Contents

Preface ............................................................................................................... XI

Typographical Conventions ......................................................................... xi
Special Notes ................................................................................................. xii
Technical Support .......................................................................................... xii

LynuxWorks U.S. Headquarters ................................................................ xiii
LynuxWorks Europe .................................................................................... xiii
World Wide Web .......................................................................................... xiii

Chapter 1  Introduction and Installation ......................................................... 1

Overview ......................................................................................................... 2
System Requirements ..................................................................................... 2
Distribution CD-ROMs .................................................................................. 3

Binary Architecture CD-ROM Structure .................................................. 4
Binary Architecture CD-ROM Directories and Components .................. 4
Board Support Package (BSP) CD-ROM Structure .................................. 6
BSP CD-ROM Directories and Components ............................................. 6
Source Architecture CD-ROM Structure .................................................. 7
Source Architecture CD-ROM Directories and Components .................. 7

Installation Procedure .................................................................................... 8

Installing the Default Configuration .......................................................... 8

Packages in the Default BlueCat Linux Configuration ......................... 10
Installing Target Board Support ............................................................... 10
Activating Support for a Target Board ...................................................... 11
Installing Sources of BlueCat Linux RPM Packages ............................. 12
Using the BlueCat Linux rpm Utility ......................................................... 13

BlueCat Linux Directory Structure ........................................................... 14

Overview ..................................................................................................... 14
BlueCat Linux Components ....................................................................... 15
Contents

Setting the BlueCat Linux Execution Environment ............................................. 16
  For Linux Hosts ...................................................................................... 16
  Unsetting the BlueCat Linux Environment .......................................... 17
  Multiple Instances of BlueCat Linux ..................................................... 17
Uninstalling BlueCat Linux ................................................................................ 17
  Uninstalling an Entire Installation ......................................................... 17
  Uninstalling Support for a Target Board .............................................. 17
  Uninstalling a BlueCat Linux Component ............................................. 18
BlueCat Linux Utilities .................................................................................. 18
  mkkernel ................................................................................................. 18
  mkrootfs ................................................................................................. 19
  mkboot .................................................................................................... 19
  osloader .................................................................................................. 19

CHAPTER 2 DEVELOPING BLUECAT LINUX APPLICATIONS ........................... 21

Development Directory ................................................................................. 21
  Kernel Tree ............................................................................................. 22
  BlueCat Linux Kernel RPMs ................................................................ 23
  BlueCat Linux Kernel versus ‘Pristine’ Linux Kernel ........................... 23
Target Tools and Files ................................................................................. 24
  Cross Development Tools ...................................................................... 24
Demo Systems .......................................................................................... 24
  Embedded System Definition ............................................................... 24
  OS Loader ............................................................................................ 25
Development Process Overview ................................................................ 26
Customizing the BlueCat Linux Kernel ........................................................ 28
  Configuring the Kernel ........................................................................... 28
    Kernel Configuration Procedure Overview .................................... 29
    Kernel Configuration Command Interface ..................................... 29
Building the Kernel .................................................................................... 30
  Building the Kernel to Boot from Floppy or Hard Disk ......................... 31
  Building a Kernel for Download using OS Loader ............................. 31
  Building the Kernel to Boot from Target ROM/Flash Memory ....... 31
Managing Multiple Kernel Profiles ............................................................. 31
  mkkernel .............................................................................................. 32
Debugging the Kernel ............................................................................... 32
  Debugging Requirements ................................................................ 33
Setting Up Serial Ports ............................................................................ 34
Starting Kernel Debugging ....................................................................... 34
Interrupting the Kernel ......................................................................... 36
Finishing Kernel Debugging ........................................................... 36
BlueCat Linux Kernel Debugger Extensions ................................. 37
Developing Application Programs .................................................. 38
Building Application Programs ..................................................... 38
Debugging Application Programs .................................................. 38
Requirements ............................................................................... 38
Procedure ................................................................................. 39
Building a Root File System .......................................................... 41
BlueCat Linux Root File System Utility – mkrootfs ......................... 41
mkrootfs Specification File ............................................................ 41
Images Created by mkrootfs ......................................................... 42
Managing Multiple Embedded Applications ................................... 42
Optimizing Footprint .................................................................... 43
Customizing the Kernel for Size ................................................... 43
Using mkrootfs to Build a Minimal File System ......................... 43
Discarding Symbols from Files .................................................... 43
Getting Necessary Shared Libraries ............................................. 43
Using Static Libraries .................................................................. 44
Using the Memory Sizing Benchmark .......................................... 44

CHAPTER 3

DOWNLOADING AND BOOTING BLUECAT LINUX ...................... 47
BlueCat Linux Boot Procedure Overview ..................................... 47
mkboot Cross Development Tool ............................................... 48
BlueCat Linux OS Loader ............................................................ 48
BlueCat Linux Loader Shell (BLOSH) .......................................... 49
BLOSH Startup Sequence ......................................................... 49
BLOSH Environment Variables .................................................. 50
Enabling Networking ................................................................... 51
Setting Up a BOOTP Server ....................................................... 51
Setting Up a TFTP Server ........................................................... 52
Setting Up an NFS Server .......................................................... 53
Booting Images from a Different Subnet ..................................... 54
Setting Up a PFTP Server ........................................................... 54
BlueCat Linux Boot Scenarios ..................................................... 56
Booting BlueCat Linux from Floppy Disk ................................... 56
Copying onto Floppy Disk .......................................................... 56
Floppy Disk Boot Mechanism ..................................................... 57
Booting BlueCat Linux from Hard Disk ..................................... 58
Copying onto Hard Disk ............................................................. 58
Contents

Booting from Hard Disk using Network and OS Loader .......... 60
Booting from DiskOnChip ......................................................... 63
Hard Disk Booting Mechanism .............................................. 65
Booting BlueCat Linux from Target ROM/Flash Memory .......... 66
  Downloading to ROM/Flash using Firmware ......................... 66
  Booting Mechanism from ROM/Flash using Firmware ............... 67
  Booting from ROM/Flash using Network and OS Loader ........ 67
  Booting from Extension BIOS on x86 .................................. 72
Booting BlueCat Linux over Network or Parallel Port .......... 74
  Booting over Network using Target Firmware .................... 74
  Booting over Network or Parallel Port using OS Loader ........ 74
Hardware Support for Net Booting with OS Loader ............. 75
IP Autoconfiguration using OS Loader ............................... 76
BLOSH Commands and Variables for Net Booting .............. 77
  Booting from RAM using a TFTP Server ............................ 78
  Booting from RAM using an NFS Server ......................... 78
  Mounting a Root File System from NFS ......................... 78
  Booting over a Parallel Port ........................................... 79
  Downloading and Executing Programs ................................ 79
Customizing BlueCat Linux OS Loader ............................... 80
  Adding New Commands to BLOSH ..................................... 80
  BLOSH Command Reference .......................................... 81
  Rebuilding BLOSH ............................................................. 85

CHAPTER 4  BLUECAT LINUX DEMO SYSTEMS ............................................................ 87
Overview ............................................................................... 87
  Demo System Components ............................................... 88
  Location .............................................................................. 88
  Contents of Demo System Directory .................................. 89
List of Supported Demo Systems ......................................... 90
Configuring a Demo System ................................................ 91
  For Hardware Devices ....................................................... 91
  For the Boot Device ........................................................ 91
  Using the Makefile to Rebuild a Demo System .................. 92
Running Demo Systems ..................................................... 93
Demo Systems Reference .................................................. 94
  Requirements ................................................................. 94
  osloader ......................................................................... 96
  showcase ......................................................................... 97
  developer ....................................................................... 98
### CHAPTE5R 5  BLUECAT LINUX ADVANCED POWER MANAGEMENT

General Architecture ................................................................. 101

Overview .................................................................................... 101
APM Modules and Components .................................................. 103
APM Event Queue ........................................................................ 104
  Pre- and Post-Events ............................................................ 104
  Event Processing ................................................................. 105
Power States ................................................................................ 107
Software Inactivity Timers ......................................................... 108
APM Interfaces ............................................................................ 108
  Kernel-Space Client Interface .............................................. 108
  PMD Device Drivers Interface .......................................... 110
  IOCTL Interface to User Space ........................................ 111
User-Space Components ............................................................ 111
  APM Daemon ................................................................. 111
  APM Control Utility ......................................................... 112
  APM proc File ................................................................. 112
CPU Power Management ........................................................... 112
Configuring APM in the Kernel .................................................. 113
APM Interfaces Reference ........................................................ 114
  Common Constants and Data Types ..................................... 114
  PMD Handles ....................................................................... 114
  Error Codes ........................................................................ 114
  PMD State Codes ............................................................. 115
  Event Code Tables ........................................................... 115
Kernel-Mode API ........................................................................ 116
  Client Handles ..................................................................... 116
  Client Callback .................................................................... 116
  Client Services .................................................................... 117
PMD API .................................................................................... 119
  PMD Callback ..................................................................... 119
PMD Requests ........................................................................... 120
  PMD Services ..................................................................... 120
IOCTL Commands ....................................................................... 121
User-Mode Interfaces ............................................................... 122
  /proc/mapm Directory ....................................................... 122
  mapmd Command Line Format ....................................... 123
  mapmd Configuration File ................................................ 123
  Handler Program Command Line Format ......................... 123
  Event Logging ..................................................................... 124
Configuring Partitions Using flash_fdisk ........................................... 154
Using the /proc/mtd File ................................................................... 154
Erasing a Flash Memory Device or Partition ..................................... 155
Writing Raw Data to Flash Memory .................................................. 155
Managing JFFS ................................................................................... 156
Downloading BlueCat Linux into Flash Memory with Flash Management Tools ................................................................. 156
Developing an MTD Driver ................................................................. 157
Registering an MTD Driver ................................................................. 157
Deregistering the MTD Driver ............................................................ 159
Configuring Partitions at Runtime ..................................................... 160

APPENDIX A

COMMAND REFERENCE ........................................................................ 161
flash_erase ........................................................................................... 161
  Description ....................................................................................... 161
  Example ............................................................................................. 161
flash_fdisk ........................................................................................... 161
  Description ....................................................................................... 161
  Configuration String Format ............................................................ 162
  Example ............................................................................................. 162
mapm_ctrl ............................................................................................ 163
  Description ....................................................................................... 163
  Parameters ....................................................................................... 163
  Examples ........................................................................................... 163
mapmd ................................................................................................. 164
  Description ....................................................................................... 164
  Options ............................................................................................. 164
mapmd.conf .......................................................................................... 164
  Description ....................................................................................... 164
  Example ............................................................................................. 165
mkboot ................................................................................................. 165
  Description ....................................................................................... 165
  Options ............................................................................................. 165
  Examples ........................................................................................... 167
mkkernel ............................................................................................. 168
  Description ....................................................................................... 168
mkrootfs ............................................................................................. 169
  Description ....................................................................................... 169
## Preface

### Typographical Conventions

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

<table>
<thead>
<tr>
<th>Kind of Text</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body text; <em>italicized</em> for emphasis, new terms, and book titles</td>
<td>Refer to the <em>LynxOS User’s Guide</em>.</td>
</tr>
<tr>
<td>Environment variables, file names, functions, methods, options, parameter names, path names, commands, and computer data</td>
<td><code>ls</code>&lt;br&gt;<code>-l</code>&lt;br&gt;<code>myprog.c</code>&lt;br&gt;<code>/dev/null</code>&lt;br&gt;<code>login: myname</code>&lt;br&gt;<code># cd /usr/home</code></td>
</tr>
<tr>
<td>Commands that need to be highlighted within body text, or commands that must be typed as is by the user are <strong>bolded</strong>.</td>
<td><code>cat &lt;filename&gt;</code>&lt;br&gt;<code>mv &lt;file1&gt; &lt;file2&gt;</code></td>
</tr>
<tr>
<td>Text that represents a variable, such as a file name or a value that must be entered by the user</td>
<td></td>
</tr>
</tbody>
</table>
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 Technical Support is available Monday through Friday (holidays excluded) between 8:00 AM and 5:00 PM Pacific Time (U.S. Headquarters) or between 9:00 AM and 6:00 PM Central European Time (Europe).

