCat Linux development environment by typing:

   ```
   $ . SETUP.sh
   ```

3. Install the BlueCat Linux Board Support Package by typing:

   BlueCat:$ **/mnt/cdrom/install** *<bsp>*

   where *<bsp>* is the specific Board Support Package to be installed. (For example, `x86` for i386 target boards.)

   This program performs all the necessary steps to install all target board-specific components onto the cross-development host, including kernel files and demo systems.

NOTE: For the name of a specific BSP, please consult the accompanying *Board Support Guide*.

---

4. Optionally, repeat the previous step to install another BSP from the Board Support Package CD-ROM.

Installation of support for a board can be done at any time. The user may install and use support for a family core and a number of boards, and then install support for a new board from a separate Board Support Package CD-ROM. Only the previous procedure needs to be performed to install target board support on top of a preinstalled core.

Activating Support for a Target Board

Installation procedures described in the preceding sections install the BlueCat Linux core and Board Support Package for a target board. To activate the package for a target board, proceed as follows:

1. From the top of the directory where BlueCat Linux is installed, run:

   ```
   $ . SETUP.sh <bsp>
   ```

   where *<bsp>* refers to the Board Support Package for a specific board.

   This script sets up all the environment variables necessary to activate the cross-development environment and tools supported by the core (if not already set), and sets all the environment variables necessary to activate support for a specified target board.

NOTE: Support for only one target board can be active at a time from a single installation. In other words, it is impossible to activate more than one target board at once (for instance, from different user sessions) even after installation of support for more than one target board has been performed.

Installing Sources of BlueCat Linux RPM Packages

The *Source Architecture* CD-ROM can be used to install the sources of prebuilt BlueCat Linux RPM packages onto the cross-development host. Upon successful installation of a Source RPM (SRPM) onto the cross-development host, the package can be rebuilt, thus providing for a fully reproducible build process.

There is one *Source Architecture* CD-ROM per a microprocessor family. This CD-ROM contains all of the files required to support installation and building of the sources for all target boards supported within the microprocessor family. The installation and build of a source RPM must be in the context of the BlueCat Linux execution environment. `SETUP.sh`, sourced at the top of the BlueCat Linux installation directory, sets up BlueCat Linux environment variables which enables the build procedure to determine the target board.

The following demonstrates rebuilding the sources for the `sed` RPM package.

1. Assuming that the Source Architecture CD-ROM is mounted at `/mnt/cdrom`, type the following to install sources on the host:

   ```
   BlueCat:$ rpm -i /mnt/cdrom/BlueCat_<cpu>/\
   target/SRPMS/sed_trg-<version>.src.rpm
   ```

   where `BlueCat_<cpu>` is the target board CPU (i386 for x86 boards) and `<version>` is the current version of the RPM.

   Upon completion of the command, the tar (compressed image) file with the sources of the `sed` RPM package (`sed-<version>.tar.gz`) can be found in the directory `$BLUECAT_PREFIX/cdt/src/bluecat/SOURCES`. For those RPM packages that patch their tar files, the same directory contains appropriate patch files.

   The RPM specification file (`sed_trg.spec`) is placed in the directory `$BLUECAT_PREFIX/cdt/src/bluecat/SPECS`.

2. Use the `.spec` file to unpack the tar file and install the patches, if any:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/cdt/src/bluecat/SPECS
   BlueCat:$ rpmbuild -bp sed_trg.spec
   ```

   Successful completion of the command creates a source files tree for the `sed` package in `$BLUECAT_PREFIX/cdt/src/bluecat/BUILD`.

3. Use this tree to rebuild the package from the sources. For example:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/cdt/src/bluecat/SPECS
   BlueCat:$ rpmbuild -ba sed_trg.spec
   ```

This places the rebuilt RPM package (`sed_trg-<version>.<cpu>.rpm`) in the `$BLUECAT_PREFIX/cdt/src/bluecat/RPMS/<cpu>` directory.

4. Reinstall this package in the BlueCat Linux execution environment with:

```
BlueCat:$ rpm -i --force
$BLUECAT_PREFIX/cdt/src/bluecat/RPMS/<cpu>/\
sed_trg-<version>.<cpu>.rpm
```

NOTE: Rebuild of certain RPM packages may require installation of appropriate optional BlueCat Linux packages.

NOTE: As the BlueCat Linux execution environment is designed for building target packages, it is impossible to build cross-development tools in the context of the BlueCat Linux execution environment.

Using the BlueCat Linux rpm Utility

The BlueCat Linux installation procedure is based on the BlueCat Linux `rpm` utility included on the CD-ROM. This version of `rpm` provides the standard functionality of the RPM facility, except that all operations are relative to the BlueCat Linux installation directory.

The BlueCat Linux `rpm` is totally independent of any RPM facility installed on the cross-development host. For example, the query function of BlueCat Linux `rpm` shows information only for the RPM packages installed using BlueCat Linux installation tools.

NOTE: To activate BlueCat Linux `rpm`, the BlueCat Linux environment must first be set up. This is done by sourcing the `SETUP.sh` script at the top of the BlueCat Linux installation directory.

NOTE: The `bash` shell must be used on the cross-development host with BlueCat Linux.

Use BlueCat Linux `rpm` as appropriate to manage the BlueCat Linux packages. For instance, the following command shows all BlueCat Linux packages installed on the cross-development host:

```
BlueCat:$ rpm -qa
```

Adding a New RPM Package to BlueCat Linux

Many utilities for the various flavors of UNIX and Linux come as archives of source files packed as SRPM files. The same utilities may be "built" to run on different target machines, and this saves the author of the software from having to produce multiple versions. The built versions are packed in the RPM files.

To add a new package to the installed BlueCat Linux environment, one of the two approaches can be chosen:

- Install the RPM package built by someone else for the specified target.
- Install the SRPM package, build the binary RPM package in the BlueCat Linux environment, and install the resultant binary RPM package into BlueCat Linux.

One can typically find both prebuilt RPM packages and SRPM archives in the public domain. Use the `www.rpmfind.org` web site to get access to the RPM database.

How to Install a Prebuilt RPM Package

To include a prebuilt RPM package to the installed BlueCat Linux environment, perform the following steps:

1. From the top of the directory in which BlueCat Linux is installed, enter:

   ```
   bash$ . SETUP.sh
   ```

2. Install the package:

   ```
   BlueCat:$ rpm -i <rpm_name>
   ```

   This step can fail due to the following reasons:

   - Before installing the package, the `rpm` command checks whether all the packages required for the correct installation of the package are already installed. If not, the `rpm` command exits without doing anything. Typically, the check fails because the names of packages in the BlueCat Linux cross environment differ from the names used in other Linux distributions, such as Red Hat Linux. For example, the package named `nfs-utils` in Red Hat Linux is named `nfs-utils_trg` in BlueCat Linux. The user can get over this by specifying the `nodeps` option to the `rpm` command:

     ```
     BlueCat:$ rpm -i --nodeps <rpm_name>
     ```

If this option is specified, the `rpm` command does not check the dependencies of the package being installed. In this case, the user becomes responsible for satisfying dependencies and should install all the necessary packages beforehand.

The package contains a pre- or postinstallation script which calls the Linux host commands that works with host directories rather than the BlueCat Linux directories. The user can avoid execution of such scripts by passing the `--noscripts` option to the `rpm` command:

```
BlueCat:$ rpm -i --noscripts <rpm_name>
```

In this case the user should perform the commands manually, replacing the references to the Linux host directories with appropriate references to the corresponding BlueCat Linux directories. For example, if the post installation script for the `zebra` package contains the following strings:

```
if [ ! -e %{_sysconfdir}/vtysh.conf ]; then
       touch %{_sysconfdir}/vtysh.conf
       chmod 640 %{_sysconfdir}/vtysh.conf
fi
```

the user should perform the similar commands on the corresponding target file:

```
if [ ! -e $BLUECAT_PREFIX%{_sysconfdir}/vtysh.conf ]; then
        touch $BLUECAT_PREFIX%{_sysconfdir}/vtysh.conf
        chmod 640 $BLUECAT_PREFIX%{_sysconfdir}/vtysh.conf
fi
```

Sometimes a host command should be changed to a command from the BlueCat Linux cross environment For example, the command:

```
BlueCat:$ /sbin/ldconfig
```

should be changed to:

```
$BLUECAT_PREFIX/cdt/sbin/ldconfig
```

How to Build and Install an SRPM Package

To build and install an SRPM package, perform the following steps:

1. From the top of the directory in which BlueCat Linux is installed, enter:

   ```
   bash$ . SETUP.sh
   ```

2. Install the RPM sources:

   ```
   BlueCat:$ rpm -i <srpm_name>
   ```

As a result, the RPM sources will be put in the `$BLUECAT_PREFIX/cdt/src/bluecat/SOURCES` directory and the RPM specification file will be put to the `$BLUECAT_PREFIX/cdt/src/bluecat/SPECS` directory as `<rpm_name>.spec`.

3. To build the RPM from the sources, enter the following command:

```
BlueCat:$ rpmbuild -ba \
$BLUECAT_PREFIX/cdt/src/bluecat/\
SPECS/<rpm_name>.spec
```

If it succeeds, the binary RPM package will be put to the `$BLUECAT_PREFIX/cdt/src/bluecat/RPMS/<target>` directory as `<rpm_name>-<rpm_version>-<rpm_release>.<target>.rpm`.

This step, however, is rarely so simple as it seems. Almost all specification files are written to build an RPM package in a Linux native environment. They can contain target commands that can not be performed in a cross environment such as BlueCat Linux. Thus, before performing this step, it is recommended to do the following trial steps:

i. Unpack the sources in the build tree by the command:

```
BlueCat:$ rpmbuild -bp \
$BLUECAT_PREFIX/cdt/src/bluecat/SPECS/\
<rpm_name>.spec
```

As a result, the `$BLUECAT_PREFIX/cdt/src/bluecat/BUILD/<rpm_name>` directory appears. It is the actual build tree with all patches applied. This command does not build anything. It only unpacks the sources and applies the necessary patches in the right order.

This step can fail due to the following reasons:

- The target architecture is not included in the `Exclusivearch` tag if it is present in the specification file. Add the target architecture and try again.
- The `BuildRequires` tag refers to a package that is absent in the BlueCat Linux environment or to a required version that is higher than what is available in BlueCat Linux. In this case, there are the following two choices:
  - Comment this tag out and try to build the package nevertheless.
  - Build and install the required packages and then return to building the ultimately required one.

ii. After the build tree is unpacked, the user can try to configure the package. To do this, change to the build directory:

```
BlueCat:$ cd \
$BLUECAT_PREFIX/cdt/src/bluecat/BUILD/<rpm_name>
```

and perform the following command:

```
BlueCat:$ ./configure <options>
```

where `<options>` are options of the configure script specified in the specification file `<rpm_name>.spec`. This step can fail for several reasons. The main reasons are the following:

- The configuration script complains about “unknown architecture.” The required architecture should be added to the list of the recognizable architectures (typically specified in the `config.guess`, `configure.in`, or `ltconfig` files).
- The configuration script checks for a system parameter, compiling a test program and then trying to execute it. The needed parameter should be passed to the configuration script. For example, while building the BlueCat Linux `findutiks_trg` package, the configuration scripts checks for the `setvbuf()` function arguments. To avoid this check in the BlueCat Linux cross environment, the specification file calls the configuration script as:

  ```
  ac_cv_func_setvbuf_reversed=no ./configure <options>
  ```

  Unfortunately, this procedure cannot be automated and requires looking through the configuration script.
- The configuration script complains that a library or a header file required by the package is absent. This can occur if the configuration script searches for it in the host directories when it should scan the BlueCat Linux target directories instead. In this case, the configuration script should be changed. For example, the string

  ```
  if [ ! -f /usr/include/stdio.h ]
  ```

  should be replaced by the string:

  ```
  if [ ! -f $BLUECAT_PREFIX/usr/include/stdio.h ]
  ```

  It is possible, of course, that the required library or header file is absent in the BlueCat Linux target directories. In this case, the RPM package that contains this object should be built and

installed beforehand. To determine which package contains a required object, the user can find the object on any Linux machine and perform the following command:

```
BlueCat:$ rpm -qf <path>
```

where `<path>` is a full name of the required file.

iii. After the configuration script has succeeded, the user can try to `make` the package. Enter the command:

```
BlueCat:$ make <options>
```

where `<options>` are the options present in the specification file of the package. This command can also fail. The following are the most frequent reasons for a failure:

- Compilation of a file fails because the `-I/usr/include/<dir>` option is passed to the `gcc` command. Check the corresponding

  Makefile. If it contains the string:

  ```
  INCLUDE=-I/usr/include/<dir>
  ```

  pass the proper `INCLUDE` variable to the `make` command as follows:

  ```
  BlueCat:$ make INCLUDE=$BLUECAT_PREFIX/usr/\
  include/<dir> <options>
  ```

- The `ld` linker fails to build an executable because the `-L/usr/include/<dir>` option is passed to it. Check the corresponding Makefile. If it contains the string:

  ```
  LIB=-I/usr/include/<dir>
  ```

  pass the proper `LIB` variable to the `make` command as follows:

  ```
  BlueCat:$ make LIB=$BLUECAT_PREFIX/usr/\
  include/<dir> <options>
  ```

- During the build, an auxiliary program is created that should generate some additional files or sources. Sometimes the special `BUILD_CC` variable is present in the Makefile which specifies the compiler to build such programs. In this case, call the `make` command with this variable set properly:

  ```
  BlueCat:$ make BUILD_CC=bluecat_native_gcc \
  <options>
  ```

  Sometimes such a variable is absent. In this case, the Makefile itself might have to be changed.

4. After the `make` command has succeeded and the necessary changes are inserted in the specification file, the user can return to the `rpmbuild -ba` command.

   Note that changes to the source files can be done in two ways:

   - A new patch file can be created. To do this, perform the following steps:

     a. Save the original build tree as *<rpm_name>*`.orig`:

        ```
        BlueCat:$ cp -dpR <rpm_name> <rpm_name>.orig
        ```

     b. Make the necessary changes to the files in the *<rpm_name>* build tree.

     c. Create a patch file as follows:

        ```
        BlueCat:$ diff -Naur <rpm_name>.orig \
        <rpm_name> >$BLUECAT_PREFIX/cdt/src/bluecat/\
        SOURCES/<rpm_name>-<date>.patch
        ```

     d. Add the following string to the specification file header:

        ```
        Patch<num>: <rpm_name>-<date>.patch
        ```

        where *<num>* is an increase of one number above the last patch present in the specification file. If the specification file contains no patch strings, the *<num>* component is 0 and the string is put after all the `Source[`*<num>*`]:` strings.

     e. Add the following string to the end of the `%build` section of the specification file:

        ```
        %patch<num> -p1
        ```

   - If a number of changes is rather small (one file, for example), the source files can be changed in the `%build` section of the specification file using the `sed` command. For example, if the single string:

     ```
     INCDIR=-I.. -I/usr/include
     ```

     in the Makefile file should be changed to:

     ```
     INCDIR=-I.. -I${BLUECAT_PREFIX}/usr/include
     ```

     it can be done by entering the following command:

     ```
     BlueCat:$ sed -e '/INCDIR=/s%-I/\
     %-I${BLUECAT_PREFIX}/%' < Makefile \
     >Makefile.tmp mv Makefile.tmp Makefile
     ```

Install the built package with the following command:

```
BlueCat:$ rpm -i $BLUECAT_PREFIX/cdt/\
src/bluecat/RPMS/<target>/<rpm_name>
```

This step can fail only in the case in which the specification file contains a post- or preinstallation section that calls the commands working with host directories. In this case, references to the host directories should be replaced by references to the BlueCat Linux target directories.

For example, if the specification file contains section:

```
%post
ldconfig /usr/lib
```

change it to:

```
%post
$BLUECAT_PREFIX/cdt/sbin/ldconfig $BLUECAT_PREFIX/usr/lib
```

Note, that if the specification file is changed, the package should be rebuilt.

BlueCat Linux Directory Structure

Overview

The installation procedure described earlier results in creation of the BlueCat Linux directory on the cross-development host. The structure of the directory is shown in the figure below:

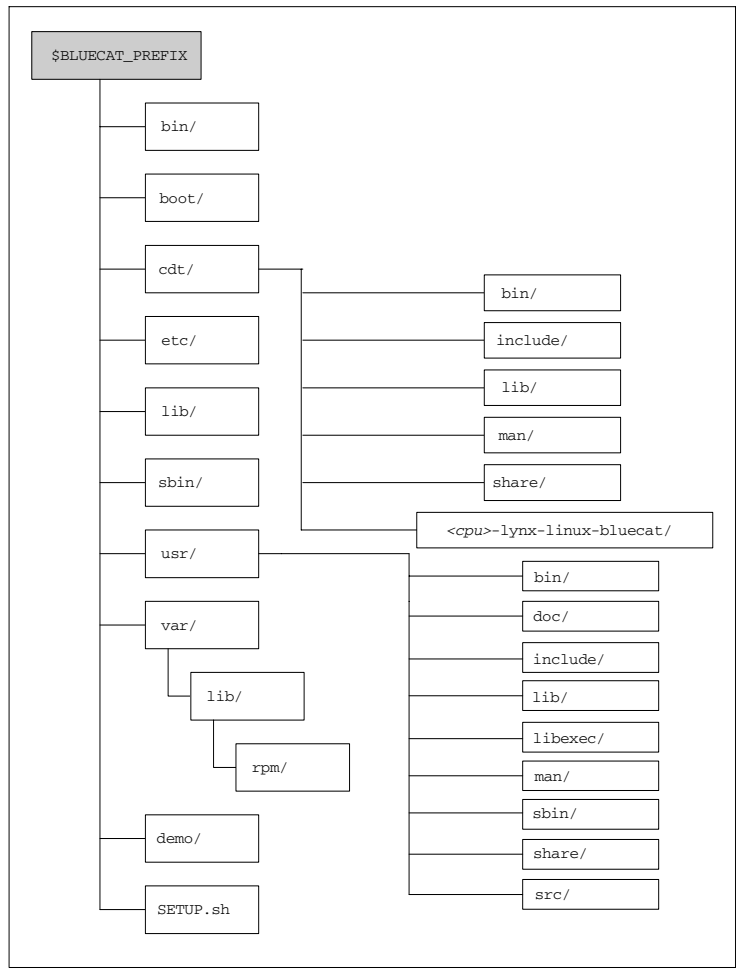

Figure 1-4: BlueCat Linux Directory Structure

BlueCat Linux Components

The following table briefly describes all the key components of the BlueCat Linux directory structure.

Table 1-4: BlueCat Linux Directory Structure Components

Node	Description
`$BlueCat_PREFIX`	Installation directory
`- bin/`	Target board binaries
`- boot/`	Target board `boot` directory
`- cdt/`	Cross-development tools
`-- bin/`	Cross-development binaries
`-- include/`	Cross-development `include` files
`-- lib/`	Cross-development libraries
`-- man/`	Cross-development tool man pages
`-- share/`	Various cross-development shared files
`--<`*`cpu`*`>-lynx-linux-bluecat/`	Cross-development binaries
`- demo/`	BlueCat Linux demo system configurations
`- etc/`	Target board configuration files
`- lib/`	Target board libraries
`- sbin/`	Target board system binaries
`- usr/`	Target board top-level `usr` directory
`-- bin/`	More target board binaries
`-- doc/`	Various documentation files
`-- include/`	Target board `include` files
`-- lib/`	More target board libraries
`-- libexec/`	Auxiliary files
`-- man/`	Man pages
`-- sbin/`	More target board system binaries
`-- share/`	Various target board shared files
`-- src/`	Main source tree

Table 1-4: BlueCat Linux Directory Structure Components (Continued)

Node	Description
- var/	Top-level target board var directory
-- lib/	Libraries
-- rpm/	RPM database
- SETUP.sh	Shell script for setting up cross-development environment

The cdt subtree contains all cross-development tools. The majority of the remaining directories contain files intended for use on the target board.

Setting the BlueCat Linux Execution Environment

For Linux Hosts

To set the correct execution environment before starting any BlueCat Linux development, perform the following steps:

1. Go to the directory where BlueCat Linux has been installed. For example:

   ```
   $ cd $HOME/BlueCat
   ```

2. Source the shell script to set up the BlueCat Linux environment variables:

   NOTE: The bash shell must be used on the cross-development host with BlueCat Linux.

   ```
   $ . SETUP.sh <bsp>
   ```

The SETUP.sh script sets up a number of environment variables used by BlueCat Linux tools. These environment variables include BLUECAT_PREFIX, which must contain the absolute path to the BlueCat Linux installation directory. If the *<bsp>* option is specified, the BLUECAT_TARGET_BSP variable is set to activate the Board Support Package for the specified target board.

Additionally, the PATH environment variable is set up so that the BlueCat Linux cross-development tools are located first, before the native host development tools (if any).

Unsetting the BlueCat Linux Environment

To unset the BlueCat Linux environment, simply close the current shell session (by closing the `xterm` window).

Multiple Instances of BlueCat Linux

BlueCat Linux is designed for use in a multiuser environment; it does not have to be installed in a system-wide manner. The entire BlueCat Linux installation and operation can be performed within a user-owned directory. Thus, multiple instances of BlueCat Linux can coexist on a single cross-development host, without interfering with each other in any way. No additional setup is required, other than each user creating his or her own installation of BlueCat Linux and setting up the correct execution environment.

Uninstalling BlueCat Linux

Uninstalling an Entire Installation

To uninstall an entire BlueCat Linux installation from the BlueCat Linux environment, remove the entire installation tree by entering the following:

```
BlueCat:$ cd $BLUECAT_PREFIX
BlueCat:$ uninstall [-f]
```

If the `-f` option is not specified, the script prompts the user before starting uninstallation. Otherwise, the entire installation tree is removed without any prompt. Close the current shell to clean up all previous environment settings.

Uninstalling Support for a Target Board

To uninstall BlueCat Linux support for a target board from the BlueCat Linux environment, remove the target board-specific files and directories by entering the following command:

```
BlueCat:$ cd $BLUECAT_PREFIX
BlueCat:$ uninstall [-f] <bsp1> <bsp2> <bspn>
```

If the `-f` option is not specified, the script prompts the user before starting uninstallation. Otherwise, target board-specific files and directories are removed

without any prompt. Also, if support for the currently active target board is being removed, close the current shell to clean all previous environment settings.

Uninstalling a BlueCat Linux Component

As most of the BlueCat Linux components come in RPM formatted packages, uninstalling a component is as easy as uninstalling a regular RPM package. This is achieved, for example, with the following command:

```
BlueCat:$ rpm -e <package_name>
```

To find the exact `<package_name>`, use the `-qa` argument with the `rpm` command. This displays a list of RPM packages installed.

CAUTION! Be careful to set up the BlueCat Linux environment (by sourcing the `SETUP.sh` script file at the top of the BlueCat Linux directory tree) before uninstalling BlueCat Linux. Failure to do so results in the risk of removing or corrupting a package in the native cross-development host installation.

BlueCat Linux Utilities

The BlueCat Linux distribution comes with a set of LynuxWorks utilities to facilitate the creation and downloading of bootable custom applications/images/device drivers/kernels.

mkkernel

`mkkernel` is a BlueCat Linux utility that rebuilds kernels. Specifically, `mkkernel` allows the user to maintain multiple kernel profiles contained in their respective `.config` configuration files. `mkkernel` copies a specific `.config` file corresponding to a given kernel profile to the `$BLUECAT_PREFIX/usr/src/linux` directory. It then runs `make oldconfig` on the `.config` file, and copies the resulting kernel image to a user-specified location.

The syntax for `mkkernel` is:

```
mkkernel <config_file> <bvmlinux_file> [<bzimage_file>]
```

The `<config_file>` file specified as the first parameter is copied to `.config` in the `$BLUECAT_PREFIX/usr/src/linux` directory. Having done this,

`mkkernel` runs `make oldconfig` and then builds the kernel. The second parameter, *<bvmlinux_file>*, specifies the location for copying the resulting kernel image. The third argument is optional: if present, it is taken as the output filename of the kernel image in the `bzImage` format (only for the x86 architecture), otherwise no `bzImage` kernel is created.

For more information, refer to “mkkernel” in Appendix A, “Command Reference.”

mkrootfs

`mkrootfs` is a LynuxWorks utility that enables building root file system images in BlueCat Linux. These images can be either bootable directly on target boards, installable on target hard disks and then used for booting, or downloadable into target Flash memory for consequent booting.

`mkrootfs` creates a root file system described in a specification file, or `.spec` file. Depending on the option passed to `mkrootfs`, it creates:

- A gzipped ext2 file system image or a tar file, (if the `-T` option is specified)
- A gzipped ext3 file system image (if the `-x` option is specified)
- A Journaling Flash File System image (if the `-J` option is specified)
- A Journaling Flash File System version 2 image (if the `-j` option is specified)
- A CD-ROM ISO9660 file system image (if the `-O` option is specified)

For more information, refer to “mkrootfs” in Appendix A, “Command Reference.”

mkboot

The `mkboot` cross-development tool can be used for copying bootable images from the cross-development host onto a floppy disk or a hard disk. Alternatively, `mkboot` can be used for creating a bootable CD-ROM image of BlueCat Linux that can be burned to a CD-R(W) disk. `mkboot` is capable of:

- Installing BlueCat Linux boot sector
- Installing compressed BlueCat Linux kernel
- Installing compressed root file system image
- Defining the root device to be mounted by the kernel

- Setting the command line to be passed to the kernel
- Creating an image composed of a BlueCat Linux kernel and a compressed file system, suitable to be programmed into ROM/Flash or downloaded over a network by the target firmware
- Creating a bootable CD-ROM image
- Creating a bootable target image file that can be copied to a bootable device outside of the BlueCat Linux environment
- Creating a partitioned disk image with an optional BlueCat Linux kernel

When called with no options, `mkboot` shows components currently installed on the media.

For more information, refer to “mkboot” in Appendix A, “Command Reference.”

osloader

The BlueCat Linux OS loader (`osloader`), which resides in the `$BLUECAT_PREFIX/demo` directory, is a tool that enables downloading and booting a BlueCat Linux application on a target board. The BlueCat Linux OS loader can be downloaded onto a hard disk, floppy disk, CD-ROM or target Flash memory and used to boot BlueCat Linux over a network using TFTP or NFS, or from a parallel port using PFTP. This is accomplished through the BlueCat Linux Loader Shell (BLOSH), a shell-like utility that allows the user to specify the target IP address and the images to be downloaded.

The advantage of the OS loader is that it can be fitted on a floppy disk and yet it maintains the ability to boot BlueCat Linux on all targets.

CHAPTER 2 *Developing BlueCat Linux Applications*

This chapter explains developing and maintaining custom BlueCat Linux applications for target boards.

Development Directory

Installing BlueCat Linux results in a development directory, which is structured as shown in the figure below.

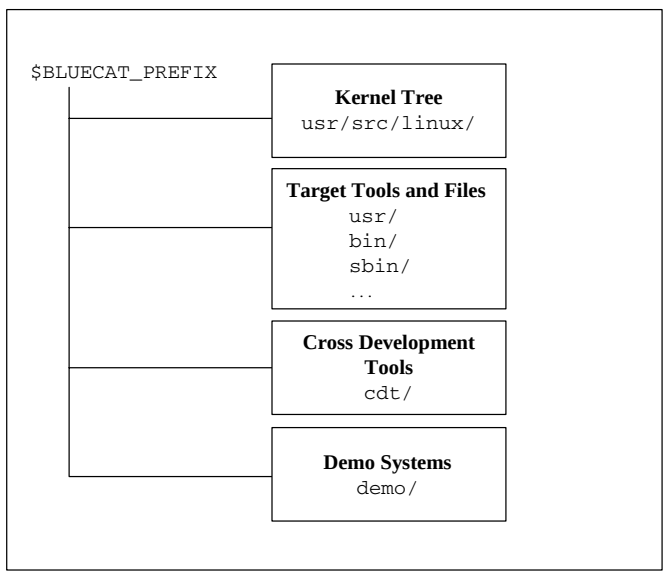

Figure 2-1: BlueCat Linux Development Directory Structure

Kernel Tree

This section describes the contents of the kernel tree.

The BlueCat Linux kernel tree is located in the following area:

```
$BLUECAT_PREFIX/usr/src/linux
```

The structure of the BlueCat Linux kernel tree is the same as that of the “standard” Linux kernel, version 2.6.13.

The location of the contents is similar to that of the Linux kernel tree. As a reminder, the following table shows important areas of the Linux kernel tree:

Table 2-1: Kernel Tree Contents

Kernel Subdirectory	Contents
`linux/init`	Functions needed to start the kernel
`linux/kernel`	Kernel core and system calls
`linux/mm`	Memory management
`linux/fs`	Implementations of various file systems supported by Linux kernel
`linux/ipc`	Sources for System V IPC, such as semaphores, shared memory, and message queues
`linux/net`	Implementation of various network protocols (TCP/IP, ARP, and so on)
`linux/lib`	Some standard C library functions
`linux/include`	Kernel-specific header files
`linux/drivers`	Device drivers for hardware components, divided into subdirectories according to the device type
`linux/arch/<cpu>`	Architecture-dependent code

BlueCat Linux Kernel RPMs

The BlueCat Linux kernel tree is unpacked from a number of RPM packages. The following packages are installed to create the kernel tree:

Table 2-2: Kernel Tree Packages

Package	Description
`kernel_trg-headers-<version>`	C header files for the BlueCat Linux kernel
`kernel_trg-source-<version>`	Source code files for the BlueCat Linux kernel and kernel object files built for the default configuration of the kernel
`kernel_trg-<target>-<version>`	Linux kernel binary built for the default configuration of the BlueCat Linux kernel
`kernel_trg-doc-<version>`	Useful documentation
`kernel_trg-bcboot-<version>`	BlueCat Linux boot record template used for copying BlueCat Linux on a hard disk, a floppy or a CD-ROM

BlueCat Linux Kernel versus 'Pristine' Linux Kernel

All changes to the *pristine* source files of the Linux kernel made by BlueCat Linux are contained as patches in the sources of the Linux kernel RPM packages. These are available on the BlueCat Linux *Source Architecture CD-ROM* in `BlueCat_<cpu>/target/SRPMS`.

This release of BlueCat Linux has the following BlueCat Linux-specific patch for the Linux kernel:

SRPMS File	`kernel_trg-<version>.src.rpm`
Patch	`linux-bc.patch.gz`
Description	BlueCat Linux changes to Linux kernel baseline

Once BlueCat Linux is installed, all the BlueCat Linux-specific changes are available in the Linux kernel subdirectory. To find these changes, search for `CONFIG_BLUECAT` and `__bluecat__`.

Target Tools and Files

The subdirectories `$BLUECAT_PREFIX/bin`, `$BLUECAT_PREFIX/sbin`, and `$BLUECAT_PREFIX/usr` contain ready-to-run tools and files for the target board. These tools and files are downloaded onto target boards by including them in the target board file system.

Cross-Development Tools

All BlueCat Linux development tools are available in the directory `$BLUECAT_PREFIX/cdt`. These tools are used to develop programs and images for target boards.

Demo Systems

The BlueCat Linux distribution package contains a number of prebuilt, ready-to-run BlueCat Linux demo systems. These can be found in the `$BLUECAT_PREFIX/demo` directory.

Each demo system contains bootable images of a BlueCat Linux kernel and a root file system that contains the programs and files required to run the BlueCat Linux features highlighted by the demo system.

The `$BLUECAT_PREFIX/demo` directory contains a number of subdirectories, each corresponding to a specific demo system. Each demo system demonstrates a particular feature of BlueCat Linux. Refer to Chapter 4, “BlueCat Linux Demo Systems” for a detailed description of all BlueCat Linux demo systems included in the distribution.

Embedded System Definition

A typical BlueCat Linux embedded demo system has two components:

- The BlueCat Linux kernel that is customized for the embedded system
- The root file system containing the tools and custom programs required to boot and run the features highlighted by the demo on the target board. Normally, the root file system also contains the embedded system application programs.

The BlueCat Linux development tools enable the creation of kernel and file system images suitable for booting and downloading onto a target board, and the creation of application programs for BlueCat Linux.

NOTE: All of the demo systems documented in this *User's Guide* may not be included a given Board Support Package (BSP). For details regarding demo systems supported on a specific target board, please refer to the relevant *Board Support Guide* (BSG).

OS Loader

The BlueCat Linux OS loader is a special configuration of BlueCat Linux that is designed as a firmware-level tool for downloading and booting a BlueCat Linux application on a target board. Refer to Chapter 3, "Downloading and Booting BlueCat Linux" for a detailed discussion of OS loader features.

Development Process Overview

The flow chart in the next figure shows the major steps in the BlueCat Linux development process.

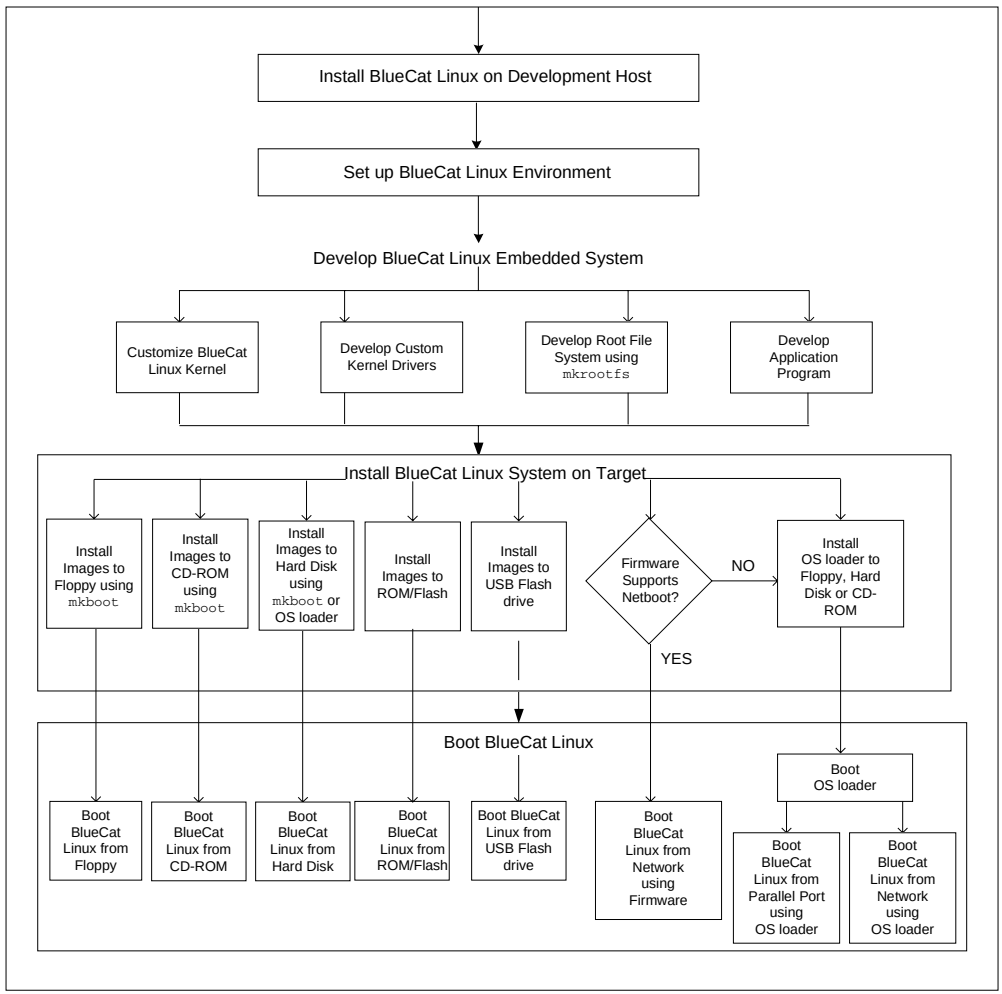

Figure 2-2: BlueCat Linux Development Flow Chart

As shown in the previous flow chart, the development process for an embedded BlueCat Linux system is as follows:

1. Install BlueCat Linux on the cross-development host. The installation procedure is detailed in Chapter 1, “Introduction and Installation.”
2. Set up the BlueCat Linux execution environment on the cross-development host. This is done by sourcing the script `SETUP.sh` from the top of the installation directory. See “Setting the BlueCat Linux Execution Environment” on page 23.
3. Develop a custom BlueCat Linux system for the target board using the cross-development tools. The actual development may involve any of the following tasks:
   - Customizing the BlueCat Linux kernel configuration
   - Developing custom kernel drivers and features
   - Developing custom application programs
   - Creating the root file system using the `mkrootfs` tool
4. Download BlueCat Linux onto the target board. BlueCat Linux supports the following boot devices:
   - Floppy disk
   - Hard disk
   - USB Flash drive
   - CD-ROM
   - Target ROM/Flash memory
   - Network (TFTP or NFS server)
   - Parallel Port

   The device from which the user wants to boot the target board determines the method for downloading the embedded system onto the target board, using one of the following:
   - Target board firmware
   - External ROM/Flash programmer device
   - `mkboot` tool
   - OS loader

The various download scenarios are described in Chapter 3, “Downloading and Booting BlueCat Linux.”