The LynuxWorks World Wide Web home page provides additional information about our products and LynuxWorks news groups.
LynuxWorks U.S. Headquarters
Internet: support@lnxw.com
Phone: (408) 979-3940
Fax: (408) 979-3945

LynuxWorks Europe
Internet: tech_europe@lnxw.com
Phone: (+33) 1 30 85 06 00
Fax: (+33) 1 30 85 06 06

World Wide Web
http://www.lynuxworks.com
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 2.4.2) 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 non-proprietary open-source kernel, a complete suite of GNU/EGCS 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. There is support for development in Java and Tcl. 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, installation 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 BlueCat Linux 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 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 pre-installed, fully operational cross development host. The host system can be either Linux (Intel IA-32 or x86 PC compatible running Red Hat Linux 6.2 or higher) or Windows (Intel IA-32 or x86 PC compatible running Windows NT, or Windows 2000).

System requirements for the cross development host are as follows:

- Standard set of Linux or Windows utilities
- 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:

![Binary Architecture CD-ROM Structure](image)

**NOTE:** In the BlueCat\_<cpu> directory, \_<cpu> implies an 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** 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** Cross Development Tools - Directory containing binary packages of the cross development tools that run on the cross development host.
• target  Target board directory - Contains 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**

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

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

![Board Support Package CD-ROM Tree](image)

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

BSP CD-ROM Directories and Components

The BlueCat Linux Board Support Package CD-ROM has a one or more directories `bsp.<bsp1>`, ..., `bsp.<bspn>`, each for a specific board. Note that the accompanying Board Support Guide is on the Documentation CD-ROM.

- `bsp.<bspn>` Target board directory containing target board-specific binary and source packages, i.e., `<bspn>` (where `<bspn>` is `cpci_cpv5360` for the x86-based CPV5350 board).

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**

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/mnt/cdrom</td>
<td>Mount point (typical)</td>
</tr>
<tr>
<td>- BlueCat_&lt;cpu&gt;/</td>
<td>BlueCat Linux tools and packages for a specific CPU architecture (for example, BlueCat_i386 for x86)</td>
</tr>
<tr>
<td>-- bsp.&lt;bspn&gt;/</td>
<td>BlueCat Linux packages for a specific target board, where <code>&lt;bspn&gt;</code> is its Board Support Package</td>
</tr>
</tbody>
</table>
Source Architecture CD-ROM Structure

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

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--- RPMS/</td>
<td>Target board-specific packages</td>
</tr>
<tr>
<td>- install</td>
<td>Installation program</td>
</tr>
</tbody>
</table>

Source Architecture CD-ROM Structure

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

![Source Architecture CD-ROM Tree](image)

**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**: Cross development tools directory containing sources of RPM packages of the cross development tools that run on the host
- **target**: Target board directory containing sources of the RPM packages configured to run on a target board
Chapter 1 - Introduction and Installation

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

### Table 1-3: Source Architecture CD-ROM Components

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/mnt/cdrom</td>
<td>Mount point (typical)</td>
</tr>
<tr>
<td>- BlueCat_&lt;cpu&gt;/</td>
<td>BlueCat Linux tools and packages for a specific CPU architecture (for example, BlueCat_i386 for x86)</td>
</tr>
<tr>
<td>-- cdt/</td>
<td>Cross development packages directory</td>
</tr>
<tr>
<td>--- SRPMS/</td>
<td>Sources of cross development packages</td>
</tr>
<tr>
<td>-- target/</td>
<td>Target board packages directory</td>
</tr>
<tr>
<td>--- SRPMS/</td>
<td>Sources of target board packages</td>
</tr>
</tbody>
</table>

### Installation Procedure

#### Installing the Default Configuration

Use the following procedure to install the BlueCat Linux core components on the cross development host:

1. First, select a cross development host type:

   A) **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.

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

   Please note that some desktops such as Red Hat Linux GNOME 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.

B) 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.

---

**Packages in the Default BlueCat Linux Configuration**

The default BlueCat Linux configuration includes all the RPM packages installed from the *Binary Architecture* CD-ROM.

**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, `cpci_cpn5350` for the x86-based CPV5350 target board.)
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 pre-installed 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:

   ```bash
   . 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, i.e., 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 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-3.02-1.src.rpm

   where Bluecat_<cpu> is the target board CPU (i386 for x86 boards).

   Upon completion of the command, the tar (compressed image) file with the sources of the sed RPM package (sed-3.02.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. Since the sed RPM package does not patch the tar file, there are no patch files in the SOURCES directory.

   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 $BLUECAT_PREFIX/cdt/src/bluecat/SPECS
   BlueCat:$ rpm -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 $BLUECAT_PREFIX/cdt/src/bluecat/SPECS
   BlueCat:$ rpm -ba sed_trg.spec
Using the BlueCat Linux rpm Utility

This places the rebuilt RPM package (sed_trg-3.02-1.<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-0.2-1.<cpu>.rpm

**NOTE:** Rebuild of certain RPM packages requires 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.

After exiting the version of rpm included in the BlueCat Linux distribution, the 
next execution of rpm calls the default host RPM facility, if any.

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
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**

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$BlueCat_PREFIX</td>
<td>Installation directory</td>
</tr>
<tr>
<td>- bin/</td>
<td>Target board binaries</td>
</tr>
<tr>
<td>- boot/</td>
<td>Target board boot directory</td>
</tr>
<tr>
<td>- cdt/</td>
<td>Cross development tools</td>
</tr>
<tr>
<td>-- bin/</td>
<td>Cross development binaries</td>
</tr>
<tr>
<td>-- include/</td>
<td>Cross development include files</td>
</tr>
<tr>
<td>-- info/</td>
<td>GNU info files</td>
</tr>
<tr>
<td>-- lib/</td>
<td>Cross development libraries</td>
</tr>
<tr>
<td>-- man/</td>
<td>Cross development tool man pages</td>
</tr>
<tr>
<td>--&lt;cpu&gt;-lynx-linux-bluecat/</td>
<td>Cross development binaries</td>
</tr>
<tr>
<td>- demo/</td>
<td>BlueCat Linux demo system configurations</td>
</tr>
<tr>
<td>- etc/</td>
<td>Target board configuration files</td>
</tr>
<tr>
<td>- lib/</td>
<td>Target board libraries</td>
</tr>
<tr>
<td>- sbin/</td>
<td>Target board system binaries</td>
</tr>
<tr>
<td>- usr/</td>
<td>Target board top-level usr directory</td>
</tr>
<tr>
<td>-- bin/</td>
<td>More target board binaries</td>
</tr>
<tr>
<td>-- doc/</td>
<td>Various documentation files</td>
</tr>
<tr>
<td>-- include/</td>
<td>Target board include files</td>
</tr>
<tr>
<td>-- info/</td>
<td>More GNU info files</td>
</tr>
<tr>
<td>-- lib/</td>
<td>More target board libraries</td>
</tr>
<tr>
<td>-- libexec/</td>
<td>Auxiliary files</td>
</tr>
<tr>
<td>-- man/</td>
<td>Main man pages</td>
</tr>
<tr>
<td>-- sbin/</td>
<td>More target board system binaries</td>
</tr>
<tr>
<td>-- share/</td>
<td>Various target board shared files</td>
</tr>
</tbody>
</table>
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**

Before starting any BlueCat Linux development, the correct execution environment must be set using the following procedure:

1. Go to the directory where BlueCat Linux has been installed:
   ```bash
   $ cd $HOME/BlueCat
   ```
2. Source the shell script to set up the BlueCat Linux environment variables:
   ```bash
   $ . 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_TSP` 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 multi-user 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 co-exist 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 LynxWorks 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 <new_config_file> <resulting_kernel_filename>
```

The `<new_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, <resulting_kernel_filename>, specifies the location for copying the resulting kernel image.

**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. If the -T option is passed to mkrootfs, it creates a tar image of the root file system. If the -J option is passed to mkrootfs, it creates a Journalling Flash File System image.

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

**mkboot**

The **mkboot** cross development tool can be used to copy bootable images of BlueCat Linux onto a floppy disk or a hard disk from the cross development host. **mkboot** is capable of copying the BlueCat Linux boot sector to the media; copying a compressed kernel image to the media; Setting the command line of the kernel; Copying the compressed root file system image to the media; defining for the kernel the root file system to be mounted at boot time.

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, 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 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 2.4.2. 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**

<table>
<thead>
<tr>
<th>Kernel Subdirectory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>linux/init</td>
<td>Functions needed to start the kernel</td>
</tr>
<tr>
<td>linux/kernel</td>
<td>Kernel core and system calls</td>
</tr>
<tr>
<td>linux/mm</td>
<td>Memory management</td>
</tr>
<tr>
<td>linux/fs</td>
<td>Implementations of various file systems supported by Linux kernel</td>
</tr>
<tr>
<td>linux/ipc</td>
<td>Sources for System V IPC, such as semaphores, shared memory, and message queues</td>
</tr>
<tr>
<td>linux/net</td>
<td>Implementation of various network protocols (TCP/IP, ARP, and so on)</td>
</tr>
<tr>
<td>linux/lib</td>
<td>Some standard C library functions</td>
</tr>
<tr>
<td>linux/modules</td>
<td>The directory in which runtime modules are held</td>
</tr>
<tr>
<td>linux/include</td>
<td>Kernel-specific header files</td>
</tr>
<tr>
<td>linux/drivers</td>
<td>Device drivers for hardware components, divided into subdirectories according to the device type</td>
</tr>
<tr>
<td>linux/arch/i386</td>
<td>Architecture-dependent code for Intel IA-32 or x86 PC compatibles</td>
</tr>
<tr>
<td>linux/arch/ppc</td>
<td>Architecture-dependent code for PowerPC processors</td>
</tr>
<tr>
<td>linux/arch/arm</td>
<td>Architecture-dependent code for ARM processors</td>
</tr>
<tr>
<td>linux/arch/sh</td>
<td>Architecture-dependent code for SH processors</td>
</tr>
<tr>
<td>linux/arch/mips</td>
<td>Architecture-dependent code for MIPS processors</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel_trg-headers-2.4.2-1</td>
<td>C header files for the BlueCat Linux kernel</td>
</tr>
<tr>
<td>kernel_trg-source-2.4.2-1</td>
<td>Source code files for the BlueCat Linux kernel and kernel object files built for the default configuration of the kernel</td>
</tr>
<tr>
<td>kernel_trg-&lt;target&gt;-2.4.2-1</td>
<td>Linux kernel binary built for the default configuration of the BlueCat Linux kernel</td>
</tr>
<tr>
<td>kernel_trg-doc-2.4.2-1</td>
<td>References to options that can be passed to the BlueCat Linux kernel at load time</td>
</tr>
<tr>
<td>kernel_trg-bcboot-2.4.2-1</td>
<td>BlueCat Linux boot record template used for copying BlueCat Linux on a hard disk or a floppy</td>
</tr>
</tbody>
</table>

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_<target_cpu>/target/SRPMS.

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

- **SRPMS File**: `kernel_trg-2.4.2-1.src.rpm`
- **Patch**: `linux-bc.patch.gz`
- **Description**: BlueCat Linux changes to Linux kernel

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 in 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.

![BlueCat Linux Development Flow Chart](image)

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 16.

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
   - 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, i.e., using:

- Target board firmware
- `mkboot` tool
- OS loader
- `install` demo system

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 either:
   - Directly from the boot device, or
   - In a two-step procedure, where the OS loader is booted from a boot device (floppy disk, hard disk, 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 2.4.2 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 four 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 GUI and uses the Tk interpreter. 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 appropriate kernels are updated to reflect the changes in configuration.

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

**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 (e.g., 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 or hard disk.

• 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 or Hard Disk

The `make bzImage` command builds the new kernel and creates the compressed kernel image ready for copying on a floppy disk or hard disk. 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 BlueCat Linux OS loader directly onto a target board. The compressed kernel, therefore, does not contain a boot sector.

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

The `bvmlinux.out` 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 approach supported by BlueCat Linux is very simple:
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.

mkkernel

The BlueCat Linux tool mkkernel included in the cross development tools is a sample implementation of this procedure. The syntax for mkkernel is:

```bash
mkkernel <new_config_file> <resulting_kernel_filename>
```

The <new_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, <resulting_kernel_filename>, specifies the location for copying the resulting kernel image.

Clearly, mkkernel does not implement all the flexibility of BlueCat Linux kernel development that may be required. It is provided as a simple example of a tool that is used to maintain multiple kernel profiles. For more information, please refer to Appendix A, “Command Reference.”

The shell script implementing mkkernel is available in the $BLUECAT_PREFIX/cdt/bin directory.

**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 kernel debugger 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 board BlueCat Linux with the kernel debugger enabled - To enable the kernel debugger, turn the `CONFIG_BLUECAT_KDBG` kernel option on by setting it to `Y` in the **Kernel Hacking** menu of an appropriate `make config` operation.
- The BlueCat Linux cross development host
- A serial line connection between the BlueCat Linux target board and the cross development host
- 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.
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.

**CAUTION!** Do not use the `set remotebaud gdb` command, because it can cause BlueCat Linux kernel debugger/GDB communication problems.

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 -nw 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 “Building the Kernel” on page 30 for a detailed description of building a `vmlinux` kernel image.

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

```
(gdb) target remote /dev/ttyS1  
Remote debugging using /dev/ttyS1  
0xc0123456 in kdbg 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.
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.
0xc0131059 in kdbg_breakinst ()
(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.
BlueCat Linux Kernel Debugger Extensions

In addition to the standard set of GDB operations, the BlueCat Linux cross GDB supports the display of contents of kernel data structures.

The following table describes additional GDB commands:

**Table 2-3: Supported GDB Commands**

<table>
<thead>
<tr>
<th>Command Format</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info proc [&lt;addr&gt;</td>
<td>&lt;pid&gt;]</td>
<td>info proc 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Displays contents of a task_struct kernel structure for task ID &lt;pid&gt;, address &lt;addr&gt;, or current task</td>
</tr>
<tr>
<td>info inode &lt;addr&gt;</td>
<td>info inode 0xc0240000</td>
<td>Displays contents of an inode structure at address &lt;addr&gt;</td>
</tr>
<tr>
<td>info block &lt;addr&gt;</td>
<td>info block 0xc0240000</td>
<td>Displays contents of a buffer_head structure at address &lt;addr&gt;</td>
</tr>
<tr>
<td>info sfiles &lt;addr&gt;</td>
<td>info sfiles 0xc0240000</td>
<td>Displays contents of a files_struct structure at address &lt;addr&gt;</td>
</tr>
<tr>
<td>info file &lt;addr&gt;</td>
<td>info file 0xc0240000</td>
<td>Displays contents of a file structure at address &lt;addr&gt;</td>
</tr>
<tr>
<td>info fifo &lt;addr&gt;</td>
<td>info fifo 0xc0240000</td>
<td>Displays contents of a pipe_inode_info structure at address &lt;addr&gt;</td>
</tr>
<tr>
<td>info sblock &lt;addr&gt;</td>
<td>info sblock 0xc0240000</td>
<td>Displays contents of a super_block structure at address &lt;addr&gt;</td>
</tr>
</tbody>
</table>
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.

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 gdbserver program, conveniently named `gdbserver`, runs on the target board.

`gdb` and `gdbserver` communicate by 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 `$BLUECAT_PREFIX/usr/bin`. The cross debugger GDB binary (`gdb`) resides in `$BLUECAT_PREFIX/cdt/bin`. 

38   BlueCat Linux User's Guide
Procedure
The following is the procedure to debug an application program:

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 43.)

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

   `<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`, type:

     ```bash
     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
     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. An unstripped copy of the program is needed 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 -nw test_prog
   ```

3. Then, 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, i.e., 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 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) image that can burned into target flash memory and then used to boot the system.

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 neither the `-T` or `-J` option is specified, `mkrootfs` creates a compressed root file system image. This image can be copied on a floppy disk, a hard disk, or a ROM. When BlueCat Linux boots on the target board, it automatically loads the file system image to RAM and mounts it as the root file system.

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.
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, \texttt{mkrootfs} runs the \texttt{ldconfig} utility to create the cache and all appropriate symbolic links.

Using the \texttt{-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.

\section*{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 (\texttt{gcc}) makes an executable program use shared libraries. If statically linking an application is desired, use the \texttt{-static} option when calling \texttt{gcc}.

\section*{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 \texttt{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 \texttt{/proc} area on the target board:

\begin{itemize}
  \item \texttt{/proc/memstattotal} \textendash{} System-wide memory usage statistics
  \item \texttt{/proc/memstatproc} \textendash{} Process-specific statistics for currently running processes
  \item \texttt{/proc/memstathist} \textendash{} Process-specific statistics for finished processes
  \item \texttt{/proc/memstattracepage} \textendash{} Traceback of kernel components that have allocated significant amounts of RAM
\end{itemize}
To get access to the data collected in the /proc files, simply read the contents using standard UNIX commands. For instance:

```
bash# more /proc/memstatotal
bash# more /proc/memstathist
```

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

/proc/memstatotal 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
- **Cache** - Memory used by the page cache
- **Shared** - Memory taken by shared memory regions
- **Swap** - Swap space currently in use
- **VMAloc** - Kernel memory allocated via vmalloc
- **KMAloc** - 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:

```
nnnnnnn 0xaaaaaaaa 0xaaaaaaaa ....
```

where `nnnnnnnn` is the total number of requested pages (in decimal), and `0xaaaaaaaa ....` 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.
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:

- Copy BlueCat Linux onto a floppy disk and then boot BlueCat Linux onto the target board from the floppy disk.
- Copy BlueCat Linux onto a hard disk and then boot the target board from the hard disk.
- Download BlueCat Linux into target ROM/flash memory and then boot the target board from ROM/flash memory.
- Boot BlueCat Linux onto the target board from a network using the target board firmware.
- Boot BlueCat Linux onto the target board from a network using the BlueCat Linux OS loader.
- Copy OS loader to floppy or hard disk and boot target with OS loader. Then boot BlueCat Linux from parallel port.

Please refer to the figure “BlueCat Linux Development Flow Chart” on page 26.


**mkboot Cross Development Tool**

The `mkboot` cross development tool can be used to copy bootable images of BlueCat Linux onto a floppy disk or a hard disk 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.

`mkboot` is capable of the following functionalities:

- Copying the BlueCat Linux boot sector to the media
- Copying a compressed kernel image to the media; the compressed kernel image is copied after the boot sector.
- Setting the command line of the kernel booted from the compressed kernel image; the kernel command line is stored in one of the first sectors on the disk.
- Copying the compressed root file system image to the media; the compressed file system is loaded by the kernel into RAM and mounted as the root file system. The file system image is stored on the media after the compressed kernel image.
- Defining for the kernel the root file system to be mounted at boot

Additionally, `mkboot` can be used to create a BlueCat Linux image composed of a Linux kernel and a compressed file system. This image can be booted onto the target board from a network using the target board’s native firmware, or programmed into target ROM/flash memory.

For more information, refer to the section “mkboot” on page 165 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 `/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. 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 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 download BlueCat Linux onto the target board every time the system or applications are updated. The advantage of 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, OS loader can be removed and the final system boots up directly from the disk. (See “Boot Mechanism using OS Loader.”)

Structurally, the OS loader is a combination of the BlueCat Linux kernel and the BlueCat 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.

---

### Setting Up a BOOTP Server

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

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

```bash
# bootptab file
.defaults:\
:dn=es.lynx.com\
:sm=255.255.255.0\
:ds=207.21.185.10\
:hn:\
matrix2:\
:ht=ether:ha=003023000001\
:ip=192.168.111.2\
:sa=192.168.111.254\
:tc=.defaults:
```
For more detailed information on the `bootptab` file format, refer to the `bootptab(5)` man page.

1. If the `bootp` daemon is not already enabled, add or uncomment the following line in the `/etc/inetd.conf` file:

   ```
   bootps dgram udp wait root/usr/sbin/tcpd in.bootpd bootpd
   ```

2. If the `/etc/inetd.conf` file has been changed, send the HUP signal to the `inetd` process to order it to re-read configuration.

   For further information on the `inetd.conf` file format refer to the `inetd.conf(5)` man page.

### Setting Up a TFTP Server

Use the following instructions to enable TFTP on a Linux host system:

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 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 provides 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>(r,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, use the following procedure:

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

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. The gateway IP address is specified by the `<gw_ip>` field. For example:

   ```
ip=1.0.3.2:172.16.1.2:1.0.3.1::bc- 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
   ```
Additional steps are required 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.

  1. 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 NT/2000**

  1. 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.
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 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, use the following procedure:

1. On the cross development host, insert a floppy disk into the floppy drive corresponding to the `/dev/fd0` special node.
2. Write permission is required to write to `/dev/fd0`. If needed, do the following as a superuser:
   ```bash
   # chmod uga+rwx /dev/fd0
   ```
3. Copy BlueCat Linux onto the floppy disk using the `mkboot` utility. For example, to copy the BlueCat Linux OS loader, execute these commands:
   ```bash
   BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
   BlueCat:$ mkboot -b /dev/fd0
   BlueCat:$ mkboot -k osloader.disk /dev/fd0
   BlueCat:$ mkboot -f osloader.rfs /dev/fd0
   BlueCat:$ mkboot -r /dev/fd0 /dev/fd0
   ```

   where:
   - `/dev/fd0` - Floppy disk; for Windows host, replace with `a`:
   - `-b` - Copies the BlueCat Linux boot sector.
   - `-k` - Copies the compressed kernel to the medium.
   - `-f` - Copies the compressed root file system image to the medium.
   - `-r` - Sets the device node on the target board to mount as the root file system or uncompress the file system image.
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), 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.
8. In the case of BlueCat Linux OS loader, the system boots up to display the OS loader prompt (>).
Booting BlueCat Linux from Hard Disk

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

Copying onto Hard Disk

To copy a BlueCat Linux system onto a hard disk from the cross development host, use the following procedure:

1. Attach the hard disk (primary master) 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/hda
   ```
   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.*

3. 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.