5. Boot BlueCat Linux on the target board. Depending on the boot device onto which the BlueCat Linux system is copied, the target board is booted one of the following ways:

   - Directly from the boot device
   - In a two-step procedure, where the OS loader is booted from a boot device (floppy disk, hard disk, CD-ROM, or ROM), which then boots the BlueCat Linux image from the same or alternative boot device

   Refer to Chapter 3, “Downloading and Booting BlueCat Linux” for a detailed description of the BlueCat Linux boot procedure.

Customizing the BlueCat Linux Kernel

This section describes configuring and building the BlueCat Linux kernel for an embedded system.

Configuring the Kernel

The need to configure the kernel depends on the nature of the user’s embedded system and application programs. For each BSP, the BlueCat Linux installation includes a complete, fully prebuilt kernel with a default set of features.

The default configuration of the BlueCat Linux kernel may be appropriate for the user’s embedded system. Users should refer to the *Board Support Guide* for a complete description of the default configuration of the BlueCat Linux kernel.

However, if the default BlueCat Linux kernel configuration is not appropriate for a specific embedded system, the kernel must be reconfigured appropriately.

In general, the user may want to reconfigure the BlueCat Linux kernel for one or more of the following reasons:

- *Customizing for functionality*—Modifying a kernel option to add or remove a kernel feature (for instance, networking support), or to modify the default feature behavior
- *Customizing for hardware devices*—Modifying a kernel option to add or remove support for a particular device

- *Customizing for size*—Removing kernel features not required in the embedded system in order to reduce kernel size
- *Customizing for performance*—Modifying a kernel option to improve the runtime performance of the kernel

Kernel Configuration Procedure Overview

The BlueCat Linux kernel can be configured in the same way that the "standard" Linux kernel version 2.6.13 is configured. The configuration procedure consists of the following steps:

1. The configuration takes as input the `.config` file located in `$BLUECAT_PREFIX/usr/src/linux` directory for information on components to be included in the kernel. The `.config` file holds the definition of the kernel configuration options and current assignments.
2. The interactive configuration interface leads the user through the kernel configuration options and allows changes to current settings.
3. Updated kernel settings are saved in the `.config` file in the `$BLUECAT_PREFIX/usr/src/linux` directory. A number of *kernel header files* are updated to reflect the changes in the kernel configuration.
4. The next kernel build uses the updated header files and rebuilds the kernel, thus setting the runtime kernel configuration exactly as described in the `.config` file.

An alternative to this procedure skips Step 2 and updates the kernel header files based on the options defined by the current `.config` file in the `$BLUECAT_PREFIX/usr/src/linux` directory. This approach ensures that the kernel configuration contained in `.config` is synchronized with the kernel header files. This is useful when one is unsure if the configuration in `.config` is currently being used.

Kernel Configuration Command Interface

There are the following ways of modifying the kernel configuration:

- By using `make config`—`make config` starts an interactive configuration script that allows for updating the BlueCat Linux kernel configuration. It does not allow for easy correction of incorrect choices.

- By using `make menuconfig`—`make menuconfig` implements a text-based menu and uses the `Curses` libraries. If an incorrect choice is made, it is easy to correct.
- By using `make xconfig`—`make xconfig` implements an X-based (Qt-based) GUI configuration tool. If an incorrect choice is made, it is easy to correct.
- By using `make gconfig`—`make gconfig` implements an X-based (GTK-based) GUI configuration tool. If an incorrect choice is made, it is easy to correct.
- By using `make oldconfig`—`make oldconfig` does not allow for kernel configuration. Instead, it simply updates the kernel header files to reflect the current settings in `.config`.

In each case when the process is complete, the updated kernel configuration file, `.config`, is saved back in the `$BLUECAT_PREFIX/usr/src/linux` directory.

The configuration process in itself does *not* rebuild the kernel. The user must explicitly rebuild the kernel to enable the new kernel configurations.

make xconfig

- **Linux host**

  For the Linux host, the `qt-devel-3.1.1-6` and `qt-3.1.1-6` packages must be installed.

  To install the Qt library on a Red Hat Linux system, insert the respective Red Hat Linux installation CD-ROM from the Red Hat Linux distribution and follow the steps described in *Red Hat Linux x86 Installation Guide*.

  If the KDE (K Desktop Environment) is used on the host, the Qt library is already installed on the host.

- **Windows host**

  Ensure that Microsoft Visual Studio 6.0 or higher is installed on the cross-development host and that the Visual Studio environment tools have been set up to allow invocation of the Microsoft Visual Studio tools in command line mode.

  Then, to install the Qt library on the Windows host, go to `www.trolltech.com` and download the Qt software for Microsoft

Visual Studio C++ Windows users. To install the Qt library, follow the instructions provided with the Qt software.

`$BLUECAT_PREFIX/usr/src/linux/scripts/kconfig/Makefile` (the `Makefile` for the Windows host) uses the `QTLIBS` environment variable to list the Qt libraries that are needed to link with the `qconf` executable used to implement `make xconfig` on the Windows host. The following default definition is used:

```
QTLIBS = qt-mteval336.lib qtmain.lib
```

This definition specifies that libraries from the Qt 3.3.6 evaluation version for Windows are needed to link with the `qconf` executable.

If the Qt version installed on the host differs from Qt 3.3.6 evaluation version, the `QTLIBS` definition must be changed to specify the correct list of libraries. This can be done either by manually editing `$BLUECAT_PREFIX/usr/src/linux/scripts/kconfig/Makefile` to modify the `QTLIBS` definition or by defining the `QTLIBS` environment variable using the **Properties**->**Advanced**->**Environment Variables** wizard in the context menu of the **My Computer** icon on the Windows desktop. The second approach allows the user to avoid changing the `Makefile` every time the BlueCat Linux is reinstalled.

make gconfig (Linux Host Only)

Before using `make gconfig`, the following GTK+ components must be installed on the host system:

- GTK+ library
- GTK+-2.*
- `gmodule-2.*`
- `libglade-2.*`

To install the GTK+, `gmodule`, and `libglade` packages on a Red Hat Linux system, insert the respective Red Hat Linux installation CD-ROM from the Red Hat Linux distribution and follow the steps described in *Red Hat Linux x86 Installation Guide*.

If the GNOME desktop environment is used on the host, the GTK+, `gmodule`, and `libglade` packages are already installed.

Building the Kernel

The following instructions for building a kernel are based on an x86 target board. For instructions and examples specific to a target board, please consult the relevant *Board Support Guide*.

As mentioned earlier, the BlueCat Linux distribution for a given target contains a prebuilt kernel configured to include the default set of features. If the default configuration of the BlueCat Linux kernel satisfies the user's embedded system needs, the user may not need to rebuild the kernel.

The BlueCat Linux kernel has to be rebuilt in either of the following cases:

- The BlueCat Linux kernel has been configured to define a kernel configuration that is different from the default kernel configuration.
- Custom updates have been made to the BlueCat Linux kernel source files or a new kernel feature (for example, a hardware device driver) has been added.

A BlueCat Linux kernel can be built in three different *boot scenarios*:

- It can be built to boot from a floppy disk, hard disk, or CD-ROM.
- It can be built for downloading directly onto the target board using the BlueCat Linux OS loader.
- It can be built to boot from target ROM/Flash memory.

Building the Kernel to Boot from Floppy, Hard Disk, or CD-ROM

The `make bzImage` command builds the new kernel and creates the compressed kernel image ready for copying on a floppy disk, hard disk, or CD-ROM. When the kernel boots, it automatically decompresses when executed.

The `make bzimage` command is executed from the `$BLUECAT_PREFIX/usr/src/linux` directory and leaves a new `bzImage` file in the `$BLUECAT_PREFIX/usr/src/linux/arch/i386/boot` directory.

Building a Kernel for Download using OS Loader

It is possible to download a compressed kernel using the BlueCat Linux OS loader directly into RAM on the target board. Unlike the scenario where booting is from floppy, hard disk, or CD-ROM, the compressed kernel does not contain a boot sector.

When executing the `make bzImage` command, an additional file called `vmlinux.bin` is created. This file, which is located in the `$BLUECAT_PREFIX/linux/usr/src/arch/i386/boot` directory, contains the boot-sector-less compressed kernel image.

The `vmlinux.bin` file is then downloaded directly onto the target board, uncompressed, and booted.

Building the Kernel to Boot from Target ROM/Flash Memory

The same `bvmlinux.out` image is used to burn a programmable image into the target ROM/Flash memory, and is then used to boot the system.

Managing Multiple Kernel Profiles

As described earlier, a specific `.config` file corresponds to a certain set of BlueCat Linux kernel settings. As the kernel is booted on the target board, these settings are used to define the runtime behavior of the kernel. The one-to-one correspondence between a kernel configuration and the `.config` file can be used in the BlueCat Linux product to manage multiple kernel profiles for different embedded applications.

The user may need to support multiple kernel profiles within a single BlueCat Linux installation. Each profile corresponds to, and is used in, a unique embedded system. The simple approach supported by BlueCat Linux is as follows:

1. Maintain a separate `.config` file as a formal description of each kernel profile.
2. To enable a kernel profile, copy the appropriate `.config` to the `$BLUECAT_PREFIX/usr/src/linux` directory and reconfigure the kernel (using `make oldconfig`).
3. Rebuild the kernel.
4. If the kernel configuration is changed by setting any configuration options to values different from those contained in the profile's initial `.config` file, copy the modified `.config` in the `$BLUECAT_PREFIX/usr/src/linux` directory back to the directory that holds the kernel profile.

Debugging the Kernel

The GnuPro Debugger (GDB) is used to debug BlueCat Linux device drivers and kernel code from the BlueCat Linux cross-development host.

Virtually all the normal GDB features are available for kernel debugging purposes:

- Source-level variable examination
- Source-level single-stepping
- Disassembling kernel memory
- Call stack chain examination
- Task support

GDB on the cross-development host actually talks to the BlueCat Linux kernel debugger, which is embedded in the kernel on the target board. The BlueCat Linux kernel debugger works as a server and performs the following basic operations to requests made by GDB:

- Memory read
- Memory write
- Register examination
- Execution resumption
- Single stepping

Hence, the target board must have the BlueCat Linux kernel with the kernel debugger enabled downloaded. See “Debugging Requirements” below.

NOTE: Debugging is available only from a remote cross-development host. There is no self or local kernel debugging.

Debugging Requirements

To use GDB for kernel debugging purposes, the setup is required:

- The target BlueCat Linux with the kernel debugger enabled - To enable the kernel debugger, set the **BlueCat kernel debugger** option to **Yes** to turn on `CONFIG_BLUECAT_KDBG` using `make menuconfig` and rebuild the kernel.
- To build a BlueCat Linux kernel for debugging at the source level, compile the device driver (or code to be debugged at the source level)

with the `-g` option. Edit the appropriate Makefile to include the `-g` option for compilation or set the **Include debugging information in the kernel binary** option to **Yes** and rebuild the kernel.

- The BlueCat Linux cross-development host - A serial line connection between the BlueCat Linux target board and the cross-development host.

Setting Up Serial Ports

A dedicated serial port for kernel debugging is required for reliable communication. This serial port can be configured using the kernel configuration procedure.

The options `CONFIG_BLUECAT_KDBG_TTYS[0-3]` are used to configure the `/dev/ttyS[0-3]` devices as a kernel debugger serial line port. The default kernel debugger serial port is `/dev/ttyS1`. The target board serial port must have parameters matching those of the cross-development host, such as baud rate, parity, bit, and type.

NOTE: The kernel debugger on the BlueCat Linux target board is configured to communicate with the cross-development host GDB using the default serial line parameters: 9600 bps baud rate, no parity, 8 data bits, 1 stop bit.

Starting Kernel Debugging

As with other components of BlueCat Linux, it is important to set up the BlueCat Linux environment before calling GDB. On the cross-development host, change to the kernel directory (`$BLUECAT_PREFIX/usr/src/linux`) and start `gdb`, specifying the kernel image as the parameter:

```
BlueCat:$ gdb vmlinux
```

NOTE: `vmlinux` is a kernel image that contains symbol information required by the cross-development host `gdb` for debugging the BlueCat Linux kernel. Refer to “Debugging Requirements” on page 42 for a detailed description of building a `vmlinux` kernel image.

To start a kernel debugging session on the host, use the `target remote` command and specify an appropriate serial port. For example:

```
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
```

```
kdbg_breakinst () at arch/i386/kernel/kdbg/i386-stub.c:58
58      }
warning: shared library handler failed to enable
breakpoint
(gdb)
```

The BlueCat Linux kernel debugger stops the kernel at an early stage in the kernel initialization procedure. This creates a synchronization point for GDB and the kernel debugger.

GDB can be run both prior to or after loading a target board kernel. When running prior to loading the kernel, GDB waits for the synchronization signal sent by the target BlueCat Linux kernel debugger at the earlier stages of initialization. If the debugging session begins after the target board kernel is up, the host GDB attaches to the kernel by sending special synchronization signals to the target board kernel debugger. Once communication is established, GDB reports the location of the kernel interrupt.

If GDB returns the following message:

```
Couldn't establish connection to remote target,
```

try the `target` command again. Persistence of this error may be due to incorrect communication port/parameters, or incorrect target board configuration (that is, not enabling the BlueCat Linux kernel debugger).

Now set breakpoints at desired kernel locations and use the `continue` command to resume the kernel. Once the kernel hits a breakpoint and stops, it is possible to examine variables and the call stack chain, single-step, and continue, as one would for user process debugging.

CAUTION! Kernel debugging stops the entire operating system; the operating system does not respond when the kernel is at a breakpoint or is stopped by the debugger.

Displaying Kernel Data Structures

The contents of any kernel data structure can be displayed using the standard GDB commands (`print` and `x`). To display the contents of a kernel data structure, the user must specify either a structure name or a memory address where the data structure is located. The following example shows how the `print` GDB command displays the contents of a `task_struct` structure located at address 0xc0214560:

```
(gdb) print *(struct task_struct *)0xc0214560
$2 = {state = 0, thread_info = 0xc0240000, usage = {counter = 6}, flags =
256,
ptrace = 0, lock_depth = 1, prio = 140, static_prio = 120, run_list = {
next = 0x100100, prev = 0x200200}, array = 0x0, sleep_avg = 0,
interactive_credit = 0, timestamp = 0, activated = 0, policy = 0,
...
```

Interrupting the Kernel

To interrupt a running kernel, press **Ctrl-C** at the cross `gdb`, while it is waiting for the target board to stop as one would for user process debugging. `gdb` sends the break-in character to stop the target board kernel.

NOTE: If the user presses **Ctrl-C** while BlueCat Linux is running a user-mode program, the following message appears on the cross-development host console:

```
(gdb) c
Continuing.
Program received signal SIGTRAP, Trace/breakpoint trap.
0xc019a632 in serial8250_console_write (co=0xffffff20,
s=0xc0266c46 ") is a 16550A\n<6>\n<4>kdbg: break command received.\n",
count=23) at include/asm/io.h:371
371     BUILDIO(b,b,char)
(gdb)
```

This is because the kernel debugger is designed not to interfere with user programs. So the kernel debugger switches into the kernel before entering the debugger and stops at the function `kdbg_breakinst`.

Finishing Kernel Debugging

To finish a debug session and let the target board kernel resume freely, detach the target board at the `gdb` prompt or quit GDB:

```
(gdb) detach
Ending remote debugging.
(gdb)
```

CAUTION! If a kernel debugging session is accidentally terminated due to a communication error or for some other unexpected reason, GDB may not have had a chance to remove breakpoints before the termination. If this happens and a new kernel debug session is started, the kernel may be trapped at a breakpoint indefinitely, until the original instruction at the breakpoint location is restored manually with a command like `print` or until the kernel is reloaded (restarted). GDB currently provides no convenient way to restore the original instructions.

Developing Application Programs

Building Application Programs

Use the BlueCat Linux cross-development tools to build custom application programs. The same GNU tools that are used to build the BlueCat Linux kernel are used to build user-space tools and programs.

The BlueCat Linux cross-development tools are configured to build binaries appropriate for target boards. The tools ensure that the correct libraries from the BlueCat Linux distribution are used to build application programs.

As with other components of BlueCat Linux, it is important to set up the BlueCat Linux environment before calling the cross-development tools to build application programs. As soon as the BlueCat Linux environment is set correctly, any reference to the GNU tools calls the cross-development tools from the BlueCat Linux `cdt/` area.

NPTL and LinuxThreads

BlueCat Linux includes a new implementation of the POSIX thread library known as NPTL (Native POSIX Thread Library). This library uses new facilities of Linux 2.6.13 kernel and the TLS (Thread Local Storage) feature of the new toolchain. The NPTL implementation shows significant improvement in performance in comparison with the old LinuxThreads implementation.

The dynamic NPTL library and a version of the `glibc` library with TLS and NPTL support enabled are located in the `$BLUECAT_PREFIX/lib/tls` directory. These are the libraries used for linking in BlueCat Linux run-time environment. Static libraries and standard links are located in the `$BLUECAT_PREFIX/usr/lib/nptl` directory.

In BlueCat Linux, by default applications are built using the NPTL `libpthread` and NPTL-oriented `libc` libraries. To build an application with the LinuxThreads `libpthread` library and/or LinuxThreads-oriented `libc` library, perform the following steps prior to building the application:

1. Remove the `nptl_trg-devel` RPM package from the BlueCat Linux installation:

   ```
   BlueCat:$ rpm -e nptl_trg-devel
   ```

2. Remove the `$BLUECAT_PREFIX/lib/tls` directory:

```
BlueCat:$ rm -rf $BLUECAT_PREFIX/lib/tls
```

Improving POSIX Conformance

Some functions in the `libc` and NPTL `libpthread` libraries (such as `gethostname()`, `sigprocmask()`, `pthread_join()`, `pthread_sigmask()`, and so on) do not check validity of their arguments by default and may cause segmentation fault when an attempt to access an invalid pointer is made.

To turn on the pointer validation check, set the `BLUECAT_CHECK_VALIDITY` environment variable to any value in the BlueCat Linux run-time environment on the target.

Debugging Application Programs

The BlueCat Linux development tools include the GDB debugger, which is configured for operation in a cross-development environment. Use it to debug application programs running on the target board from the cross-development host.

Requirements

This setup involves two components of the GDB debugger:

- GDB, configured as a GDB client, runs on the cross-development host.
- The GDB server program, conveniently named `gdbserver`, runs on the target board.

`gdb` and `gdbserver` communicate via a serial line or a TCP/IP connection, using the standard `gdb` remote protocol.

In the BlueCat Linux directory tree, the `gdbserver` binary can be found in `$BLUECAT_PREFIX/usr/bin`. The cross debugger GDB binary (`gdb`, `insight`) resides in `$BLUECAT_PREFIX/cdt/bin`.

Procedure

To debug an application program, perform the following steps:

1. Place a copy of the application program to be debugged onto the target board.

`gdbserver` does not need the program symbol table, so the user can strip the program, if necessary, to save space. `gdb` on the cross-development host does all the symbol handling. (See "Discarding Symbols from Files" on page 53.)

2. `gdbserver` must be told how to communicate with `gdb`. It must also be given the name of the user program and arguments for the program. The syntax is:

   ```
   bash# gdbserver <COMM> <PROGRAM> [ <ARGS> ... ]
   ```

   where *<COMM>* is either a device name (to use a serial line) or a network target board name and port number.

   For example:

   - To debug a program named `test_prog` with the argument `foo.txt` and to communicate with `gdb` over the serial port `/dev/ttyS1`, enter:

     ```
     bash# gdbserver /dev/ttyS1 test_prog foo.txt
     ```

     `gdbserver` waits for the cross-development host `gdb` to communicate with it.

   - To use a network connection instead of a serial line, enter:

     ```
     bash# gdbserver <host_IP>:<port_number> \
     test_prog  foo.txt
     ```

     The only difference between the two examples is that the first argument specifies that the user is communicating with the cross-development host GDB via network. The *<host_IP>*:*<port_number>* argument means that `gdbserver` is to expect a network connection from the machine *<host_IP>* to the local network port, *<port_number>*. (Currently, the *<host_IP>* component is ignored.)

     Any number the user wishes can be chosen for the port number, as long as it does not conflict with network ports already in use on the target board (for example, `23` is reserved for `telnet`). If the user chooses a port number that conflicts with another service, `gdbserver` prints an error message and exits.

The same port number must be used with the cross-development host GDB `target remote` command.

1. Ensure that an unstripped copy of the program is on the cross-development host because the GDB client needs symbols and debugging

information. Do not strip the program binary on the cross-development host.

2. Start up the cross GDB from the BlueCat Linux environment using the name of the local copy of the program as the first argument.

   (The `--baud` option may also be needed if the serial line is running at anything other than 9600 bps.)

   For instance, if an unstripped copy of the program is located in the current directory, start GDB as follows:

   ```
   BlueCat:$ gdb test_prog
   ```

3. Use `target remote` to establish communication with `gdbserver`. Its argument is either a serial device name or a network port descriptor in the form `<target_IP>:<port_number>`. For example:

   ```
   (gdb) target remote /dev/ttyS1
   ```

   communicates with the target board via the serial line `/dev/ttyS1`, and

   ```
   (gdb) target remote 1.0.0.1:2345
   ```

   communicates via a TCP connection to port `2345` on the target `1.0.0.1`.

NOTE: For TCP connections, the `gdbserver` must be started prior to using the `target remote` command. Failure to do so may result in an error message on the cross-development host.

4. Once the connection between the cross-development host GDB and gdbserver is established, use the `gdb` command line interface to debug the application program.

Building a Root File System

This section describes building a root file system for an embedded BlueCat Linux system. BlueCat Linux requires a root file system for booting and initializing the embedded system.

BlueCat Linux Root File System Utility— mkrootfs

`mkrootfs` is a cross-development tool that creates a single file (that is, the image containing a root file system that can be booted on a target board). Depending on selected options, `mkrootfs` creates an image of one of the following types:

- *Bootable*—A compressed ext2 or ext3 RAM file system image that can be directly booted onto the target board. When the kernel is booted on the target board, it automatically unpacks the file system into RAM.
- *Installable*—A tar image that can be copied into a hard disk on the target board or NFS-mounted and then used to boot the embedded system.
- *Downloadable to target Flash memory*—A Journalling Flash File System (JFFS) or Journalling Flash File System (JFFS2) image that can burned into target Flash memory and then used to boot the system.
- *Burnable*—An image containing an ISO9660 read-only root file system that can be burned into a blank CD-R(W) and then booted on the target.

Refer to Chapter 3, “Downloading and Booting BlueCat Linux” for a detailed discussion of the BlueCat Linux boot procedure. Also refer to “mkrootfs” in Appendix A, “Command Reference.”

mkrootfs Specification File

The `mkrootfs` utility creates a root file system that is described in a specification file. The specification file defines the exact layout of the files and directories included in the root file system.

The `mkrootfs` specification file syntax includes a comprehensive set of commands to create a minimal file system for the target board. For instance, `mkrootfs` allows for the creation of a directory and placement of individual files in that directory. Alternatively, the user can copy an entire directory recursively into the target board root file system and then remove certain files and subdirectories that are not needed for custom applications.

For each file and directory, the owner ID and access permissions can be specified as appropriate for the application at hand. Device nodes and symbolic links can be established for the creation of a bootable root file system.

If the `-l` option is specified in the call to `mkrootfs`, it automatically includes all the shared libraries required for all the tools and programs included in a root file system. This is done by scanning each executable and including in the file system all the shared libraries on which it depends.

Images Created by mkrootfs

If the `-T` option is specified in the call to `mkrootfs`, it creates a tar file containing all the files and directories that make up the root file system. This tar file can be copied as-is into a partition on a hard disk, and then mounted as a root file system when BlueCat Linux boots on the target board. Alternatively, the tar file can be copied to an NFS server and then NFS mounted as the target root file system.

If the `-J` option is specified, `mkrootfs` creates a Journalling Flash File System (JFFS) image. This image can be burned into target Flash memory and then mounted as a root file system when BlueCat Linux boots on the target board.

If the `-j` option is specified, `mkrootfs` creates a Journalling Flash File System version 2 (JFFS2) image.

If the `-O` option is specified, `mkrootfs` creates a CD-ROM image with a ISO9660 file system on it.

If no option is specified, `mkrootfs` creates a compressed ext2 root file system image. If the `-x` option is specified, `mkrootfs` creates a compressed ext3 file system image. The file system image can be copied on a floppy disk, a hard disk, a CD-ROM, or a ROM. When BlueCat Linux boots on the target board, it automatically uncompresses the file system image to RAM and mounts it as the ext2/ext3 root file system.

All file system parameters (such as file system size, the number of inodes, and so on) are set according to the file tree layout described in the specification file (refer to “mkrootfs Specification File” on page 50 for more information about the `mkrootfs` spec file). By default, `mkrootfs` reserves as free blocks only a small fraction of the total number of used blocks in file system. The total number of blocks is calculated using the formula:

```
max (500, 100 + (size + size/10)/1024)
```

where `size` is the amount of data to be put in the file system.

The number of reserved blocks can be increased by specifying the `-r` option to `mkrootfs`.

For instance, issue the following command to reserve 500 free blocks on the target file system, in addition to the normally reserved amount (refer to the formula above):

```
# mkrootfs -lv -r 500 osloader.spec osloader.rfs
```

Managing Multiple Embedded Applications

As described earlier, a typical BlueCat Linux system is composed of the following components:

- The BlueCat Linux kernel customized for the embedded application; the kernel configuration of a particular kernel image is fully defined by a `.config` file.
- The root file system containing all the tools and custom programs used by the embedded application; the root file system is fully defined by a specification file and may contain application programs.

The simplest approach to develop and manage multiple embedded systems is to maintain system-specific `.config` and specification files in a separate directory. The same approach is used for the BlueCat Linux demo systems. The `mkkernel` tool is used to rebuild kernels for embedded systems. The `mkrootfs` tool is used to build root file system images.

Optimizing Footprint

Customizing the Kernel for Size

The user can configure the BlueCat Linux kernel to include only those kernel features that are needed for the embedded application. Getting rid of unnecessary kernel features may often result in considerable reduction of the kernel image size.

Using mkrootfs to Build a Minimal File System

The `mkrootfs` tool lets the user handpick files and directories included in the target board file system used by the embedded BlueCat Linux system. If the size of the target board file system is an important factor for the embedded system, optimize the `mkrootfs` specification file to include only those files and directories that are absolutely necessary.

For instance, if the specification file uses a `cp` command to include an entire directory in the target board file system and there are files in that directory that are not needed for the application, use the `rm` command to remove them. Alternatively, copy files one at a time instead of copying entire directories.

Discarding Symbols from Files

Use the `strip on/off` command in the `mkrootfs` specification file to choose program files and libraries to be stripped of symbols. When file stripping is on, all subsequent copy commands are performed using appropriate stripping options. If the file is an executable, all symbols are removed (the `-S` option in `objcopy`). If the file is a library, only the debugging symbols are stripped (the `-g` option).

Stripping symbols saves a considerable amount of space in the resulting target board file system. In general, one should not strip symbols of only those executables and libraries that need symbols because of debugging or dynamic/runtime linking requirements. All other files may be safely stripped.

Getting Necessary Shared Libraries

Use the `mkrootfs -l` option to turn on automatic adding of all the shared libraries used by the tools and programs included in the target board file system. With this option, each executable is scanned and all of the shared libraries it depends on are added to the target board file system. Furthermore, after the target board file system is built, `mkrootfs` runs the `ldconfig` utility to create the cache and all appropriate symbolic links.

Using the `-l` option ensures that the target board file system contains only those shared libraries that are used in the embedded application.

By default all shared libraries are debug-stripped on copy.

Using Static Libraries

If the embedded system configuration is small and the target board runs a single application program, or multiple different application programs but not concurrently, using shared libraries may be a more expensive option in terms of disk and memory space usage, than statically linking libraries into the application programs. This is because statically linked programs can be fully stripped, while shared library versions must retain all symbols. Carefully weigh trade-offs to decide whether to use shared libraries in an embedded configuration.

By default, The GNU Cross Compiler (`gcc`) makes an executable program use shared libraries. If statically linking an application is desired, use the `-static` option when calling `gcc`.

Using the Memory Sizing Benchmark

Use the BlueCat Linux memory sizing benchmark to determine the memory requirements for an embedded system. The memory sizing benchmark is based on extensions to the Linux kernel that are dedicated to the task of collecting various memory usage statistics. The extensions can be configured into the Linux kernel using the `CONFIG_BLUECAT_MEMSIZE` kernel configuration option.

When the memory sizing option is enabled in the kernel, the following runtime memory usage statistics are collected in a number of files residing in the `/proc` area on the target board:

- `/proc/memstattotal`—System-wide memory usage statistics
- `/proc/memstatproc`—Process-specific statistics for currently running processes
- `/proc/memstathist`—Process-specific statistics for finished processes
- `/proc/memstattracepage`—Traceback of kernel components that have allocated significant amounts of RAM

To get access to the data collected in the `/proc` files, simply read the contents using standard UNIX commands. For instance:

```
bash# more /proc/memstattotal
bash# more /proc/memstathist
```

The format of each `/proc` file maintained by the benchmark is described below:

- `/proc/memstattotal` contains the following fields:
  - `Total`—Total amount of RAM controlled by BlueCat Linux
  - `Used`—Memory allocated by the kernel for internal data, processes data, caches, and others
  - `Required`—Memory allocated by the kernel excluding memory allocated for the buffer and page caches (Minimum recommended size for each cache is also displayed.)
  - `Buffer`—Memory used by the buffer cache (always set to 0)
  - `Cache`—Memory used by the page cache
  - `Shared`—Memory taken by shared memory regions
  - `Swap`—Swap space currently in use
  - `VMAlloc`—Kernel memory allocated via `vmalloc`

  - `KMAlloc`—Kernel memory allocated via `kmalloc`

  In addition to the current values for each field, except for `Total`, the file shows a maximum value reached by the respective field since the most recent read of the file (or system boot). Tracebacks for kernel calls during which the maximum value was achieved are listed at the end of the file.

- `/proc/memstatproc` shows the following fields for each running process:

  - `Process`—Process name and PID
  - `Libraries`—Libraries loaded by the process
  - `Total`—Total amount of virtual memory allocated for the process
  - `Stack`—Stack size
  - `Data`—Size of process data including static data and heap
  - `Locked`—Memory locked by the process (that is, process memory that cannot be swapped out to disk)
  - `Mapped`—Size of memory regions mapped from files (including the binary image and libraries)
  - `Shared`—Memory shared with other processes

  For all the fields, the current and maximum values are shown.

- `/proc/memstathist` contains the same information as `/proc/memstatproc` but only for the finished processes (only maximum values are listed).

- `/proc/memstattracepage` contains information about memory allocators in the kernel. Each line of this file has the following format:

  ```
  nnnnnnnn 0xaaaaaaaaa 0xaaaaaaaaa ....
  ```

  where `nnnnnnnn` is the total number of requested pages (in decimal), and `0xaaaaaaaaa ....` is a traceback of the memory allocator.

  The memory tracking facility requires some memory itself. For example, the memory for the text reports is allocated via `vmalloc`.

NOTE: A read of each file clears the corresponding statistics. The user has to use the system to its maximum memory allocation.

CHAPTER 3 *Downloading and Booting BlueCat Linux*

This chapter explains the downloading and booting of BlueCat Linux onto embedded systems. *All boot scenarios are exemplified using an x86 target board.*

The BlueCat Linux OS loader is discussed in detail.

BlueCat Linux Boot Procedure Overview

BlueCat Linux can be booted on the target board in one of the following ways:

- Copying BlueCat Linux onto a floppy disk and then booting BlueCat Linux onto the target board from the floppy disk.
- Copying BlueCat Linux onto a hard disk and then booting the target board from the hard disk.
- Copying BlueCat Linux onto a CD-ROM and then booting the target board from the CD-ROM.
- Downloading BlueCat Linux into target ROM/Flash memory and then booting the target board from ROM/Flash memory.
- Booting BlueCat Linux onto the target board from a network using the target board firmware.
- Booting BlueCat Linux onto the target board from a network using the PXE Netboot
- Copying the OS loader to floppy, hard disk, CD-ROM, or ROM/Flash and booting the target with the OS loader. Then booting BlueCat Linux from parallel port or network.
- Creating the bootable partitioned hard disk image with the partitions containing the root file system and optional additional file systems. The image can be created directly on the hard disk device or as a file, which

can be copied to a bootable device (a hard disk, a CF card or USB Flash drive) outside of the BlueCat environment.

Please refer to Figure 2-1 and Figure 2-2 on page 34.

mkboot Cross-Development Tool

The `mkboot` cross-development tool can be used to copy bootable images of BlueCat Linux onto a floppy disk, hard disk, or prepare a bootable CD-ROM image of BlueCat Linux from the cross-development host. As soon as `mkboot` completes the copy successfully, the floppy disk or hard disk can be connected to the target board, which can then be booted from the newly connected disk. If `mkboot` is used to prepare a bootable CD-ROM image, the image can be burned to a CD-R(W) disk.

`mkboot` is capable of the following functionalities:

- Installing BlueCat Linux boot sector
- Installing compressed BlueCat Linux kernel
- Installing compressed root file system image
- Defining the root device to be mounted by the kernel
- Setting the command line to be passed to the kernel
- Creating an image composed of a BlueCat Linux kernel and a compressed file system, suitable to be programmed into ROM/Flash or downloaded over a network by the target firmware
- Creating a bootable CD-ROM image
- Creating a bootable target image file that can be copied to a bootable device outside of the BlueCat Linux environment
- Creating a partitioned disk image with an optional BlueCat Linux kernel

For more information, refer to the section “mkboot” of Appendix A, “Command Reference.”

BlueCat Linux OS Loader

Before outlining various boot methods, this section introduces the BlueCat Linux OS loader used in several of the boot scenarios.

The OS loader resides in the `$BLUECAT_PREFIX/demo` directory. Look in the `$BLUECAT_PREFIX/demo/osloader` directory for the files required to build the BlueCat Linux OS loader.

The OS loader is a special configuration of BlueCat Linux that is designed as a firmware-level tool for downloading and booting a BlueCat Linux application on a target board. The OS loader provides a shell-like utility on the target that allows for setting the target IP address and the kinds of images to be downloaded.

The OS loader is capable of booting BlueCat Linux over a network, using such protocols as BOOTP, TFTP and NFS. (See “Boot Mechanism using the OS loader” above.) Alternatively, the OS loader is capable of booting BlueCat Linux from a cross-development host over a parallel port.

This capacity provides a convenient and flexible cross-development environment because the user does not need to install BlueCat Linux onto the target board every time the system or applications are updated. The advantage of the OS loader is that it can be fitted on a floppy disk and yet it maintains the ability to boot BlueCat Linux on all targets.

Once system development is complete, the OS loader supports downloading of a BlueCat Linux system onto the hard disk or Flash memory on the target board. Once BlueCat Linux is downloaded, the OS loader can be removed and the final system boots up directly from the disk. (See “Boot Mechanism using the OS loader.”)

Structurally, the OS loader is a combination of the BlueCat Linux kernel and the BlueCat Linux Loader Shell (BLOSH) command interpreter. BLOSH provides a command line interface with its own set of utilities and environment variables to implement booting.

The OS loader is available in full source. It can be enhanced to support custom boot devices and protocols.

BlueCat Linux Loader Shell (BLOSH)

The BlueCat Linux OS loader is based on a shell-like utility, BlueCat Linux Loader Shell (BLOSH), which provides the user interface for booting BlueCat Linux. The BLOSH shell and the BLOSH prompt (>) are launched when `osloader` boots up on a target board.

BLOSH Startup Sequence

The BlueCat Linux OS loader launches BLOSH as an `init` process. Thus, initially, BLOSH is the only process that runs on the target board. As with any `init`-like process, BLOSH is started after the `osloader` kernel has mounted the root file system in the RAM disk.

At startup, BLOSH reads the configuration file `/etc/blosh.rc` and executes commands contained in that file. If no `/etc/blosh.rc` file is present in the root file system, no commands are executed.

Having completed the processing of `/etc/blosh.rc`, BLOSH enters interactive mode and prompts the user for command input.The user can then use BLOSH commands, and set environment variables to point to the BlueCat Linux system to be downloaded. If interactive operation with BLOSH is not desired, place all required commands in `/etc/blosh.rc`.

BLOSH Environment Variables

BLOSH uses a number of environment variables to configure the BlueCat Linux boot process. These environment variables may be set up by the `osloader` kernel or explicitly defined by the user via the BLOSH `set` command.

The following environment variables are used by BLOSH:

- `CMD`—Command line to be passed to the kernel booted by BLOSH (empty by default)
- `FILE`—File downloaded to the RAM disk-based root file system by the `read` command. This variable has the same format as the `KERNEL` environment variable. If IP autoconfiguration is enabled in the BlueCat Linux OS loader, this variable is set automatically.
- `HOME`—Default working directory
- `HOST`—IP address of the cross-development host from which BLOSH downloads BlueCat Linux images. If IP autoconfiguration is enabled in the BlueCat Linux OS loader, this variable is set automatically.

- IF—Name of the network interface used by BLOSH to download BlueCat Linux images
- IP—IP address of the target board. If IP autoconfiguration is enabled in the BlueCat Linux OS loader, this variable is set automatically.
- KERNEL—BlueCat Linux kernel image to be loaded by the boot command; the format of this variable is as follows:

  <boot_type> <type_specific_parameters>

  The following *<boot_types>* are supported:

  - file <filename> boots image from the specified file in the local file system
  - tftp *<filename>* boots image from the specified file on the TFTP server
  - nfs *<directory>* *<filename>* boots image from the specified file in a specified directory on the NFS server
  - pftp *<filename>* boots image from the specified file on the PFTP server
- PPORT—Name of a character device that represents a parallel port from which to load BlueCat Linux image. In the standard configuration, this device is /dev/bpar0.
- RFS—BlueCat Linux root file system image loaded by the boot command - this variable has the same format as the KERNEL environment variable.