   First, obtain write permission for the hard disk. Then, proceed to copy OS loader onto the disk.
   ```
   # chmod uga+r /dev/hda
   BlueCat:$ cd $BLUECAT_PREFIX/demo/osloader
   BlueCat:$ mkboot -b /dev/hda
   BlueCat:$ mkboot -k osloader.disk /dev/hda
   BlueCat:$ mkboot -f osloader.rfs /dev/hda
   BlueCat:$ mkboot -r /dev/fd0 /dev/hda
   ```
   where:
   - **hda** – Hard disk; for Windows host, replace with ....
   - **-b** – Copies the BlueCat Linux boot sector.
   - **-k** – Copies the compressed kernel to the medium.
Copying onto Hard Disk

- **-f** - Copies the compressed root file system image to the medium.
- **-r** - Sets the device node on the target board to mount as the root file system or uncompress the file system image.

For this boot scenario this step completes the copy.

1. If the file system is mounted from a partition on the hard disk, create the partition using `fdisk`.

2. Create the file system on the newly made partition.
   ```
   # mke2fs /dev/hda2
   ```

3. 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:
   ```
   # mount /dev/hda2 /mnt1
   # cd /mnt1
   # tar xvf osloader.tar
   <BlueCat_Linux_installation_point>/demo/osloader/
   ```

4. Unmount the hard disk:
   ```
   # cd /
   # umount /mnt1
   ```

5. 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 osloader.disk /dev/hda
   BlueCat:$ mkboot -r /dev/hda2 /dev/hda
   ```
   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 48.)

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

**NOTE:** When duplicating a hard drive image via Norton Ghost, Drive Image, etc., the boot record is not duplicated. Duplicated disks require the boot sector to be rewritten as per Step 5.
Booting from Hard Disk using Network and OS Loader

Root File System Copied to Partition on Target Hard Disk

To download BlueCat Linux from a TFTP server onto a target hard disk using the OS loader, proceed as follows:

1. Attach the hard disk to the target board.
2. Copy the OS loader in bootable form onto a floppy disk. The osloader demo system can be found in $BLUECAT_PREFIX/demo/osloader. See “Copying onto Floppy Disk” on page 56.
3. Boot the OS loader on the target board.

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

4. A successful boot brings up the BLOSH (BlueCat OS Loader Shell) prompt on the target board console (>). 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
5. Create a partition on the hard disk to hold the kernel. For example:

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

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.*

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

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

6. Create a second partition to hold the root file system using `fdisk`.

7. Create a file system on the newly made partition. For example:

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

8. Mount the partition on the hard disk. For example:

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

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

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

10. Copy the kernel image, specifying the root file system. (See “mkboot Cross Development Tool” on page 48.)

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

11. Remove the boot floppy containing OS loader 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

The procedure in “Root File System Copied to Partition on Target Hard Disk” 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, proceed as follows:

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.
3. Boot the target board with OS loader.
4. A successful boot displays the BLOSH (BlueCat Loader Shell) prompt on the target board console (>). 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

**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.
5. Create a boot partition to hold the kernel. For example:

   > exec fdisk /dev/hda

   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.*

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

   > sync
   > sync
   > reset

6. Copy the kernel and root file system images on the hard disk. (See “mkboot Cross Development Tool” on page 48.)

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

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

   > sync
   > sync
   > reset

---

**Booting from DiskOnChip**

Support for a DiskOnChip device in BlueCat Linux is based on a binary driver for DiskOnChip distributed by the vendor. Currently, support is available for the x86 target board only.

**Adding Support for DiskOnChip in the BlueCat Linux Kernel**

1. Download a binary DiskOnChip software for Linux 2.4.x i386 from the M-Systems Web site (www.m-sys.com) to a temporary directory /tmp.
2. Unpack the driver as described in the DiskOnChip documentation:

   BlueCat:$ cd /tmp
   BlueCat:$ tar xzf <archive_location>/driver_4.2.1.tgz

3. Follow the steps described in the README.kit file of the /tmp/Linux_DOC_4.2.1/driver directory using $BLUECAT_PREFIX as a value for the kernel_src_dir variable.

   NOTE: The current version of the driver is designed for Linux v2.4.0. To make the driver work with BlueCat Linux based on Linux v2.4.2, the user must make the following changes in the files listed below. These files are located in the $BLUECAT_PREFIX/usr/src/linux/drivers/block/ directory.

      * Makefile
        Old line: SUB_DIRS += doc
        New line: subdir-y += doc
      * doc/Makefile
        Old line: O_OBJS := fldrvlnx.o libosak.a
        New line: obj-y := fldrvlnx.o libosak.a
      * doc/fldrvlnxc.c
        Old line
        #if LINUX_VERSION_CODE > KERNEL_VERSION(2,4,0)
        #error "Linux Kernel Version newer than 2.4.0 not supported"
        New line
        #if LINUX_VERSION_CODE > KERNEL_VERSION(2,4,2)
        #error "Linux Kernel Version newer than 2.4.2 not supported"

These changes may not be required for newer versions of the driver.

4. In the BlueCat Linux embedded system, turn on the DiskOnChip kernel configuration option (CONFIG_BLK_DEV_MSYS_DOC). For example, run the standard configuration tool make xconfig to enable the M-Systems DOC device support option in the Block Devices submenu.

5. Rebuild the kernel.

6. The M-Systems 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 100 and the minor number corresponding to a disk partition (0 for entire disk, 1 for the first partition, and so forth).

The install and osloader demo systems already have special device files for the DiskOnChip driver integrated in the root file system. They are named /dev/tffs, /dev/tffs1, /dev/tffs2, /dev/tffs3, and /dev/tffs4. If support for DiskOnChip is added to the kernel (See “Adding Support for DiskOnChip in the BlueCat Linux Kernel” on page 63.), it can be enabled in the osloader or install demo systems 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.

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 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, use the following procedure:

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 command creates 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 48.)

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.
Chapter 3 - Downloading and Booting BlueCat Linux

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

   BlueCat:$ mkboot -m -k showcase.kernel -f showcase.rfs showcase.kdi

(See “mkboot Cross Development Tool” on page 48.)

To download a composite BlueCat Linux image onto ROM/flash memory on the target board using the OS loader, use the following procedure:

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 Floppy Disk” on page 56.

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-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) partition, and the kernel then made to mount it at boot.

- **JFFS Image Built on the Host**
  
  To download BlueCat Linux with an JFFS root file system image built on the cross development host using OS loader, use the following procedure:

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

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

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 Floppy Disk” on page 56.

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 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 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 image to be burned into the second flash memory partition:

   > set FILE tftp /tftpboot/showcase.jffs

8. Download the JFFS image of the root file system specifying the target flash memory partition created for it. For example the following command places the JFFS image into the second flash memory partition:

   > flash /dev/mtdchar2 erase

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

   > reset

• Creating a Root File System in a JFFS Partition on the Target

The procedure in “JFFS Image Built on the Host” 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 management tools.

To download BlueCat Linux into target flash memory with a JFFS root file system created from the tar archive, use the following procedure:

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 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 48.)

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 Floppy Disk” on page 56.

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. For example:

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

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

    ```
    > cd /mnt
    > ntar
    ```
Chapter 3 - Downloading and Booting BlueCat Linux

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, use the following procedure:

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.

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.

---

**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.
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 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 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 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 Network or Parallel Port using OS Loader

This section explains booting BlueCat Linux onto a 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 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 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 48 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. OS loader downloads a compressed kernel image, and optionally, a compressed root file system image onto the target board memory.
2. OS loader shuts down its kernel.
3. OS loader moves the loaded kernel and file system images to the appropriate places in RAM.
4. 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 bootstrapping 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 OS loader is also applicable to the `i_osloader` demo system.

**Hardware Support for Net Booting with 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`), BlueCat Linux bidirectional parallel port driver (`CONFIG_BLUECAT_BPAR`), device driver for a particular parallel port
IP Autoconfiguration using 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 file name 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 84 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:

```bash
# /etc/blosh.rc
# autoboot from TFTP host

set IP <target_IP_address>
set HOST <host_IP_address>
set IF <ethernet_interface>
set KERNEL tftp /tftproot/<program>.kernel
set RFS tftp /tftproot/<program>.gz

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. 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 Floppy Disk” on page 56.
Chapter 3 - Downloading and Booting BlueCat Linux

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/<program>.kernel
> set RFS tftp /tftpboot/<program>.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 <program>.kernel
> set RFS nfs /nfsboot <program>.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):
Booting over a Parallel Port

The following BLOSH commands boot BlueCat Linux from a parallel port:

> set PPORT /dev/bpar0
> set KERNEL pftp /pftpboot/<program>.kernel
> set RFS pftp /pftpboot/<program>.gz
> 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/<program>
> read /<program>
> exec chmod +x /<program>
> exec /<program>
Chapter 3 - Downloading and Booting BlueCat Linux

Customizing 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:

```c
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;
```

- **name**: Command name
- **do_cmd**: The function that implements the command--It must return one of the following values:
  - BLOSH_SUCCESS,
  - BLOSH_FAILURE
  - BLOSH_USAGE
- **usage**: Usage text--If `do_cmd()` returns BLOSH_USAGE, the text is prefixed with Usage: and displayed.
- **description**: Short command description to be displayed by the `help` command
- **support_level**: 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 Filesystem**

```
mount <device> <directory>
```

The `mount` command mounts a filesystem 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 (i.e., 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>
> 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 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 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 filesystem 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 filesystem on a hard disk for an x86 target. The kernel installed by this
example boots from a hard disk, then uncompressed 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/<program>.disk
> set RFS tftp /tftpboot/<program>.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:

```plaintext
> set IP <target_IP>
> set HOST <host_IP>
> set IF network_interface
> set FILE tftp /tftpboot/script.<target_IP>
> 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.
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 (i.e., 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 `.config` file that can be used to rebuild the kernel. Use the `mkkernel` (or an equivalent) tool to build the kernel image from the `.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 `.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

<table>
<thead>
<tr>
<th>File/ Subdirectory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;demo&gt;.config</code></td>
<td>Kernel configuration file</td>
</tr>
<tr>
<td><code>&lt;demo&gt;.spec</code></td>
<td>Root file system specification file</td>
</tr>
<tr>
<td><code>&lt;demo&gt;.kernel</code></td>
<td>Prebuilt, compressed kernel image suitable for booting onto a target board over a network using BlueCat Linux OS loader</td>
</tr>
<tr>
<td><code>&lt;demo&gt;.disk</code></td>
<td>Prebuilt, compressed kernel image suitable for copying onto a floppy or hard disk</td>
</tr>
<tr>
<td><code>&lt;demo&gt;.rfs</code></td>
<td>Prebuilt, compressed RAM disk root file system image suitable for booting onto a target board over a network using BlueCat Linux OS loader, or for loading from a floppy disk or hard disk</td>
</tr>
<tr>
<td><code>&lt;demo&gt;.tar</code></td>
<td>Tar image of the root file system suitable for copying on a hard disk partition or for NFS-mounting</td>
</tr>
<tr>
<td><code>&lt;demo&gt;.kdi</code></td>
<td>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</td>
</tr>
<tr>
<td>Makefile</td>
<td>Makefile to build the demo system</td>
</tr>
<tr>
<td>src</td>
<td>Source files of the custom programs used in the demo system</td>
</tr>
<tr>
<td>local</td>
<td>Configuration files specific to the demo system</td>
</tr>
</tbody>
</table>
List of Supported Demo Systems

The $BLUECAT_PREFIX/demo directory contains the demo systems listed in the table below:

Table 4-2: Demo Systems and their Requirements

<table>
<thead>
<tr>
<th>Subdirectory</th>
<th>Demo System</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>developer</td>
<td>A combination of a simple environment demo, simple networking demos, and a demo for the Visuallynux program</td>
<td>Storage: Medium RAM: Large Network: Yes Disk: None Special: Host and target machines must be connected by a serial line.</td>
</tr>
<tr>
<td>osloader</td>
<td>BlueCat Linux OS loader</td>
<td>Storage: Tiny RAM: Tiny Network: Yes Disk: None Special: None</td>
</tr>
<tr>
<td>showcase</td>
<td>Configures an Apache Web Server</td>
<td>Storage: Small RAM: Small Network: Yes Disk: None Special: None</td>
</tr>
</tbody>
</table>
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, etc.). 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, he or she must reconfigure the kernel to add support for the hard disk.

To support booting from a floppy or a hard disk, the user may also need to configure the hardware device driver for the floppy or hard disk, 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 4-3: Typical Makefile Targets**

<table>
<thead>
<tr>
<th>Makefile Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel</td>
<td>Builds <code>&lt;demo&gt;.kernel</code> and <code>&lt;demo&gt;.disk</code></td>
</tr>
<tr>
<td>rootfs</td>
<td>Builds <code>&lt;demo&gt;.rfs</code> and <code>&lt;demo&gt;.tar</code></td>
</tr>
<tr>
<td>kdi</td>
<td>Builds <code>&lt;demo&gt;.kdi</code></td>
</tr>
<tr>
<td>this</td>
<td>Builds custom programs in <code>src/</code></td>
</tr>
<tr>
<td>all</td>
<td>Builds all of the above</td>
</tr>
</tbody>
</table>
| xconfig         | • Copies `<demo>.config` to the kernel `.config`  
                  • Calls `make xconfig`  
                  • Copies the updated kernel `.config` into  
                  `<demo>.config` |
| clean           | Removes all prebuilt binaries |
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 4-4: Demo System Boot Procedure

<table>
<thead>
<tr>
<th>Boot Device</th>
<th>Boot Procedure</th>
<th>Detailed Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floppy Disk</td>
<td>• Copy the demo system onto a floppy from cross development host using <code>mkboot</code></td>
<td>See “Booting BlueCat Linux from Floppy Disk” in Chapter 3, “Downloading and Booting BlueCat Linux.”</td>
</tr>
<tr>
<td></td>
<td>• Boot the target board from floppy disk</td>
<td></td>
</tr>
<tr>
<td>Hard Disk</td>
<td>• Copy the demo system onto a hard disk either from the cross development host using <code>mkboot</code> or from the target board using the OS loader</td>
<td>See “Booting BlueCat Linux from Hard Disk” in Chapter 3, “Downloading and Booting BlueCat Linux.”</td>
</tr>
<tr>
<td></td>
<td>• Boot the target board from a hard disk</td>
<td></td>
</tr>
<tr>
<td>ROM/Flash Memory</td>
<td>• Download a demo system into target ROM/flash memory using firmware, external device-specific tools or from the target board using the OS loader</td>
<td>See “Booting BlueCat Linux from Target ROM/Flash Memory” in Chapter 3, “Downloading and Booting BlueCat Linux.”</td>
</tr>
<tr>
<td></td>
<td>• Boot the target board from ROM/flash memory</td>
<td></td>
</tr>
<tr>
<td>Network or Parallel Port using OS Loader</td>
<td>• Download the OS loader onto a floppy disk, hard disk, or into target ROM/flash memory</td>
<td>See “Booting over Network or Parallel Port using OS Loader” in Chapter 3, “Downloading and Booting BlueCat Linux.”</td>
</tr>
<tr>
<td></td>
<td>• Boot the OS loader on the target board</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Boot a demo system from target RAM using a network (TFTP or NFS) or the parallel port (PFTP) using the OS loader</td>
<td></td>
</tr>
<tr>
<td>Network using Firmware</td>
<td>• Boot a demo system from the network using the firmware netboot option</td>
<td>See “Booting over Network using Target Firmware” in Chapter 3, “Downloading and Booting BlueCat Linux.”</td>
</tr>
</tbody>
</table>
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 non-volatile storage (floppy, hard disk, or ROM/flash memory) on the target board. Storage values can be any of the following:

<table>
<thead>
<tr>
<th>Target Board</th>
<th>Tiny</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
<th>Huge</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>2 MB</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>&gt;16 MB</td>
</tr>
<tr>
<td>x86</td>
<td>1 MB</td>
<td>2 MB</td>
<td>4 MB</td>
<td>8 MB</td>
<td>&gt;8 MB</td>
</tr>
<tr>
<td>PowerPC</td>
<td>2 MB</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>&gt;16 MB</td>
</tr>
<tr>
<td>MIPS</td>
<td>2 MB</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>&gt;16 MB</td>
</tr>
</tbody>
</table>

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:
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, make sure that the specified kernel option is passed in the kernel command line. 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 -e` 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.

---

**Table 4-6: Memory Size Options**

<table>
<thead>
<tr>
<th>Target Board</th>
<th>Tiny</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>&gt;16 MB</td>
</tr>
<tr>
<td>Intel IA-32 or x86 PC compatible</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>&gt;16 MB</td>
</tr>
<tr>
<td>PowerPC</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>&gt;16 MB</td>
</tr>
<tr>
<td>MIPS</td>
<td>4 MB</td>
<td>8 MB</td>
<td>16 MB</td>
<td>&gt;16 MB</td>
</tr>
</tbody>
</table>
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
```

**SYNOPSIS**

Sets up an Apache web server.

**REQUIREMENTS**

- **Storage**  Small
- **RAM**  Small
- **Network**  Yes
- **Disk**  None
- **Special**  None
- **Kernel Option**  `ramdisk_size=4096`

**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.

The system boots in single-user mode. `init` starts `bash` without a login prompt. Bring up the network interface(s) manually using the `ifconfig` command and, optionally, set up the kernel routing table using the `route` command. For instance:

```
bash# ifconfig eth0 172.17.3.4
```

```
bash# route add default gw 172.17.0.1
```

If the HTTP daemon does not run, type the following command:

```
bash# httpd
```

Now the Apache Server is accessible from any networked machine using the IP address 172.17.3.4 and serves the `index.html` page located in the `showcase` subtree in the `demo` directory.
**SYNOPSIS**

Various functionalities are combined in one BlueCat Linux system: developer demonstrates the use of an FTP client, the bash shell, simple networking, remote debugging with GDB, and the VisualLynux program.

**REQUIREMENTS**

<table>
<thead>
<tr>
<th>Storage</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>Large</td>
</tr>
<tr>
<td>Network</td>
<td>Yes</td>
</tr>
<tr>
<td>Disk</td>
<td>None</td>
</tr>
</tbody>
</table>

Special: The host and target machines must be connected by a serial line to use remote debugging via a serial line. Serial tty devices (nodes) are available in the root file system image are /dev/ttyS0 and /dev/ttyS1.

Kernel Option: ramdisk_size=28472

**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
bash# ftp 195.239.208.81
```

To ftp to the target board from another machine on the network:

```bash
[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
bash# ifconfig eth0 172.16.1.62
bash# route add default gw 172.16.0.1
```

2. Then start `gdbserver` with the simple test program `test_prog` included in the root file system image:

```bash
bash# gdbserver target_ip:2345 /test_prog
```

3. To connect `gdbserver` via a serial line connected to COM2 use the following command:

```bash
bash# gdbserver /dev/ttyS1 /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:

   ```bash
   BlueCat:bash# gdb -nw ./test_prog
   ```

6. Use the `target remote` command to connect to `gdbserver` on the target board:

   ```bash
   (gdb) target remote target_ip:2345
   or use the standard GDB commands to debug `test_prog`:
   ```bash
   (gdb) target remote /dev/ttyS1
   ```

**NOTE:** Use COM2 for the Windows hosts.
The VisualLynux program demonstration configures the target board for the required remote procedure call connections in order to establish a target board that is VisualLynux friendly, as well as a debug connection from the cross development host platform to the client target board.

This system builds a Kernel Downloadable Image that contains TCP/IP, FTP, Telnet and other networking components, as well as enough OS utilities to support communication as a networked development target.
This chapter describes Advanced Power Management (APM) support implemented in BlueCat Linux. BlueCat Linux APM support, which allows support of diverse target board architectures while using the same software interfaces, is defined in a generic manner.

BlueCat Linux APM software defines an open, architecture-independent interface with low-level device drivers servicing power-managed devices (PMDs). The upper layers of BlueCat Linux APM use this interface to interact with specific APM hardware devices in an architecture-independent fashion.

BlueCat Linux APM defines a number of user-visible, architecture-independent interfaces, both for the kernel space and the user space. These are used by clients of APM to control all aspects of power management.

BlueCat Linux APM provides the ability to control the power management aspects of the CPU and system operation.

**General Architecture**

**Overview**

BlueCat Linux APM places each power-managed device (PMD) under the control of the software. The software can force each individual PMD into low-power states, either immediately or upon expiration of a software timer, provided no activity is observed at the PMD.

Forcing a PMD into a low-power state is negotiated with the registered kernel-space clients of the APM, each of which can reject switching the PMD to a low-power state. Additionally, the transition of a PMD to a different power state is reported to the user-space daemon (`mapmd`). This reacts in an appropriate manner
according to the user-specified configuration file. Reaction to an APM event may include:

- Sending an informational message to the system console
- Placing an event record in the log file
- Invocation of a user-defined program
- Any combination of the above

In addition to the `mapmd` daemon, the BlueCat Linux APM software includes the user-space control utility (`mapm_ctrl`). This is used to explicitly control PMDs and to provide user-readable APM status information.

All interactions with clients of the APM software are accomplished through a comprehensive architecture-independent interface. This includes both interactions between clients and APM for triggering actions on individual PMDs, and between APM and clients for notification of APM events.

It is important to note that the APM software implements a clearly defined state machine, configured and controlled from the outside of the APM by clients, either kernel-space or user-space, or both. Policy making decisions are made by the APM clients and are made known to the APM software via a set of clearly defined architecture-independent Application Programming Interfaces (APIs).

To support the various features and capabilities implemented by the hardware PMDs on the target boards supported by BlueCat Linux, the APM software defines an open, platform-independent API to low-level PMD device drivers. A PMD device driver registers itself with the core APM modules in order to implement appropriate PMD management functions via the hardware-independent PMD API. The core APM modules call appropriate callbacks, implemented by a PMD driver, to trigger a specific action at the PMD controlled by the driver.

The CPU itself is viewed by the APM as another PMD. The BlueCat Linux APM software includes a special CPU PMD driver which, when enabled, controls the power state of the CPU. The CPU PMD driver is bundled with APM-aware code in the core kernel. This code monitors the overall kernel activity. Whenever it detects that the system is inactive, except for running the kernel scheduler, it switches the CPU into a low-power state until the next interrupt or system clock occurs.
**APM Modules and Components**

The figure below shows the layout of the BlueCat Linux APM modules and its components:

![Diagram of APM Modules and Components](image)

**Figure 5-1: BlueCat Linux APM Modules and Components**

The core of the BlueCat Linux APM software is the APM core module. This is a kernel-space component that implements the APM state machine for each of the PMDs, and provides the APIs to the other APM components.

The APM driver implements an architecture-independent interface for client kernel-space device drivers. The kernel-space interface allows client device drivers to explicitly control the power management related aspects of hardware device functionality.

The callback interface is considered to be an extension of the kernel-space interface. Client drivers register their callback functions, which are called by the APM core module, before and after each APM event. This synchronization mechanism allows a client driver to react appropriately when the respective device changes its power state (for instance, as a reaction to an explicit user request via the `mapm_ctrl` utility).

The APM core module is closely bundled with a special kernel-space client called the APM user-space interface driver. The APM user-space interface driver
implements the IOCTL interface to the \texttt{mapmd} daemon and the \texttt{mapmd\_ctrl} control utility. The IOCTL interface is used by the \texttt{mapmd} daemon to access APM events, to allow reaction at the user-space level. The IOCTL interface is used by the \texttt{mapmd\_ctrl} control utility to explicitly control the APM and power-managed devices.

Another user interface is through the APM \texttt{proc} file (/\texttt{proc/mapm}). The APM core module maintains this file to relay APM status to the user space.

At the other end, the APM core module implements the PMD interface. This is used to call low-level PMD device drivers in an architecture-independent manner. PMD drivers implement device-specific management of individual power-managed devices.

**APM Event Queue**

The APM core module maintains the APM event queue, which is used to sequentially process APM events. Event processing occurs on a first-come-first-processed basis and does not commence until the previous event in the queue has been completely processed.

**Pre- and Post-Events**

As a general rule, there are two types of events:

- \textit{Pre-events} - events of this type are associated with a request to change the power state of a PMD. For instance, a client driver requests that the PMD it is servicing be switched to a lower power state.

- \textit{Post-events} - events of this type are associated with an action that has occurred at a PMD, for example, when the PMD has been switched to a lower power state.

Events are placed onto the queue by the APM core module:

- In reaction to the \textit{invocation of an API service} that requires placing an event in the queue (such as a request to change the power state of a PMD)

- As a result of \textit{processing an earlier event}, in particular, a post-event is placed in the queue provided that a pre-event has resulted in the successful change of the power state of a PMD.

- If a PMD driver reports a \textit{hardware-driven state transition} at the PMD

- As a result of an \textit{inactivity timer expiration}
Event Processing

The event queue is processed by a special event processing kernel task maintained by the APM core module. Events in the queue are processed one at a time. If there are no events in the queue, the task sleeps. The task is given such priority as to ensure that processing of APM events does not result in blocking interrupt handlers or other critical kernel tasks.

Event processing occurs in the following manner:

1. For each kernel client registered for processing events at the PMD associated with the event, the client callback is called. The client callback can take as long as required to react to the event.

2. Eventually the client callback has to return either a success or a failure.

3. If a success is returned for a pre-event, the client does not object to the power-state change at the PMD. If a failure is returned for a pre-event, the client indicates that the power-state transition cannot be executed.

4. A client callback must always return a success on a post-event. In other words, the callback return code is ignored by the event processing task for post-events.

5. All registered client callbacks are called sequentially. Once the last callback returns, event processing is complete for a post-event.

6. For a pre-event, processing continues as follows: If at least one of the callbacks has returned a failure for a pre-event, the event processing task fails to carry out the power state transition associated with the pre-event. In this case, the event processing is complete.

7. If all the clients have indicated agreement to the power state transition, the event processing task calls an appropriate function of the PMD device driver to carry out the power state transition. It then updates its internal tables to reflect the change at the PMD.

8. As a final step in event processing, an appropriate post-event is placed in the event queue.
Figure 5-2: Event Processing
For each PMD, the APM core module supports the following power states:

- **ON** - The PMD is completely on.
- **STOP** - The PMD is suspended, while the wakeup logic of the PMD is enabled. When a wakeup event occurs at the PMD, it automatically returns to the **ON** power state.
- **OFF** - The PMD is completely off. The wakeup logic is disabled.
- **AUTO** - The PMD is in a device-specific, architecture-dependent power state. This state is typically associated with various hardware-based autonomous power management capabilities, such as an autonomous hardware inactivity timer.
- **SPECIAL** - The PMD is in a device-specific architecture-dependent power state. This state is intended as a placeholder for custom power management features or future extensions.

For each PMD, the APM core module maintains the following machine state:

- While a PMD is in the **ON** state, a client can make a request to switch the PMD to the **STOP**, **OFF**, **SPECIAL**, or **AUTO** power states. The change is negotiated with all registered clients via the event callbacks mechanism, and is either rejected or agreed upon.
- While a PMD is in the **STOP**, **AUTO**, or **SPECIAL** state, it may automatically return to the **ON** state. For instance, the return from **STOP** to **ON** state occurs upon a wakeup event at the PMD. The PMD device driver may be able to detect a return to the **ON** state and report it to the APM core module. Such a notification is handled by the ARM core module by placing a post-event in the queue. As explained, no client negotiation occurs in this case.
- It is possible, however, that a PMD switches to the **ON** state from the **STOP**, **SPECIAL**, or **AUTO** states without the APM core module receiving any notification of such a transition. To cover this scenario, the APM core module allows clients to transition to the same set of power states from either of the **ON**, **STOP**, **SPECIAL**, or **AUTO** states.
Software Inactivity Timers

The APM core module maintains an optional inactivity timer for each PMD. From the point of view of the client interface, each request to perform an action at a PMD is accompanied by an inactivity period parameter. If the inactivity period is non-zero, the APM core module starts a software timer for the PMD. If the inactivity timer expires without clients having notified the APM core that the PMD has been active, the APM core starts the process of negotiating the power state transition.

If a notification arrives before the inactivity timer expires, the APM core resets the timer, and restarts the inactivity countdown.

While an inactivity timer can be associated with any client request, this feature is typically used for the conditional switch of a PMD into a low-power state. If a PMD is inactive for a specific period, the APM core switches it to a low-power state. To prevent this occurrence, clients must recurrently notify the APM core that the PMD is active.

Conceptually, the inactivity timer feature is similar to the AUTO power state. The key difference is that the AUTO state is implemented in the hardware in a device-specific, architecture-dependent manner, while the inactivity timer is a software concept implemented by the APM core module.

APM Interfaces

This section discusses the APIs defined by the BlueCat Linux APM software in greater detail. The following APIs are defined:

- Kernel-space client interface
- PMD device drivers interface
- IOCTL interface to the user space

Each of the above interfaces is implemented in a thread-safe, reentrant manner. Meaningful error codes are returned in case a requested operation cannot be carried out.

Kernel-Space Client Interface

This interface is from the APM core module to the kernel-space clients. Typically, a client is a device driver for a power management-aware device. In addition to implementing the core functionality of the I/O device, the device driver uses the APM interface to control the power management aspects of device functionality.
A client device driver registers itself with the APM core module. The client must supply a client name (character string), a pointer to a client private data structure and a callback function used by the APM core to asynchronously notify the client of various APM-related events.

If registration is successful, the client gets an opaque client handle used to further access the APM core.

As soon as a client is done using the APM software, it can unregister itself by using a special API service.

The client callback is called by the APM core any time there is an event that requires client attention or reaction. For each APM event, the callback is called with an event descriptor as a parameter. The event descriptor includes the identifier of the event, a handle to the PMD at which the event has occurred, as well as a pair of optional event-specific parameters.

Two special events are used to notify a client of the creation or removal of a PMD device driver (in other words, the existence of a PMD). These are needed to ensure that each client knows the PMDs present in the system.

A client can control one or more PMDs. For each one the client wishes to control, the client driver is given an opaque handle to the PMD. The PMD handle is available in the new PMD event descriptor, as described above.

A client uses the PMD handle to specify for the APM core module a set of events occurring at the PMD, of which the client wishes to be notified. The set can include just one event (for instance, a pre-event for a request to switch the device to the STOP state), or all events that may occur at the PMD, or any subset of events.

Once the set of events is specified, the APM core calls the client callback for any event in the set. Initially (upon registration), the set of events is empty for each PMD reported to a client.

As described above, for each event, the callback is called with the event descriptor as a parameter. A callback uses the event identifier to distinguish different events. A callback uses the handle to determine the PMD at which the event has occurred.

For pre-events, a callback can either reject the action requested by the event or agree that the action can be carried out by the APM core. In either case, an appropriate return code must be returned by the callback.

For post-events, a callback must always return a success.

The client API defines a service for client drivers to request a change of the power state at a PMD. The PMD handle is used to specify a particular PMD. A state identifier is passed to the service, along with a pair of optional state-specific parameters.
parameters to specify the state to which a switch is suggested. If a non-zero inactivity timer period is passed as a parameter, the request to change the power state is conditional, depending on the activity of the PMD.

The clients use a special API service to notify the APM core that a particular PMD has been active.

The API defines a special service allowing clients to retrieve the current state of a PMD.

PMD Device Drivers Interface

This is an open, platform-independent interface with low-level PMD device drivers that implement device-specific control of individual PMDs.

A PMD uses a registration service to register itself with the APM core. The registration service must be given the name of the PMD (character string), a pointer to a PMD device driver private data structure, and a callback function used by the APM core to request specific actions at the PMD.

If the registration is successful, the PMD is given an opaque PMD handle, which is used for further interactions with the APM core.

A PMD device driver can unregister itself at the APM core at any time.

The APM core calls a PMD callback to trigger a specific action at the PMD. The callback is given an action identifier, as well as a pair of action-specific parameters. The action identifier is used by the PMD driver to switch on a specific action requested by the APM core.

A PMD device driver uses an API service to notify the APM core that the PMD has changed its power state. The APM core handles such notification by placing an appropriate post-event in the event queue. Another API service is used to notify the APM core that the PMD has been active.

In an implementation note, a PMD driver does not need to be in a separate device driver, or even reside in a separate source file from the client device driver for the power-managed device. It is quite possible that a single device driver includes the core I/O device code, the APM client code, and the PMD device driver. The developer is free to choose the approach to code modularization conducive to the overall system structure and/or the ultimate application at hand.
IOCTL Interface to User Space

The IOCTL interface is implemented by a special kernel-space client called the APM user-space interface driver. This driver propagates any APM events to user-space programs via a special `ioctl()` call. The user-space program interacting with the driver blocks on the `ioctl()` call until an event is available. The APM daemon (`mapmd`) is intended as the only user-space program that interacts with the APM user-space interface driver.

Another IOCTL is used to propagate requests from the user space to the APM core. Logically, the same set of requests is supported for a user-space as for a kernel-space client. This IOCTL is used by the APM control utility (`mapm_ctrl`).

User-Space Components

This section discusses the user-space components of the BlueCat Linux APM software in greater detail. The following user-space components are supported:

- APM daemon (`mapmd`)
- APM control utility (`mapm_ctrl`)
- APM proc file (`/proc/mapm`)

APM Daemon

The APM daemon (`mapmd`) defines how the APM software reacts to particular APM events at the user level. The APM daemon is user-configurable. User configuration is read by the daemon at startup from the `/etc/mapmd.conf` configuration file. Refer to the `mapmd.conf (8)` manual page for the format of the configuration file.

One line of the configuration file defines the reaction of the APM for events from a particular PMD. APM reaction to an event can be any of the following:

- A call to a user-space event handling program
- Starting a script for printing an event-describing message on the system console
- Placing a record in the log file
- Any combination of the above

See the section “mapmd” in Appendix A, “Command Reference" for details.
The APM daemon interacts with the APM core module via the APM user-space interface driver. The core algorithm implemented by the APM daemon does the following:

1. Blocks on the event receiving IOCTL
2. As soon as an event is available, processes it as defined by the user configuration file
3. Returns to Step 1

**APM Control Utility**

The APM control (mapm_ctrl) utility allows the user to send explicit requests to the APM core module to perform a specific action at an individual PMD. The APM control utility interacts with the APM core via the APM user-space interface driver. For details, see “mapm_ctrl” in Appendix A, “Command Reference.”

**APM proc File**

The APM proc file (/proc/mapm) provides the user with an easy way to get the current APM status. The user can read the file at any time. The file contains current status data per PMD.

The APM core module maintains the /proc/mapm file from the kernel-space context.

**CPU Power Management**

From the point of view of BlueCat Linux APM software architecture, the CPU is just another PMD. As such, it is served by an architecture-dependent PMD device driver that controls low-level aspects of CPU power management.

An important feature of CPU power management, as supported by the BlueCat Linux APM software, is the ability of the system to suspend the CPU in case the kernel is only running the kernel scheduler. In BlueCat Linux this concept is implemented as follows:

1. To enable the CPU suspend mode, a request to switch the CPU to the AUTO state is issued to the APM core module (for instance, using the mapm_ctrl utility). This request is negotiated with registered clients and, provided the change is agreed upon, comes to the CPU PMD driver.
2. When given a request to switch to the AUTO state, the CPU PMD driver enables a global variable mapm_cpu_suspendable, which is used in the kernel null process (also known as process 0) to determine whether it can switch the CPU into a suspend state.

3. Whenever the null process is entered and mapm_cpu_suspendable is on, the null process directly calls an appropriate service in the CPU PMD device driver to switch the CPU to the suspend state. This service is architecture-specific and must be implemented so as to ensure that the CPU returns to the ON state at any interrupt (a system clock tick, etc.).

### Configuring APM in the Kernel

The kernel part of the APM software is controlled by a number of kernel configuration options. These options determine and control the APM software components included in the kernel. This enables the user to remove undesired functionality, thus saving on kernel code size and runtime memory requirements. When disabled, APM has no impact on kernel size.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIG_BLUECAT_APM</td>
<td>This option controls the presence of the APM core module in the kernel. Disabling it automatically turns off all other APM components.</td>
</tr>
<tr>
<td>CONFIG_BLUECAT_APM_USER</td>
<td>This option controls the presence of the APM user-space interface driver in the kernel. It can be set to m indicating that the driver is compiled as a runtime kernel module.</td>
</tr>
<tr>
<td>CONFIG_BLUECAT_APM_TEST</td>
<td>A test PMD device driver</td>
</tr>
<tr>
<td>CONFIG_BLUECAT_APM_CPU</td>
<td>A CPU PMD skeleton device driver</td>
</tr>
</tbody>
</table>

**NOTE:** The APM core uses the standard kernel option CONFIG_PROC_FS (kernel support for the /proc file system) to enable or disable /proc/mapm interface support.
APM Interfaces Reference

This section lists the common constants and data types used in BlueCat Linux APM. These are defined in the following header file:

$BLUECAT_PREFIX/usr/include/linux/mapm.h

Common Constants and Data Types

PMD Handles

A PMD is identified by an opaque handle of type `apm_pmd_handle_t`. The handle is provided to the PMD device drivers as a result of successful registration, as well as to clients, as a part of the event descriptor.

Error Codes

Each APM service returns a value of type `apm_error_t`. The possible return codes are shown in the following table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APM_SUCCESS</td>
<td>Service successful</td>
</tr>
<tr>
<td>APM_ERROR_INVAL</td>
<td>Invalid argument</td>
</tr>
<tr>
<td>APM_ERROR_REJECT</td>
<td>Request rejected</td>
</tr>
<tr>
<td>APM_ERROR_RETRY</td>
<td>Out of resources</td>
</tr>
<tr>
<td>APM_ERROR_SYSTEM</td>
<td>OS error</td>
</tr>
<tr>
<td>APM_ERROR_NOTIMP</td>
<td>Not implemented</td>
</tr>
<tr>
<td>APM_ERROR_GENERAL</td>
<td>General error</td>
</tr>
<tr>
<td>APM_ERROR_NOPMD</td>
<td>No PMD with such handle</td>
</tr>
<tr>
<td>APM_ERROR_PMD</td>
<td>PMD internal error</td>
</tr>
<tr>
<td>APM_ERROR_NOCLIENT</td>
<td>No client with such handle</td>
</tr>
<tr>
<td>APM_ERROR_BUSY</td>
<td>Resource busy</td>
</tr>
<tr>
<td>APM_ERROR_MEM</td>
<td>Out of memory</td>
</tr>
</tbody>
</table>
PMD State Codes

A PMD state is represented by the code type `apm_state_t`. The possible PMD states are listed in the table below:

### Table 5-3: PMD States

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APM_STATE_ON</td>
<td>On state</td>
</tr>
<tr>
<td>APM_STATE_OFF</td>
<td>OFF state</td>
</tr>
<tr>
<td>APM_STATE_STOP</td>
<td>STOP state</td>
</tr>
<tr>
<td>APM_STATE_AUTO</td>
<td>AUTO state</td>
</tr>
<tr>
<td>APM_STATE_SPECIAL</td>
<td>SPECIAL state</td>
</tr>
</tbody>
</table>

Event Code Tables

A PMD event is identified by the code type `apm_event_code_t`. The possible event codes are shown in the following tables:

### Table 5-4: Event Codes

<table>
<thead>
<tr>
<th>Pre-Event Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APM_EVENT_REQUEST_ON</td>
<td>Request to switch to ON state</td>
</tr>
<tr>
<td>APM_EVENT_REQUEST_OFF</td>
<td>Request to switch to OFF state</td>
</tr>
<tr>
<td>APM_EVENT_REQUEST_STOP</td>
<td>Request to switch to STOP state</td>
</tr>
<tr>
<td>APM_EVENT_REQUEST_AUTO</td>
<td>Request to switch to AUTO state</td>
</tr>
<tr>
<td>APM_EVENT_REQUEST_SPECIAL</td>
<td>Request to switch to SPECIAL state</td>
</tr>
</tbody>
</table>
Kernel-Mode API

Client Handles
Clients are identified by client handles obtained by clients at the time of registration. Client drivers supply the client handle to each call to APM client services. The client handle is defined as opaque of the type `apm_client_handle_t`.

Client Callback
Clients receive event notifications using the callback function specified at the time of registration. The callback function type is defined as follows:

```c
typedef apm_error_t (*apm_client_callback_t)(apm_client_handle_t handle, const apm_event_t * event, void * user);
```

Table 5-5: Post-Event Codes

<table>
<thead>
<tr>
<th>Post-Event Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APM_EVENT_ON</td>
<td>Transition to ON state</td>
</tr>
<tr>
<td>APM_EVENT_OFF</td>
<td>Transition to OFF state</td>
</tr>
<tr>
<td>APM_EVENT_STOP</td>
<td>Transition to STOP state</td>
</tr>
<tr>
<td>APM_EVENT_AUTO</td>
<td>Transition to AUTO state</td>
</tr>
<tr>
<td>APM_EVENT_SPECIAL</td>
<td>Transition to SPECIAL state</td>
</tr>
</tbody>
</table>

Table 5-6: PMD Driver Event Codes

<table>
<thead>
<tr>
<th>PMD Driver Event Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APM_EVENT_NEW</td>
<td>New PMD driver registers</td>
</tr>
<tr>
<td>APM_EVENT_DELETE</td>
<td>Existing PMD driver unregisters</td>
</tr>
</tbody>
</table>
When APM calls the client callback, it provides the following parameters to the client callback:

- handle - Client handle
- event - Event descriptor
- user - Pointer to a client’s private data provided by the client at the time of registration

Client Services
One of the client services registers a new client specified by the info structure. On success, the new client handle is stored in the location pointed to by the handle argument. The \textit{apm\_client\_info\_t} structure is defined as follows:

\begin{verbatim}
#include <apmdefs.h>

typedef struct apm_client_info_t
{
    aprm_client_callback_t callback;
    char * name;
} apm_client_info_t;

apm_error_t apm_client_register (const apm_client_info_t * info, apm_client_handle_t * handle, void * user);

apm_error_t apm_client_unregister (apm_client_handle_t handle);

apm_error_t apm_client_event_mask (apm_client_handle_t handle, apm_pmd_handle_t pmd, unsigned int mask);

apm_error_t apm_client_state_transition (apm_client_handle_t handle, apm_pmd_handle_t pmd, aprm_arg_i arg_i, aprm_arg_p arg_p);

apm_error_t apm_client_set_timeouts (apm_client_handle_t handle, aprm_timeout_callback_t timer, aprm_client_timeout_t timeout);

\end{verbatim}

APM unregisters the client specified by the handle.

APM sets the event mask. The client receives event notifications from the specified PMD as defined by the \textit{mask} argument. The mask is bitwise or composed of event codes (APM\_EVENT\_*).

\begin{verbatim}
#include <apmdefs.h>

apm_error_t apm_client_event_mask (apm_client_handle_t handle, apm_pmd_handle_t pmd, unsigned int mask);

\end{verbatim}

APM makes a request for a state transition. The PMD specified by the \textit{pmd} argument is requested to switch to the state, \textit{state}. The state-specific parameters are specified by the \textit{arg\_i} and \textit{arg\_p} parameters. The request is negotiated with other clients registered to receive notifications of the state transitions at the PMD. If the \textit{timer} parameter is not NULL, the inactivity timer is primed for this PMD.
apm_error_t apm_client_request(
    apm_client_handle_t handle,
    apm_pmd_handle_t pmd,
    apm_state_t state,
    unsigned int arg_i,
    void * arg_p,
    struct timespec * timer);

Client services makes a raw request for a state transition. The PMD specified by the pmd argument is requested to switch to the state, state. The state-specific parameters are specified by the arg_i and arg_p parameters. No request negotiations or queuing takes place. The request is passed immediately to the PMD driver layer.

apm_error_t apm_client_raw_request(
    apm_client_handle_t handle,
    apm_pmd_handle_t pmd
    apm_state_t state,
    unsigned int arg_i,
    void * arg_p );

APM gets information about a PMD specified by the pmd parameter. On success, the service returns the PMD name, current PMD state and state-specific parameters in the locations pointed to by the name, state, arg_i and arg_p arguments, respectively.

apm_error_t apm_client_pmd_info(
    apm_client_handle_t handle,
    apm_pmd_handle_t pmd,
    char ** name,
    apm_state_t * state,
    unsigned int * arg_i,
    void ** arg_p );

The following indicates that the PMD is currently active and is being used by the client. A call to this service restarts the inactivity timer for this PMD, if any.

apm_error_t apm_client_pmd_active
( apm_client_handle_t handle,
  apm_pmd_handle_t pmd );

The following returns the pointer to a string that names the specified error code. This is intended to be used for composing informational and error messages.

char * apm_error_name( apm_error_t error );
The following returns the pointer to a string that names the specified PMD state. This is intended to be used for composing informational and error messages.

```c
char * apm_state_name( apm_state_t state);
```

The following gets general information on the APM core. This is intended to be used to detect the presence of the APM core software. The version returned is composed of the major version number in the most significant byte and the minor version in the least significant byte.