Enabling Networking

Setting Up a DHCP Server

The following steps are necessary to set up the DHCP daemon on a *Linux* host:

Information about the IP and hardware addresses of the servers to be autoconfigured using the DHCPD protocol must be added to the /etc/dhcpd.conf file. Each server is described in the following format:

```
allow bootp;
subnet  192.168.111.0   netmask        255.255.255.0 {
        option subnet-mask             255.255.255.0;
```

```
        option domain-name-servers      207.21.185.10

        host matrix2 {
                hardware ethernet       00:30:23:00:00:01;
                filename                "/tftpboot/ATC/uImage";
                fixed-address           192.168.111.2;
                option host-name        "matrix2";
        }

}
```

For more detailed information on the `dhcpd.conf` file format, refer to the `dhcpd.conf(5)` man page.

If the `dhcpd` daemon is not already enabled, do the following:

1. Install the `dhcp-3.0pl1-23.rpm` package on the Linux host.
2. Use the following commands to enable `dhcpd` on startup:

   `chkconfig --add dhcpd`
   `chkconfig dhcpd on`

   or:

   Start it manually:

   `/etc/rc.d/initd/dhcpd start`

Setting Up a TFTP Server

To enable TFTP on a *Linux* host system, perform the following steps:

1. Edit `/etc/ethers` to include the hexadecimal Ethernet address and the hostname of the target board. The *`<target_hostname>`* is the user-assigned name of the target system. The hex number is a MAC address; every Ethernet card has one assigned from the manufacturer, which is usually printed on the Ethernet card.

   `# ` **`vi /etc/ethers`**

```
# hex_target_ethernet_address    target_hostname
  08:00:3E:23:8C:BD              fpc1
```

Figure 3-1: /etc/ethers File

2. Edit the `/etc/hosts` file to include the hostname and IP address of the target board:

```
# vi /etc/hosts
```

```
# target_IP_address         target_hostname
192.168.1.2                 fpc1
```

Figure 3-2: /etc/hosts File

3. Create the `/tftpboot` directory:

```
# mkdir /tftpboot
```

4. Enable TFTP by editing the `/etc/xinetd.d/tftp` file.

```
# cd /etc/xinetd.d/
# vi tftp
```

5. In the `disable` field, type **no** to enable TFTP.
6. In the `server_args` field type **/tftpboot**.

   The following is a sample `tftp` file.

```
# default: off
# description: The tftp server serves files using the trivial file transfer \
# protocol. The tftp protocol is often used to boot diskless \
# workstations, download configuration files to network-aware printers, \
# and to start the installation process for some operating systems.
service tftp
{
socket_type = dgram
protocol = udp
wait = yes
user = root
server = /usr/sbin/in.tftpd
server_args = /tftpboot
disable = no
}
```

Figure 3-3: Sample tftp Configuration File

7. Manually restart the `xinetd` services:

```
# cd /etc/rc.d/init.d
# ./xinetd restart
```

Setting Up an NFS Server

To enable the BlueCat Linux OS loader to download kernel and root file system images from an NFS server, the directory that the images reside in must be exported from the NFS server.

1. To export a directory from a *Linux server*, a line in the following format must be added to the `/etc/exports` file:

   ```
   /nfsboot <target_or_client_ip>(wr,no_root_squash)
   ```

2. If `/etc/exports` has been modified, the `exportfs` utility must be run in order for the changes to take effect.

For more detailed information on the `/etc/exports` file format and the `exportfs` utility, refer to the corresponding man pages.

Booting Images from a Different Subnet

To enable the BlueCat Linux OS loader to download the kernel and root file system images from an NFS or TFTP server on a different subnet, perform the following steps:

1. Turn on the `CONFIG_IP_PNP` kernel configuration option in the `osloader.config` file. This is done by running `make menuconfig` in the `osloader` directory and enabling the **IP:kernel level autoconfiguration** option in the **Networking Options** submenu of the **Networking** menu.

2. Rebuild the OS loader and copy it on bootable media, passing it a kernel command line in the following format:

   ip=*<client_ip>*:*<server_ip>*:*<gw_ip>*:*<netmask>*: \
   <hostname>:*<device>*:*<autoconf>*

   This command line provides the Linux kernel with routing information. For example:

   ```
   ip=1.0.3.2:172.16.1.2:1.0.3.1::test2:eth0
   ```

   The command line is passed to the BlueCat Linux kernel using the `-c` option of the `mkboot` utility. Refer to the `mkboot(1)` man page for a detailed description of the `-c` option.

3. Boot the OS loader on the target board. The OS loader now has all the routing data required to load images from a different subnet.

NOTE: The value of the `IP` BLOSH environment variable must match the `client_ip` field specified in the kernel command line.

Setting Up a PFTP Server

To enable the parallel port booting feature of BlueCat Linux, the PFTP server must be run on the cross-development host. Please refer also to the section “pftpd” of Appendix A, “Command Reference.”

1. To start the PFTP server, execute the following command in the BlueCat Linux environment on the cross-development host:

   ```
   BlueCat:bash$ pftpd start
   ```

2. To stop the PFTP server, execute the following command in the BlueCat Linux environment:

   ```
   BlueCat:bash$ pftpd stop
   ```

3. Perform additional steps depending on the host operating system:

   - Linux

     Before running the PFTP server on the Linux host, the low-level driver must be loaded to allow the daemon to access the parallel port. To load the low-level driver, execute the following commands under a superuser account:

     ```
     BlueCat:bash$ cd
     $BLUECAT_PREFIX/cdt/lib/pftpd/module
     BlueCat:bash$ su
     Password: <root_password>
     BlueCat:bash# bash install.sh
     ...
     The module has been installed successfully.
     BlueCat:bash# exit
     ```

This action must be repeated after each reboot.

NOTE: On some Linux systems, the parallel port is not configured by default, which prevents the server from finding the port. To fix this, the line `alias parport_lowlevel parport_pc` must be added to `/etc/conf.modules` or `/etc/modules.conf` (Choose the file that exists on the Linux installation.).

- Windows XP Professional

  To start the server, log in as a system administrator.

NOTE: Before using the PFTP server on an x86 machine, the parallel port may need to be configured in BIOS. Some ports require setting the port mode to EPP, ECP, or ECP/EPP to allow the server to function. For additional information on configuring the port mode in BIOS, refer to the BIOS user's manual.

NOTE: The `pftpd` daemon running on the Windows host interprets paths in a Cygwin notation taking into account mount table entries. In other words, if `pftproot` is set to `/` in the `$BLUECAT_PREFIX/cdt/etc/pftpd.rc` file and the mount table contains a standard entry `C:\cygwin32 on /`, the `pftpd` daemon will search for files to download in the `C:\cygwin32` directory. If search in `C:\` is needed, the `pftproot` variable must be set to `/cygdrive/c`.

BlueCat Linux Boot Scenarios

In all the boot scenarios outlined below, be sure to enable the BlueCat Linux environment by running `SETUP.sh <bsp_name>`. Also, make note of the file formats and BLOSH environment variables used in each boot scenario.

Booting BlueCat Linux from a Floppy Disk

This section explains booting BlueCat Linux on the target from a floppy disk.

Copying onto Floppy Disk

To create a bootable floppy disk containing BlueCat Linux, perform the following steps:

NOTE: If the procedure described below is used on the Windows host, `/dev/fd0` must be replaced with `a:`.

---

1. On the cross-development host, insert a floppy disk into the floppy drive corresponding to the `/dev/fd0` special node.
2. For Linux host only:

   Write permission is required to write to `/dev/fd0`. If needed, do the following as a superuser:

   ```
   # chmod ugo+rw /dev/fd0
   ```

3. Copy BlueCat Linux onto the floppy disk using the `mkboot` utility. For example, to copy the BlueCat Linux OS loader, execute the following commands:

   a. Go to the `$BLUECAT_PREFIX/demo/osloader` directory and copy the BlueCat Linux boot sector to floppy disk using `mkboot -b`:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
   BlueCat:$ mkboot -b /dev/fd0
   ```

   b. Copy the compressed kernel to the medium using `mkboot -k`:

   ```
   BlueCat:$ mkboot -k osloader.disk /dev/fd0
   ```

   c. Copy the compressed root file system image to the medium using `mkboot -f`:

   ```
   BlueCat:$ mkboot -f osloader.rfs /dev/fd0
   ```

   d. Set the device node on the target board to mount as the root file system or uncompress the file system image using `mkboot -r`:

   ```
   BlueCat:$ mkboot -r /dev/fd0 /dev/fd0
   ```

   For more information on the `mkboot` command, refer to the section “mkboot” of Appendix A, “Command Reference.”

Floppy Disk Boot Mechanism

On an Intel-based PC target board, booting BlueCat Linux from a floppy disk works as follows. After the user has inserted the floppy disk in the drive and pressed the **Reset** button:

1. First, the BIOS loads the first sector (the boot sector) of the floppy disk, and executes the code found there.
2. The boot loader found in the boot sector loads the compressed BlueCat Linux kernel, the kernel command line, and the setup code.
3. The setup code is called.
4. The setup code obtains certain system parameters from the BIOS (memory size, and so on), and stores them for the BlueCat Linux kernel.
5. It then enters protected mode and calls the BlueCat Linux kernel entry point.
6. The kernel decompresses itself and begins the OS bootstrapping process.
7. Depending on the root device settings and command line parameters, the root file system is mounted from a hard disk, a Journalling Flash File System (JFFS)/Journalling Flash File System version 2 (JFFS2), a network file system using NFS, or the compressed root file system image on the floppy disk is decompressed into a RAM disk and mounted as the root file system.

NOTE: In the case of BlueCat Linux OS loader, the system boots up to display the OS loader prompt (>).

Booting BlueCat Linux from a Hard Disk or DiskOnChip

This section explains how to boot BlueCat Linux on the target from a hard disk and DiskOnChip.

Copying onto Hard Disk (Linux Host only)

To copy a BlueCat Linux system onto a hard disk from the cross-development host, perform the following steps:

1. Attach the hard disk (primary slave) to the cross-development host.

2. Create a boot partition to hold the kernel. This operation requires root privileges, so be sure to switch to the `root` account.

   ```
   # fdisk /dev/hdb
   ```

   and then proceed to create a partition on the disk.

NOTE: `fdisk` shows the number of bytes contained in a disk cylinder. Use this number to calculate a boot partition size in cylinders sufficient for the compressed kernel image.

Failure to allocate sufficient space results in BlueCat Linux crashing at boot.

If the *boot is from the compressed root file system*, copying onto the hard disk is exactly the same as for floppy disk, except that the hard disk device node for x86 is used instead of the floppy device node.

3. Obtain write permission for the hard disk. Then proceed to copy the `i_osloader` version of the BlueCat Linux OS loader onto the disk. Enter:

   ```
   # chmod ugo+rw /dev/hdb
   ```

   Then do the following:

   a. Go to the `$BLUECAT_PREFIX/demo/osloader` directory and copy the BlueCat Linux boot sector to the hard disk using `mkboot -b`:

      ```
      BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
      BlueCat:$ mkboot -b /dev/hdb
      ```

   b. Copy the compressed kernel to the medium using `mkboot -k`:

      ```
      BlueCat:$ mkboot -k i_osloader.disk /dev/hdb
      ```

   c. Copy the compressed root file system image to the medium using `mkboot -f`:

      ```
      BlueCat:$ mkboot -f i_osloader.rfs /dev/hdb
      ```

   d. Set the device node on the target board to mount as the root file system or uncompress the file system image using `mkboot -r`:

      ```
      BlueCat:$ mkboot -r /dev/hda /dev/hdb
      ```

   For this boot scenario, this step completes the copy.

   If the *file system is mounted from a partition on the hard disk*, perform the steps described below.

4. Create the file system on the newly made partition:

```
# mke2fs /dev/hdb2
```

5. Copy the root file system from the tar file created by `mkrootfs -T` to the newly made partition. For instance, the following commands copy the root file system of the OS loader:

```
# mkdir /mnt1
# mount /dev/hdb2 /mnt1
# cd /mnt1
# tar xvf <BlueCat_Linux_installation_point>/demo/\
osloader/i_osloader.tar
```

6. Unmount the hard disk:

```
# cd /
# umount /mnt1
```

7. At this point, return to the BlueCat Linux environment. Copy the BlueCat Linux boot sector and kernel to the hard disk using the `mkboot` tool:

```
BlueCat:$ mkboot -b -k i_osloader.disk /dev/hdb
BlueCat:$ mkboot -r /dev/hda2 /dev/hdb
```

   These commands make the kernel reside at the beginning of the disk, and configure it to mount `/dev/hda2` as the root file system. (See “mkboot Cross-Development Tool” on page 58.)

8. Shut down the cross-development host, disconnect the hard disk, and attach the disk to the target board.

Installing on Hard Disk Using the Disk Image

To install BlueCat Linux on the hard disk using the disk image, perform the following steps:

1. Attach the hard disk to the target board as described in the previous section.

2. Create a disk image using the `mkboot` utility:

```
BlueCat:$ mkboot -b2 -k i_osloader.disk -f \
i_osloader.rfs disk.img
```

The disk image containing two partitions (the compressed kernel partition and the noncompressed RFS partition) is created.

NOTE: An image with three and more partitions can be created using the `mkboot` utility. For more information on the `mkboot` command, refer to the section "mkboot" on page 196.

---

3. Copy the image to the hard disk.

```
BlueCat:$ dd if=disk.img of=/dev/hdb
```

Adding Support for DiskOnChip in the BlueCat Linux Kernel

Support for a DiskOnChip device in BlueCat Linux is based on the Linux Memory Technology Devices (MTD device drivers).

To enable support for DiskOnChip in BlueCat Linux, perform the following steps:

1. Run the `make menuconfig` tool:

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
BlueCat:$ make menuconfig
```

2. Go to the **Device Drivers** menu.
3. In the **Memory Technology Devices** submenu turn on the following configuration options:
   - **Memory Technology Device (MTD) support** (`CONFIG_MTD`)
   - **MTD partitioning support** (`CONFIG_MTD_PARTITIONS`)
   - **NFTL (NAND Flash Translation Layer) support** (`CONFIG_NFTL`)
   - **Write support for NFTL** (`CONFIG_NFTL_RW`).
4. In the **Self-Contained MTD device drivers** submenu turn on one of the following three options:
   - **M-Systems Disk-On-Chip 2000 and Millennium** (`CONFIG_MTD_DOC2000`)
   - **M-Systems Disk-On-Chip Millennium-only alternative drv** (`CONFIG_MTD_DOC2001`)
   - **M-Systems Disk-On-Chip Millennium plus** (`CONFIG_MTD_DOC2001PLUS`)

5. In the **NAND FLASH DEVICE DRIVERS** submenu turn on the following configuration options:
   - **NAND Device Support** (`CONFIG_MTD NAND`)
   - **DiskOnChip 2000, Millennium and Millennium Plus (NAND reimplement)** (`CONFIG_MTD_NAND_DISKONCHIP`)
   - **Allow BBT writes on DiskONChip Millennium and 2000TSOP** (`CONFIG_MTD_NAND_DISKONCHIP_BBTWRITE`)
6. Rebuild the kernel.

   The DiskOnChip driver is now enabled in the kernel.

Copying BlueCat Linux to DiskOnChip

The DiskOnChip presents itself to the system as a hard disk. It can be accessed via device nodes with the major number `93` and the minor number corresponding to a disk partition (`0` for entire disk, `1` for the first partition, and so forth).

The `osloader` demo system already has special device files for the DiskOnChip driver integrated in the root file system. They are named `/dev/nftl`, `/dev/nftl1`, `/dev/nft2`, `/dev/nft3`, and `/dev/nft4`. If support for DiskOnChip is added to the kernel (see “Adding Support for DiskOnChip in the BlueCat Linux Kernel” on page 71.), it can be enabled in the `osloader` demo system and used to copy BlueCat Linux onto the DiskOnChip device using the same procedure as for a conventional hard disk.

Some important notes on copying BlueCat Linux onto DiskOnChip:

- After partitioning DiskOnChip using `fdisk`, the system must be reset in order for the new partitioning to take effect.
- If the target board has both a DiskOnChip and an ordinary hard disk, there may be a need to reprogram the DiskOnChip firmware with alternative images. These images, provided by M-Systems, ensure a correct search sequence for the disk boot devices on the target board, as

dictated by the existing application. These issues are described in detail in the documentation that comes with the DiskOnChip hardware.

NOTE: The most convenient way to copy BlueCat Linux onto a DiskOnChip device is to boot BlueCat Linux from the floppy disk on a system with a floppy controller and a DiskOnChip. Copy the target BlueCat Linux system onto DiskOnChip, and then move the DiskOnChip device to the otherwise diskless target board.

Booting from Hard Disk and DiskOnChip using Network and OS Loader

Root File System Copied to Partition on Target Hard Disk or DiskOnChip

To download BlueCat Linux from a TFTP server onto a target hard disk using the OS loader, perform the following steps:

1. Attach the hard disk to the target board.
2. Copy the `osloader` version of the OS loader in bootable form onto a floppy disk. See “Copying onto Floppy Disk” on page 67.
3. Boot the `i_osloader version` of the BlueCat Linux OS loader on the target board over a network See “Booting BlueCat Linux over a Network or Parallel Port” on page 96.

NOTE: Make sure that support for the type of boot disk being used is configured in the OS loader. If necessary, reconfigure the OS loader to add support for the hard disk.

A successful boot brings up the BLOSH prompt on the target board console (`>`).

4. Create two partitions on the hard disk to hold the kernel and the root file system using `fdisk`:
   - To create partitions on an IDE hard disk:

     ```
     > exec fdisk /dev/hda
     ```

     and then proceed to create two partitions: `/dev/hda1` to hold the kernel and `/dev/hda2` to hold the root file system.

- To create partitions on DiskOnChip:

  ```
  > exec fdisk /dev/nftl
  ```

  and then proceed to create two partitions: `/dev/nftl1` to hold the kernel and `/dev/nftl2` to hold the root file system.

  When copying onto a DiskOnChip device the target board must be reset after partitioning:

  ```
  > sync
  > sync
  > reset
  ```

5. Create a file system on the newly made partition. For example,

   - For an IDE hard disk:

     ```
     > exec mke2fs /dev/hda2
     ```

   - For DiskOnChip:

     ```
     > exec mke2fs /dev/nftl2
     ```

6. Mount the partition on the hard disk. For example,

   - For an IDE hard disk:

     ```
     > mount /dev/hda2 /mnt
     ```

   - For DiskOnChip:

     ```
     > mount /dev/nftl2 /mnt
     ```

7. Set the BLOSH environment variables, specifying the BlueCat Linux system to copy. For example:

   ```
   > set IP 1.0.3.2
   > set HOST 1.0.3.1
   > set IF eth0
   > set KERNEL tftp /tftpboot/<system>.disk
   > set FILE tftp /tftpboot/<system>.tar
   ```

   where:

   - `IP` is the target IP address.
   - `HOST` is the cross-development host IP address.
   - `IF` points to the Ethernet interface for the TFTP server being used to download BlueCat Linux.
   - `KERNEL` points to the kernel image of the system to be booted.

- `FILE` points to the tar image of the root file system.
- *<system>* is to be substituted by the name of the BlueCat Linux system.

8. Untar the root file system and copy it from the TFTP server:

```
> cd /mnt
> ntar
```

9. Copy the kernel image, specifying the root file system. (See "mkboot Cross-Development Tool" on page 58.)
   - For an IDE hard disk:

```
> mkboot -b -r /dev/hda2 /dev/hda
```

   - For DiskOnChip:

```
> mkboot -b -r /dev/nftl2 /dev/nftl
```

10. Remove the boot floppy containing `i_osloader` and reset the target board. This boots the BlueCat Linux *<system>* on the target.

```
> sync
> sync
> reset
```

Compressed Root File System Copied to Hard Disk or DiskOnChip

The procedure in "Root File System Copied to Partition on Target Hard Disk or DiskOnChip" describes copying a root file system into a partition on the hard disk. Alternatively, BlueCat Linux can be booted from a TFTP server using a compressed root file system image copied onto a hard disk.

To copy BlueCat Linux with a compressed root file system onto a hard disk attached to the target board using the OS loader, perform the following steps:

1. Attach the hard disk to the target board.
2. Copy the OS loader on a floppy disk. The `osloader` demo system can be found in the `$BLUECAT_PREFIX/demo/osloader` directory.

NOTE: Make sure that support for the boot disk being used is configured in the OS loader. If necessary, reconfigure the OS loader to add support for the hard disk.

---

3. Boot the target board with the OS loader.

   A successful boot displays the BLOSH (BlueCat Linux Loader Shell) prompt on the target board console (>).

4. Create a boot partition to hold the kernel. For example,

   - For an IDE hard disk:

     ```
     > exec fdisk /dev/hda
     ```

   - For DiskOnChip:

     ```
     > exec fdisk /dev/ntfl
     ```

NOTE: `fdisk` shows the number of bytes contained in a disk cylinder. Use this number to calculate a boot partition size in cylinders sufficient for the compressed kernel image.

Failure to allocate sufficient space results in BlueCat Linux crashing at boot.

   When copying onto a DiskOnChip device, the target board must be reset after partitioning:

   ```
   > sync
   > sync
   > reset
   ```

5. Set the BLOSH environment variables, specifying the BlueCat Linux embedded system to copy. For example:

   ```
   > set IP 1.0.3.2
   > set HOST 1.0.3.1
   > set IF eth0
   > set KERNEL tftp /tftpboot/<system>.disk
   > set RFS tftp /tftpboot/<system>.rfs
   ```

   where:

   - `IP` is the target IP address.
   - `HOST` is the cross-development host IP address.
   - `IF` points to the Ethernet interface for the TFTP server being used to download BlueCat Linux.
   - `KERNEL` points to the kernel image of the system to be booted.
   - `RFS` points to the root file system.

- *<system>* is to be substituted by the name of the BlueCat Linux system.

6. Copy the kernel and root file system images either on an IDE hard disk or DiskOnChip. (See "mkboot Cross-Development Tool" on page 58.)

   - For an IDE hard disk:

     ```
     > mkboot -b -r /dev/hda /dev/hda
     ```

   - For DiskOnChip:

     ```
     > mkboot -b -r /dev/nftl /dev/nftl
     ```

7. Remove the floppy disk containing the OS loader and reset the target board. This boots the BlueCat Linux *<system>* on the target:

   ```
   > sync
   > sync
   > reset
   ```

Hard Disk Booting Mechanism

On an Intel-based PC system, booting BlueCat Linux from a hard disk works as follows:

1. First, the BIOS loads the first sector (the master boot record) of the hard disk, and executes the code found there.
2. The boot loader found in the first sector loads the compressed BlueCat Linux kernel, the kernel command line, and an additional piece of code, the setup code.
3. The setup code is called.
4. The setup code obtains certain system parameters from the BIOS (memory size, and so forth) and stores them for the BlueCat Linux kernel.
5. It then enters protected mode and calls the BlueCat Linux kernel entry point.
6. The kernel decompresses itself and begins the OS bootstrapping process.
7. Depending on the root device settings and command line parameters, the root file system is mounted from a hard disk or a network file system using NFS, or the compressed root file system image on the hard disk is decompressed into a RAM disk and mounted as the root file system.

Booting from a USB Flash Drive

This section explains how to boot BlueCat Linux on the target from a USB Flash drive.

Copying onto USB Flash Drive (Linux Host only)

To copy a BlueCat Linux system onto a USB Flash drive from the cross-development host, perform the following steps:

1. Attach the USB Flash drive to the host.
2. Copy the compressed kernel and root file system into the Flash drive using the `mkboot` command:

   ```
   BlueCat:$ echo rootdelay=10 > cl.txt
   BlueCat:$ mkboot -b2 -k osloader.disk -f |
   osloader.rfs -c cl.txt /dev/sda
   ```

   For more information on the `mkboot` command, refer to the section "mkboot" of Appendix A, "Command Reference."

   The bootable image with two partitions is created. The first partition contains the compressed kernel and the second partition contains the root file system (noncompressed ext2 or ext3 image).

   NOTE: The attached USB Flash drive device name can be `/dev/sdb`, `/dev/sdc`, and so on, depending upon the other USB mass-storage or SCSI devices attached to the host.

   The `rootdelay=10` kernel parameter specifies the time (in seconds) for the USB Flash device to settle before mounting. The actual delay value can be fewer depending on the particular USB device.

   NOTE: To guarantee that the actual disk partition is mounted instead of the RFS copy in the RAM disk, the **BlueCat RFS support** kernel option should be disabled in the BlueCat kernel.

3. Disconnect the USB Flash drive from the host and attach it to the target.

Booting the BlueCat Linux System from the USB Flash Drive

To boot the BlueCat Linux system from the USB Flash drive, perform the following steps:

1. Connect the USB Flash drive to the target.
2. Reset the target and enter the BIOS configuration menu.
3. In the BIOS configuration menu, set the USB Flash drive as the first boot device.
4. Save settings and reboot the target.

The BlueCat Linux demo system boots on the target.

Adding Support for USB Flash Drive in the BlueCat Linux Kernel

To enable support for a USB Flash drive in BlueCat Linux, perform the following steps:

1. Run the `make menuconfig` tool:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
   BlueCat:$ make menuconfig
   ```

2. Go to the **Device Drivers** menu.
3. In the **SCSI device support** submenu, enable the following options:
   - **SCSI device support**
   - **SCSI disk support**
4. In the **USB support** submenu, enable the following options:
   - **Support for Host-side USB**
   - **EHCI HCD (USB 2.0) support**
   - **UHCI HCD (most Intel and VIA) support**
   - **USB Mass Storage support**
5. Rebuild the kernel.

The USB Flash support is now enabled in the kernel.

USB Flash Drive Booting Mechanism

On an Intel PC based target board, booting BlueCat Linux from a USB Flash drive is the same as booting from the hard disk. For the description of the booting mechanism refer to “Hard Disk Booting Mechanism” on page 77.

Booting BlueCat Linux from a CD-ROM

This section explains how to boot BlueCat Linux on the target from a CD-ROM.

Creating a Bootable CD-ROM Image

To create a bootable CD-ROM image with a BlueCat Linux system from the cross-development host, use one of the following methods.

Creating a Bootable CD-ROM image with the ISO9660 Root File System

To create a bootable CD-ROM image with an ISO9660 root file system, perform the following steps:

1. Use the `mkrootfs` utility to create an ISO9660 image with the file system. For example, to create an ISO9660 image with the OS loader file system, execute the following commands:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
   BlueCat:$ mkrootfs -lvO osloader.spec \
   osloader.isofs
   ```

   where:

   - `osloader.isofs` is the ISO9660 file system image.
   - `osloader.spec` is the file system spec file.

2. Make the created image bootable using the `mkboot` utility:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
   BlueCat:$ mkboot -o -k osloader.disk \
   -r <cdrom_device> -g osloader.isofs osloader.iso
   ```

   where:

   - `osloader.disk` is the prebuilt, compressed kernel image suitable for copying onto a CD-ROM.

- `<cdrom_device>` is a device node on the target board that specifies a CD-ROM device. It may be either `/dev/hdc` for the IDE CD-ROM or `/dev/scd0` for the SCSI CD-ROM.
- `osloader.isofs` is the ISO9660 file system image created by `mkrootfs` at the previous step.
- `osloader.iso` is the resultant bootable CD-ROM image.

For this boot scenario to work, the target kernel must be compiled with the **ISO9660 CD-ROM File System Support** option from the **File System**–>**CD-ROM/DVD File System** submenu enabled.

When creating a CD-ROM image to be booted from the IDE CD-ROM, the **IDE/ATAPI CD-ROM Support** kernel configuration option from the **Device Drivers**–>**ATA/ATAPI /RLL Support** submenu must be enabled in the target kernel. When creating a CD-ROM image to be booted from the SCSI CD-ROM, the **SCSI CD-ROM** kernel configuration option from the **Device Drivers**–>**SCSI Device Support** submenu must be enabled in the target kernel.

Creating a Bootable CD-ROM Image with the Compressed Root File System

To create a bootable CD-ROM image with a compressed root file system (`demo.rfs`), perform the following steps:

1. Use the `mkboot` utility to create a CD-ROM image with the kernel and the root file system. For example, to create a CD-ROM image with the BlueCat Linux OS loader, execute the following commands:

   ```
   BlueCat:$ mkboot -o -k osloader.disk -f \
   osloader.rfs osloader.iso
   ```

   where:

   - `osloader.disk` is the prebuilt, compressed kernel image suitable for copying onto a CD-ROM.
   - `osloader.rfs` is the prebuilt, compressed RAM disk root file system image suitable for loading from a CD-ROM.
   - `osloader.iso` is the resultant bootable CD-ROM image.

Burning BlueCat Linux to CD-ROM

It is important to understand that the `mkboot` utility is used only to create a bootable CD-ROM image. This image can be burned to a CD-R(W) disk using special software (for example, the `cdrecord` utility on the Linux host).

NOTE: The `cdrecord` utility is not included into BlueCat Linux distribution.

CD-ROM Disk Booting Mechanism

On an Intel-based PC target board, booting BlueCat Linux from a CD-ROM disk works as follows:

1. The BIOS on the target loads the CD-ROM Boot Record Volume sector (as defined by the El Torito Specification). The sector contains a reference to the boot image.
2. The Boot Record Volume is coded to specify use of no-emulation mode. That is, BIOS loads 2048 bytes of the boot image and executes the code found there.
3. The code in the boot image loads to RAM the BlueCat Linux kernel, the kernel command line, the kernel setup code and, optionally, the compressed root file system.
4. The kernel setup code is called.
5. The kernel setup code obtains certain system parameters from BIOS, and stores them in RAM for the BlueCat Linux kernel.
6. It then enters a protected mode and calls the BlueCat Linux kernel entry point.
7. The kernel decompresses itself and begins the OS bootstrapping process.
8. Depending on the boot device settings and command line parameters, the root file system is mounted from a hard disk, a ISO9660 file system on the CD-ROM, a Journalling Flash File System (JFFS/JFFS2) in Flash, a network file system using NFS, or the compressed root file system image on the CD-ROM disk decompressed by the previous steps into RAM disk.

Booting BlueCat Linux from a CompactFlash Card

This section demonstrates how to create a BlueCat Linux bootable image that can be copied to a bootable device even outside of the BlueCat Linux environment. The bootable device with the BlueCat Linux image can be used to boot the target board.

In the scenario described in this section, the CompactFlash card is used as a bootable device and the BlueCat Linux `developer` demo system as an example of a BlueCat Linux target system.

Booting BlueCat Linux on the target from a CompactFlash card consists of the following steps:

1. Creating a bootable BlueCat Linux image
2. Installing the BlueCat Linux bootable image onto the CompactFlash card from the development host (The bootable BlueCat Linux image can be copied to the CompactFlash card from both inside and outside of the BlueCat Linux environment.)
3. Booting the BlueCat Linux system from the CompactFlash card

Creating a Bootable BlueCat Linux Image

To create a bootable BlueCat Linux image, perform the following steps.

1. Go to the `demo/developer` directory and make sure that the root file system image has been built:

   ```
   BlueCat:$ cd demo/developer
   BlueCat:$ make rootfs
   ```

2. Enable IDE disk support in the kernel configuration file for the `developer` demo system. Use either the `make menuconfig` or `make xconfig` command and set the following kernel configuration options to **YES**:

   - **Device Drivers->ATA/ATAPI/MFM/RLL support->ATA/ATAPI/MFM/RLL support**
   - **Device Drivers->ATA/ATAPI/MFM/RLL support->Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support**
   - **Device Drivers->ATA/ATAPI/MFM/RLL support->Include IDE/ATA-2 DISK support**
   - **Device Drivers->ATA/ATAPI/MFM/RLL support->PCI IDE chipset support**
   - **Device Drivers->ATA/ATAPI/MFM/RLL support->Generic PCI IDE Chipset Support**

- **Device Drivers->ATA/ATAPI/MFM/RLL support->Generic PCI bus-master DMA support**

3. Build the kernel:

```
BlueCat:$ make kernel
```

4. Prepare a text file containing the command line to be passed to the kernel. It must include the `rw ramdisk_size=20000` argument. For example:

```
BlueCat:$ echo "rw ramdisk_size=20000" > cl.txt
```

5. Create a a bootable `developer` image using the `mkboot` command with either the `-b` or the `-b2` option:

   - Use the `mkboot` command with the `-b` option to create a bootable `developer` image with the compressed file system:

```
BlueCat:$ mkboot -b -k developer.disk -f \
developer.rfs -c cl.txt -r /dev/hda developer.img
```

   - Use the `mkboot` command with the `-b2` option to create a bootable `developer` image with two partitions. The first partition contains the BlueCat Linux kernel image, and the second partition contains the root file system (not compressed).

```
BlueCat:$ mkboot -b2 -k developer.disk -f \
developer.rfs -c cl.txt developer.img
```

The created `developer` image is ready to be copied to the CompactFlash card.

Installing the Bootable BlueCat Linux Image onto the CompactFlash Card

To install the bootable BlueCat Linux image onto the CompactFlash card, perform the following steps:

1. Plug the CompactFlash card into the USB CF reader/writer and connect the device to a USB port on the host. The USB storage device should become visible on the host system as a SCSI disk. In other words, it

should become accessible through one of the SCSI disk device nodes, such as `/dev/sda` or `/dev/sdb`.

NOTE: Which particular device node is used depends on the host operating system, the particular model of the USB CF adapter, and whether any other SCSI devices are attached to the system.

On the Linux host, it is possible to determine which SCSI device node has been assigned to a USB storage device (the CompactFlash adapter) by examining the `/proc/scsi/scsi` file.

---

2. To copy the BlueCat Linux image onto the CompactFlash card, do the following on the development host:

   - Linux

     Issue the following commands, assuming that the CF adapter is accessible as `/dev/sdb`:

     ```
     bash# dd if=developer.img of=/dev/sdb
     bash# sync
     ```

   - Windows

     From the public domain, download a Windows version of the utility that allows copying an image to the CompactFlash card. For instance, such utilities can be found at the following addresses:

     ```
     http://ftp.si-linux.co.jp/pub/utils/ddforwindows/DDWin_0_95.exe
     http://uranus.it.swin.edu.au/%7Ejn/linux/rawwrite/dd.htm
     ```

     Use the newly installed utility to copy the Bluecat image to the CompactFlash card.

3. Disconnect the USB CF reader and unplug the CompactFlash card.

Booting the BlueCat Linux System from the CompactFlash Card

To boot the BlueCat Linux system installed onto the CompactFlash card, perform the following steps:

1. Power down the target and install the CompactFlash card into the CF-IDE adapter on the target.
2. Power up the target.

The BlueCat Linux demo system boots on the target.

CompactFlash Booting Mechanism

On an Intel PC based target board, booting BlueCat Linux from CompactFlash is the same as booting from the hard disk. For the description of the booting mechanism refer to “Hard Disk Booting Mechanism” on page 77.

Booting BlueCat Linux From a Network Using PXE Netboot

To boot BlueCat Linux from a network using PXE Netboot, perform the following steps:

1. Set up and start the DHCP and TFTP servers.
2. Install and configure the PXELINUX server.
3. Configure PXE Netboot on the target.
4. Boot a BlueCat Linux demo system on the x86 board from a network using the PXE Netboot.

Setting Up and Starting the DHCP Server

Perform the following steps to install, configure, and start the DHCP server on a Red Hat Linux machine:

1. Set up the DHCP server as described in the *BlueCat Linux User's Guide*.
2. Add the following lines to the `/etc/dhcpd.conf` file:

```
allow booting;
allow bootp;
filename "pxelinux.0";
option dhcp-client-identifier "PXEClient";
always-reply-rfc1048 on;
```

   where `pxelinux.0` is the boot loader for Linux used by the PXE network booting protocol, and `PXEClient` is the DHCP server client identifier. The `pxelinux.0` will be taken from the `syslinux` package and located in the `/tftpboot` directory. Refer to “Installing and Configuring the PXELINUX Server” on page 87 for more information.

   What follows in the example of the `/etc/dhcpd.conf` file:

```
allow booting;
allow bootp;
filename "pxelinux.0";

subnet 172.17.0.0 netmask 255.255.0.0
{
```

```
option routers 172.17.0.1;
range 172.17.1.20 172.17.1.30;
option dhcp-client-identifier "PXEClient";
always-reply-rfc1048 on;
}
```

3. Start the DHCP server as described in the *BlueCat Linux User's Guide*.

Setting Up and Starting the TFTP Server

Refer to the *BlueCat Linux User's Guide* for the instructions how to install, configure, and start the TFTP server.

Installing and Configuring the PXELINUX Server

Perform the following steps to install and configure the PXELINUX server on a Red Hat Linux machine:

1. Install the `syslinux` RPM package from the native Linux distribution and extract the `pxelinux.0` file from the downloaded RPM package.

   `rpm -i syslinux-2.00-4.i386.rpm`

2. Copy the `/usr/lib/syslinux/pxelinux.0` file to the `/tftpboot` directory:

   `cp /usr/lib/syslinux/pxelinux.0 /tftpboot`

3. Create the `pxelinux.cfg` subdirectory in `/tftpboot`:

   `mkdir /tftpboot/pxelinux.cfg`

4. In the `/tftpboot/pxelinux.cfg` directory, create a file named `default` with the following contents:

```
default linux
prompt 0

label linux
kernel <demo>.disk
append initrd=<demo>.rfs root=/dev/ram0 rw ramdisk_size=30000
```

where *<demo>* is the name of a BlueCat Linux demo system to boot and the following parameters are important:

`default`	Sets the default command line.
`prompt` *<flag_val>*	• If *<flag_val>* is 1, the boot prompt is always displayed • If *<flag_val>* is 0, the boot prompt is displayed only if **Shift** or **Alt** is pressed, or **Caps Lock** or **Scroll Lock** is set (this is the default).
`label` *<label>* `kernel` *<kernel>* `append` *<append_options>*	If *<label>* is entered, PXELINUX uses a *<kernel>* to boot with the specified *<append_options>*. The `append` option `initrd=`*<filename>* specifies the BlueCat Linux RFS image.

Configuring PXE Netboot on the Target Board

To enable the PXE Netboot on the target board, perform the following steps:

1. When the BIOS starts, enter the `BIOS setup` utility.
2. In `BIOS setup`, set the first boot device to `LAN`.
3. Enable the `LAN boot ROM` option.
4. Save setting and exit the `BIOS setup` utility.

Booting a BlueCat Linux Demo System onto the Target Board Using PXE Netboot

To boot a BlueCat Linux demo system on to the target board using the PXE Netboot, perform the following steps:

1. Create *<demo>*`.disk` and *<demo>*`.rfs` images as described in the *BlueCat Linux User's Guide*.

   where *<demo>* is the name of a BlueCat Linux demo system to boot.
2. Put *<demo>*`.disk` and *<demo>*`.rfs` to the `/tftpboot` directory.
3. Reboot the target board.

The BlueCat Linux demo system boots on the target.

Booting BlueCat Linux from Target ROM/Flash Memory

This section explains how to boot BlueCat Linux on the target board from target ROM/Flash memory.

Downloading to ROM/Flash using Firmware

BlueCat Linux can be downloaded onto target ROM/Flash memory using either target board firmware or the BlueCat Linux OS loader. This boot scenario assumes that the firmware has a user command or an equivalent feature for passing control to target ROM/Flash memory.

To download BlueCat Linux onto target ROM/Flash memory from the target board firmware, perform the following steps:

1. Create an image suitable for booting from target ROM/Flash memory using firmware.

   On the cross-development host, use `mkboot -m` to create a BlueCat Linux image composed of a compressed kernel image and a compressed root file system. For instance, the following commands create a BlueCat Linux image for the `showcase` demo system.

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/demo/showcase
   BlueCat:$ mkboot -m -k showcase.disk -f \
   showcase.rfs  showcase.kdi
   ```

   (See “mkboot Cross-Development Tool” on page 58.)

2. Download the BlueCat Linux image onto target ROM/Flash memory using an appropriate firmware command. Typically, there is a special command (or a set of commands) that downloads an image over a network, and programs it to target ROM/Flash memory. Refer to the appropriate *Board Support Guide* for specific commands that can be used to download an image onto target ROM/Flash memory from firmware.

Booting Mechanism from ROM/Flash using Firmware

Use an appropriate target firmware command or an equivalent autoboot feature to make a jump to the entry point of the BlueCat Linux image in Flash memory.

Refer to the appropriate *Board Support Guide* for the target board for a description of the appropriate firmware command.

On a target board, booting BlueCat Linux from ROM/Flash memory works as follows:

1. The firmware looks in target Flash memory for the BlueCat Linux boot image.
2. Once it is found, the BlueCat Linux image entry point is called.
3. The kernel decompresses itself from Flash memory into RAM and begins the OS bootstrapping process.
4. If a root file system image is programmed into Flash memory, the kernel decompresses it into a RAM disk and mounts it as the root file system. Otherwise, a root file system is mounted from a hard disk or an NFS server on the cross-development host.

Booting from ROM/Flash using Network and OS Loader

This section explains downloading over a network and booting BlueCat Linux from target ROM/Flash memory using the OS loader.

Booting Kernel and File System Image

This section explains how to download a BlueCat Linux image composed of a *compressed kernel image* and a *compressed file system* onto target ROM/Flash memory. The image is downloaded from a TFTP server on the host. Such an image can be created on the cross-development host using the `mkboot -m` command.

The following command creates the BlueCat Linux image for the `showcase` demo system:

```
BlueCat:$ mkboot -m -k showcase.disk -f \
showcase.rfs showcase.kdi
```

(See “mkboot Cross-Development Tool” on page 58.)

To download a *composite* BlueCat Linux image onto ROM/Flash memory on the target board using the OS loader, perform the following steps:

1. Boot the OS loader on the target board. The boot can be from a floppy disk, network, or any other boot device. See “Booting BlueCat Linux from a Floppy Disk” on page 66.
2. Partition the target Flash memory device using the `Flash_fdisk` utility.

This requires creating a partition that resides at the beginning of target Flash memory and is large enough to hold the BlueCat Linux image. The precise geometry of the partition depends on the size of the BlueCat Linux image to be downloaded and where the target board firmware expects to find a bootable image in Flash memory.

For example, assuming target Flash memory sectors have a size of 64 KB, the following command creates two partitions. The first partition resides in the beginning of target Flash memory and can hold a BlueCat Linux image of up to 640 KB:

```
> exec flash_fdisk /dev/mtdchar0 0-9:10-15
```

3. Set the BLOSH environment variables, so that the `FILE` variable points to the BlueCat Linux image to be downloaded. For example:

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set FILE tftp /tftpboot/showcase.kdi
```

4. Download the BlueCat Linux image onto the target Flash memory partition created for the image. Use the `erase` option of the `flash` command to erase the Flash memory partition before burning the BlueCat Linux image into it. For example:

```
> flash /dev/mtdchar1 erase
```

5. Reset the target board to boot the `showcase` demo system:

```
> reset
```

Booting a Kernel and JFFS/JFFS2-Based Root File System

The procedure in "Booting Kernel and File System Image" above shows the use of a compressed root file system contained in a composite BlueCat Linux image. Alternatively, a root file system can be downloaded onto a Journalling Flash File System (JFFS/JFFS2) partition, and the kernel then made to mount it at boot.

To download BlueCat Linux with an JFFS/JFFS2 root file system image built on the cross-development host using the OS loader, perform the following steps:

1. Prepare a BlueCat Linux image composed of a compressed kernel image, but *not including a compressed file system*. Instead, specify the device node of the target Flash memory partition onto which the JFFS/JFFS2 root file system is to be downloaded. The following example assumes that the root file system is to be downloaded onto the second partition:

```
BlueCat:$ mkboot -m -k showcase.kernel -r 1f02 \
showcase.kdi
```

2. Prepare the JFFS/JFFS2 image of the root file system using the `mkrootfs` utility. See “mkrootfs” in Appendix A, “Command Reference.”

   - To prepare the JFFS image:

     ```
     BlueCat:$ mkrootfs -1vJ showcase.spec \
     showcase.jffs
     ```

   - To prepare the JFFS2 image:

     ```
     BlueCat:$ mkrootfs -1vj showcase.spec \
     showcase.jffs2
     ```

3. Boot the OS loader on the target board. The boot can be from a floppy disk, network, or any other boot device. See “Booting BlueCat Linux from a Floppy Disk” on page 66.

4. Partition the target Flash memory device using the `flash_fdisk` utility. Create at least two partitions: one for the kernel image, another for the JFFS/JFFS2 root file system image. Make sure that the sizes of the first and the second partition are large enough to hold the kernel image and the JFFS/JFFS2 image respectively. For example:

   ```
   > exec flash_fdisk /dev/mtdchar0 0-4:5-10
   ```

5. Set the BLOSH environment variables, so that the `FILE` variable points to the BlueCat Linux kernel image. For example:

   ```
   > set IP 1.0.3.2
   > set HOST 1.0.3.1
   > set IF eth0
   > set FILE tftp /tftpboot/showcase.kdi
   ```

6. Download the BlueCat Linux kernel image, specifying the Flash memory partition created for it. For example, the following command places the BlueCat Linux kernel image into the first Flash memory partition:

   ```
   > flash /dev/mtdchar1 erase
   ```

7. Set the BLOSH environment variable `FILE` to point to the JFFS/JFFS2 image to be burned into the second Flash memory partition:

   ```
   > set FILE tftp /tftpboot/showcase.jffs
   ```

8. Download the JFFS/JFFS2 image of the root file system specifying the target Flash memory partition created for it. For example the following

command places the JFFS/JFFS2 image into the second Flash memory partition:

```
> flash /dev/mtdchar2 erase
```

9. Reset the target board to boot the `showcase` demo system:

```
> reset
```

The previous procedure demonstrates downloading a root file system into target Flash memory as a prebuilt Journalling Flash File System image. Alternatively, a root file system can be downloaded into target Flash memory using the tar archive of the file system and runtime target JFFS/JFFS2 management tools.

To download BlueCat Linux into target Flash memory with a JFFS/JFFS2 root file system created from the tar archive, perform the following steps:

1. Prepare a BlueCat Linux image composed of a compressed kernel image, but *not including a compressed file system*. Instead, specify the device node number for the Flash memory partition on which the JFFS/JFFS2 root file system is to be downloaded. The following example assumes that the root file system is to be downloaded in the second Flash memory partition:

```
BlueCat:$ mkboot -m -k showcase.kernel -r 1f02 \
showcase.kdi
```

(See “mkboot Cross-Development Tool” on page 58.)

2. Prepare a tar image of the root file system using the `mkrootfs` utility. For example:

```
BlueCat:$ $BLUECAT_PREFIX/demo/showcase
BlueCat:$ mkrootfs -lvT showcase.spec showcase.tar
```

(See “mkrootfs” in Appendix A, “Command Reference.”)

3. Boot the OS loader on the target board. The boot can be from a floppy disk, network, or any other boot device. See “Booting BlueCat Linux from a Floppy Disk” on page 66.

4. Partition the target Flash memory device using the `flash_fdisk` utility. Create at least two partitions: one for the kernel image, another for the tar root file system image. Make sure that the sizes of the first and second partitions are large enough to hold the BlueCat Linux kernel image and the tar root file system image, respectively. For example:

```
> exec flash_fdisk /dev/mtdchar0 0-4:5-10
```

5. Set the BLOSH environment variables so that the `FILE` variable points to the BlueCat Linux kernel image. For example:

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set FILE tftp /tftpboot/showcase.kdi
```

6. Download the BlueCat Linux kernel image, specifying the target Flash memory partition created for it. The following command places the BlueCat Linux kernel image in the first Flash memory partition:

```
> flash /dev/mtdchar1 erase
```

7. Empty the target Flash memory partition onto which the root file system is to be downloaded. Make sure to reset the `FILE` environment variable before calling the `flash` command. For example:

```
> set FILE ""
> flash /dev/mtdchar2 erase
```

8. Set the BLOSH `FILE` environment variable to point to the tar image of the root file system. For example:

```
> set FILE tftp /tftpboot/showcase.tar
```

9. Mount the target Flash memory partition created for the root file system as a JFFS/JFFS2. For example:

```
> mount /dev/mtdblock2 /mnt
```

10. Untar the root file system into the JFFS/JFFS2 copying it from the TFTP server:

```
> cd /mnt
> ntar
```

11. Unmount the target Flash memory partition:

```
> cd /
> umount /mnt
```

12. Reset the target board to boot the `showcase` demo system:

```
> reset
```

Booting from Extension BIOS on x86

This section explains booting BlueCat Linux on the x86 target board from an extension BIOS.

Downloading BlueCat Linux onto Extension BIOS

To download BlueCat Linux onto ROM/Flash memory on the target board, perform the following steps:

1. Download BlueCat Linux ROM Boot BIOS as an extension BIOS using target Flash memory and BIOS management tools provided with the target board hardware. The BlueCat Linux ROM Boot BIOS image, `romboot.img` is located in the `$BLUECAT_PREFIX/boot/` directory.

NOTE: The BlueCat Linux ROM Boot BIOS image included in the distribution is an example of a Boot BIOS developed to support booting of BlueCat Linux on a reference board based on the x86 architecture (PC-680 Mobile Industrial Computer of Octagon Systems Corporation). It may be necessary to make changes in the Boot BIOS code to support specifics of custom target board hardware.
Refer to the comments in the source files of the ROM Boot BIOS residing in the `$BLUECAT_PREFIX/usr/src/linux/arch/i386/boot/romboot` directory.

---

2. Create a BlueCat Linux image composed of a compressed kernel and a compressed file system. To do this, use the `mkboot -m` command on the development host. For instance:

   ```
   BlueCat:$ mkboot -m -k showcase.disk -f \
   showcase.rfs showcase.kdi
   ```

   creates an image, `showcase.kdi`, that can be programmed into target ROM/Flash memory.

3. Program the image created in the previous step into target Flash memory using the Flash memory and BIOS management tools provided with target board hardware.

4. Enable support for extension BIOS on the target board, either in BIOS or using on-board jumpers, depending on target board hardware.

5. Reset the target board. BlueCat Linux boots from target ROM/Flash memory.

Booting Mechanism from Extension BIOS

On an Intel PC based target board, booting BlueCat Linux from target ROM/Flash memory works as follows:

1. At initialization, the BIOS performs the ROM-Scan procedure in search of BIOS extensions. BlueCat Linux ROM Boot BIOS is found and called for initialization.
2. BlueCat Linux ROM Boot BIOS intercepts the `INT19` interrupt handler (Bootstrap Operating System). The old `INT19` handler vector is saved as `INT18`.
3. The BIOS proceeds with the initialization and eventually calls `INT19` to begin the OS bootstrapping process, thus calling the BlueCat Linux ROM Boot BIOS.
4. BlueCat Linux ROM Boot BIOS copies the kernel image and the root file system image from target Flash memory into RAM.
5. It then calls the new kernel in RAM, which decompresses itself and begins the OS bootstrapping process.
6. The compressed root file system image in the RAM is decompressed into a RAM disk, and is mounted as the root file system.

Booting BlueCat Linux over a Network or Parallel Port

This section outlines booting BlueCat Linux on an target board using a network or a parallel port. Previous sections have already introduced the use of a network, specifically, a TFTP server, in booting BlueCat Linux. “Booting from Hard Disk and DiskOnChip using Network and OS Loader” focuses on installing to and booting from the hard disk. “Booting from ROM/Flash using Network and OS Loader” explains download and booting from ROM/Flash. However, in all these instances, the BlueCat Linux OS loader has been used to boot the target board before downloading an image(s) over the network.

Booting over a Network using Target Firmware

This section explains booting BlueCat Linux onto the target board over a network using the target board firmware. (See also “Booting Mechanism from ROM/Flash using Firmware.”) This boot scenario assumes that the firmware has a command or an equivalent feature that allows downloading an image from a TFTP server on a cross-development host.

Booting over a Network or Parallel Port using OS Loader

This section explains booting BlueCat Linux onto an target board over a network or parallel port using the OS loader. This boot scenario implies that the first step of the boot procedure downloads the OS loader onto the target board.

- Downloading the OS loader

  The BlueCat Linux OS loader can be downloaded onto a hard disk, floppy disk, or target Flash memory and used to boot BlueCat Linux over a network using TFTP or NFS, or from a parallel port using PFTP. The procedure to download the OS loader onto a bootable medium is the same as for any other BlueCat Linux system.

- Boot Mechanism using the OS loader

  The BlueCat Linux OS loader provides a command interface to boot a target board with a BlueCat Linux system. Refer to the section entitled “BlueCat Linux OS Loader” on page 59 for details on the OS loader user interface and features.

On a target board, booting BlueCat Linux using the OS loader works as follows:

1. The OS loader downloads a compressed kernel image, and optionally, a compressed root file system image onto the target board memory.
2. The OS loader shuts down its kernel.
3. The OS loader moves the loaded kernel and file system images to the appropriate places in RAM.
4. The OS loader prepares parameters for the new kernel, including the command line.
5. It then calls the new kernel, which decompresses itself and begins the OS bootstraping process.
6. The compressed root file system image in RAM is decompressed into a RAM disk, and is mounted as the root file system.

NOTE: The procedure for booting BlueCat Linux on a target board using the OS loader is also applicable to the `i_osloader` demo system.

Hardware Support for Network or Parallel Port Booting with the OS Loader

The BlueCat Linux OS loader kernel must be configured with support for devices and media from which the kernel and root file system images are loaded. The following is a list of kernel configuration options required for a particular boot method to work:

- `TFTP`—Networking (`CONFIG_NET`), TCP/IP (`CONFIG_INET`), device driver for network interface
- `NFS`—Networking (`CONFIG_NET`), TCP/IP (`CONFIG_INET`), NFS (`CONFIG_NFS_FS`, `CONFIG_SUNRPC`, `CONFIG_LOCKD`), device driver for network interface
- `FILE`—Support for a disk device and file system type on which the image resides
- `PFTP`—Generic parallel port support (`CONFIG_PARPORT`), device driver for a particular parallel port (PC-style hardware support, `CONFIG_PARPORT_PC`, for x86), IEEEI284 transfer modes support (`CONFIG_PARPORT_I284`), BlueCat Linux bidirectional parallel port driver (`CONFIG_BLUECAT_BPAR`), and optionally polling mode support (`CONFIG_BLUECAT_BPAR_POLL`). Additionally, the kernel command line must contain the `parport=auto` parameter.

IP Autoconfiguration using the OS Loader

The BlueCat Linux kernel supports automated configuration of the target IP address and routing tables using BOOTP or RARP protocols. The BlueCat Linux OS loader supports IP autoconfiguration to set the environment variables necessary for network booting using TFTP or NFS. When IP autoconfiguration succeeds, the following BLOSH environment variables are set at startup:

- `IP`—Set to the target IP address acquired via BOOTP
- `HOST`—Set to the IP address of the server that answered the BOOTP request
- `FILE`—Set to the boot filename acquired via BOOTP

To enable IP autoconfiguration in the `osloader` kernel, the `CONFIG_IP_PNP` and `CONFIG_IP_PNP_BOOTP` configuration options must be enabled.

NOTE: IP autoconfiguration without a properly set up BOOTP server pauses kernel loading for about a minute, and no BLOSH variables are set.

See the use of BLOSH commands in "script— Process List of Commands in a File" on page 106 to autoconfigure environment variables necessary for network booting.

Autobooting BlueCat Linux

This example shows the contents of a sample `/etc/blosh.rc` used to autoboot the system from a TFTP server without any manual intervention:

```
# /etc/blosh.rc
# autoboot from TFTP host

set IP <target_IP_address>
set HOST <host_IP_address>
set IF <ethernet_interface>
set KERNEL tftp /tftpboot/<name>.kernel
set RFS tftp /tftpboot/<name>.rfs
boot
```

Target Board-Specific Autoboot of BlueCat Linux

This example shows how a combination of the IP autoconfiguration and the `read` and `script` commands is used to execute a target board-specific auto-boot sequence from the same OS loader image.

The `/etc/blosh.rc` script is set up as follows:

```
read /<my_script>
script /<my_script>
boot
```

The sequence of target board-specific auto-boot events is as follows:

1. The IP autoconfiguration process sends a BOOTP request to the network.
2. The BOOTP host replies with the IP addresses of the target board, the TFTP host, and the target-specific file stored in the cross-development host file system. The BlueCat Linux OS loader uses these values to set up the environment variables `IP`, `HOST`, and `FILE`, respectively.

3. BLOSH is started and finds the `/etc/blosh.rc` script.
4. The `read` command copies the *<my_script>* file from the TFTP host.
5. The `script` command executes the *<my_script>* script file. This can, for instance, set up the `KERNEL`, `RFS`, and `CMD` environment variables.
6. Finally, the `boot` command auto-boots the target board.

BLOSH Commands and Variables for Net Booting

This section shows some simple examples of how the BlueCat Linux OS loader can be used to boot embedded systems from different kinds of servers. *All examples assume that the BlueCat Linux OS loader has already been used to boot the target board*. (See, for example, "Booting BlueCat Linux from a Floppy Disk.")

Booting from RAM using a TFTP Server

The following sequence of BLOSH commands shows how a BlueCat Linux system can be booted on the target board from a TFTP server on a host. Both the kernel image as well as the root file system image are downloaded.

```
> set IP <target_IP_address>
> set HOST <host_IP_address>
> set IF <ethernet_interface>
> set KERNEL tftp /tftpboot/<name>.kernel
> set RFS tftp /tftpboot/<name>.rfs
> boot
```

Booting from RAM using an NFS Server

The following sequence of BLOSH commands shows how a BlueCat Linux system can be booted on the target board from an NFS server on a host. Both the kernel image as well as the root file system image are downloaded.

```
> set IP <target_IP_address>
> set HOST <host_IP_address>
> set IF <ethernet_interface>
> set KERNEL nfs /nfsboot <name>.kernel
> set RFS nfs /nfsboot <name>.rfs
> boot
```

On a Linux NFS server, ensure that the following line is present in the `/etc/exports` file:

```
/nfsboot <target_IP_address>(r,no_root_squash)
```

Mounting a Root File System from NFS

This example shows how the BlueCat Linux OS loader can be used to boot a BlueCat Linux kernel that mounts an NFS-based file system as the root file system. The example assumes that the BlueCat Linux kernel is configured to mount an NFS-based file system (versus mounting a RAM disk-based file system downloaded into the image).

```
> set IP <target_IP_address>
> set HOST <host_IP_address>
> set IF <ethernet_interface>
> set KERNEL tftp /tftpboot/<name>.kernel
> set CMD root=/dev/nfs rw \
nfsroot=<host_IP_address>:/nfsboot \
ip=<target_IP_address>:<host_IP_address>:::::off \
panic=1
> boot
```

Booting over a Parallel Port

The following BLOSH commands boot BlueCat Linux from a parallel port:

```
> set PPORT /dev/bpar0
> set KERNEL pftp /pftpboot/<name>.kernel
> set RFS pftp /pftpboot/<name>.rfs
> boot
```

Downloading and Executing Programs

This example shows how the BlueCat Linux OS loader is used to download an application program from the network server and execute it onto the target board. This example does not boot a new BlueCat Linux kernel on the target board, but relies instead on the fact that the OS loader is simply an embedded configuration of BlueCat Linux.

```
> set IP <target_IP_address>
> set HOST <host_IP_address>
> set IF <ethernet_interface>
> set FILE tftp /tftpboot/<name>
```

```
> read /<name>
> exec chmod +x /<name>
> exec /<name>
```

Customizing the BlueCat Linux OS Loader

Adding New Commands to BLOSH

Command names and the functions that implement them are listed in the table provided in the `blosh_cmd.c` file in the BLOSH source directory.

Each table entry has the following structure, defined in the `blosh_cmd.h` file:

```
typedef struct blosh_cmd_entry_s
{
  const char*        name;
  blosh_cmd_result_t (*do_cmd)(int argc, char** argv);
  const char*        usage;
  const char*        description;
  char*              (*support_level)(char* buf, size_t size);
}
blosh_cmd_entry_t;
```

where:

- `name` is command name.
- `do_cmd` is the function that implements the command. It must return one of the following values:

  ```
  BLOSH_SUCCESS,
  BLOSH_FAILURE
  BLOSH_USAGE
  ```

- `usage` is usage text. If `do_cmd()` returns `BLOSH_USAGE`, the text is prefixed with `Usage:` and displayed.
- `description` is a short command description to be displayed by the `help` command.
- `support_level` is the function that reports the command support level. The text the function places in the buffer pointed to by the `buf` parameter is displayed by the `help` command right after the command description.

To complete the new BLOSH command, place the name of the source module to the new command in the `OBJS` list in the Makefile and rebuild BLOSH.

BLOSH Command Reference

BLOSH implements a number of built-in commands. Each command prints a `Usage error` message if used incorrectly. A command name may be reduced to any number of characters as make it unique. For example, `boot` can be abbreviated as `b`, `bo`, or `boo`. (See also Appendix B, “BLOSH Commands.”)

cd—Change Current Working Directory

cd [*<directory>*]

The `cd` command sets the current working directory for BLOSH. If no directory is specified, the value of the `HOME` environment variable is used.

mount—Mount a File System

mount *<device> <directory>*

The `mount` command mounts a file system at the specified directory.

set—Show or Modify Environment Variables

set *<var> <value>*

The `set` command shows or modifies the environment variables. If no variable is specified, the command shows all the environment variables and their respective values. If a variable name but no value is specified, the current value of the variable is shown. Finally, set with two arguments sets the variable to a new value.

Quoting is not mandatory in the `set` command. The remainder of the line, excluding the leading space character, is considered to be the new value of the variable.

boot—Booting a BlueCat Linux Kernel

boot

The `boot` command boots a BlueCat Linux system. The location of the kernel image and optional root file system image, as well as the kernel boot parameters, are specified by other BLOSH environment variables.

If the root file system is specified, the booted kernel loads the file system image into RAM and mounts it as a root file system. If the root file system variable is not set (that is, the `RFS` environment variable is set to an empty string), the booted kernel image must mount something else as the root file system. This can be, for instance, a file system on a local disk or an NFS-based file system.

If booting is from the network, the networking-related environment variables must be set to appropriate values. Also, the network server (either a TFTP or an NFS server) must be configured to allow downloading of images to the target.

If booting is from a parallel port, the `PPORT` variable must be set. Also, the PFTP server must be set up to allow downloading of images to the target.

The following command sequence shows booting of a BlueCat Linux system from a TFTP host. Both kernel and root file system images are specified:

```
> set IP <target_IP_address>
> set HOST <host_IP_address>
> set IF <ethernet_interface>
> set KERNEL tftp /tftpboot/<kernel>.kernel
> set RFS tftp /tftpboot/<rootfs>.rfs
> boot
```

exec—Execute a Program

exec [**-r**] *<program>* [*<params>*]

The `exec` command executes the specified program found on the BlueCat Linux root file system as a new process. If the `-r` flag is specified, the new program completely replaces BLOSH in RAM. The *<params>* string, if provided, is passed to the process as the parameters.

For instance, the following command shows the contents of the BlueCat Linux OS loader root directory.

```
> exec /bin/ls -lt /
```

(This example assumes that the `ls` utility is contained in the `/bin` directory, which is not the case by default. However, arbitrary utilities and files can be added to the BlueCat Linux OS loader file system.

flash— Program Image into Flash

flash /dev/mtdchar<*n*> [**erase**]

The `flash` command downloads the file specified by the `FILE` environment variable and installs it on the specified Flash device. If an optional `erase` argument is supplied, the full erase of the specified Flash device is performed before programming begins.

mkboot— Create a Bootable Disk

mkboot [**-b**] [**-r** <*root*>] **/dev/***xxx*

The `mkboot` command functions similarly to the LynuxWorks `mkboot` utility included in the set of BlueCat Linux cross-development tools. The only difference is that the kernel image, root file system image, and the command line are specified by the BLOSH environment variables as follows:

- `KERNEL`—Specifies the kernel image to install.
- `RFS`—If set, specifies the compressed root file system image to install.
- `CMD`—Specifies the kernel command line.

The following command sequence shows the installation of a BlueCat Linux kernel and root file system on a hard disk for an x86 target. The kernel installed by this example boots from a hard disk, then uncompresses the file system in the RAM, and mounts it as the root file system.

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set KERNEL tftp /tftpboot/<name>.disk
> set RFS tftp /tftpboot/<name>.rfs
> mkboot -b -r /dev/hda /dev/hda
```

ntar— Download and Unpack a tar Archive

ntar

The `ntar` command downloads and unpacks a tar archive into the current directory. The archive to work with is specified by the `FILE` environment variable. If the archive is located on a network, the networking-related environment variables must be set to the appropriate values. Also, the network server (either a TFTP or an NFS server) or the parallel port server must be configured to allow downloading of images to the target.

The following command sequence shows the creation of a BlueCat Linux root file system on a partition of the local disk. The archive is copied from a TFTP server.

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set FILE tftp /tftpboot/<root_file_system>.tar
> mount /dev/hda1 /mnt
> cd /mnt
> ntar
```

script— Process List of Commands in a File

script *<file>*

The `script` command sequentially executes BLOSH commands contained in the specified file. If any command fails, the script is halted.

The script file can contain another script command, thus allowing recursive scripting. This feature may be especially useful in a scenario where a script must be downloaded from the network.

Empty lines or lines starting with a `#` character are considered to be comments and ignored by the script processor.

The following example shows a script file that sets the network environment variables:

```
# sample script file
# set up network variables
set IP 1.0.3.2
set HOST 1.0.3.1
set IF eth0
# end of script
```

read—Download an Arbitrary File

read *<file>*

The `read` command downloads the file specified by the `FILE` environment variable and places it in the BlueCat Linux OS loader root file system under the file named *<file>*.

The intended use of this command is to download a BLOSH script file. Alternatively, the `read` command can be used to copy an executable file to the `osloader` root file system to execute.

As an example, the following sequence copies a BLOSH script from a TFTP server and then executes BLOSH commands contained in the script:

```
> set IP <target_IP_address>
> set HOST <host_IP_address>
> set IF network_interface
> set FILE tftp /tftpboot/script.<target_IP_address>
> read /my_script
> script /my_script
```

reset— Reboot the System

reset

The `reset` command unconditionally shuts down the BlueCat Linux OS loader and performs a hardware reset.

help – Print Help Message

help [<*name*>]

The `help` command shows help messages. If no argument is specified, the list of all supported commands is shown. `help` with a single argument shows the usage string for the specified command.

Rebuilding BLOSH

To rebuild BLOSH, execute `make` in the BLOSH source directory.

CHAPTER 4 *BlueCat Linux Demo Systems*

This chapter describes BlueCat Linux demo systems. Demo systems can be used to jump-start the user's own development of custom embedded systems and applications for target boards, because sources are included in the demo system.

NOTE: Not all demos described in this chapter may be supported on a specific target board. Consult the relevant *Board Support Guide* for this information.

Overview

Once BlueCat Linux is installed onto the cross-development host, a number of prebuilt, ready-to-run demo systems are available for booting on the target board. Each demo system, whether a specially configured kernel, or a sample application, displays a particular feature of BlueCat Linux.

This chapter provides a detailed description of each demo system. LynuxWorks recommends using these systems to get used to BlueCat Linux (that is, booting, rebuilding kernels and root file systems) without having to learn the development environment in detail.

A feature of demo systems especially useful for developers is that each system includes all the files and tools required to rebuild the system from scratch. Thus, there is a set of templates from which to jump-start the user's own development. The recommended approach is to find a demo or set of demos that is closest to the user's embedded application, and use it as a starting point for custom development. This approach has the advantage of always having a working prototype that can be tested on the target board at any point in the development process.

Because the demo systems included in BlueCat Linux span a wide range of features, each user can find a different starting point suitable to their custom embedded system.

Demo System Components

A demo system directory contains all of the files required to build the demo system. A demo system is composed of the following:

- A customized BlueCat Linux kernel represented by a prebuilt, compressed kernel image and a `<demo>.config` file that can be used to rebuild the kernel. Use the `mkkernel` tool (or an equivalent) to build the kernel image from the `<demo>.config` file.
- A root file system containing the tools and custom programs required for the demo system to boot up and run. This is represented by a prebuilt, compressed root file system image and a specification file that can be used to rebuild the root file system. Use the `mkrootfs` tool to build the root file system image from the `<demo>.spec` file.
- Optionally, a demo system contains custom applications and files such as sample application programs, developed especially for the demo system.

Location

Once BlueCat Linux has been installed, the demo systems can be found in the `$BLUECAT_PREFIX/demo` directory. This directory contains a number of subdirectories, each containing its own embedded demo system.

Contents of Demo System Directory

A demo system directory typically contains the files and subdirectories listed below.

Table 4-1: Demo System Files and Subdirectories

File/ Subdirectory	Description
<*demo*>.config	Kernel configuration file
<*demo*>.spec	Root file system specification file
<*demo*>.kernel	Prebuilt, compressed kernel image suitable for booting onto a target board over a network using the BlueCat Linux OS loader
<*demo*>.disk	Prebuilt, compressed kernel image suitable for copying onto a floppy, hard disk, or CD-ROM
<*demo*>.rfs	Prebuilt, compressed RAM disk root file system image suitable for booting onto a target board over a network using the BlueCat Linux OS loader, or for loading from a floppy, hard disk, or CD-ROM
<*demo*>.tar	Tar image of the root file system suitable for copying on a hard disk partition or for NFS-mounting
<*demo*>.kdi	Image composed of the compressed kernel image (.disk) and optional compressed RAM disk root file system (.rfs) suitable for booting onto a target board from a network using firmware, or programming into target ROM/Flash memory
Makefile	Makefile to build the demo system
src	Source files of the custom programs used in the demo system
local	Configuration files specific to the demo system

Configuring a Demo System

Each demo system has its own kernel configuration. See the relevant `<demo>.config` for a complete specification of the demo kernel configuration.

For Hardware Devices

Each demo system is configured to support the feature displayed in the kernel (for example, FTP capabilities). The user might still need to reconfigure the kernel in case the target board hardware is different from that on which the demo kernel image is built. For instance, the user may have to enable a network driver for the target board's specific Ethernet interface, as opposed to the Ethernet interfaces supported in the demo system by default. Similarly, if a demo system supports a particular hard disk, the user may need to reconfigure the hardware driver for the target board's disk controller.

For specifications of the hardware supported by the prebuilt demo system kernel, refer to specific demo system descriptions in this chapter and the *BlueCat Linux Board Support Guide* for the appropriate target board.

For the Boot Device

Depending on the nature of a demo system and the boot options supported by a target board, different demo systems have support for different boot devices configured in their kernels. For instance, in a BlueCat Linux distribution for an x86-based target board, those demo systems that can fit onto a floppy disk (1.44 MB) have floppy support configured in the kernel. The user can copy such demo systems onto a floppy as is. To keep the image small, support for a hard disk is disabled (unless the demo system displays operations with a hard disk). If the user wants to copy the kernel onto a hard disk or a CD-ROM, he or she must reconfigure the kernel to add support for the hard disk or CD-ROM respectively.

To support booting from a floppy, a hard disk or a CD-ROM, the user may also need to configure the hardware device driver for the floppy, hard disk, or CD-ROM respectively. For boot options supported by the demo system by default, refer to specific demo system descriptions in this chapter and the *BlueCat Linux Board Support Guide* for the appropriate target board.

Using the Makefile to Rebuild a Demo System

Use the Makefile in the demo system directory to rebuild the system images. A typical Makefile can achieve the targets listed in the table below:

Table 4-2: Typical Makefile Targets

Makefile Target	Description
`kernel`	Builds *<demo>*`.kernel` and *<demo>*`.disk`
`rootfs`	Builds *<demo>*`.rfs` and *<demo>*`.tar`
`kdi`	Builds *<demo>*`.kdi`
`this`	Builds custom programs in `src/`
`all`	Builds all of the above
`xconfig`	• Copies *<demo>*`.config` to the kernel `.config` • Calls `make xconfig` • Copies the updated kernel `.config` into *<demo>*`.config`
`menuconfig`	• Copies *<demo>*`.config` to the kernel `.config` • Calls `make menuconfig` • Copies the updated kernel `.config` into *<demo>*`.config`
`gconfig`	• Copies *<demo>*`.config` to the kernel `.config` • Calls `make gconfig` • Copies the updated kernel `.config` into *<demo>*`.config`
`clean`	Removes all prebuilt binaries

NOTE: Since the Qt X-based and the Gtk X-based libraries are not included in the BlueCat Linux distribution, the respective library must be installed on the host if `make xconfig` or `make gconfig` are used. Refer to “make xconfig” and “make gconfig (Linux Host Only)” in Chapter 2, “Developing BlueCat Linux Applications.”

Running Demo Systems

A demo system on the target board can be booted from one of the devices in the table below, depending on the boot options supported by the target board:

Table 4-3: Demo System Boot Procedure

Boot Device	Boot Procedure	Detailed Description
Floppy Disk	• Copy the demo system onto a floppy from cross-development host using `mkboot`. • Boot the target board from floppy disk.	See "Booting BlueCat Linux from a Floppy Disk" in Chapter 3, "Downloading and Booting BlueCat Linux."
Hard Disk	• Copy the demo system onto a hard disk either from the cross-development host using `mkboot` or from the target board using the OS loader. • Boot the target board from a hard disk.	See "Booting BlueCat Linux from a Hard Disk or DiskOnChip" in Chapter 3, "Downloading and Booting BlueCat Linux."
CD-ROM	• Optionally, create an ISO9660 file system. • Create a bootable image of the demo system using `mkboot`. • Burn the image created on the previous step into blank CD-R(W). • Boot the target board from CD-ROM.	See "Booting BlueCat Linux from a CD-ROM" in Chapter 3, "Downloading and Booting BlueCat Linux."
ROM/Flash Memory	• Download a demo system into target ROM/Flash memory using firmware, external device-specific tools, or from the target board using the OS loader. • Boot the target board from ROM/Flash memory.	See "Booting BlueCat Linux from Target ROM/Flash Memory" in Chapter 3, "Downloading and Booting BlueCat Linux."
USB Flash drive	• Create a demo system on the USB Flash drive using `mkboot`. • Boot the target from the USB Flash drive.	See "Booting from a USB Flash Drive" in Chapter 3, "Downloading and Booting BlueCat Linux."

Table 4-3: Demo System Boot Procedure (Continued)

Boot Device	Boot Procedure	Detailed Description
Network or Parallel Port using OS Loader	• Download the OS loader onto a floppy disk, hard disk, or into target ROM/Flash memory. • Boot the OS loader on the target board. • Boot a demo system from target RAM using a network (TFTP or NFS) or the parallel port (PFTP) using the OS loader.	See "Booting BlueCat Linux over a Network or Parallel Port" in Chapter 3, "Downloading and Booting BlueCat Linux."
Network using Firmware	• Boot a demo system from the network using the firmware netboot option.	See "Booting over a Network using Target Firmware" in Chapter 3, "Downloading and Booting BlueCat Linux."

Kernel Command Line Options

If the user boots a demo system on the target board using the OS loader, the kernel command line is specified in the `CMD` environment variable of BLOSH. Alternatively, if the user copies the demo system onto bootable media or boots the target board from a BlueCat Linux image composed of a compressed kernel and a root file system, `mkboot -c` must be used to pass a kernel option to the kernel command line. Refer to the man page for `mkboot(1)` for details. Also refer to the "mkboot" section in Appendix A, "Command Reference."

NOTE: If a demo system requires a specific kernel command line option, the prebuilt demo system image composed of a compressed kernel and a root file system is built to include a correct kernel command line in the image.

Useful Kernel Command Line Options

ramdisk_size

The `ramdisk_size` kernel parameter defines the maximum size in KBytes the kernel will let the RAM disk grow to.

When a size value for the RAM disk is provided to the kernel via `ramdisk_size`, the kernel *would* allow the RAM disk to grow to the specified size. Typically, RAM disk is used to hold an ext2 root file system created by the `mkrootfs(1)` tool on the development host. It is important to understand that if the relationship between the RAM disk size and the file system size. If the user attempts to put a large file system into a smaller RAM disk, the file system will simply not fit and the target bootstrap will fail with an appropriate message. This is why it is important to pass an appropriate `ramdisk_size` parameter for an embedded system that requires a large root file system.

On the other hand, even if a large RAM disk size is passed to the kernel as a `ramdisk_size` parameter, it does not mean that the ext2 file system put into the RAM disk will take the entire RAM disk. On the contrary, the geometry of the file system placed to the RAM disk is fixed (by `mkrootfs`), which has an implication that the RAM disk size will grow to exactly the size of the file system put into it but no more.

Some applications require additional free blocks in the root file system, compared to what is available by default. This need may be due to a requirement that relatively large files be created in the root file system at run time. For such applications, the user must calculate the desired number of free blocks in the file system and then pass that number to `mkrootfs` as a `-r` parameter. This will make `mkrootfs` allocate the required number of free blocks (and the underlying inodes and so on) in the file system. Then the user must make sure that the size of the RAM disk as defined by `ramdisk_size` is the same or larger as the file system being placed into the RAM disk.

console

The `console` kernel parameter is used to redirect the system console to a device other than the kernel default console device:

```
console=/dev/ttyS<x>
```

For example, on the x86 targets for all demo systems, the VGA console and PC keyboard are configured as the default Linux console. To redirect the system console to one of the serial ports, the following kernel parameter must be added to the kernel command line:

- To redirect output to COM1:

  ```
  console=ttyS0
  ```

- To redirect output to COM2:

  ```
  console=ttyS1
  ```

Demo Systems Reference

This section contains a detailed description of all the demo systems included in BlueCat Linux. For each demo, a description and basic requirements for target board hardware and demo system environment are provided.

NOTE: Not all demos listed here may be supported on a specific target board. Consult the relevant *Board Support Guide* for this information.

Requirements

This section contains target board hardware requirements for each demo system. The "*Storage*" entry under each demo system description shows minimal requirements for nonvolatile storage (floppy, hard disk, or ROM/Flash memory) on the target board. Storage values can be any of the following:

Table 4-4: Storage Size Options

Target Board	Tiny	Small	Medium	Large	Huge
ARM	2 MB	4 MB	8 MB	16 MB	>16 MB
x86	1 MB	2 MB	4 MB	8 MB	>8 MB
PowerPC	2 MB	4 MB	8 MB	16 MB	>16 MB
MicroBlaze	2 MB	4 MB	8 MB	16 MB	>16 MB
MIPS	2 MB	4 MB	8 MB	16 MB	>16 MB

The "*RAM*" entry under each demo system description shows minimal requirements for system memory on the target board. RAM values can be any of the following:

Table 4-5: Memory Size Options

Target Board	Tiny	Small	Medium	Large
ARM	4 MB	8 MB	16 MB	>16 MB
Intel IA-32 or x86 PC compatible	4 MB	8 MB	16 MB	>16 MB
PowerPC	4 MB	8 MB	16 MB	>16 MB
MicroBlaze	4 MB	8 MB	16 MB	>16 MB
MIPS	4 MB	8 MB	16 MB	>16 MB

If a demo system requires a *network*, it means that the target board must have an Ethernet connection to the local area network.

If a demo system requires a *disk*, it means that the target board must have a hard disk connected to it.

If a demo system requires a specific *kernel option*, it means that the specified kernel option must be passed in the kernel command line.

List of Supported Demo Systems

The `$BLUECAT_PREFIX/demo` directory typically contains the demo systems listed in the table below.

Table 4-6: Demo Systems and their Requirements

Subdirectory	Demo System	Requirements
`developer`	A combination of a simple environment demo and simple networking demos	Storage: Large RAM: Large Network: Yes Disk: None Special: In case of remote debugging via a serial line, the host and target machines must be connected by a serial line.
`osloader`	BlueCat Linux OS loader	Storage: Tiny RAM: Tiny Network: Yes Disk: None Special: None
`showcase`	Configures an Apache Web Server	Storage: Small RAM: Medium Network: Yes Disk: None Special: None
`install_light`	Boots BlueCat Linux from a removable boot media (CD-ROM, CompactFlash) and installs the BlueCat Linux system onto the hard disk	Storage: Large RAM: Large Network: Yes Disk: Small Special: None

NOTE: Refer to *Board Support Guide* for a precise list of the demo systems supported on a particular target.

osloader

DEMO

BlueCat Linux OS loader

SYNOPSIS

This demo system is the BlueCat Linux OS loader that can be used to boot BlueCat Linux from various boot media.

REQUIREMENTS

Storage	Tiny
RAM	Tiny
Network	Yes
Disk	None
Special	None

DESCRIPTION

The system boots up in single-user mode. `init` starts the BLOSH (BlueCat Linux Loader Shell) shell. The BLOSH command interface is used to boot a BlueCat Linux system on the target board. Refer to Chapter 3, “Downloading and Booting BlueCat Linux” for a detailed description of the BlueCat Linux OS loader.

There is an alternative version (`i_osloader`) of this demo system built in the `osloader` demo system directory. The demo system is extended with support for a hard disk and intended to perform a copy of BlueCat Linux onto a hard disk. The `i_osloader` demo system is slightly larger than `osloader`.

showcase

DEMO

`showcase` demo system

SYSNOPSIS

Sets up an Apache web server.

REQUIREMENTS

Storage	Small

RAM	Medium
Network	Yes
Disk	None
Special	None
Kernel Option	`ramdisk_size=8192`

DESCRIPTION

This demo system starts and configures the `apache` HTTP daemon turning the target board into a web server. Web pages are accessible from any remote cross-development host that has a web browser installed.

Network interfaces are set up in the `.bashrc.<board>` file, located in the `$BLUECAT_PREFIX/demo/showcase/local/etc` directory. When `showcase` is booted up, the IP address values (`<target_IP_address>` and `<gateway_IP_address>`) in this file are automatically used. If changing the default IP address, it is recommended that this be done before building the `showcase` demo. Refer to “Modifying Target Board IP and Gateway Addresses” on page 121 for more information.

The system boots in single-user mode. `init` starts `bash` without a login prompt. `bash` brings up the network interface(s) using the `ifconfig` command, optionally, sets up the kernel routing table using the `route` command, and runs the HTTP daemon.

Now the Apache server is accessible from any networked system using the IP address specified in the `.bashrc.<board>` file and serves the `index.html` page located in the `showcase` subtree in the `demo` directory.

Modifying Target Board IP and Gateway Addresses

The `showcase` demo system includes an Apache web server feature.

The Apache web server is a robust, commercial-grade, featureful, and freely-available source code implementation of an HTTP (web) server. See `http://www.apache.org` for further information.

Users may wish to define a unique IP address for their target board, rather than use the default address (`172.17.1.12`) defined in the `showcase` demo system. To change the default IP address for the `showcase` demo system, open the

`showcase` demo system's `.bashrc.<board>` file with any text editor (such as `vi`), and perform the following steps:

1. Change directory to the location of the `.bashrc` file:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/demo/ \
   showcase/local/etc
   ```

2. Change the `showcase` demo system's `.bashrc.<board>` default target board IP address (`172.17.1.12`) to a new user-selected one. For example, to change it to 216.100.252.140, edit it as follows:

   Find this line:

   ```
   TARGET_IP=172.17.1.12
   ```

   Edit it to read as follows:

   ```
   TARGET_IP=216.100.252.140
   ```

3. Set a gateway IP address by entering a value at this line in the `.bashrc.<board>` file:

   ```
   GATE_IP= <gateway_IP_addr>
   ```

4. Rebuild the `showcase` demo system by entering the following commands in the `$BLUECAT_PREFIX/demo/showcase` directory:

   ```
   BlueCat:$ make clean
   BlueCat:$ make all
   ```

developer

DEMO

`developer` demo system

SYNOPSIS

Various functionalities are combined in one BlueCat Linux system: `developer` demonstrates the use of an FTP client, the bash shell, simple networking, and remote debugging with GDB.

REQUIREMENTS

Storage	Large
RAM	Large
Network	Yes

Disk None

Special If remote debugging via a serial line is desired, the host and target machines must be connected by a serial line. Serial `tty` devices (nodes) available in the root file system image are `/dev/ttyS0` and `/dev/ttyS1`.

Kernel Option `ramdisk_size=32768`

DESCRIPTION

- `developer` demonstrates the Bourne-Again Shell (`bash`), which provides a simple environment. The file system includes the `ls`, `ps`, `reboot`, and `shutdown` commands. To invoke the shell, log in as `root` with a blank or null password and then test included commands.
- To test simple networking, bring up the network interface(s) manually using the `ifconfig` command and, optionally, reset the kernel routing table using the `route` command. Network functionality can be tested with the `ping` command. For instance:

```
bash# ifconfig eth0 172.16.1.62
bash# route add default gw 172.16.0.1
bash# ping 195.239.208.81
```

- To use the FTP functionality, bring up the network interface(s) manually using the `ifconfig` command and optionally, set up the kernel routing table using the `route` command.

  To `ftp` to another machine on the network:

```
bash# ftp 195.239.208.81
```

  To `ftp` to the target board from another machine on the network:

```
[user@host user]$ ftp 172.16.1.62
```

- To debug an application program on the target board from remote GDB connected through a serial line or network, ensure that the host and target machines are connected through a serial line. Serial `tty` devices (nodes) available in the root file system image are `/dev/ttyS0` and `/dev/ttyS1`.

  1. On the target board, to connect GDB to Gdbserver via network use the `ifconfig` command to bring up a network interface(s) and, optionally, set up the kernel routing table using the `route` command. For instance:

```
bash# ifconfig eth0 172.16.1.62
bash# route add default gw 172.16.0.1
```

2. Start `gdbserver` with the simple test program `test_prog` included in the root file system image:

   ```
   bash# gdbserver <target_ip>:2345 /tmp/test_prog
   ```

3. To connect `gdbserver` via a serial line connected to `COM2` use the following command:

   ```
   bash# gdbserver /dev/ttyS1 /tmp/test_prog
   ```

4. After starting `gdbserver` on the target board, change the working directory on the cross-development host to the directory containing the source of the demo system (`$BLUECAT_PREFIX/demo/gdb/src`).

5. Start `gdb` specifying the program name as a parameter:

   ```
   BlueCat:bash# gdb ./test_prog
   ```

6. Use the `target remote` command to connect to `gdbserver` on the target board via a network:

   ```
   (gdb) target remote <target_ip>:2345
   ```

   or use the standard GDB commands to debug `test_prog` via a serial line:

   ```
   (gdb) target remote /dev/ttyS1
   ```

NOTE: Use `COM2` for the Windows hosts.

install_light

DEMO

`install_light` demo system

SYNOPSIS

Boots BlueCat Linux on the target from a removable boot media (CD-ROM, CompactFlash), and installs the BlueCat Linux system onto the hard disk. After installation to the hard disk, the BlueCat Linux system is ready for booting from the hard disk.

REQUIREMENTS

Storage	Large
RAM	Large
Network	Yes
Disk	Small

DECRIPTION

The `install_light` demo system contains the kernel and the root file system images for the BlueCat Linux system to be installed to the hard disk.

Booting the install_light Demo to the Target from CD-ROM

This section describes how to boot the `install_light` demo system onto the target from CD-ROM and then install the BlueCat Linux system to the hard disk.

To create a bootable CD-ROM with the `install_light` demo system on it, perform the steps below.

1. Go to the `$BLUECAT_PREFIX/demo/install_light` directory:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/demo/install_light
   ```

2. Edit the `Makefile` file.

   Since the `mkboot` tool requires a CD-ROM device name to create a bootable CD-ROM, make sure that the correct device name is specified in the `$(KDI_NAME).iso` section of the `Makefile` file. This section has the following command:

   ```
   mkboot -o -k $(KDI_NAME).disk -r /dev/hdc -g $(KDI_NAME).isofs -c \
   $(KDI_NAME).txt $(KDI_NAME).iso
   ```

In the `mkboot` command line shown above, `/dev/hdc` is the CD-ROM device filename. If the CD-ROM device is not `/dev/hdc`, enter a correct CD-ROM device name instead.

3. Build the demo system:

```
BlueCat:$ make
```

4. Burn the `install_light.iso` image to a CD-R(W) disk using special software.
5. To boot BlueCat Linux on the target from the bootable CD-ROM with the `install_light` demo system, reset the target board, enter the BIOS setup and set CD-ROM device as the first boot device. Refer to the hardware documentation for the details about boot device priority in the BIOS setup.
6. Insert the disk with the `install_light` demo system and reset the target board.

After that, the `install_light` demo system will be installed to the hard disk using the GUI-based Installer.

Installing the install_light Demo System to the Hard Disk Using the GUI-based Installer

To install the `install_light` to the hard disk using the GUI-based Installer from the bootable CD-ROM, perform the following steps:

1. The GUI-based Installer starts, and the **Installation language module** window appears.

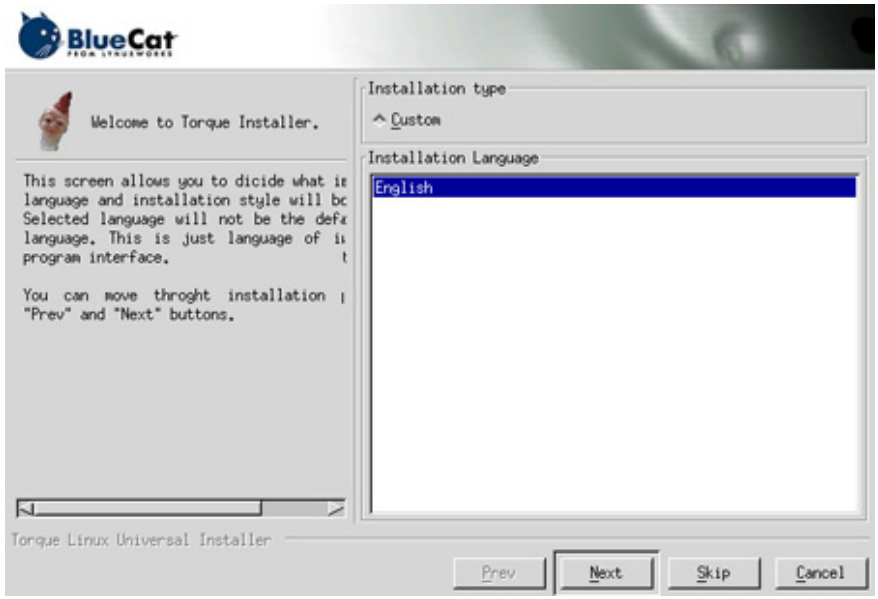

Figure 4-1: The Installation Language Modules Window

Since there is no choice in this window, click the **Next** button to go to the next window. The **Preparing drives** window appears.

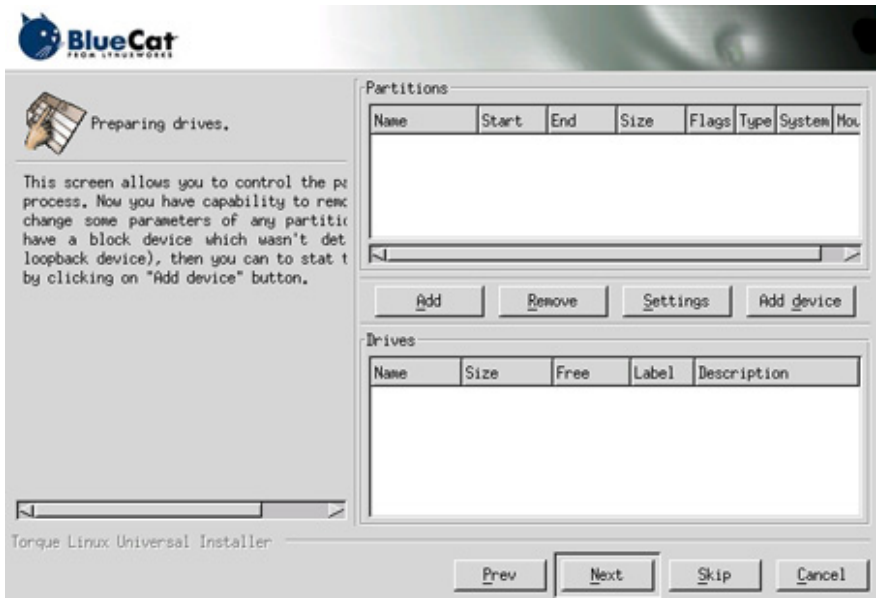

Figure 4-2: The Preparing drives Window

2. Use the **Preparing drivers** window to create partitions on the hard disk. First, a hard disk device should be selected: use the **Add device** button to open the **Enter block device path** window. Then, type the hard disk device name (the default is `/dev/hda`) and select **Ok**.

Figure 4-3: The Enter block device path Window

If there is no partition table on the hard disk the **Disk label name** window will appear to select a disk label name. Use `msdos` as a hard disk label.

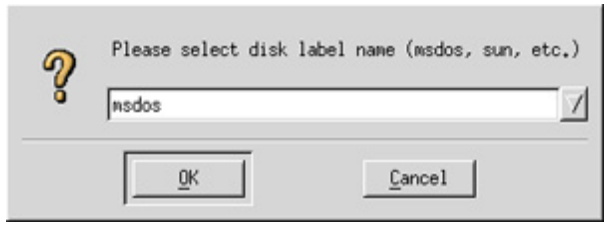

Figure 4-4: The Disk label name Window

At this point, the hard disk device should appear in the **Drives** section of the **Preparing drives** window.

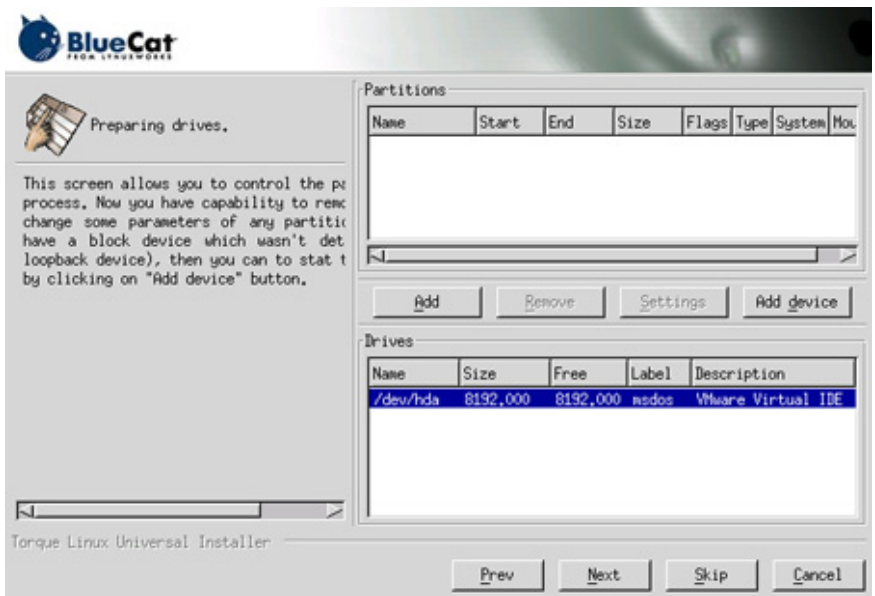

Figure 4-5: The Preparing drives Window with a Hard Disk Device

Then, two partitions should be created on the target hard disk. Set their parameters as follows:

A)Set the following parameters for the bootable partition (`/dev/hda1`):

- Size: at least 2 MB
- File system type: ext2

- Mount point: leave this field blank

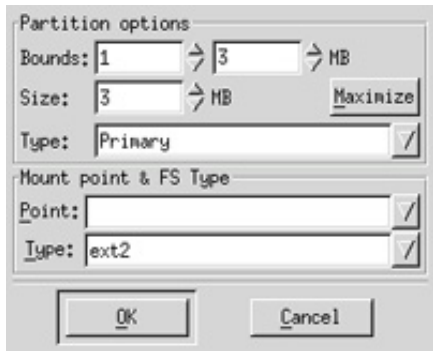

Figure 4-6: The Partition Options Dialog with the Options Set for the Bootable Partition

Click **OK** to return back to the **Preparing drives** window.

B)Set the following parameters for the partition that will hold the root file system (`/dev/hda2`):

Size: at least 30 MB

File system type: ext2

Mount point: `/`

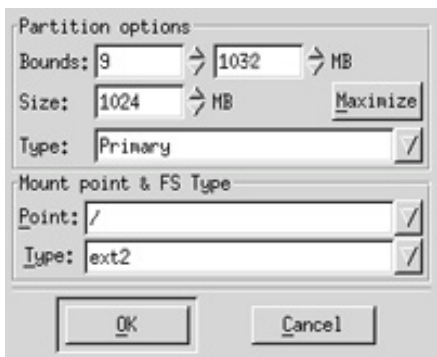

Figure 4-7: The Partition Options Dialog with the Options Set for the Root File System Partition

Click **OK** to return back to the **Preparing drives** window.

C)Click the **Next** button to go to the next step.

The **Formatting drives** window is displayed.

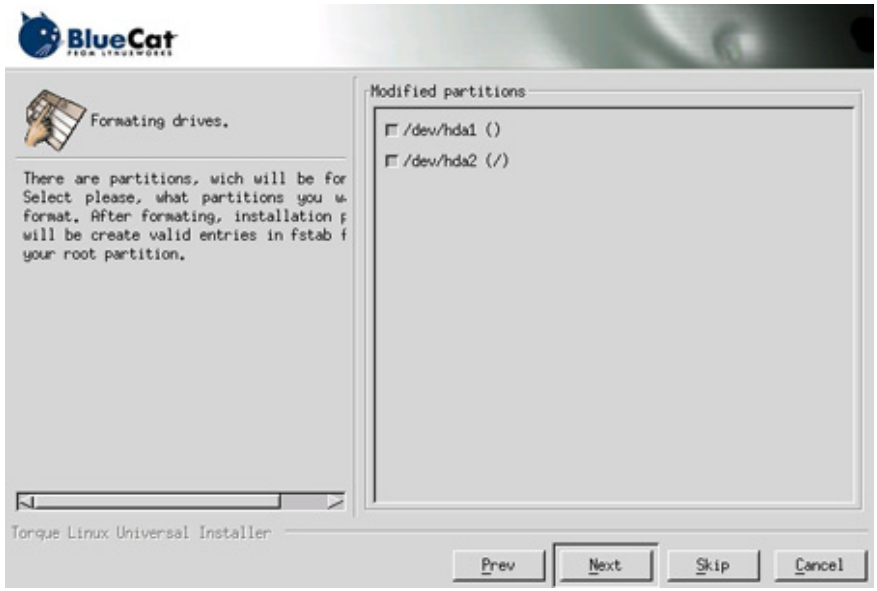

Figure 4-8: The Formatting drives Window

3. Use this window to select both items in the **Modified partitions** box and click **Next** to start formatting selected partitions.

As soon as formatting completes, the **Network setup** window appears.

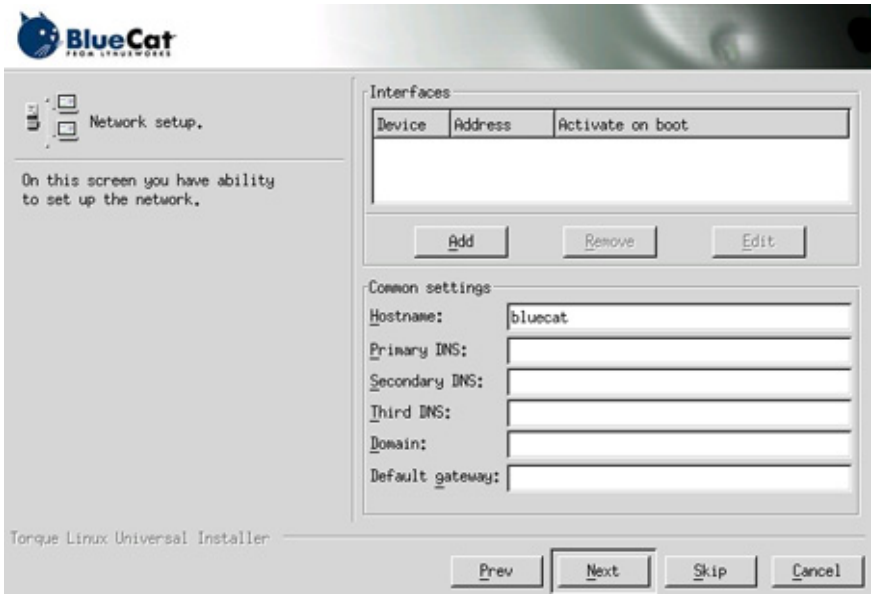

Figure 4-9: The Network setup Window

4. Use this window to add network interfaces:

   - Click the **Add** button to display the **Interface type** dialog box and click the **Loopback** radio button to add a loopback interface.

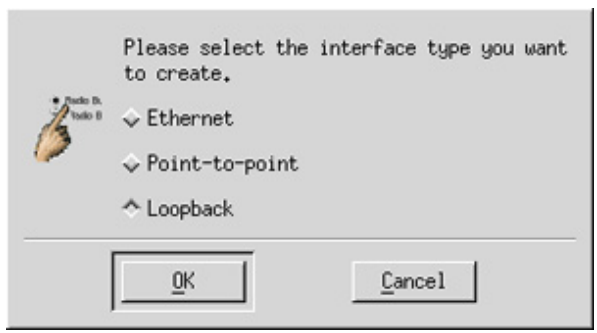

Figure 4-10: The Interface type Dialog with the Loopback Option Selected

Click **OK** to return to the **Network setup** main window. A new item (`lo`) appears in the **Interfaces** box.

- Click the **Add** button again to add an **Ethernet** interface.

Figure 4-11: The Interface type Dialog with the Ethernet Option Selected.

Click **OK** to display the **Interface settings** dialog.

5. Set the Ethernet interface parameters (the IP address, subnet mask, gateway address):

Figure 4-12: The Creating eth0 interface Dialog

Click **OK** to return to the **Network setup** main window.

6. Enter the other network parameters in the **Common settings** field:

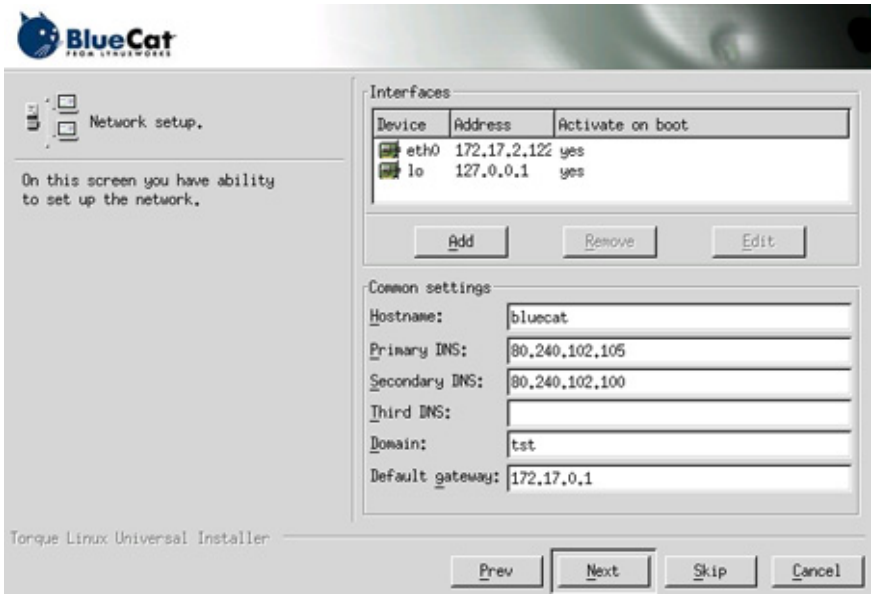

Figure 4-13: The Network setup Window with Selected interfaces and Common Settings

Click **Next** to go to the next step.

The **Software selection** window appears.

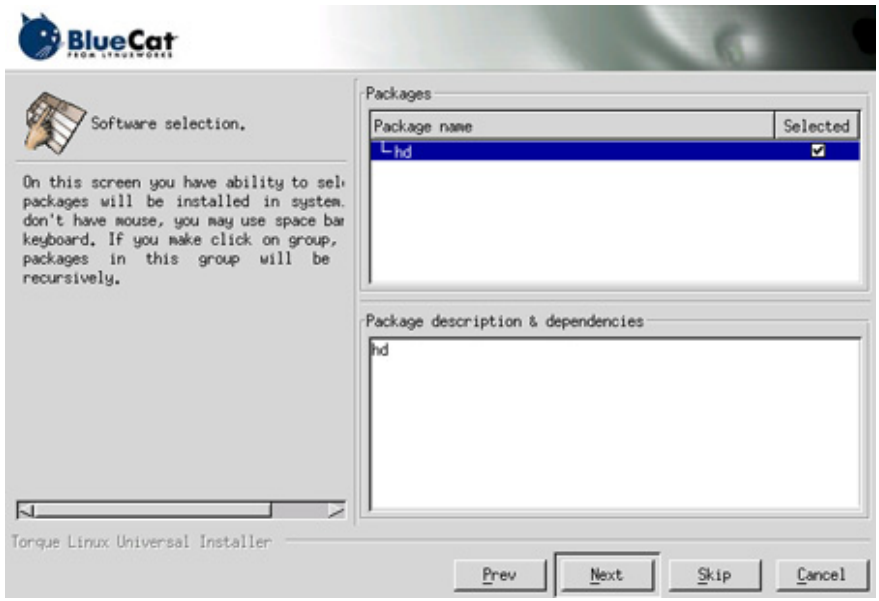

Figure 4-14: The Software selection" Window

7. In the **Packages** box, make sure that the **hd** package is selected to be installed (there is a check mark in the **Selected** field).
8. Click **Next** and then **Yes** to start installation.

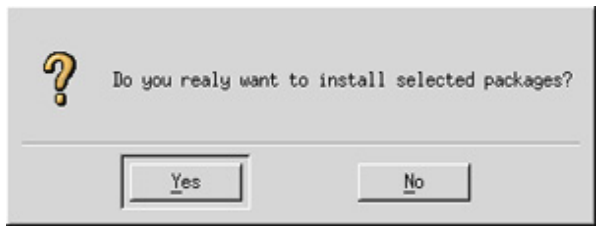

Figure 4-15: The Confirmation Dialog

As soon as installation is complete, the **Installation complete** window appears.

Figure 4-16: The Installation Complete Window

9. Click the **Finish** button to reboot the system.

As a result, the BlueCat Linux installed from the CD-ROM using the `install_light` demo system can be booted from the hard disk.

Booting the BlueCat Linux System Installed on the Hard Disk

To boot the BlueCat Linux system installed on the hard disk, do the following:

1. Reset the target.
2. Enter BIOS setup and set the hard disk as a first boot device. Refer to the hardware documentation for details about boot device priority.
3. Save BIOS setup and reset the board.

The BlueCat Linux will be booted from the hard disk after reset.

Using Selected RPM Packages

The following sections describe how to use selected RPM packages included in the BlueCat Linux distribution that are frequently deployed in the embedded systems environment.The described functionalities are illustrated by a step-wise procedure explaining creation and use of respective features.

Using the BusyBox RPM Package

The BusyBox RPM package combines tiny versions of many common UNIX utilities into a single small executable. It provides minimalist replacements for most UNIX utilities, such as `fileutils`, `shellutils`, `findutils`, `textutils`, `grep`, `gzip`, and `tar`. BusyBox provides a fairly complete POSIX environment for any small or embedded system.

The utilities in BusyBox generally have fewer options than their full-featured GNU counterparts; however, the options that are included provide the expected functionality and behave much like their GNU correlates.

The following sections describe the steps necessary for creating and booting a BlueCat Linux system containing BusyBox and demonstrate the use of the BusyBox utilities.

Creating a BlueCat Linux System for BusyBox

To create and boot a BlueCat Linux system containing BusyBox, perform the following steps:

1. Create a new directory by typing:

   ```
   BlueCat:$ mkdir -p \
   $BLUECAT_PREFIX/demo/busybox/local
   ```

2. Set up the BlueCat Linux kernel configuration using the standard kernel configuration tools, and copy the kernel configuration file to the `$BLUECAT_PREFIX/demo/busybox` directory. For instance, type the following commands:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/usr/src/linux
   BlueCat:$ make xconfig
   ```

   Select **Save and Exit** to update the `.config` file, then type the following command:

```
BlueCat:$ cp .config \
$BLUECAT_PREFIX/demo/busybox/busybox.config
```

NOTE: The kernel `.config` file for the `developer` demo (`$BLUECAT_PREFIX/demo/developer/developer.config`) is also recommended as a starting point.

3. Create a BlueCat Linux Kernel Downloadable Image (`busybox.kernel`).

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/busybox
BlueCat:$ mkkernel ./busybox.config \
./busybox.kernel ./busybox.disk
```

4. Create a `.spec` file (`busybox.spec`) with the following minimal directives:

```
strip on

mkdir /dev
mknod /dev/console c 5 1

mkdir    /lib
mkdir -p /usr/lib
mkdir    /bin
mkdir    /sbin
mkdir -p /etc/rc.d
mkdir    /proc
mkdir -p /etc/init.d

cp ./local/fstab ./local/inittab  /etc
cp ./local/rc.sysinit             /etc/init.d/rcS

lcd ${BLUECAT_PREFIX}/sbin
cp reboot busybox /sbin

ln -s /sbin/busybox /bin/msh
ln -s /sbin/busybox /sbin/init
ln -s /sbin/busybox /sbin/ifconfig
ln -s /sbin/busybox /sbin/route
ln -s /sbin/busybox /bin/mount
ln -s /sbin/busybox /bin/cat
ln -s /sbin/busybox /bin/ls
ln -s /sbin/busybox /bin/uname
ln -s /sbin/busybox /bin/date
ln -s /sbin/busybox /bin/chmod
ln -s /sbin/busybox /bin/echo
ln -s /sbin/busybox /bin/ping

chmod 755 /bin /sbin /etc/init.d/rcS
# End of File
```

5. Create the `local/fstab` file with the following contents:

```
proc    /proc   proc    defaults        0       0
```

6. Create the `local/inittab` file with the following contents:

```
# This is run first except when booting in single-user mode.
#
::sysinit:/etc/init.d/rcS

# /bin/sh invocations on selected ttys
#
# Start an "askfirst" shell on the console (whatever that may be)
::askfirst:-/bin/msh
```

NOTE: The first two fields in every record of the inittab file are ignored by the BusyBox init, so they must be empty. For example, the line `1:12345:respawn:/bin/sh` is not valid.

7. Create the `local/rc.sysinit` file with the following contents:

```
#!/bin/msh

PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH
>/etc/mtab
mount none /proc -t proc -n
cat /proc/mounts > /etc/mtab
```

8. Create a root file system image (`busybox.rfs`) by entering the following command:

```
BlueCat:$ mkrootfs -lv ./busybox.spec ./busybox.rfs
```

9. Create a composite BlueCat Linux image:

```
BlueCat:$ mkboot -m -k busybox.disk -f \
busybox.rfs busybox.kdi
```

Refer to the *BlueCat Linux User's Guide* for more information about a BlueCat Linux composite image.

NOTE: The Makefile for the `developer` demo system can be used to produce the BusyBox kernel and RFS images. To do so, modify the Makefile as follows:

i. Change the line `KDI_NAME = developer` to `KDI_NAME = busybox`.

ii. Comment out the following lines:

```
# IS_NEED_REBUILD_SRC=$(shell if ! make -q -C src; then echo this ; fi)
# cd src; make all
```

iii. Change the following lines:

```
clean    :
       rm -f *.rfs *.tar *.kernel *.disk *.kdi *.srec; cd src; make clean
```

to read as follows:

```
clean    :
       rm -f *.rfs *.tar *.kernel *.disk *.kdi *.srec
```

iv. Run the `make all` command.

v. If the following message appears:

```
make: circular busybox <- busybox dependency dropped.
make: *** No rule to make target `.rfs', needed by `busybox'. stop.
```

edit the Makefile to delete the following line:

```
KDI_NAME = busybox
```

Then retype the line in. There may be trailing characters on this line that cause errors in the Makefile.

Booting BusyBox Images from a Network

To boot the BlueCat Linux with the BusyBox utility over a network using the BlueCat Linux OS loader, perform the step below. Refer to Chapter 3 of the *BlueCat Linux User's Guide*, "Downloading and Booting BlueCat Linux" for details on the OS loader.

At the OS loader prompt (>), type the following commands:

```
> set IF eth0
> set IP <target_board_IP>
> set HOST <development_host_IP>
> set KERNEL tftp busybox.kernel
> set RFS tftp busybox.rfs
> boot
```

where *<target_board_IP>* is an IP address of the target and *<development_host_IP>* is an IP address of the development host.

Using BusyBox Utilities

This section provides the examples of using the BusyBox utilities. Entering a command from the following list results in the respective output:

- ls

```
# ls /
bin         etc         lost+found  sbin
dev         lib         proc        usr
```

- cat

```
# cat /etc/inittab
# This is run first except when booting in single-user mode.
#
::sysinit:/etc/init.d/rcS

# /bin/sh invocations on selected ttys
#
# Start an "askfirst" shell on the console (whatever that may be)
::askfirst:-/bin/msh
```

- chmod

```
# chmod a-x /sbin/reboot
# ls -la /sbin/reboot
-rw-r--r--    1 0        0           12410 Apr 20 09:13 /sbin/reboot
# chmod 755 /sbin/reboot
# ls -la /sbin/reboot
-rwxr-xr-x    1 0        0           12410 Apr 20 09:13 /sbin/reboot
```

- echo

```
# echo !!!!!!!
!!!!!!!
```

- date

```
# date
Fri Apr 21 16:18:50 UTC 2006
```

- uname

```
# uname -a
Linux (none) 2.6.13.4 #3 Thu Apr 20 05:12:09 EDT 2006 i686 unknown
```

- mount

```
# mount
/dev/root on / type ext2 (rw)
none on /proc type proc (rw,nodiratime)
```

- ifconfig

NOTE: In the example below, for the machine to ping the local host (itself), first enter `ifconfig lo 127.0.0.1` to make the local interface available.

```
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:E0:29:3D:C7:1D
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:5

# ifconfig eth0 172.17.1.2
# ping -c 5 172.17.0.1
PING 172.17.0.1 (172.17.0.1): 56 data bytes
64 bytes from 172.17.0.1: icmp_seq=0 ttl=255 time=2.7 ms
64 bytes from 172.17.0.1: icmp_seq=1 ttl=255 time=0.1 ms
64 bytes from 172.17.0.1: icmp_seq=2 ttl=255 time=0.1 ms
64 bytes from 172.17.0.1: icmp_seq=3 ttl=255 time=0.1 ms
64 bytes from 172.17.0.1: icmp_seq=4 ttl=255 time=0.1 ms

--- 172.17.0.1 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.6/2.7 ms
```

Using the TinyLogin RPM Package

The TinyLogin RPM package is a suite of tiny UNIX utilities for handling logging into, being authenticated by, changing one's password for, and otherwise maintaining users and groups on an embedded system. It also provides shadow password support to enhance system security.

The following sections describe the steps necessary for creating and booting a BlueCat Linux system containing TinyLogin and demonstrate the use of the TinyLogin utility.

Creating a BlueCat Linux System for TinyLogin

To create a BlueCat Linux image for TinyLogin, perform the following steps:

1. Create a new directory by typing:

   ```
   BlueCat:$ mkdir -p \
   $BLUECAT_PREFIX/demo/tinylogin/local
   ```

2. Set up the BlueCat Linux kernel configuration by using the standard kernel configuration tools, and copy the kernel configuration file to the `$BLUECAT_PREFIX/demo/tinylogin` directory.

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/usr/src/linux
   BlueCat:$ make menuconfig
   ```

   Select **Save and Exit** to update the `.config` file, then type the following command:

   ```
   BlueCat:$ cp .config \
   $BLUECAT_PREFIX/demo/tinylogin/tinylogin.config
   ```

NOTE: The kernel `.config` file for the `developer` demo (`$BLUECAT_PREFIX/demo/developer/developer.config`) is also recommended as a starting point.

---

3. Create a BlueCat Linux Kernel Downloadable Image (`tinylogin.kernel`):

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/demo/tinylogin
   BlueCat:$ mkkernel ./tinylogin.config \
   ./tinylogin.kernel ./tinylogin.disk
   ```

4. Create a `.spec` file (`tinylogin.spec`) that contains the following minimal directives:

```
strip on

mkdir /dev
mknod /dev/console c 5 1
ln -s /dev/console /dev/tty
ln -s /dev/console /dev/tty1

mkdir    /bin
mkdir    /sbin
mkdir -p /etc/rc.d
mkdir    /proc
mkdir    /tmp
mkdir -p /usr/bin

mkdir /root

mkdir /dev/pts
mknod /dev/ptmx c 5 2

chmod 0666 /dev/ptmx

cp ./local/fstab ./local/passwd ./local/inittab   /etc
cp ./local/securetty ./local/shadow               /etc
cp ./local/rc.sysinit                             /etc/rc.d
cp ${BLUECAT_PREFIX}/etc/shells                   /etc
chmod 644 /etc/shells
cp ${BLUECAT_PREFIX}/etc/group                    /etc

lcd ${BLUECAT_PREFIX}/sbin
cp reboot init mingetty /sbin

cp ${BLUECAT_PREFIX}/usr/bin/tinylogin            /usr/bin
ln -s /usr/bin/tinylogin                          /usr/bin/passwd
ln -s /usr/bin/tinylogin                          /bin/login

lcd ${BLUECAT_PREFIX}/bin
cp mount bash ls cat hostname /bin
ln -s /bin/bash /bin/sh

chmod 711 /etc/rc.d/rc.sysinit

chmod 755 /bin /sbin /usr/bin

chmod 04755 /usr/bin/tinylogin
# End of File
```

NOTE: In this `.spec` file, the `/bin/login` and `/usr/bin/passwd` symbolic links point to `/usr/bin/tinylogin`. This allows the user to change his/her password by simply typing **`passwd`**.

5. Create the `local/fstab` file with the following contents:

```
none /proc    proc
none /dev/pts devpts
```

6. Create the `local/inittab` file with the following contents:

```
id:1:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

1:12345:respawn:/sbin/mingetty tty1
```

7. Create the `local/securetty` file with the following contents:

```
console
tty1
```

8. Create the `local/passwd` file with the following contents:

```
root:x:0:0:/root:/:/bin/bash
guest:x:500:10::/:/bin/bash
```

9. Create the `local/shadow` file:

```
root::10942:0:99999:7:::
guest::500:10:99999:7:::
```

10. Create the `local/rc.sysinit` file with the following contents:

```
#!/bin/sh

PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH

mount -a
hostname myhostname
```

11. Create a root file system image (`tinylogin.rfs`) by entering the following command:

```
BlueCat:$ mkrootfs -lv ./tinylogin.spec \
./tinylogin.rfs
```

NOTE: The Makefile for the `developer` demo system can be used to produce the TinyLogin kernel and RFS images. To do so, modify the Makefile as follows:

i. Change the line `KDI_NAME = developer` to `KDI_NAME = tinylogin.`

ii. Comment out the following lines:

```
# IS_NEED_REBUILD_SRC=$(shell if ! make -q -C src; then echo this ; fi)
# cd src; make all
```

iii. Change the following lines:

```
clean    :
       rm -f *.rfs *.tar *.kernel *.disk *.kdi *.srec; cd src; make clean
```

to read as follows:

```
clean    :
       rm -f *.rfs *.tar *.kernel *.disk *.kdi *.srec
```

iv. Run the `make all` command.

v. If the following message appears:

```
make: circular tinylogin <- tinylogin dependency dropped.
make: *** No rule to make target `.rfs', needed by `tinylogin'. stop.
```

edit the Makefile to delete the following line:

```
KDI_NAME = tinylogin
```

Then retype the line in. There may be trailing characters on this line that cause errors in the Makefile.

Booting the TinyLogin Images from a Network

To boot BlueCat Linux with the TinyLogin utility from a network using the BlueCat Linux OS loader, perform the steps listed below. Refer to Chapter 3, "Downloading and Booting BlueCat Linux" in the *BlueCat Linux User's Guide* for additional details about the BlueCat Linux OS loader.

At the OS loader prompt, type the following commands:

```
> set IF eth0
> set IP <target_board_IP>
> set HOST <development_host_IP>
> set KERNEL tftp tinylogin.kernel
> set RFS tftp tinylogin.rfs
> boot
```

where <*target_board_IP*> is an IP address of the target and <*development_host_IP*> is an IP address of the development host.

The TinyLogin utility is loaded onto the target board and then automatically started.

Using the TinyLogin Utility

This section provides examples of using the TinyLogin utility:

- Changing the guest password:

```
myhostname login: guest
-bash-3.00$ passwd
Changing password for guest
Enter the new password (minimum of 5, maximum of 8 characters)

Please use a combination of upper and lower case letters and numbers.
Enter new password: <new_guest_password>
Re-enter new password: <new_guest_password>
passwd[14]: password for `guest' changed by user `guest'
Password changed.
-bash-3.00$ exit
```

- Changing the root password:

```
myhostname login: root
login[16]: root login  on `console'

-bash-3.00# passwd
Changing password for root
Enter the new password (minimum of 5, maximum of 8 characters)

Please use a combination of upper and lower case letters and numbers.
Enter new password: <new_root_password>
Re-enter new password: <new_root_password>
passwd[17]: password for `root' changed by user `root'
Password changed.
-bash-3.00# exit
myhostname login: root
Password: <new_root_password>
login[18]: root login  on `console'

-bash-3.00# exit
```

- Getting the root permissions:

```
myhostname login: guest
Password: <guest_password>
-bash-3.00$ tinylogin su
Password:
login[17]: root login  on `console'

-bash-3.00$
```

Using the Zebra RPM Package

GNU Zebra is free software that manages a TCP/IP-based routing protocol. It takes a multiserver and multithread approach to resolve the current complexity of the Internet.

GNU Zebra supports BGP4, BGP4+, OSPFv2, OSPFv3, RIPv1, RIPv2, and RIPng.

GNU Zebra is intended to be used as a Route Server and a Route Reflector. It is not a toolkit; it provides full routing power under a new architecture. GNU Zebra is unique in design in that it has a process for each protocol.

The following sections describe the steps necessary for creating and booting a BlueCat Linux system containing Zebra and demonstrate use of the Zebra utility.

Creating a BlueCat Linux System for Zebra

To create a BlueCat Linux image for Zebra, perform the following steps:

1. Create a new directory by typing:

   ```
   BlueCat:$ mkdir -p $BLUECAT_PREFIX/demo/zebra/local
   ```

2. Set up the BlueCat Linux kernel configuration by using the standard kernel configuration tools and copy the kernel configuration file to the `$BLUECAT_PREFIX/demo/zebra` directory. For instance, type the following commands:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/usr/src/linux
   BlueCat:$ make menuconfig
   ```

   Select **Save and Exit** to update the `.config` file, then type the following command:

   ```
   BlueCat:$ cp .config \
   $BLUECAT_PREFIX/demo/zebra/zebra.config
   ```

3. Create a BlueCat Linux Kernel Downloadable Image (`zebra.kernel`):

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/zebra
BlueCat:$ mkkernel ./zebra.config ./zebra.kernel \
./zebra.disk
```

4. Create a `.spec` file (`zebra.spec`) that contains the following minimal directives:

```
strip on

mkdir /dev
mknod /dev/null c 1 3
mknod /dev/console c 5 1
ln -s /dev/console /dev/tty
ln -s /dev/console /dev/tty1
# Standard 16550 serial driver device
mknod /dev/ttyS0   c 4 64
mknod /dev/ttyS1   c 4 65

mkdir -p /lib/security
mkdir -p /usr/lib
mkdir    /bin
mkdir    /sbin
mkdir -p /etc/rc.d
mkdir -p /etc/pam.d
mkdir -p /etc/xinetd.d
mkdir -p /etc/zebra
mkdir    /proc
mkdir    /tmp
mkdir -p /usr/bin
mkdir -p /usr/sbin
mkdir -p /var/run
mkdir -p /usr/libexec

mkdir -p /var/log/zebra

mkdir /root

mkdir /dev/pts
mknod /dev/ptmx c 5 2

chmod 0666 /dev/ptmx

cp ./local/fstab ./local/passwd ./local/inittab ./local/mtab /etc
cp ./local/securetty                                /etc/
chmod 0600 /etc/securetty
cp ./local/rc.sysinit                               /etc/rc.d
cp ./local/hosts                                    /etc
cp ./local/protocols                                /etc
cp ./local/resolv.conf                              /etc
cp ${BLUECAT_PREFIX}/etc/pwdb.conf                  /etc
cp ${BLUECAT_PREFIX}/etc/nsswitch.conf              /etc
cp ${BLUECAT_PREFIX}/etc/services                   /etc

cp ${BLUECAT_PREFIX}/etc/security /etc

cp ./local/shadow                                   /etc
cp ./local/pam.d/*                                  /etc/pam.d
cp ./local/xinetd.d/*                               /etc/xinetd.d
cp ./local/zebra.conf                               /etc/zebra/
```

```
cp ${BLUECAT_PREFIX}/lib/libnss_files-*.so       /lib
cp ${BLUECAT_PREFIX}/lib/libnss_dns-*.so         /lib
cp ${BLUECAT_PREFIX}/lib/security/*              /lib/security

cp ./local/empty   /var/log/wtmp

lcd ${BLUECAT_PREFIX}/sbin
cp reboot shutdown init mingetty ifconfig /sbin

cp ${BLUECAT_PREFIX}/etc/xinetd.conf             /etc

cp ${BLUECAT_PREFIX}/usr/bin/telnet              /usr/bin

cp ${BLUECAT_PREFIX}/etc/shells                  /etc
chmod 644 /etc/shells

cp ${BLUECAT_PREFIX}/etc/group                   /etc

#
# General Binaries
#
lcd ${BLUECAT_PREFIX}/bin
cp ping mount bash cat ls hostname ps            /bin
cp login /bin
ln -s /bin/bash /bin/sh

cp ${BLUECAT_PREFIX}/usr/bin/vtysh               /usr/bin

# internet services utils
cp ${BLUECAT_PREFIX}/usr/sbin/xinetd             /usr/sbin
cp ${BLUECAT_PREFIX}/usr/sbin/in.telnetd         /usr/sbin
cp ${BLUECAT_PREFIX}/usr/sbin/zebra              /usr/sbin

chmod 711 /etc/rc.d/rc.sysinit

chmod 755 /bin /sbin /usr/bin /usr/sbin

# End of File
```

5. Create the `local/inittab` file with the following contents:

```
id:1:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/sbin/halt
l6:6:wait:/sbin/reboot

ca::ctrlaltdel:/sbin/shutdown -t3 -r now

pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting
Down"

pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown
Cancelled"

1:12345:respawn:/sbin/mingetty tty1
```

6. Create the `local/rc.sysinit` file with the following contents:

```
#!/bin/sh

PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH

mount -a
xinetd -stayalive -reuse

hostname myhostname

zebra -d
```

7. Create the `local/zebra.conf` file with the following contents:

```
!
! zebra configuration file
!
hostname Router
password zebra
enable password zebra
!
! Interface's description.
!
interface lo
ip address 127.0.0.1/8

interface eth0
ip address 172.17.1.3/16

!
! Static default route.
!
ip route 80.240.0.0 255.255.0.0 172.17.0.1

log stdout
```

NOTE: This configuration file sets the password to `zebra`. The user has to enter this password when connecting to Zebra or changing the Zebra configuration mode by entering the `enable` command at the command prompt.

8. Copy the `fstab`, `passwd`, `mtab`, `hosts`, `protocols`, `resolv.conf`, `securetty`, `shadow`, `pam.d/*`, `xinetd.d/*`, and `empty` files from the `$BLUECAT_PREFIX/demo/developer/local` directory to the `$BLUECAT_PREFIX/demo/zebra/local` directory.
9. Create a root file system image (`zebra.rfs`) by entering the following command:

```
BlueCat:$ mkrootfs -lv ./zebra.spec ./zebra.rfs
```

NOTE: The Makefile for the `developer` demo system can be used to produce the Zebra kernel and RFS images. To do so, modify the Makefile as follows:

i. Change the line `KDI_NAME = developer` to `KDI_NAME = zebra`.

ii. Comment out the following lines:

```
# IS_NEED_REBUILD_SRC=$(shell if ! make -q -C src; then echo this ; fi)
# cd src; make all
```

iii. Change the following lines:

```
clean    :
       rm -f *.rfs *.tar *.kernel *.disk *.kdi *.srec; cd src; make clean
```

to read as follows:

```
clean    :
       rm -f *.rfs *.tar *.kernel *.disk *.kdi *.srec
```

iv. Run the `make all` command.

v. If the following message appears:

```
make: circular zebra <- zebra dependency dropped.
make: *** No rule to make target `.rfs', needed by `zebra'. stop.
```

edit the Makefile to delete the following line:

```
KDI_NAME = zebra
```

Then retype the line in. There may be trailing characters on this line that cause errors in the Makefile.

Booting the Zebra Images from a Network

To boot BlueCat Linux with the Zebra utility from a network using the BlueCat Linux OS loader, perform the steps listed below. Refer to Chapter 3, "Downloading and Booting BlueCat Linux" in the *BlueCat Linux User's Guide* for additional details about the BlueCat Linux OS loader.

At the OS loader prompt, type the following commands:

```
> set IF eth0
> set IP <target_board_IP>
> set HOST <development_host_IP>
> set CMD ramdisk_size=8192
> set KERNEL tftp zebra.kernel
> set RFS tftp zebra.rfs
> boot
```

where *<target_board_IP>* is an IP address of the target and *<development_host_IP>* is an IP address of the development host.

The Zebra utility is loaded onto the target board and then automatically started.

Using the Zebra Utility

This section provides examples of using the Zebra utility:

```
myhostname login: root
login(pam_unix)[21]: session opened for user root by (uid=0)
 -- root[21]: ROOT LOGIN ON console
-bash-3.00# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:01:00:00:FE:44
          inet addr:172.17.1.7  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:23 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1530 (1.4 KiB)  TX bytes:0 (0.0 b)
          Interrupt:4

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

-bash-3.00# ping -c 2 172.17.0.1
PING 172.17.0.1 (172.17.0.1) 56(84) bytes of data.
64 bytes from 172.17.0.1: icmp_seq=0 ttl=255 time=10.0 ms
64 bytes from 172.17.0.1: icmp_seq=1 ttl=255 time=0.000 ms

--- 172.17.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1010ms
rtt min/avg/max/mdev = 0.000/5.000/10.000/5.000 ms, pipe 2
-bash-3.00# ping -c 3 172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.
64 bytes from 172.17.0.3: icmp_seq=0 ttl=64 time=10.0 ms
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.000 ms
64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.000 ms

--- 172.17.0.3 ping statistics ---
```

```
3 packets transmitted, 3 received, 0% packet loss, time 2020ms
rtt min/avg/max/mdev = 0.000/3.333/10.000/4.714 ms, pipe 2
-bash-3.00# telnet localhost 2601
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Hello, this is zebra (version 0.93b).
Copyright 1996-2002 Kunihiro Ishiguro.


User Access Verification

Password:
Router> enable
Password:
Router# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       B - BGP, > - selected route, * - FIB route

S>* 80.240.0.0/16 [1/0] via 172.17.0.1, eth0
C>* 127.0.0.0/8 is directly connected, lo
C>* 172.17.0.0/16 is directly connected, eth0
Router#
```

Using the OpenSSH RPM Package

OpenSSH is the SSH (Secure SHell) protocol implementation. SSH replaces `rlogin` and `rsh` to provide secure encrypted communications between two hosts over an insecure network. X11 connections and arbitrary TCP/IP ports can also be forwarded over the secure channel. Public key authentication may be used for a passwordless access to servers.

To enable OpenSSH support in the BlueCat Linux `developer` demo system, perform the following steps:

1. Activate the BlueCat installation:

   ```
   bash$ . SETUP.sh x86
   ```

2. Enter the `developer` demo system directory:

   ```
   BlueCat:$ cd $BLUECAT_PREFIX/demo/developer
   ```

3. Add the following lines at the end of the `local/rc.sysinit` file:

   ```
   chmod 0111 /var/empty/sshd
   /etc/rc.d/init.d/sshd start
   ```

4. Add the following lines to the end of the `developer.spec` file:

   ```
   mkdir -p /etc/rc.d/init.d

   cp ${BLUECAT_PREFIX}/etc/rc.d/init.d/functions /etc/rc.d/init.d/
   ```

```
cp ${BLUECAT_PREFIX}/sbin/consoletype /sbin
mknod /dev/urandom c 1 9

mkdir -p /etc/ssh
mkdir -p /usr/libexec/openssh
mkdir -p /var/empty/sshd
mkdir -p /var/lock/subsys

cp ${BLUECAT_PREFIX}/etc/ssh/moduli /etc/ssh
cp ${BLUECAT_PREFIX}/usr/bin/ssh-keygen  /usr/bin
cp ${BLUECAT_PREFIX}/usr/libexec/openssh/ssh-keysign    /etc/ssh

cp ${BLUECAT_PREFIX}/etc/ssh/ssh_config    /etc/ssh
cp ${BLUECAT_PREFIX}/usr/bin/scp           /usr/bin
cp ${BLUECAT_PREFIX}/usr/bin/sftp          /usr/bin
cp ${BLUECAT_PREFIX}/usr/bin/slogin        /usr/bin
cp ${BLUECAT_PREFIX}/usr/bin/ssh          /usr/bin
cp ${BLUECAT_PREFIX}/usr/bin/ssh-add      /usr/bin
cp ${BLUECAT_PREFIX}/usr/bin/ssh-agent    /usr/bin
cp ${BLUECAT_PREFIX}/usr/bin/ssh-keyscan  /usr/bin

cp ${BLUECAT_PREFIX}/etc/pam.d/sshd        /etc/pam.d
cp ${BLUECAT_PREFIX}/etc/rc.d/init.d/sshd   /etc/rc.d/init.d
cp ${BLUECAT_PREFIX}/etc/ssh/sshd_config     /etc/ssh
cp ${BLUECAT_PREFIX}/usr/sbin/sshd        /usr/sbin
cp ${BLUECAT_PREFIX}/usr/libexec/openssh/sftp-server
/usr/libexec/openssh
```

5. Add the following lines to the end of the `local/passwd` file:

   ```
   sshd:*:74:74:Privilege-separated SSH:/var/empty/sshd:
   ```

6. Add the following lines to the end of the `local/shadow` file:

   ```
   sshd:*:10942:0:99999:7:::
   ```

7. Rebuild the `developer` root file system:

   ```
   BlueCat:$ make rootfs
   ```

Using the ssh Client Utility and the ssh Server

This section provides ssh client and server demonstration:

1. Boot the `developer` demo system onto a target board from a network (refer to the *BlueCat Linux User's Guide* for more information). As soon as booting is complete, login as `root` and bring up the network interface(s) manually using the `ifconfig` command, and set up the kernel routing table using the `route` command.

   ```
   myhostname login: root
   login(pam_unix)[46]: session opened for user root by LOGIN(uid=0)
    -- root[46]: ROOT LOGIN ON console
   -bash-3.00# ifconfig eth0 172.17.1.27
   -bash-3.00# route add default gw 172.17.0.1
   ```

2. Connect to a remote host using the `ssh` utility:

```
bash-3.00# ssh 80.240.102.251
The authenticity of host '80.240.102.251 (80.240.102.251)' can't be
established.
RSA key fingerprint is
dd:e5:b1:58:87:23:83:5d:68:1e:e4:e1:df:d3:6b:92.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '80.240.102.251' (RSA) to the list of known
hosts.
root@80.240.102.251's password:
Last login: Mon Mar 21 12:57:07 2005 from pulsar
[root@cash root]#
```

3. Connect to the target board using the `ssh` utility:

```
[root@cash root]# ssh 172.17.1.27
The authenticity of host '172.17.1.27 (172.17.1.27)' can't be
established.
RSA key fingerprint is
81:82:59:86:be:37:3e:aa:c0:a8:e0:82:86:f0:e1:24.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.17.1.27' (RSA) to the list of known
hosts.
root@172.17.1.27's password:
sshd(pam_unix)[54]: session opened for user root by (uid=0)
-bash-3.00#
```

Supported RPM Packages

Table 4-7 lists all the packages included in the default BlueCat Linux configuration.

NOTE: Please note that the set of supported RPMs for MMU-less targets differs from the one listed in Table 4-7. For the list of RPMs supported by BlueCat Linux for MMU-less platforms, refer to the *Board Support Guide* document.

Table 4-7: Supported RPM Packages

Package	Description	Target / CDT
`apr`	Apache Portable Runtime library.	Target
`apr-util`	Apache Portable Runtime Utility library.	Target
`autoconf`	A GNU tool for automatically configuring source code.	CDT

Table 4-7: Supported RPM Packages (Continued)

Package	Description	Target/ CDT
autoconf213	A GNU tool for automatically configuring source code.	CDT
automake	A GNU tool for automatically creating Makefiles.	CDT
automake14	A GNU tool for automatically creating Makefiles.	CDT
bash	The GNU Bourne Again shell (bash) version 3.0.	Target
bc	The miscellaneous BlueCat Linux utilities.	CDT
binutils	A GNU collection of binary utilities.	CDT
busybox	Statically linked binary providing simplified versions of system commands.	Target
bzip2	A file compression utility.	CDT
coreutils	The GNU core utilities.	Target/CDT
coreutils	The GNU core utilities.	Target
cpio	A GNU archiving program.	Target
cpp	The C Preprocessor.	CDT
cpufreq-utils	CPU Frequency changing related utilities.	Target
cracklib	A password-checking library.	Target/CDT
db4	The Berkeley DB database library (version 4) for C.	Target
demo	The BlueCat Linux demo.	Target
dhclient	The DHCP client daemon and dhclient-script.	Target
dhcp	A DHCP (Dynamic Host Configuration Protocol) server and relay agent.	Target
diffutils	A GNU collection of diff utilities.	Target/CDT
e2fsprogs	Ext2 filesystem-specific static libraries and headers.	Target/CDT
expat	A library for parsing XML.	Target

Table 4-7: Supported RPM Packages (Continued)

Package	Description	Target / CDT
ffs	Userland tools interfacing to the JFFS filesystem.	Target
file	A utility for determining file types.	CDT
findutils	The GNU versions of find utilities (find and xargs).	Target
flex	A tool for creating scanners (text pattern recognizers).	CDT
fpga-bitstream	Prebuilt bitstream for the Xilinx ML507 boards.	Target
freetype	A free and portable TrueType font rendering engine.	Target/CDT
ftp	The standard UNIX FTP (File Transfer Protocol) client.	Target
gawk	The GNU version of the awk text processing utility.	Target/CDT
gcc	GNU development tools.	CDT
gdb	A GNU source-level debugger for C, C++ and Fortran.	CDT
gdbserver	A server for GNU source-level debugger for C, C++ and Fortran.	Target
gettext	GNU libraries and utilities for producing multi-lingual messages.	CDT
glib	The GIMP ToolKit (GTK+) and GIMP Drawing Kit (GDK) support library.	Target/CDT
glib2	The GIMP ToolKit (GTK+) and GIMP Drawing Kit (GDK) support library.	Target
glibc	The GNU libc libraries.	Target/CDT
grep	The GNU versions of grep pattern matching utilities.	Target/CDT
gtk+	Development tools for GTK+ (GIMP ToolKit) applications.	Target

Table 4-7: Supported RPM Packages (Continued)

Package	Description	Target/ CDT
gzip	The GNU data compression program.	Target
httpd	Apache HTTP Server.	Target
info	A stand-alone TTY-based reader for GNU texinfo documentation.	CDT
initscripts	The inittab file and the /etc/init.d scripts.	Target
iproute	Advanced IP routing and network device configuration tools.	Target
iputils	Network monitoring tools including ping.	Target
irshd	rshd server with modifications to run interactive commands.	Target
kernel	The source code for the Linux kernel.	Target
krb5	The shared libraries used by Kerberos 5.	Target
lddyn_fix	A package to make the cross tools work in a cross-development environment.	CDT
less	A text file browser.	CDT
libgcc	GCC version 3.2 shared support library.	CDT
libstdc++	GNU Standard C++ Library.	Target
libtermcap	Development tools for programs which will access the termcap database.	Target/CDT
libtool	The GNU Portable Library Tool.	CDT
libuser	A user and group account administration library.	Target
libxml	The libXML library.	Target
logrotate	Rotates, compresses, removes and mails system log files.	Target
m4	The GNU macro processor.	CDT
mailcap	Associates helper applications with particular file types.	Target

Table 4-7: Supported RPM Packages (Continued)

Package	Description	Target/ CDT
make	A GNU tool which simplifies the build process for users.	CDT
man-pages	Man (manual) pages from the Linux Documentation Project.	Target
mingetty	A compact getty program for virtual consoles only.	Target
mkboot	The BlueCat Linux utility for installing kernel upon a bootable media.	Target/CDT
mkrootfs	The BlueCat Linux utility for building a filesystem image according to a specfile.	CDT
mktemp	A small utility for safely making /tmp files.	Target/CDT
mod	SSL/TLS module for the Apache HTTP server.	Target
module-init-tools	Kernel module management utilities.	Target
ncompress	Fast compression and decompression utilities.	Target
ncurses	A CRT screen handling and optimization package.	Target/CDT
net-tools	Basic networking tools.	Target
nfs-utils	NFS utlilities and supporting daemons for the kernel NFS server.	Target
nptl	Header files and static libraries for development using NPTL library.	Target
openssh	OpenSSH clients.	Target
pam	A security tool which provides authentication for applications.	Target
parted	The GNU disk partition manipulation program.	Target
passwd	The passwd utility for setting/changing passwords using PAM.	Target

Table 4-7: Supported RPM Packages (Continued)

Package	Description	Target/ CDT
patch	The GNU patch command, for modifying/upgrading files.	CDT
pcdsrvr	PosixWorks Cross Desk Server.	Target
pcre	Perl-compatible regular expression library.	Target/CDT
pftpd	The BlueCat Linux parallel port booting daemon.	CDT
popt	A C library for parsing command line parameters.	Target
portmap	A program which manages RPC connections.	Target
procps	System and process monitoring utilities.	Target
psmisc	Utilities for managing processes on your system.	Target
readline	A library for editing typed command lines.	Target/CDT
rpm	The RPM package management system.	CDT
rsh	Client and servers for remote access commands (rsh, rlogin, rcp).	Target
sed	A GNU stream text editor.	Target/CDT
setup	A set of system configuration and setup files.	Target
shadow-utils	Utilities for managing accounts and shadow password files.	Target
slang	The shared library for the S-Lang extension language.	Target
sysfsutils	sysfsutils, library interface to sysfs.	Target
sysklogd	System logging and kernel message trapping daemons.	Target
SysVinit	Programs which control basic system processes.	Target
tar	A GNU file archiving program.	Target
tcp	A security tool which acts as a wrapper for TCP daemons.	Target

Table 4-7: Supported RPM Packages (Continued)

Package	Description	Target / CDT
telnet	The client and server program for the Telnet remote login protocol.	Target
texinfo	Tools needed to create Texinfo format documentation files.	CDT
tftp	The client and server for the Trivial File Transfer Protocol (TFTP).	Target
tinylogin	TinyLogin is a suite of tiny Unix utilities.	Target
torque	Torque installer.	Target
util-linux	A collection of basic system utilities.	Target
vim	The VIM editor.	Target
wu-ftpd	An FTP daemon provided by Washington University.	Target
XFree86	The basic fonts, programs and docs for an X workstation.	CDT
xinetd	A secure replacement for inetd.	Target
xorg-x11	The basic fonts, programs and docs for an X workstation.	Target
zebra	Routing daemon.	Target
zlib	The zlib compression and decompression library.	Target/CDT

CHAPTER 5 *Flash Support and Journalling Flash File System*

This chapter provides a detailed description of Flash memory support and the Journalling Flash File System (JFFS) and the Journalling Flash File System version 2 (JFFS2) in BlueCat Linux.

Flash Support and JFFS/JFFS2 Architecture

This section provides a general overview of the BlueCat Linux Flash memory support architecture.

BlueCat Linux Interfaces to Flash Memory

BlueCat Linux supports the following interfaces to Flash memory devices for user-space processes:

- `mtdchar` character-device interface
- `mtdblock` block-device interface
- JFFS or JFFS2 file system

Regardless of the interface is used to access Flash memory, access to an actual Flash memory device occurs via the Memory Technology Device (MTD) interface. The MTD interface provides an abstraction layer, which allows the upper layers of the Flash memory support software to perform specific operations on Flash memory via an open, device-independent interface.

Flash memory support architecture implemented by BlueCat Linux is shown in the figure below.

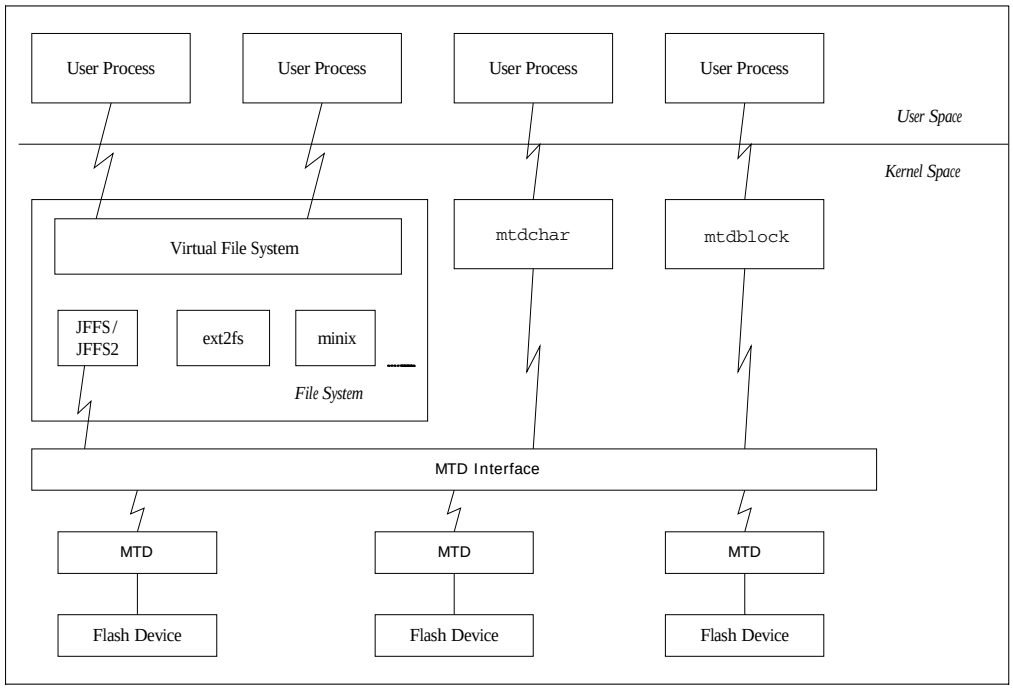

Figure 5-1: Flash Memory Support Architecture

The mtdchar Interface

The `mtdchar` interface provides access to an entire Flash memory device or partition using the character-device interface. The `mtdchar` interface lets the user access Flash memory as a file, using the standard `open()`, `read()`, `lseek()`, and other POSIX system calls, all of which have their standard interpretations.

All `mtdchar` operations are synchronous; each call results in physical access to Flash memory, unless a call is a logical operation.

The `ioctl()` call is supported for `mtdchar`. It implements a number of Flash memory-specific commands, such as erasing a specified Flash memory sector. BlueCat Linux Flash memory support includes a special tool, `flash_erase`, used to erase an entire Flash memory device or partition. This tool makes use of Flash

memory-specific IOCTL commands implemented by the `mtdchar` interface. See "flash_erase" in Appendix A, "Command Reference" for more information.

NOTE: The `mtdchar` interface does not perform an erase of appropriate Flash memory sectors on a `write()` call. It is the user's responsibility to ensure that Flash memory is erased before it is written to with the `mtdchar` interface.

The `mtdchar` interface is designed to provide raw access to Flash memory. The user always has full control of Flash memory operations, so that the actual device is erased and written to when the user commands `mtdchar` to do so.

Access to the `mtdchar` interface is through character device special nodes with a major number of `MTD_CHAR_MAJOR (90)`. BlueCat Linux uses the device nodes `/dev/mtdchar<x>` to access the `mtdchar` interface, although character device files with other names can also be used, as long as the major number is set to `MTD_CHAR_MAJOR`.

The minor number of an `mtdchar` device node is used to distinguish between Flash memory devices and partitions within a single Flash memory device. This is discussed under "Flash Memory Partitioning" on page 168.

The mtdblock Interface

The `mtdblock` interface provides access to an entire Flash memory device or partition using the block-device interface. The `mtdblock` interface presents Flash memory as an entity that can host a file system. In fact, the only recommended use of the `mtdblock` interface is in the `mount` command to refer to a Flash memory device or a Flash memory partition that needs to be mounted either as a Journalling Flash File System (JFFS) or as a Journalling Flash File System version 2 (JFFS2).

Like the `mtdchar` interface, `mtdblock` can be accessed using the standard POSIX operations, all of which have their standard interpretations. Unlike `mtdchar`, however, `mtdblock` is not synchronous. From the point of view of kernel architecture, `mtdblock` implements a block device, so when the user attempts to access it as a file, all operations are subject to every block device access mechanism implemented by the Linux kernel.

Thus, upon return from a system call, the `mtdblock` interface cannot guarantee that the contents of physical Flash memory are coherent with the data in the kernel block device buffers. In general, access to a Flash memory device as a file should always be via `mtdchar` rather than `mtdblock`.

The `mtdblock` interface supports the `ioctl()` call. Only the standard block device IOCTL commands (`BLKGETSIZE` and `BLKFLSBUF`) are supported. There is no command to erase a Flash memory sector or perform any other Flash memory-specific operations.

Access to the `mtdblock` interface is provided by means of block device special nodes with a major number of `MTD_BLOCK_MAJOR (31)`. BlueCat Linux uses the device nodes `/dev/mtdblock<x>` to access the `mtdblock` interface.

Like `mtdchar`, the minor number of an `mtdblock` device node is used to distinguish between Flash memory devices, and partitions within a single device.

Journalling Flash File System (JFFS/JFFS2) Interface

The Journalling Flash File System (JFFS)/Journalling Flash File System version 2 (JFFS2) is a POSIX-compliant file system implemented on Flash memory. JFFS/JFFS2 resides underneath the Virtual File System (VFS) layer. This means that, as with any other type of file system supported by Linux, any system call on a JFFS/JFFS2 or its files and directories goes through the VFS, which directs appropriate tasks to the JFFS/JFFS2 layer.

JFFS/JFFS2 is designed for the efficient use of Flash memory devices. It has built-in wear leveling, power loss recovery, and bad block mapping features.

JFFS/JFFS2 supports the following types of files defined by the POSIX.1 standard:

- Regular files
- Directories
- Special device files
- Symbolic links
- Named pipes

The following file system-specific calls defined by the POSIX.1 standard are supported by the JFFS/JFFS2 on the above types of files:
`chmod()`, `chown()`, `close()`, `closedir()`, `ioctl()`, `lseek()`, `mkdir()`, `mkfifo()`, `mknod()`, `open()`, `opendir()`, `read()`, `readdir()`, `readlink()`, `rename()`, `rmdir()`, `stat()`, `symlink()`, `truncate()`, `unlink()`, `write()`.

All writes are performed synchronously in JFFS. This means that upon return from a `write()` call, all data (and meta data) is physically written to Flash memory. Read accesses use the standard Linux cache and buffering mechanisms.

A user process specifies a Flash memory device or partition that a JFFS must be created in by using an appropriate block device node (`/dev/mtdblock<x>`) as a parameter to the `mount` command.

MTD Interface

The Memory Technology Device (MTD) interface is an abstraction layer between the upper layers of the Flash memory support software and low-level device drivers for specific Flash memory devices. The JFFS/JFFS2, `mtdchar`, and `mtdblock` all use the MTD interface, rather than directly implementing any aspects of low-level programming of a particular Flash memory device.

The MTD interface allows the upper layers of Flash memory support software to perform specific operations on Flash memory via an open, device-independent interface. In other words, to perform a particular task on a Flash memory device, the upper layers call the appropriate entry point of an MTD driver responsible for support of the Flash memory device. The MTD driver translates a device-independent request into low-level, device-specific operations performed on the actual Flash memory device.

Adding support for a new Flash memory device is as simple as developing a new MTD driver. The upper layers and user interfaces all remain unchanged.

To register at the MTD interface, an MTD driver allocates and populates a data structure of the `struct mtd_info` type. This data structure contains information about the supported device and pointers to the driver's access routines. Then, the MTD driver calls a registration service passing the `mtd_info` structure as a parameter. Provided the registration is successful, the MTD interface finds out about the new MTD and how to call it for specific Flash memory operations.

Access routines implemented by an MTD driver must conform to the rules specified by the MTD interface. This allows for seamless interplay between the upper Flash memory support layers and MTD drivers.

For successful Flash memory device support, there are a few entry points defined by the MTD interface that must be implemented by an MTD driver. These mandatory entry points include erase, write, and read operations. The upper layers of the Flash memory software handle all device-independent aspects of Flash memory management and call an MTD driver's entry point only when an operation on the physical Flash memory is required. An MTD driver functions independent of the reason that a particular low-level operation is needed or whether it has been initiated at JFFS/JFFS2, `mtdchar`, or `mtdblock`.

An interesting illustration of this general concept is the support of Flash memory devices composed of sectors of nonuniform size. When an MTD for such a device registers itself at the MTD interface, it provides a full description of the Flash memory device geometry in the `mtd_info` structure. The upper layers of the Flash memory software make use of this information, thus ensuring that access to a logical Flash memory region results in a correct sequence of calls to the MTD driver.

Flash Memory Partitioning

BlueCat Linux supports multiple partitions in a single Flash memory device. This feature is very important for many embedded applications, as it allows implementation of an arbitrary storage hierarchy using a single Flash memory chip. For instance, it is possible to maintain more than one Journalling Flash File System/Journalling Flash File System version 2 in a single Flash memory device, or use particular sectors of Flash memory as a JFFS/JFFS2 while leaving remaining sectors for other use (for instance, as storage for binary data).

Partitioning Method

Partitioning occurs at the MTD layer and works as follows: An MTD driver that needs to maintain several partitions in the Flash memory device it services, calls the MTD registration service for each partition, in addition to the initial registration call for the entire Flash memory device. The `mtd_info` structure passed to the MTD layer, at the time when a partition is being registered, describes the geometry of the Flash memory partition rather than the entire device's geometry.

The upper layers of the Flash memory software ensure that access to a Flash memory device or partition results in the use of an appropriate `mtd_info` structure. This, in turn, ensures that logical access to Flash memory is translated into the terms of the geometry described by the appropriate `mtd_info` structure, whether it corresponds to an entire Flash memory device or only to a partition.

It is possible that a Flash memory device is not partitioned at all. In this case, the MTD registers itself just once, passing the geometry of the entire Flash memory device to the MTD interface.

Also, it is important to note that since the partitioning is at the MTD layer, the upper layers use a concrete Flash memory entity (a device or a partition) in the most appropriate manner for the embedded application at hand. In other words, a Flash memory device or partition can be either mounted as a JFFS/JFFS2 via a

block device node (`/dev/mtdblockx`) or raw-accessed via a character device node (`/dev/mtdcharx`).

Flash Memory Entities and Device Nodes

A user process specifies a particular Flash memory device or partition by using an appropriate device node. There is a one-to-one relationship between Flash memory device nodes and Flash memory entities maintained by BlueCat Linux.

To elaborate further, a Flash memory entity (a device or a partition) has one device node corresponding to it within a device group (character devices or block devices). This means that each Flash memory entity can, in fact, have two device nodes corresponding to it: One for the character interface (`mtdchar`), and the other for the block interface (`mtdblock`).

A Flash memory character device node (`/dev/mtdcharx`) always has a major number of `MTD_CHAR_MAJOR (90)`. A Flash memory block device node (`/dev/mtdblockx`) always has a major number of `MTD_BLOCK_MAJOR (31)`.

The minor number is used to distinguish between Flash memory devices, and then between partitions within a device. The function used to determine the minor number for a concrete Flash memory entity (a device or a partition) is very simple: the minor number of the entity is equal to the number of Flash memory entities that have registered at the MTD interface before that entity.

Consider, for instance, three MTDs:

- MTD1 creates three partitions
- MTD2 does not create any partitions
- MTD3 creates two partitions.

Provided the MTDs register in the order they are defined above, MTD1 performs four registrations: one for the entire device and one each for the three partitions. Consequently, the minor number for the MTD1 device is `0`, while the minor numbers for its partitions are `1` to `3`. MTD2 does not create any partitions, so it registers only once for the entire device. It has the minor number `4`. MTD2 registers three times and has the minor number `5` for the entire device and the minor numbers `6` and `7` for its two partitions.

Typically, for a concrete embedded system, there is just one Flash memory device and, therefore, just one MTD. Furthermore, the logical layout of the Flash memory is often decided upon at the time of application design. Hence, it is not changed at runtime. This means that there is a fixed number of partitions in the Flash memory device, so the minor numbers for Flash memory entities are known *a priori*.

BlueCat Linux, however, allows runtime partitioning of Flash memory devices. In other words, theoretically, in some advanced Flash memory configurations, the user might have difficulty calculating the minor number for a concrete Flash memory device or partition. To facilitate administering such advanced configurations, BlueCat Linux maintains a `proc` file, `/proc/mtd`. This file can be read from the user space at any time. Each line of the file has a description of a Flash memory entity. The order of the lines is the same as that in which the entities have registered themselves with the MTD interface.

Partition Configuration

Any of the following means may be chosen to define the sectors of a Flash memory device to be used for a concrete partition:

- Define the partition configuration as a set of kernel build-time configuration parameters.
- Pass the partition configuration to the kernel as a set of kernel runtime parameters at boot time.
- Use the target board `flash_disk` utility to define the partition configuration at runtime.

Please refer to the remainder of this chapter for a detailed description of the Flash memory management tools and mechanisms.

JFFS Internals

JFFS Layout

All data physically stored in JFFS is divided into chunks. Each chunk starts with a control structure called `raw_inode`:

```
struct jffs_raw_inode
{
  __u32 magic;       /* A constant magic number.  */
  __u32 ino;         /* Inode number.  */
  __u32 pino;        /* Parent's inode number.  */
  __u32 version;     /* Version number.  */
  __u32 mode;        /* The file's type or mode.  */
  __u16 uid;         /* The file's owner.  */
  __u16 gid;         /* The file's group.  */
  __u32 atime;       /* Last access time.  */
  __u32 mtime;       /* Last modification time.  */
  __u32 ctime;       /* Creation time.  */
```

```
  __u32 offset;     /* Where to begin to write.  */
  __u32 dsize;      /* Size of the node's data.  */
  __u32 rsize;      /* How much are going to be replaced?  */
  __u8 nsize;       /* Name length.  */
  __u8 nlink;       /* Number of links.  */
  __u8 spare   : 6; /* For future use.  */
  __u8 rename  : 1; /* Is this a special rename?  */
  __u8 deleted : 1; /* Has this file been deleted?  */
  __u8 accurate;     /* The inode is obsolete if accurate ==\ 0.*/
  __u64 call_num    /* write() call number  */
  __u32 num_nodes   /* Number of the nodes written within a\
call
                     */
  __u32 alignment;  /* This makes an alignment of fields\
predictable
                    */
  __u32 dchksum;   /* Checksum for the data.  */
  __u16 nchksum;   /* Checksum for the name.  */
  __u16 chksum;    /* Checksum for the raw inode.  */
};
```

`raw_inode` is followed by a filename and a piece of file data, according to the values of the `nsize` and `dsize` members of the above structure, respectively. The filename stored with `raw_inode` is not absolute, but relative to the parent directory. The `pino` (parent inode number) field of the `raw_inode` structure is used to maintain a complete directory tree.

A directory is represented in JFFS by a `raw_inode` with the `mode` field that has the `S_IFDIR` bit set and with no node data following the directory name. Each `raw_inode` of the files local to the directory has the `pino` field pointing to the `raw_inode` of the directory.

Both the filename and the file data may not be present in a node. There is no need for the filename to be present in each `raw_inode` associated with the file because all `raw_inodes` for a file have the same value of the `ino` field.

When the file is renamed, a new `raw_inode` with the same `ino` field, the new filename, but no file data is created. If only the mode or access permissions or the modification time of the file is changed, then a new `raw_inode` with the appropriate `mode`, `mtime`, and `atime` fields, but without the filename and data, is created.

There is a limitation on the length of a chunk, depending on the sector size of the underlying Flash memory parts. A chunk cannot be longer than half of the largest Flash memory sector.

When a file is stored in the file system, it is split into several data chunks, so that the `offset` field of the `raw_inode` structure points to where the chunk's data is located at the beginning of the file.

When a file stored in the file system is modified (renamed, deleted, has data appended to it, and so on), no actual deletions or modifications are performed on

the data already stored in Flash memory. Instead, `raw_inodes` with up-to-date control information and/or file data and an incremented `version` field are written to the free space of Flash memory. The underlying concept of a log structure in JFFS is that the JFFS is a recording of the VFS commands. Reading a file is like replaying the VFS commands stored in Flash memory in the correct order.

The relevant fields are set as follows:

- `ino`—node number associated with the file
- `pino`—Inode number of the parent directory
- `version`—Highest version of all `raw_inodes` of the file previously written + 1
- `offset`—Position in the file at which `write()` is performed
- `dsize`—Size of data appended to the file
- `rsize`—`0`
- `nsize`—`0`

If data is written to a position within the file, then a `raw_inode` is created. The `rsize` field is used to indicate the amount of data being erased (rewritten). The relevant fields are set as follows:

- `ino`—Inode number associated with the file
- `pino`—Inode number of the parent directory
- `version`—Highest version of all `raw_inodes` of the file previously written + 1
- `offset`—Position in the file at which `write()` is performed
- `dsize`—Size of data rewritten in the file
- `rsize`—Size of data rewritten in the file
- `nsize`—`0`

The JFFS maintains an in-RAM list of pointers to all the actual nodes of each file. Hence, when a file is being read, the system looks through the list to find where the data requested is located in Flash memory.

Consider the following code fragment, which writes data to a file and then overwrites two portions of it:

```
char msg1[1000];
char msg2[100];
char msg3[50];
```

```
int hdl = open ("test",O_CREAT+O_RDWR);
memset(msg1,'A',sizeof(msg1));
write(hdl,msg1,sizeof(msg1));

lseek(hdl,600,SEEK_SET);
memset(msg2,'B',sizeof(msg2));
write(hdl,msg2,sizeof(msg2));

lseek(hdl,200,SEEK_SET);
memset(msg3,'C',sizeof(msg3));
write(hdl,msg3,sizeof(msg3));
```

Thus, after three writes, the file should contain a pattern like:

```
aaaacaaaaaaabbaaaaaa
```

where each lower-case character represents 50 of the equivalent uppercase characters. The table that follows shows the contents of the JFFS and the in-RAM list after the sample code above has completed.

JFFS and in RAM List After Completed Sample Code

Flash Memory Contents	
Offset	Description
0x0	raw_inode created during open() Relevant fields are set as follows: offset = 0 nsize = 4 (name size) dsize = 0 (data size) rsize = 0 (removed size) version = 1 test - filename
0x3c	
0x40	raw_inode created during first write() Relevant fields are set as follows: offset = 0 nsize = 0 dsize = 1000 rsize = 0 version = 2 AAA....AAA
0x7c	
0x464	raw_inode created during second write() Relevant fields are set as follows: offset = 600 nsize = 0 dsize = 100 rsize = 100 version = 3 BBB....BBB
0x4a0	
0x504	raw_inode created during third write() Relevant fields are set as follows: offset = 200 nsize = 0 dsize = 50 rsize = 50 version = 4 CCC....CCC
0x540	

In RAM List	
Node	Contents
1	dsize = 0 flash_offset = 0x40
2	dsize = 200 flash_offset = 0x7c
3	dsize = 50 flash_offset = 0x540
4	dsize = 350 flash_offset = 0x7c + 250 = 0x176
5	dsize = 100 flash_offset = 0x4a0
6	dsize = 300 flash_offset = 0x7c + 700 = 0x338

JFFS is mounted using the standard `mount` command. The `-t jffs` flag is passed directly to the kernel to specify that JFFS is being mounted.

An `mtdblock` special device node is used as a parameter for `mount` to specify a Flash memory device on which to mount the file system. BlueCat Linux allows for mounting a file system on a partition, as well as on an entire device.

When JFFS is being mounted, Flash memory is scanned and an in-memory representation of the file system is built. Flash memory is searched for the most recent version of each chunk of the file. The `offset` field of each `raw_inode` indicates where the chunk starts in the file. The `size` field indicates how big the chunk is. If there is only a node in the system for file "`A`" with start position `0` and length `32768`, it is the most recent version. If there are two nodes, the one with the highest version count is correct. The next node in the file is the one with the starting position `32768`, even if its version number is less than the one for the logically prior node.

Power Loss Recovery

When scanning Flash memory for `raw_inodes`, the JFFS scan algorithm described in "JFFS Layout" searches for the `raw_inode` number to identify each chunk. When a chunk is found, the `checksums` for `raw_inode`, and, if present, the filename and chunk data are calculated and compared with the values stored in the `raw_inode` structure in Flash memory. If a power loss has occurred during a write operation, then the `checksums` are incorrect and the node is rejected. This means that if a less recent node for this part of the file is present in Flash memory, it is used instead of the reject. The stock result is as if no operation that caused the rejected node to be created has occurred.

Each POSIX I/O call that leads to a Flash memory update is enumerated and its number is stored in all the `raw_inodes` to be written during call processing. The number of nodes written by a call is stored along with the call number. So, if the power fails during the multinode `write()` call, the successfully written nodes of the partially complete write are found and rejected by the scan procedure during the file system mount. The scan procedure also finds the largest call number stored in Flash memory and initializes the call counter appropriately.

JFFS, therefore, guarantees that if a power loss occurs at a Flash memory update, then the JFFS is restored to its previous state upon reboot, as if the failing POSIX I/O call never occurred.

Call number storage uses 64 bits. Supposing there are 32 MB of Flash memory and each write stores one `raw_inode` (about 70 bytes) without the filename and file data: It takes less than 2^{19} calls to fill the Flash memory completely; 64 bits can

hold the value of 2^{64}. Hence, many holes are burned in Flash memory before the call counter overflows.

Wear Leveling

Wear Leveling and Garbage Collection Algorithm

Wear leveling is maintained by the following algorithm implemented in the JFFS. Data stored in Flash memory is maintained as a circular array. There are two pointers maintained in RAM. The first (the head) points to the beginning of the used/dirty space, and the second (the tail) points to the position where the used/dirty space ends and free space begins.

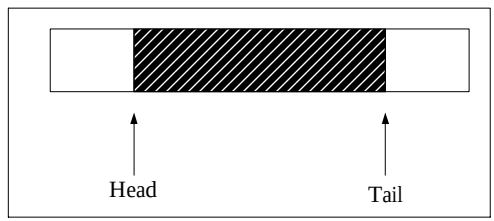

Figure 5-2: Data Pointers

Writing to Flash memory is always performed at the tail. This means that even if a file is being deleted from the JFFS, no actual deletions are performed immediately in Flash memory. Instead, meta data indicating that a certain file has been deleted is written to Flash memory at the tail position. The tail pointer is moved accordingly and the Flash memory space used by the deleted file is marked as dirty in the in-RAM data structures. Obviously, as more files are written, the free Flash memory space is exhausted. When this happens (in fact, some time before this happens), the Flash memory sectors at the head position are erased to free some space.

If the sector to be erased contains only dirt, that is older versions of files that have been modified or deleted, then the sector merely gets erased and its space marked as free. But if the sector contains some actual data, that is, files that have not been modified and are “current,” then it cannot be erased right away, because the “current” data must be saved first. In this case, the data is copied to the tail position in Flash memory, thus making obsolete the instance in the head sector.

The process of erasing the sectors at the head position on Flash memory while saving the actual data on an as-needed basis at the tail is called the *garbage collection*. Erasing a single sector is called a single iteration of garbage collection.

NOTE: Unless otherwise stated, garbage collection should be understood as a single iteration of the garbage collection.

Apart from being initiated from the user space via an IOCTL, there are several rules governing garbage collection. If not in an emergency, a separate kernel thread handles garbage collection. The thread is activated by a signal under certain conditions: if the dirty space in Flash memory is more than a third of the Flash memory size or the free space is below five percent. These garbage collection criteria are evaluated when sending a signal to the garbage collection thread.

Criteria evaluation occurs at two points during JFFS operation: The first is at the end of the `jffs_insert_node()` function and is called every time the in-RAM representation of a file is being changed. This occurs upon completion of any write-to-Flash memory operation, as well as during a file system mount. The second is when processing the `write_super()` call issued by VFS, which occurs about every five seconds.

Garbage collection is designed to be called and to function only when needed and during idle cycles. The separation of the garbage collector thread supports this approach. However, when writes and file deletions are intensive, there may not be enough free space in Flash memory to write the next node. In this case, the garbage collector is called explicitly and, provided the dirty space is larger than the smallest Flash memory sector, it recovers some space. This delays the writing of the node, because the multithreaded nature of the garbage collection is not utilized due to the sequential blocking call to the garbage collection system.

Upon reboot, when mounting the file system, the head and tail pointers are set to their previous locations. Therefore, the wear leveling algorithm described here works across reboots.

Synchronous Operations

In the JFFS, all writes to Flash memory are performed synchronously. Therefore one can be sure that upon the return of a `write()` call, all data (and meta data) has been physically written to Flash memory. This feature is ensured by the MTD layer. MTD drivers must implement the `write()` callback in a blocking manner, that is, the callback does not return until the write to Flash memory is actually

completed. The user does not need to force synchronicity by using the `O_SYNC` flag when opening a file.

There is no buffering for writing. The VFS file system may (and JFFS does) use caching for reading data. JFFS uses caching for the read operation because it supplies the `Linux generic_file_read()` function in `struct file_operations` when registering the file system with VFS. The `generic_file_read()` function uses the standard Linux memory page caching mechanism and calls the `inode->i_op->readpage()` function (implemented by the particular file system) for the actual low-level read operation (before each `write()` call returns, the `invalidate_inode_pages()` function is called). When the user makes a `write()` call, the control reaches the `jffs_file_write()` function immediately. Since the blocking MTD driver `write()` callback is used in this function, it is guaranteed that all writes to Flash memory are physically completed upon return from the call.

Automatic Bad Block Mapping

The following approach is used in JFFS to provide an automatic bad block mapping capability.

When a Flash memory sector is being updated, the Flash memory chip's status register is checked for possible errors. If an error is detected, the sector to which a write has been attempted is marked as bad, data written to another sector, and no further attempts to write to the bad sector performed until the next reboot. Sectors are marked bad only in RAM. The bad block mapping information is not stored in Flash memory and, therefore, is valid only until the next reset/power-down.

The table of bad blocks is not stored persistently in Flash memory for the following reasons:

- The problem may have corrected itself with power cycling, thus enabling the sector to be used successfully.
- Storing the information requires an extra sector, which in turn can become bad, thus requiring a reinitialization of the bad block array.
- Bad blocks should not occur very often in use. The overhead of mapping bad block information across reboots is deemed not valuable.

Write operation validation is performed by the MTD driver during the `write()` and `erase()` calls. If any error occurs, an appropriate error code is returned to the JFFS layer, which handles the code and maintains the table of bad blocks.

The mtdchar Interface Reference

The `mtdchar` character device interface implements Flash memory-specific IOCTL commands defined below. These commands can be used if the `linux/mtd/mtd.h` file is included.

`MEMGETINFO`

This IOCTL command copies the MTD driver information to the user space. It is returned in the `mtd_info_user` structure pointed to by the `arg` parameter of the `ioctl()` call.

The `mtd_info_user` structure is defined as follows:

```
struct mtd_info_user
{
  u_char type;      /* Type of memory technology */
  u_long flags;     /* Device capabilities */
  u_long size;      /* Size of the device in bytes */
  u_long n_regions; /* Number of Flash regions */
  u_long oobblock   /* Size of block that has out-of-band data
                     */
  u_long oobsize;   /* Size of each out-of-band area */
  u_long ecctype;   /* Error correction type */
  u_long eccsize;   /* Size of blocks for automatic */
                    /* error correction
                     */
};
```

`MEMGETREGIONS`

This IOCTL command copies information about the Flash memory regions defined by the MTD driver to the user space. It is returned in an array of the `mtd_flash_region_user` structures pointed to by the `arg` parameter of the `ioctl()` call. The `mtd_flash_region_user` structure is defined as follows:

```
struct mtd_flash_region_user
{
  loff_t start_offset;  /* Region starting offset */
  loff_t size;          /* Region size */
  u_long erasesize;     /* Size of the sectors */
  int   _sectors;       /* Number of sectors */
};
```

`MEMERASE`

This IOCTL command erases a specified area in Flash memory. This area is specified with the `erase_info_user` structure pointed to by the `arg` parameter of the `ioctl()` call. The `erase_info_user` structure is defined as follows:

```
struct erase_info_user
{
  unsigned long start;   /* Offset to start erase from */
  unsigned long length; /* The length of the area to be\   erased
                           */
};
```

The values of the `start` and `length` fields of the above structure must be aligned to a sector boundary. Also, the area defined by the structure must be a part of a single Flash memory region.

`MEMWRITEOOB`

This IOCTL command writes out-of-band data specified with the `mtd_oob_buf` structure pointed to by the `arg` parameter of the `ioctl()` call. The `mtd_oob_buf` structure is defined as follows:

```
struct mtd_oob_buf
{
  loff_t start;        /* Starting offset of the oob_area */
  ssize_t length;      /* The length of data to be written */
  unsigned char *ptr;  /* Pointer to the data to be written */
};
```

`MEMREADOOB`

This IOCTL command reads out-of-band data to the `mtd_oob_buf` structure pointed to by the `arg` parameter of the `ioctl()` call. The `mtd_oob_buf` structure is defined as follows:

```
struct mtd_oob_buf
{
  loff_t start;        /* Starting offset of the oob_area */
  ssize_t length;      /* The length of data to be read */
  unsigned char *ptr;  /* Pointer to the memory where \
  data hould be stored  */
};
```

`MEMDEFPARTTABLE`

This IOCTL command modifies the partition configuration of the MTD device. The new partition configuration is specified in the `mtd_partition_conf` structure pointed to by the `arg` parameter of the `ioctl()` call. The `mtd_partition_conf` structure is defined as follows:

```
struct mtd_partition_conf
{
  char * conf;         /* Configuration string */
  int    size          /* Size of the configuration string */

};
```

JFFS IOCTL Command Reference

To use the commands described below, the `linux/jffs.h` file must be included. The commands can be issued on any file contained in the JFFS.

`JFFS_GET_BAD_TABLE`

This IOCTL command provides user-space programs with access to the table of bad blocks.

The semantics of the command is as follows: The user allocates memory to hold the bad block table. The user then supplies `struct jffs_bad_table` with the `num_sectors` field containing the number of bytes already allocated for the bad block table and with the `bad_block_table` field pointing to the allocated space. Information about each sector in the table takes one byte. If the byte is not "`0`," it means that the corresponding sector is marked as bad. If the number of sectors associated with the partition is less than `num_sectors`, then on return `num_sectors` is set to the actual number of sectors.

The `jffs_bad_table` structure is defined as follows:

```
struct jffs_bad_table
{
  char * bad_block_table;  /* The array representing device\
blocks */
  int    num_sectors;      /* The size of the array */
};
```

`JFFS_GARBAGE_COLLECT`

The IOCTL command initiates a garbage collection procedure. The `arg` parameter of the `ioctl()` call must have one of the following values:

```
JFFS_GC_TRIGGER
JFFS_GC_ONCE
JFFS_GC_COMPLETE
```

If `JFFS_GC_ONCE` is specified, then the garbage collection procedure is run until at least one Flash memory sector is erased.

If `JFFS_GC_TRIGGER` is specified, then the procedure of `JFFS_GC_ONCE` is executed, provided the dirty area is larger than a third of the Flash memory entity size or the amount of free space is less than five percent of the total partition size.

If `JFFS_GC_COMPLETE` is specified, then the garbage collection procedure is run when the amount of dirty space is larger than a smallest Flash memory sector of the partition.

JFFS2 IOCTL Command Reference

To use the command described below, the `linux/jffs2_ioctl.h` file must be included. The commands can be issued on any file contained in the JFFS2.

`JFFS2_GARBAGE_COLLECT`

The IOCTL command initiates a garbage collection procedure. The `arg` parameter of the `ioctl()` is not used and must be set to 0.

MTD Interface Reference

This section describes the general structure of the Memory Technology Device (MTD) subsystem and its interfaces. The MTD system is divided into two types of modules: *users* and *drivers*. *Drivers* are kernel modules that provide raw read/write/erase access to the physical memory devices. *Users* are kernel modules that use the MTD drivers and provide a higher-level interface to the user space. The term “module” does not automatically imply Linux-loadable modules; MTD modules can be linked statically to the kernel.

The idea here is simple: The MTD interface provides for an open-interface, extensible approach that makes it easy to add support for Flash memory devices (and other types of memory devices) without the need to update any of the user interfaces, such as JFFS, `mtdchar`, or `mtdblock`.

Writing an MTD driver is simple:

1. Allocate and populate `struct mtd_info` with information about the supported device and pointers to the driver’s access routines.
2. Register it at the MTD interface by calling:

   `int add_mtd_device(struct mtd_info *mtd)`

Access routines implemented by an MTD driver must conform to the rules specified by the MTD interface.

What follows is a description of `struct mtd_info` that provides the interface to access an MTD driver from MTD users:

```
struct mtd_info {
  char name[32];
```

The name of the device is rarely used, but presented to the user via the `/proc/mtd` interface. When the `proc` file system support is compiled into the

kernel and the file system mounted, one can inspect the MTD drivers registered within the system by looking through the `/proc/mtd` file.

```
u_char type;
```

The type of memory technology used in this device may be one of the following:

- `MTD_ABSENT`—No technology
- `MTD_RAM`—RAM
- `MTD_ROM`—ROM
- `MTD_NORFLASH`—NOR Flash memory
- `MTD_NANDFLASH`—NAND Flash memory
- `MTD_PEROM`—EPROM
- `MTD_OTHER`—Other
- `MTD_UNKNOWN`—Unknown

```
u_long flags;
```

Device capabilities expressed as a bit mask that can include any of the following flags:

- `MTD_CLEAR_BITS`—Bits can be cleared (Flash memory)
- `MTD_SET_BITS`—Bits can be set
- `MTD_ERASEABLE`—Has erase function
- `MTD_WRITEB_WRITEABLE`—Direct IO is possible
- `MTD_VOLATILE`—Set for RAM
- `MTD_XIP`—eXecute-In-Place possible
- `MTD_OOB`—Out-of-band data (NAND Flash memory)
- `MTD_ECC`—Device capable of automatic ECC

Total size in bytes:

```
loff_t size;
```

Number of regions with sectors of the same size:

```
u_char n_regions;
```

List of structures describing each Flash memory region in detail:

```
struct mtd_flash_region;
```

Pointer to the partition layout information:

```
struct mtd_partition *part;
```

A pointer to the MTD driver of the entire Flash memory device is set to a nonzero value only when registering additional MTD entries for partition access:

```
struct mtd_info *driver;
```

Callback to the driver that reconfigures the partitions:

```
int (*modify_part_table) (char * new_table);
```

Some memory technologies support out-of-band data, for example, NAND Flash memory has 16 extra bytes per 512-byte page for error correction or meta data. `oobsize` and `oobblock` hold the size of each out-of-band area, and the number of bytes of real memory with which each is associated, respectively. For example, NAND Flash memory has `oobblock == 512` and `oobsize == 16` for 16 bytes of OOB data per 512 bytes of Flash memory.

```
u_long oobblock;
u_long oobsize;
```

Some types of hardware not only allow access to Flash memory or similar devices, but also have ECC (error correction) capabilities built-in to the interface:

```
u_long ecctype;
```

The `ecctype` field is an enumeration and can be one of the following:

- `MTD_ECC_NONE` No automatic ECC available
- `MTD_ECC_RS_DiskOnChip` Automatic ECC on DiskOnChip

`eccsize` holds the size of the blocks on which the hardware can perform automatic ECC.

```
u_long eccsize;
```

When a driver is a kernel-loadable module, this field is a pointer to the `struct module` of the module. It is used to increase and decrease the module's usage count as appropriate. The user modules are responsible for increasing and decreasing the usage count of the driver as appropriate, by calling `__MOD_INC_USE_COUNT (mtd->module)` in the open routine, for example.

```
struct module *module;
```

The following routine adds `struct erase_info` to the erase queue for the device. The routine may sleep until the erase is complete or it may simply queue the request and return immediately. The exact behavior of the routine is defined by the particular MTD driver implementation. Currently, the MTD interface does not

provide means to instruct the driver for a specific type of operation (sleep or return). `struct erase_info` contains a pointer to a callback function, which is called by the MTD driver when the erase is complete.

```
int (*erase) (struct mtd_info *mtd,
              struct erase_info *instr);
```

For devices that are entirely memory-mapped and which can be mapped directly into user-space page tables, support for execute-in-place (XIP) mapping of data on Flash memory may be possible. The precise semantics of this are yet to be defined, so it is probably best not to implement or attempt to use these two functions at the moment. Currently, in BlueCat Linux there is no support for the XIP feature.

```
int (*point) (struct mtd_info *mtd,
              loff_t from,
              size_t len,
              u_char **mtdbuf);
void (*unpoint) (struct mtd_info *mtd,
                 u_char * addr);
```

The following exemplifies `read()` and `write()` functions for the memory device. These may sleep and should not be called from the IRQ context or with locks held. The `buf` argument is assumed to be in the kernel space. Currently, the MTD interface does not provide means to instruct the driver for a specific type of operation (sleep or return).

```
int (*read) (struct mtd_info *mtd,
             loff_t from,
             size_t len,
             u_char *buf);

int (*write) (struct mtd_info *mtd,
              loff_t to,
              size_t len,
              const u_char *buf);
```

For the devices that support automatic ECC generation or checking, the routines below behave exactly as the `read/write()` functions above. Additionally, the `write_ecc()` function places the generated ECC data into `eccbuf`, and the `read_ecc()` function verifies the ECC data and attempts to correct any errors it detects.

```
int (*read_ecc) (struct mtd_info *mtd,
                 loff_t from,
                 size_t len,
                 u_char *buf,
                 u_char *eccbuf);
```

```
int (*write_ecc) (struct mtd_info *mtd,
                  loff_t to,
                  size_t len,
                  const u_char *buf,
                  u_char *eccbuf);
```

These functions provide access to out-of-band data for devices that have it:

```
int (*read_oob) (struct mtd_info *mtd,
                 loff_t from,
                 size_t len,
                 u_char *buf);

int (*write_oob) (struct mtd_info *mtd,
                  loff_t to,
                  size_t len,
                  const u_char *buf);
```

The following routine sleeps until all pending Flash memory operations are complete. This call is not used currently by any existing MTD drivers. Instead, their `write()` and `erase()` calls are implemented in the blocking manner.

```
  void (*sync) (struct mtd_info *mtd);
```

The following is used as a pointer to data that is private to the MTD driver:

```
  void *priv;
}; /* end of struct mtd_info */
```

The starting offset of the region relative to the beginning of the partition:

```
struct mtd_flash_region
{
  loff_t start_offset;
```

The size of the region:

```
  loff_t size;
```

The size of sectors within the region:

```
  u_long erasesize;
```

The number of sectors within the region:

```
  int n_sectors;
```

A pointer to the next region:

```
  struct mtd_flash_region * next;
}; /* end of struct mtd_flash_region */
```

The starting offset of the partition relative to the beginning of the physical device:

```
struct mtd_partition
{
  loff_t start_offset;
  loff_t start_offset;
```

Size of the partition:

```
  loff_t size;
```

A pointer to the next partition:

```
  struct mtd_partition * next;
} /* end of struct mtd_partition */
```

Flash Memory Management Tools and Mechanisms

This section explains in detail the control of Flash memory devices in embedded applications.

Configuring Flash Memory Partitions

The first step is to configure Flash memory partitions. This step is optional, as it can well be that an embedded application requires the use of an entire Flash memory device as a single entity, to hold a JFFS, for example.

Configuring Partitions at Build Time

An MTD driver can be configured during the build of a BlueCat Linux kernel. A configurable MTD driver must support a set of configuration parameters that define the layout of the Flash memory partitions maintained by the MTD. If no configuration parameters are specified at build time, the MTD assumes the entire Flash memory is a single entity.

The MTD drivers included in the user's distribution use the configuration parameters in the format shown below. It is recommended that the user preserve the format and semantics of the parameters if there is a need to develop an MTD for a new Flash memory device.

BlueCat Linux limits the number of Flash memory partitions that can be configured by each MTD to four. Build-time parameters have the format shown below:

```
CONFIG_MTD_<driver>_PART="0,4:1-3:5-34"
```

The numbers in quotation marks correspond to the sectors that are allocated to particular partitions. Colons separate configuration input for the partitions. In the example above, sectors `0` and `4` are allocated to the first partition, sectors `1`, `2`, `3` to the second partition, and sectors `5` to `34` to the third one.

Configuring Partitions Using Kernel Boot-Time Parameters

The second method is a boot-time configuration through kernel command-line parameters. The MTD drivers included in the user's distribution are written to recognize boot-time parameters in the format shown below:

```
<driver>_part_conf="0,4:1-3:5-34"
```

The numbers in quotation marks have the same interpretation as for the build-time configuration parameters.

Configuring Partitions Using flash_fdisk

The third method runs the runtime configuration utility `flash_fdisk(1)` to configure Flash memory partitions. `flash_fdisk` performs `ioctl()` calls to the `mtdchar` interface specifying the command `MEMDEFPARTTABLE` and supplying configuration information for a partition. `mtdchar`, in turn, calls the underlying MTD driver's callback defined in `modify_part_table` of `struct_mtd_info`. The MTD driver calls `del_mtd_device()` to unregister previously registered partitions (if any) and then calls `add_mtd_device()` to register the newly created partition. `flash_fdisk` has the following syntax:

```
flash_fdisk <mtdchar_node> <configuration_string>
```

The configuration string has the same format as in the first two configuration methods. Please refer to "flash_fdisk" in Appendix A, "Command Reference."

The following example command creates the same partitions as shown in the examples for the first two partitioning methods:

```
# flash_fdisk /dev/mtchar0 0,4:1-3:5-34
```

NOTE: The partition configuration created by any of the above configuration methods is not written to Flash memory and needs to be reestablished after reboot.

Using the /proc/mtd File

Read `/proc/mtd` to determine the current configuration of the Flash memory devices. The `/proc/mtd` file is composed of a number of lines. Each line corresponds to a single Flash memory entity (device or partition) registered at the MTD interface.

The format of one such line is as follows:

`mtd<minor_#>: <size> <id_string>`

For example, the following snapshot shows output for the configuration that has one Flash memory device with three partitions on it.

```
bash# cat /proc/mtd

mtd0: 00100000 "Flash on the CMA120 Willow board"
mtd1: 00008000 "Flash on the CMA120 Willow board"
mtd2: 00018000 "Flash on the CMA120 Willow board"
mtd3: 000E0000 "Flash on the CMA120 Willow board"
```

The first line in the output corresponds to the entire Flash memory device, while the next three lines correspond to Flash memory partitions.

The first column contains the minor number of the device node that must be used to access the corresponding Flash memory entity. For instance, in the example above, `/dev/mtdblock2` must be used as a parameter to the `mount` command to mount a JFFS on the second partition.

The second column is the hexadecimal size of the Flash memory entity in bytes.

Finally, the third column is an identification string supplied by the MTD driver at registration time.

Erasing a Flash Memory Device or Partition

Use the `flash_erase` utility to erase an entire Flash memory device or partition. For instance, given the configuration shown in the example in “Configuring Partitions Using flash_fdisk” on page 188 the following command erases the second partition:

```
# flash_erase /dev/mtdchar2
```

In general, when first using a new partition, it is advisable to erase it before mounting it as an FFS, or writing raw data to it. Refer to “flash_erase” in Appendix A, “Command Reference.”

Writing Raw Data to Flash Memory

Use the `mtdchar` interface to access raw data to Flash memory. Use a custom application code or any of the Linux target board tools to access raw data via `mtdchar`. For example, the following simple command sequence copies an image to Flash memory:

```
# flash_erase /dev/mtdchar2
# cat /images/flash.img > /dev/mtdchar
```

Managing JFFS/JFFS2

To create a JFFS in a Flash memory device or partition, use the `mount` command with the flag `-t jffs`. For instance, the following command creates a JFFS in the second Flash memory partition:

```
# mount /dev/mtdblock2 /mnt -t jffs
```

The same command mounts an already existing JFFS in a Flash memory entity specified by the block device node parameter.

To create a JFFS2 in a Flash memory device or partition, use the `mount` command with the flag `-t jffs2`. For instance, the following command creates a JFFS2 in the second Flash memory partition:

```
# mount /dev/mtdblock2 /mnt -t jffs2
```

The same command mounts an already existing JFFS2 in a Flash memory entity specified by the block device node parameter.

Once a JFFS/JFFS2 is mounted, it can be used in the same manner as a file system of any other type. For instance:

```
# cd /mnt
# mkdir tmp
# cd tmp
# cp /tmp/test_file .
# ls -lR /mnt
```

When done using a JFFS, unmount it:

```
# cd /
# umount /mnt
```

A clean unmount of a JFFS calls the garbage collector. This ensures that no time is spent on garbage collection the next time the user mounts the JFFS.

Downloading BlueCat Linux into Flash Memory with Flash Management Tools

Refer to Chapter 3, “Downloading and Booting BlueCat Linux” for a detailed explanation of how to use the BlueCat Linux Flash memory management tools and mechanisms to download a BlueCat Linux system onto an target board.

Developing an MTD Driver

This section shows the skeleton of a sample MTD driver.

Registering an MTD Driver

The following sample code shows the registration of a sample MTD driver.

```
/* Service function that parses configuration string
 * and calls add_mtd_device() as appropriate.
 */
extern int mtd_create_partitions(const char      *  layout,
                                 struct mtd_info ** part_mtds,
                                 int                max_num,
                                 struct mtd_info *  driver);

/* mtd_info structures of the configured partitions.
 */
static struct mtd_info * mtd_part_mtds[4];

/* The number of the configured partitions.
 */
static int               configured_parts;

/* Partitions configuration string. Initialized during
 * kernel build-time configuration. May be changed at boot time
 * during the kernel command line parsing in init/main.c
 */
char * mtd_sample_part_conf = CONFIG_MTD_SAMPLE_PART;

/* mtd_info structure corresponding to the whole device.
 */
static struct mtd_info   mymtd;

    /* Callbacks declarations.
     */
static int sample_erase(struct mtd_info *mtd,
                        struct erase_info *instr);
static int sample_read(struct mtd_info *mtd,
                       loff_t from,
                       size_t len,
                       size_t *retlen,
                       u_char *buf);
static int sample_write(struct mtd_info *mtd,
                        loff_t to,
                        size_t len,
```

```
                        size_t *retlen, const
                        u_char *buf);
static int sample_runtime_part_conf(char* str);

int init_mtdsample(void)
{
      int                       res;
      struct mtd_flash_region * region;

           /* Allocate memory for the Flash regions info.
            * This example assumes that the device has 4
regions
            * with sectors of the same size.
            */
      mtd_info->flash_regions = (struct mtd_flash_region *)
kmalloc(sizeof(struct mtd_flash_region) * 4, GFP_KERNEL);
      if (mtd_info->flash_regions == 0)
      {
          res = -ENOMEM;
          goto Done;
      }

      region = mtd_info->flash_regions;

                 /* One 16K sector.
                  */
           region->size               = 16 * 1024;
           region->erasesize          = 16 * 1024;
           region->n_sectors          = 1;
           region->start_offset       = 0;
           region->next               = region + 1;
           region->next->start_offset = region->start_offset +
region->size;
           region++;

                  /* Two 8K sectors.
                  */
           region->size               = 8 * 1024 * 2;
           region->erasesize          = 8 * 1024;
           region->n_sectors          = 2;
           region->next               = region + 1;
           region->next->start_offset = region->start_offset +
region->size;
           region++;

                  /* One 32K sector.
                   */
           region->size               = 32 * 1024;
           region->erasesize          = 32 * 1024;
           region->n_sectors          = 1;
           region->next               = region + 1;
           region->next->start_offset = region->start_offset +
region->size;
           region++;

                  /* Thirty one 64K sectors.
                  */
           region->size      = 64 * 1024 * 31;
           region->erasesize = 64 * 1024;
           region->n_sectors = 31;
           region->next = 0;
```

```
                    /* Setup the MTD structure.
                     */
        mymtd->name = "MTD sample device";
        mymtd_info->size = 2048*1024;
        mymtd->n_regions = 4;
        mymtd_info->erase = sample_erase;
        mymtd->read = sample_read;
        mymtd->write = sample_write;
        mymtd->modify_part_table = sample_runtime_part_conf;

        if (add_mtd_device(mymtd))
        {
                    /* Registration failed.
                     */
                    ...
        }

                    /* Optionally configure partitions.
                     */
configured_parts = mtd_create_partitions(mtd_sample_part_conf,
                                         mtd_part_mtds,
                                         4,
                                         mymtd);
        if (configured_parts < 0)
        {
              /* Partitions configuration failed.
               */
              ...
        }

        res = 0;
        Done:
        return res:
}
```

Deregistering the MTD Driver

The following sample code shows how an MTD driver can complete its operation.

```
void cleanup_mtdsample(void)
{
        int i;

              /* First unregister configured partitions.
               */
        for (i = 0; i < configured_parts; i++)
        {
              if (mtd_part_mtds[i])
              {
                    del_mtd_device(mtd_part_mtds[i]);
                    kfree(mtd_part_mtds[i]->part);
                    kfree(mtd_part_mtds[i]->flash_regions);
                    kfree(mtd_part_mtds[i]);
              }
        }

                    /* Unregister the MTD driver.
                     */
    del_mtd_device(mtd_info);
}
```

Configuring Partitions at Runtime

The following sample code shows how to write an MTD driver callback for runtime partition configuration.

```
static int sample_runtime_part_conf(char* str)
{
  int res = 0;

    /* First, unregister previously created partitions, if\
any.
     */
      while (configured_parts > 0)
      {
         if (mtd_part_mtds[configured_parts - 1])
         {
          if (del_mtd_device(mtd_part_mtds[configured_parts \ -1]))
          {
             res = -EBUSY;
             goto Done;
          }
           kfree(mtd_part_mtds[configured_parts - 1]->part);
           kfree(mtd_part_mtds[configured_parts - 1]
                 ->flash_regions);
           kfree(mtd_part_mtds[--configured_parts]);
         }
      }

        /* Register new partitions.
         */
           configured_parts = mtd_create_partitions(str,
                                                    mtd_part\
mtds,
                                                    4,
                                                    mymtd);
      if (configured_parts < 0)
      {
     /* Partitions configuration failed.
      */
         ...
         res = -EINVAL;
         goto Done;
      }

       Done:
       return res;
}
```

APPENDIX A *Command Reference*

This appendix includes man pages describing the BlueCat Linux utilities.

flash_erase

Erases a Flash memory device or a Flash memory partition

flash_erase *<device_node>*

Description

The `flash_erase` utility erases a Flash memory device or partition corresponding to a Flash memory character device node specified as a command line parameter.

Example

Assuming there are three partitions on a Flash memory device, the following command erases the first partition of the device:

```
# flash_erase /dev/mtdchar1
```

See also the `flash_fdisk(1)` man page.

flash_fdisk

Partitions a Flash memory device at runtime

Description

flash_fdisk *<device_node> <configuration_string>*

The `flash_fdisk` utility modifies the partition configuration of a specified Flash memory device at runtime. The device node specified as a parameter must correspond to a Flash character device node for an entire Flash memory device.

NOTE: Please bear in mind that the partition information is not written to Flash memory, and needs to be reestablished at every boot.

Configuration String Format

The configuration string has the following format:

```
single_part_conf[:single_part_conf]…
```

where:

```
single_part_conf={<number>[,|-<number>]}
```

<number> is a decimal number.

Numbers in the configuration string correspond to the sectors allocated to the particular partition. Configuration for different partitions is separated by a colon (`:`).

Example

In the following example sectors `0` to `3` are allocated to the first partition, sector `4` is allocated to the second partition, and sectors `5` and `6` are allocated to the third partition:

```
# flash_fdisk /dev/mtdchar0 0-3:4:5,6
```

mkboot

Copies bootable images from the cross-development host onto a floppy disk, a hard disk or a Flash drive. Alternatively, creates a bootable CD-ROM image of BlueCat Linux that can be burned to a CD-R(W) disk.

mkboot `[options] device|file|stdout`

Description

The `mkboot` utility is capable of performing the following tasks:

- Installing BlueCat Linux boot sector
- Installing compressed BlueCat Linux kernel
- Installing compressed root file system image
- Defining the root device to be mounted by the kernel
- Setting the command line to be passed to the kernel
- Creating an image composed of a BlueCat Linux kernel and a compressed file system, suitable to be programmed into ROM/Flash or downloaded over a network by the target firmware
- Creating a bootable CD-ROM image
- Creating a bootable target image file that can be copied to a bootable device outside of the BlueCat Linux environment
- Creating a partitioned disk image with an optional BlueCat Linux kernel

When called with no options, `mkboot` shows components currently installed on the media.

Options

- `-b`

  Installs the BlueCat Linux boot sector. Note that installing the BlueCat Linux boot sector onto a hard disk removes any boot loader present on the disk. The user will not be able to boot operating systems other than BlueCat Linux from the disk he has updated using the `mkboot` utility. Also, reinstalling the boot sector invalidates all the booting options set by previous calls to `mkboot` for this disk.

  The `-b` option can be used with an optional argument to set the number of created partitions. This argument must be entered without any space after `-b` (for example, `-b2`). Currently, only two partitions are supported. The first partition contains the BlueCat Linux kernel image, and the second one contains the root file system (not compressed). If the `-r` option is not present, and no `root=` parameter is specified in the kernel command line, the BlueCat Linux kernel will automatically detect and mount the second partition as a root file system.

- `-d`

  Used in conjunction with the `-b` option. Sets the target board drive BIOS ID manually. For instance, `0` corresponds to the first floppy, and `128` corresponds to the first hard disk in the boot sequence. By default, `mkboot` attempts to determine the ID automatically.

- `-s`

  Used in conjunction with the `-b` option. Sets the number of target board drive sectors per track. By default, `mkboot` attempts to determine the drive geometry automatically.

- `-h`

  Used in conjunction with the `-b` option. Sets the target board drive heads number. By default, `mkboot` attempts to determine the drive geometry automatically.

- `-c` *<none>|<file>|<stdin>*

  Sets the command line for the kernel installed on the media; *<none>* resets command line; *<stdin>* takes the command line from the standard input.

- `-k` *<file>|<stdin>*

  Installs the compressed kernel to the media; *<stdin>* takes the image of the kernel from the standard input.

- `-f` *<none>|<file>|<stdin>*

  Installs the compressed root file system image to the media; *<none>* removes compressed root file system; *<stdin>* takes the image of the root file system from the standard input.

- `-g` *<file>*

  Uses ISO9660 file system from *<file>*.

- `-r` *<xxxx>|<device>*

  Sets the device node on the target board to mount as the root file system or from which to uncompress the file system image; for instance, `200` corresponds to `/dev/fd0`, and `801` corresponds to `/dev/sda1`. Instead of the major/minor number, the root file system can be specified as the standard name of the device node. For example: `/dev/hd**`, `/dev/sd**`, `/dev/fd**`, `/dev/tffs*`.

- `-o`

  Tells `mkboot` to create a bootable BlueCat Linux ISO9660 image.

- `-m`

  Tells `mkboot` to create an image composed of a BlueCat Linux kernel (specified by the `-k` flag) and a compressed file system (specified by the `-f` flag) suitable for programming into target ROM/Flash memory or downloading over a network by the target board firmware

- `-i`

  Does not automatically install target board-specific parameters into the kernel command line

- `-q`

  Quiet mode; only error messages are printed on a console.

- `-C <file>`

  Tells `mkboot` to create a partitioned disk image. No other options should be specified. A `<file>` argument is a description filename. The description file is a text file consisting of a sequence of sections. The section consists of an optional header and a sequence of properties, one per line.

  Common Section

  If present, the common section should be specified at the beginning of the description file. The common section describes the overall disk layout. It has no header and may contain the following properties:

  `size = <size><unit>`

  where `<size>` is an integer value, and `<unit>` is one of `s`, `K`, `M`, and `G`. This property specifies the total disk size in 512-byte sectors, kilobytes, megabytes, and gigabytes respectively. If this property is omitted, `mkboot` will calculate the disk size summing up sizes of each partition.

  `extended = <number>`

  where `<number>` is an integer value in range 1-4 specifying the number of the primary partition slot, which contains the extended partition.

Partition Section

The partition section describes a partition layout. It starts from the following header:

```
[Partition]
```

followed by a sequence of partition properties. The partition section may contain the following properties:

`name = <partition_name>`

Specifies the partition name. *<partition_name>* is an alphanumerical word (50 characters maximum, without spaces). This name is used internally by `mkboot` for the `kp1`, `kp2`, `up1`, `up2`, and `op` properties in the common section.

`layout = <partition_layout>`

Specifies the partition layout (location). *<partition_layout>* may be an integer value in range 1-4 specifying the number of the primary partition slot or the word `logical` specifying that the partition is a logical partition on the extended partition.

`source = <file_name>`

Specifies the file containing the image to be used to populate this partition.

`type = <type>`

Specifies the type of the image specified in the source property. Possible values of *<type>* are as follows:

- `fs`

  Means that the source is the compressed FS image created by the `mkrootfs` utility which should be uncompressed to the partition

- `resized-fs`

  The same as `fs`, but with expansion of the file system to the size of the partition. This value can be specified for ext2 and ext3 filesystems only.

- `kernel`

  Means that the source is the BlueCat Linux kernel.

- `raw`

  Means that the source is an image, which should be copied to the partition as is.

- `empty`

  Means that the partition will be filled with zeroes.

`size = <size>`

Specifies the partition size exactly as the size property in the common section does. Additionally, `<size>` may be specified as the word `maximum`. In this case the partition size will be set to fill all the remaining disk space. If omitted, the partition size will be calculated based on the source property. If the type of the partition is kernel, the following additional properties may be specified:

`cmdfile = <file_name>`

Specifies the file containing the kernel command line as in the `mkboot -c` option.

`cmdline = <command_line>`

Specifies the kernel command line. The command line starts from the first nonspace character after `=`, and continues up to the end of the line (255 characters maximum).

Note, that command line can contain the `root=bc-boot-auto-<linux_partition_number>` string, where `<linux_partition_number>` is the RFS partition number in the terms of Linux partition numeration:

- 1-4 are the primary partitions, the numbers greater than 4 are logical partitions.

  If specified, this string tells the kernel to mount as RFS the `<linux_partition_number>` partition of the device it booted from. The boot device is detected automatically, the `rootdev` option should not be specified in this case.

`rfs = <file_name>`

Specifies the file containing the RFS image to be placed in the same partition directly after the kernel.

`rootdev = <device>`

Specifies the device containing the root file system for the kernel. Has the same meaning as the `mkboot -r` option.

- `-D property=value,property=value,...`

  Tells `mkboot` partitioned disk properties. An argument is a comma-separated list of the properties, followed by the `=` sign followed by the value of the property. The list must not contain spaces. See the description of the `-C` option in `Common Section` for the allowed disk properties.

  No other options, except `-p` should be specified.

  `-p property=value,property=value,...`

  Tells `mkboot` properties of a partition. This option may be used several times, corresponding to the number of partition. Each `-p` option defines the properties of exactly one partition. An argument is a comma-separated list of the properties, followed by the `=` sign, followed by the value of the property. The list must not contain spaces. See `-C` option, Partition section description for the allowed partition properties. No other options, except `-D` should be specified.

Examples

To copy the `osloader` demo system onto a floppy:

- On the Linux host:

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
BlueCat:$ mkboot -b -k osloader.disk -f \
osloader.rfs -r /dev/fd0 /dev/fd0
```

- On a Windows host:

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
BlueCat:$ mkboot -b -k osloader.disk -f \
osloader.rfs -r /dev/fd0 a:
```

To copy the `developer` demo system onto an IDE hard disk for x86 target board:

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/developer
BlueCat:$ mkboot -b -k developer.disk -f developer.rfs \
-r /dev/hda /dev/hda
```

To copy the `osloader` demo system on a floppy and set the kernel command line, use one of the following sequences:

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
BlueCat:$ echo console=441 >cl.txt
BlueCat:$ echo mem=4M >>cl.txt
```

```
BlueCat:$ mkboot -b /dev/fd0
BlueCat:$ mkboot -k osloader.disk -f osloader.rfs -r \
/dev/fd0 /dev/fd0
BlueCat:$ mkboot -c cl.txt /dev/fd0
```

or:

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
BlueCat:$ mkboot -b /dev/fd0
BlueCat:$ mkboot -k osloader.disk -f osloader.rfs -r \
/dev/fd0 /dev/fd0
BlueCat:$ echo "mem=4M console=441" | mkboot -c stdin \
/dev/fd0
```

To use the floppy created by the example above use a file system contained in `/dev/sda1` as the root file system at boot time:

- On a Linux host:

  ```
  BlueCat:$ mkboot -f none -r /dev/sda1 /dev/fd0
  ```

- On a Windows host:

  ```
  BlueCat:$ mkboot -f none -r /dev/sda1 a:
  ```

To create a bootable image suitable for writing later onto a CF card or a hard disk and containing the `showcase` demo system:

```
BlueCat:$ mkboot -m -h 4 -s 32 -d 128 -k showcase.disk \
-f  showcase.rfs -r /dev/hda osloader.img
```

This example assumes a CF card or a hard disk has 4 heads, has 32 sectors, and is attached as a first disk on the target board.

To create a firmware-downloadable or Flash memory-programmable image containing the `osloader` demo system:

```
BlueCat:$ mkboot -m -k osloader.disk -f osloader.rfs \
osloader.kdi
```

To create a bootable ISO9660 image containing the `showcase` demo system:

```
BlueCat:$ mkboot -o -k showcase.disk -f showcase.rfs \
showcase.iso
```

To create a bootable ISO9660 image with kernel from `showcase` demo system and the ISO9660 root file system:

```
BlueCat:$ mkboot -o -k showcase.disk -r /dev/hdc -g \
showcase.isofs showcase.iso
```

To create a bootable multi-partition image:

```
BlueCat:$ mkboot -C disk.txt /dev/sda
```

where the `disk.txt` file has the following content:

```
size = 128M
extended = 4

[ Partition ]
layout = 1
type = kernel
size = 4M
source = developer.disk
cmdline = root=bc-boot-auto-5

[ Partition ]
layout = logical
type = fs
source = developer.rfs

[ Partition ]
layout = logical
type = fs
source = usr.rfs

[ Partition ]
layout = logical
size = 10M
type = empty

[ Partition ]
layout = logical
size = maximum
type = resized-fs
source = files.rfs
```

The image is created on the `/dev/sda` device.

To create a bootable multi-partition image using the `-D` and `-p` options:

```
BlueCat:$ echo root=bc-boot-auto-5 > cmdline.txt
```

```
BlueCat:$ mkboot -D size=128M,extended=4 \
-p layout=1,type=kernel,size=4M,source=developer.disk,\
cmdfile=cmdline.txt \
-p layout=logical,type=fs,source=developer.rfs \
-p layout=logical,type=fs,source=usr.rfs \
-p layout=logical,type=empty,size=10M \
-p layout=logical,type=resized-
fs,size=maximum,source=files.rfs \
disk.img
```

The `disk.img` image is created.

mkkernel

Builds the BlueCat Linux kernel

```
mkkernel <config_file> <bvmlinux_file> \
[<bzimage_file>]
```

Description

`mkkernel` is a shell script that builds the BlueCat Linux kernel with the kernel options defined by the specified configuration file. The name of the configuration file must be given as the first argument. The second argument specifies the name of the output file for the kernel image. The third argument is optional. If present, it is taken as the name of the output file for the kernel image in the `bzImage` format (only meaningful for x86 architecture), otherwise no `bzImage` kernel is created.

`mkkernel` creates the output files in the current directory. The actual kernel build is performed in the `$BLUECAT_PREFIX/usr/src/linux` tree.

No options can be applied to `mkkernel`.

mkrootfs

Builds a target board file system image according to a specification file

Description

```
mkrootfs [-DilLxTOv][-J[<chunk_size>]] \
[-j[<chunk_size>] [-e <sector_size>]] [-N inodes] \
[-r blocks] <spec_file> <output_file>
mkrootfs -t [-ilLv] <spec_file>
```

Depending on the option passed to `mkrootfs`, it creates:

- A gzipped target file system image or a tar file (if the `-T` option is specified)
- A gzipped ext3 file system image (if the `-x` option is specified)
- A Journalling Flash File System image (if the `-J` option is specified)
- A Journalling Flash File System version 2 image (if the `-j` option is specified)

- A CD-ROM ISO9660 file system image (if the `-O` option is specified)

The utility can be used to create a minimal root file system for the target system.

A *<spec_file>* describes the target board file system contents and configuration. Its syntax is similar to a shell script and is explained in "Specfile Format" on page 206.

Currently, `mkrootfs` supports four target board file system types, the Linux `ext2` and `ext3` file systems (for compressed images), the ISO9660 file system (for use on CD-ROM), and the Linux JFFS/JFFS2 file systems (for use directly from Flash).

Specfile Format

<spec_file> consists of lines of one of three types: *comment*, *setting*, or *command*.

On any line, `mkrootfs` ignores all comments starting with a "`#`" character. Blank lines, or lines that contain only comments starting with a "`#`" character, are ignored.

Any line of the form *left = right* is considered to be a setting. The result of such a setting is that the environment variable named `left` is assigned the value `right`. Referencing syntax is similar to the `sh` shell, but curly braces (`{}`) should be used to delimit variable names (for example, `${BLUECAT_PREFIX}/usr/local/bin`), because `mkrootfs` treats most `sh` delimiters as ordinary symbols.

Command lines start with a `mkrootfs` command (`include`, `cp`, `rm`, and so on) and are subject to argument parsing and environment-variable substitution (just as any other line). Parsing, quoting, and substitution rules resemble those used in `sh`, so they are not explained here.

The following commands are implemented:

include *<path_to_another_specfile>*

Parses another specfile.

strip [**on** | **off**]

Turns file stripping on/off.

When on, all subsequent copy commands (up to the next strip off) are performed using `objcopy` with the appropriate stripping option, depending on the type of the source file. If the source file is an executable, all symbols

are removed (`-S` option). If it is a library, only the debugging symbols are stripped (`-g` option).

binary [**on** | **off**]

Turns binary mode on/off.

Binary mode allows for files to be copied without interpretation. This is useful when copying other platforms' binaries, in which case an attempt to find library dependencies may result in failure.

cd *<absolute_dst_path>*

Sets the current working directory for the target file system paths (affects all subsequent relative target file system paths, up to the next `cd`).

lcd *<absolute_src_path>*

Sets the current working directory for `mkrootfs` to find files to place on the target file system (affects all subsequent relative source paths, up to the next `lcd`). To reset the source path prefix to the initial working directory (where `mkrootfs` was started), use the `lcd .` command.

cp *<src> <src...> <src_dst/path>*

Copies files to the target board file system.

Wildcard characters can be used in *<src>*. Globbing is performed just like in `sh`. When a directory matches *<src>*, it is copied recursively. There is no special `-R` option. Permissions and owner IDs are preserved. Symbolic links are resolved and actual files are copied. Use the `ln` command to create links on the target board file system.

rm *<dst ... dst>*

Removes files/directories from target board file system.

Allows any wildcards. Globbing is performed just like in `sh`. Removes directories recursively.

ln [-s] *<src_path> <dst_path>*

Creates target board file system links.

No wildcards are allowed. The `-s` option indicates that a symbolic link should be made.

mkdir [**-p**] [**-m** *<mode>*] [**-u** *<user_id>*] *<dst_path...dst_path>*

Creates directories in the target board file system.

With `-p`, `mkdir` creates intermediate directories if required. The `-m` option allows the user to specify permissions for the directory to be created (must be octal). The `-u` option sets the owner of the directory.

mknod [**-m** *<mode>*] *<full_dst_path> <type> <major> <minor>*

Creates target board file system special (devices) files.

File type can be `b`, `c`, or `p` to indicate a block device, a character device or a FIFO (respectively). *<major>* and *<minor>* specify the major/minor device numbers, which can be specified for the `b` and `c` types only. `-m` sets permission modes for the special file (in the same format as directory permissions are set).

chmod *<mode> <dst ... dst>*

Changes permission modes. *<mode>* must be octal.

If *<dst>* is a directory, it recursively sets the mode of all files and subdirectories as well as of the directory itself.

chown *<uid> <dst ...>*

Changes the file/directory owner.

It works recursively in the same manner as `chmod`.

`mkrootfs` supports a simple `if-else-endif` construct and has the following syntax:

```
if <arg1> = <arg2>
...
[else]
...
endif
```

This can be used to arrange for conditional parsing of the specification file. For example:

```
if $BLUECAT_TARGET_CPU = ppc
... # PowerPC <part>
else
... # non-PowerPC <part>
endif
```

Note that variable substitution and argument parsing is performed as with any other commands.

NOTE: On the Windows hosts if `mkrootfs` is terminated (either by an error or by a signal), it tries to clean all its temporary files before exiting, but due to certain peculiarities of CygWin, such temporary files can remain uncleaned in the `/tmp` directory. It is recommended that the `/tmp` directory be regularly checked and cleaned up.

Options

- `-D`

  Does not create a `lost+found` directory on the target board file system.

- `-i`

  Ignores nonfatal errors. By default, `mkrootfs` fails on errors that can result in target board file system inconsistency, such as `shared library not found`. This option allows the user to override this behavior.

- `-l`

  Adds all required shared libraries. With this option, each executable is scanned and all shared libraries that it depends on are copied to the target board file system. Furthermore, after building the target board file system, `mkrootfs` runs the `ldconfig` utility to create the cache and all appropriate symlinks. All shared libraries are debug-stripped on copy.

- `-L`

  Does not add required shared libraries, but runs `ldconfig` anyway. This can be useful when custom libraries are being used and there is no need to copy standard libraries automatically.

- `-N <inodes>`

  Creates the specified number of `<inodes>` for the target board file system instead of the default value, which is calculated based on the file system size and can be incorrect for the user's embedded system.

- `-r <freespace>`

  Reserves specified free space on the target board file system. The argument must be a numeric value indicating the number of free 1 KB blocks that should be allocated.

- `-t`

  Test mode. No image is created; all required actions are printed instead.

- `-T`

  Creates a tar file on output. When this option is specified, no image file is created, but the *<output_file>* argument is used as a filename for the resulting tar archive.

- `-x`

  Create an ext3 journalling file system image on output.

- `-O`

  Creates a CD-ROM image with ISO9660 file system on it.

- `-J [`*chunk_size*`]`

  Creates a Journalling Flash File System (JFFS) image. This image can be downloaded into Flash memory and then mounted as a root file system when BlueCat Linux boots on the target board. The `chunk_size` argument is optional and can be used to specify the maximum size of a data chunk for the resulting image. By default this value is 4096 bytes, which is suitable in most cases.

- `-j [`*chunk_size*`]`

  Creates a JFFS2 image on output. The optional `chunk_size` argument can be used to specify the desired size of the data granularity of the files in the resulting image. If `chunk_size` is omitted, the default size of 4096 bytes is used instead.

- `-e [`*sector_size*`]`

  Specifies size of erase block to use. Valid for the JFFS2 file system only. The argument must be a numeric value indicating the number of bytes (for kilobytes, use the K suffix) of the Flash memory sector size. If `sector_size` is omitted, the default size of 131072 (128K) bytes is used instead.

- `-v`

  Verbose mode. Produces verbose output

pftpd

BlueCat Linux PFTP daemon

Description

```
pftpd {start|stop}
pftpd --help
```

`pftpd` is a daemon program that allows booting a BlueCat Linux embedded system from the BlueCat Linux cross-development host running the daemon via the PFTP protocol. PFTP is a Parallel Port File Transfer Protocol used to transfer files over an ECP parallel cable connection.

Parameters

- `start`—Starts the `pftpd` daemon.
- `stop`—Stops all running `pftpd` daemon processes.
- `--help`—Displays a short description of the command line options.

Configuration File

The configuration file contains specifications of files that can be loaded onto the target board.

The configuration file is named `$BLUECAT_PREFIX/cdt/etc/pftpd.rc`. The user can modify this file to customize local settings.

Configuration File Format

The configuration file consists of lines in the following formats:

- `pftproot = value`—Defines the root directory from which the file searching starts.
- `allow = value`—Defines subdirectories of the root directory to which client access is enabled; specifies an empty value to allow access to all files in the `pftproot` directory.

- `# comment`—The comment text is ignored.

NOTE: In order to enable the daemon, the system administrator must perform installation of a low-level driver.

APPENDIX B *BLOSH Commands*

BLOSH Commands

BLOSH implements a number of built-in commands. Each command prints a Usage error message if used incorrectly. A command name may be reduced to as many characters as make it singular. For example, `boot` can be abbreviated as `b`, `bo`, or `boo`.

boot

Boots a BlueCat Linux Kernel

`boot`

The `boot` command boots a BlueCat Linux system. The location of the kernel image and optional root file system image, as well as the kernel boot parameters, are specified by their respective environment variables.

If the root file system is specified, the booted kernel loads the file system image into RAM and mounts it as a root file system. If the root file system variable is not set (that is, the `RFS` environment variable is set to an empty string), the booted kernel image must mount something else as the root file system. This can be a file system on a local disk, or an NFS-based file system.

If booting from the network, the networking-related environment variables must be set to appropriate values. Also, the network server machine (either a TFTP or NFS server) must be configured to allow downloading of images onto the target board.

If booting from a parallel port, the `PPORT` variable must be set. Also, the PFTP server must be set up to allow downloading of images onto the target board.

The following command sequence demonstrates booting a BlueCat Linux system from a TFTP server. Both kernel and root file system images are specified:

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set KERNEL tftp /tftpboot/<demo>.kernel
> set RFS tftp /tftpboot/<demo>.rfs
> boot
```

cd

Changes current working directory

cd [*<directory>*]

The `cd` command sets the current working directory for BLOSH. If no directory is specified, the value of the `HOME` environment variable is used.

exec

Executes a program

exec [**-r**] *<program>* [*<params>*]

The `exec` command executes the specified program found in the BlueCat Linux root file system as a new process. If the `-r` flag is specified, the new program completely replaces BLOSH in RAM. The <*params*> string, if provided, is passed to the process as the parameters.

For instance, the following command shows the contents of the BlueCat Linux OS loader `root` directory.

```
> exec /bin/ls -lt /
```

(This example assumes that the `ls` utility is contained in the `/bin` directory, which is not the default case. However, arbitrary utilities and files can be added to the BlueCat Linux OS loader file system.

flash

Programs (burns) an image into Flash memory

flash /dev/mtdchar *<n>* [**erase**]

The `flash` command downloads the file specified by the `FILE` environment variable into the specified Flash memory device. If an optional `erase` argument is supplied, the full erase of the specified Flash memory device is performed before programming begins.

help

Prints a help message

help [*<name>*]

The `help` command shows help messages. If no argument is specified, the list of all supported commands is shown. `help` with a single argument shows the usage string for the specified command.

mkboot

Creates a bootable disk

mkboot [**-b**] [**-r** *<root>*] **/dev/***xxx*

The `mkboot` command functions similar to the `mkboot` utility included in the BlueCat Linux cross-development tools. The `mkboot` command differs in that the kernel image, root file system image, and the command line are specified by the BLOSH environment variables as follows:

- `KERNEL`—Specifies the kernel image to be downloaded.
- `RFS`—If set, specifies the compressed root file system image to be downloaded.
- `CMD`—Specifies the kernel command line.

The following command sequence shows the downloading of a BlueCat Linux kernel and root file system onto a hard disk for an x86 target board. The kernel boots from a hard disk, uncompresses the file system in the RAM, and mounts it as the root file system.

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
```

```
> set KERNEL tftp /tftpboot/showcase.disk
> set RFS tftp /tftpboot/showcase.rfs
> mkboot -b -r /dev/hda /dev/hda
```

mount

Mounts a file system

mount *<device> <directory>*

The `mount` command mounts a file system at the specified directory.

ntar

Downloads and unpacks a tar archive

ntar

The `ntar` command downloads and unpacks a tar archive into the current directory. The archive is specified by the `FILE` environment variable. If the archive is located on a network, networking-related environment variables must be set to appropriate values. Also, the network server machine (either TFTP or NFS) or the parallel port server must be configured to allow downloading of images onto the target board.

The following command sequence shows the creation of a BlueCat Linux root file system on a partition of the local disk. The archive is copied from a TFTP cross-development host.

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set FILE tftp /tftpboot/root.tar
> mount /dev/hda1 /mnt
> cd /mnt
> ntar
```

read

Downloads an arbitrary file

read *<file>*

The `read` command downloads the file specified by the `FILE` environment variable and places it in the BlueCat Linux OS loader root file system under the filename *<file>*.

This command is used to download a BLOSH script file. Alternatively, the `read` command can be used to copy an executable file to the BlueCat Linux OS loader root file system.

The following sequence copies a BLOSH script from a TFTP server and executes BLOSH commands contained in the script:

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set FILE tftp /tftpboot/script.1.0.3.2
> read /my_script
> script /my_script
```

reset

Reboots the system

reset

The `reset` command unconditionally shuts down the BlueCat Linux OS loader and performs a hardware reset.

script

Processes a list of commands in a file

script *<file>*

The `script` command sequentially executes BLOSH commands contained in the specified file. If any command fails, the script is halted.

The script file can contain another script, thus allowing recursive scripting. This feature is especially useful in a scenario where a script must be downloaded over the network.

Empty lines or lines starting with a `#` character are considered to be comments and are ignored by the script processor.

The following is an example of a script file that sets up the network environment variables:

```
# sample script file
# set up network variables
set IP 1.0.3.2
set HOST 1.0.3.1
set IF eth0
# end of script
```

set

Shows or modifies environment variables

set *<var> <value>*

The `set` command shows or modifies environment variables. If no variable is specified, the command shows all the environment variables and their respective values. If a variable name is specified without a value, the current value of the variable is shown. Finally, `set` with two arguments sets the variable to a new value.

Quoting is not mandatory in the `set` command. The remainder of the line, excluding the leading blank space, is considered to be the new value of the variable.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z