```c
apm_error_t apm_info_get( char ** name,
                       unsigned short * version );
```

**PMD API**

**PMD Callback**

APM makes requests to the PMD driver using a callback provided by the PMD driver at the time of registration. The callback type is defined as follows:

```c
typedef apm_error_t (*apm_pmd_callback_t)
    (apm_pmd_handle_t handle,
     apm_pmd_request_t request,
     unsigned int arg_i,
     void * arg_p,
     void * user );
```

When APM calls the PMD callback, it provides the following parameters:

- `handle` - PMD handle
- `request` - Request code
- `arg_i` - Request-specific integer argument
- `arg_p` - Request-specific pointer argument
- `user` - Pointer to PMD private data provided at PMD registration time
PMD Requests

The requests made by the APM core to the PMD driver are identified by the request code value of type `apm_pmd_request_t`. The possible PMD request code values are shown in the table below:

**Table 5-7: Possible PMD Request Code Values**

<table>
<thead>
<tr>
<th>Request Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APM_PMD_REQUEST_ON</td>
<td>Switch PMD to ON state</td>
</tr>
<tr>
<td>APM_PMD_REQUEST_OFF</td>
<td>Switch PMD to OFF state</td>
</tr>
<tr>
<td>APM_PMD_REQUEST_STOP</td>
<td>Switch PMD to STOP state</td>
</tr>
<tr>
<td>APM_PMD_REQUEST_AUTO</td>
<td>Switch PMD to AUTO state</td>
</tr>
<tr>
<td>APM_PMD_REQUEST_STATE</td>
<td>Report current device state</td>
</tr>
<tr>
<td>APM_PMD_REQUEST_SPECIAL</td>
<td>Switch to the SPECIAL state</td>
</tr>
</tbody>
</table>

PMD Services

PMD services register a new PMD driver specified by the `info` parameter. On success, the new PMD handle is stored in the location pointed to by the `handle` argument. The `apm_pmd_info_t` structure is defined as follows:

```c
typedef struct apm_pmd_info_s
{
    char * name;
    apm_pmd_callback_t callback;
} apm_pmd_info_t;
```

The following unregisters the PMD driver specified by the `handle` argument.

```c
apm_error_t apm_pmd_unregister(apm_pmd_handle_t handle);
```

The following indicates that the PMD is active and is being used by someone.

```c
apm_error_t apm_pmd_active(apm_pmd_handle_t handle);
```
The following reports that a hardware-driven state transition has occurred at the PMD. The new PMD state is specified by the `state` parameter.

```c
apm_error_t apm_pmd_change( apm_pmd_handle_t handle,
                            apm_state_t state,
                            unsigned int arg_i,
                            void * arg_p);
```

**IOCTL Commands**

**APM_IOCTL_GET_EVENT**

This IOCTL command blocks until an event occurs. The event code and its associated PMD handle is returned in the `apm_ioctl_event_t` structure pointed to by the `arg` parameter to the `ioctl()` call. The `apm_ioctl_event_t` structure is defined as follows:

```c
typedef struct apm_ioctl_event_s
{
    apm_pmd_handle_t pmd;
    apm_event_code_t code;
    unsigned int arg_i;
    void * apg_p;
    char arg_buf[APM_MAX_ARG]
} apm_ioctl_event_t;
```

**APM_IOCTL_REQUEST**

This IOCTL command makes a state transition request. The target board state and state-specific parameters are passed in the `apm_ioctl_request_t` structure pointed to by the `arg` parameter to the `ioctl()` call. The `apm_ioctl_request_t` structure is defined as follows.

```c
typedef struct apm_ioctl_request_s
{
    apm_pmd_handle_t pmd;
    apm_state_t request;
    unsigned int arg_i;
    void * arg_p;
    struct timespec timer;
} apm_ioctl_request_t;
```
User-Mode Interfaces

/proc/mapm Directory

The /proc/mapm directory has a number of files. A read from each of the files yields text describing the current state of a particular APM software component.

The /proc/mapm/pmds file lists all the PMD drivers registered. Each line in the file corresponds to a PMD driver. PMDs are listed and sorted by the handle value. The format is as follows:

```
<handle> [ <name> ] <state> <timer> <client> <event>
```

- `<handle>` - Hexadecimal value of the PMD handle
- `<name>` - Name provided by the PMD at registration
- `<state>` - The current PMD state, one of the following: ON, OFF, STOP, AUTO or SPECIAL.
- `<timer>` - The value of the inactivity timer, if set for the PMD; otherwise this field is 0.
- `<client>` - If an inactivity timer has been set for the PMD, this is the hexadecimal value of the client handle that has primed the timer; otherwise, this field is 0.
- `<event>` - If an inactivity timer has been set for the PMD, this is the target board state to make the transition to timer expiration. It is one of the following: ON, OFF, STOP, AUTO, SPECIAL.

For example, a CPU PMD in the ON state, with an inactivity timer primed to set the CPU into AUTO state in 1 second by the client with handle 2, is described by the following line:

```
0  [CPU]  ON  100  2  AUTO
```

The /proc/mapm/clients file lists all registered clients. Each line in the file corresponds to a client. Clients are listed and sorted by the handle value. The format is:

```
<handle> [ <name> ]
```

- `<handle>` - Hexadecimal value of the client handle
- `<name>` - Name provided by the client at registration
For example, the APM user-space interface driver with handle 0 is described by the following line:

```
0 [ioctl]
```

**mapmd Command Line Format**

The `mapmd` command line format is as follows:

```
mapmd [-f <file>]
```

where:

- `-f <file>` Specifies an alternate configuration file for `mapmd` operation - if not specified, `mapmd` reads its configuration from the `/etc/mapmd.conf` file.

For details, please refer to “mapmd” in Appendix A, “Command Reference.”

**mapmd Configuration File**

The `mapmd` configuration defines how `mapmd` reacts to particular `mapmd` events. The settings are defined separately for each PMD. Each line of the configuration file corresponds to settings for one PMD.

The format of a line is as follows:

```
\[<PMD_handle_or_name>\] <command>[,<command>...]<PMD_handle_or_name> Specifies the PMD name or PMD handle (a hexadecimal number) of the PMD.
<command>[,<command>...] The list of programs to be executed in response to the PMD state transition
```

Lines starting with “#” are considered comments, to be ignored by `mapmd`.

`mapmd` can be forced to re-read the configuration by sending the HUP signal to the `mapmd` process.

For details, please refer to “mapmd.conf” in Appendix A, “Command Reference.”

**Handler Program Command Line Format**

When `mapmd` calls program in response to an `mapmd` event, it provides command line parameters in the following format:
user_program <PMD_handle_or_name> <state> <arg>[,<arg>...]

<PMD_handle_or_name> Specifies the PMD (in the same form as the configuration file for this PMD).

<state> Specifies the state the PMD has been switched to, and is one of ON, OFF, AUTO, STOP, or SPECIAL.

<arg>[,<arg>...] State-specific arguments are provided in these parameters.

Event Logging

mapmd is accompanied by two shell scripts, /etc/mapmd_log.sh and /etc/mapmd_msg.sh. These output a log message describing an APM event to the /etc/mapm.log and the system console, respectively. A user can specify these scripts in the command list of the mapmd configuration file in order to output information about APM events.

mapmd Operation

When started, mapmd reads the configuration file, becomes a daemon program, and operates according to the configuration file using the IOCTL and /proc/mapm interfaces for interactions with the APM kernel-space software.

APM Control Utility

APM Control Utility Command Line Format

The APM control utility command line format is as follows:

mapm_ctrl [ <PMD_handle_or_name> <state> [ <timer> | [ <arg>[,<arg>...] ] ] ]

When called without any parameters, the utility outputs the APM status information.

With parameters, the utility places a request to switch the specified PMD to the specified state. The state parameter can be one of the following: ON, OFF, STOP, AUTO, or SPECIAL. State-specific arguments are provided in the arg parameters. If the timer parameter is specified, the inactivity timer is primed for the PMD.
mapm_ctrl Operation

mapm_ctrl utilizes the APM IOCTL and /proc/mapm interfaces for interactions with the APM kernel-space software. For details, please refer to “mapm_ctrl” in Appendix A, “Command Reference.”

Developing APM Drivers

This section provides some sample code that can be used in an APM client and a PMD device driver.

Sample APM Client

This section shows the skeleton of a sample APM client device driver.

Registering an APM Client

The following sample code shows the registration of an APM client device driver:

```c
/* Client driver initialization. */
void client_init( void )
{
    apm_client_info_t client_info;
    ...

    /* Register client. */
    client_info.name = "Sample APM Client Driver";
    client_info.callback = client_callback;
    if ( apm_client_register( &client_info, &handle, NULL ) != APM_SUCCESS )
    {
        /* Registration failed */
        ...
    }
}
```

Deregistering an APM Client

The following sample code shows how an APM client can complete its operation:

```c
/* Client driver deinitialization. */
void client_term( void )
{
    ...

    /* We do not use the device anymore.
    * If nobody is using it, turn the device off.
    */
```
/* Unregister from the APM core. */
apm_client_unregister( handle );
```

}  

Processing APM Events

The following sample code shows the possible implementation of an APM client callback:

```c
/* Client callback to process PMD events. */
apm_error_t client_callback( apm_client_handle_t handle,
const apm_event_t * event,
void * user )
{
  switch( event->code )
  {  
    case APM_EVENT_NEW:
      /* Is it the PMD for the device we are working with? */
      if ( event->pmd == CLIENT_OUR_PMD )
      {
        apm_state_t state;
        char * name;
        /* Register to receive all event notifications from the PMD. */
        apm_client_event_mask( handle, event->pmd, APM_EVENT_ALL );
        /* Check the device state. */
        apm_client_pmd_info( handle, event->pmd, &name,
                          &state, NULL, NULL );
        /* If the device is in low-power state, enable it prior
        * to performing device initialization. In this case,
        * the actual initialization is delayed until
        * the device comes up and we get the
        * state change event. */
        if ( state != APM_STATE_ON )
          
        apm_client_request( handle, CLIENT_OUR_PMD, APM_STATE_ON,
                          0, NULL, NULL );
      }
      else /* The device is on, initialize the device. */
      {
        ...
        client_hardware_specific_init( ... );
        ...
      }
      break;
    case APM_EVENT_ON:
      /* The device has been turned on. Initialize the device. */
      ...
      client_hardware_specific_init( ... );
      ...
      break;
    case APM_EVENT_REQUEST_OFF:
      /* The device is being removed. */
      /* Do something */
    break;
    }
```
case APM_EVENT_REQUEST_STOP:
    case APM_EVENT_REQUEST_AUTO:
        /* We are using the device. Reject all low-power states. */
        return APM_ERROR_REJECT;
    break;
default:
    }

    return APM_SUCCESS;

 SAMPLE PMD DRIVER

This section shows the skeleton of a sample PMD device driver.

Registering a PMD Driver

The following sample code shows the registration of a PMD device driver:

    /* Driver initialization. */
    void pmd_init( void )
    {
        apm_pmd_info_t pmd_info;
        ...
    }
    /* Register PMD. */
    pmd_info.name = "Sample APM PMD driver";
    pmd_info.callback = pmd_callback;
    if ( apm_pmd_register( &pmd_info, &handle, NULL )
        != APM_SUCCESS)
        /* Registration failed */
        ...
    }

Deregistering a PMD Driver

The following sample code shows how a PMD driver can complete its operation:

    /* Driver deinitialization. */
    void pmd_term( void )
    {
        apm_pmd_unregister( handle );
    }
Processing Requests in a PMD Driver

The following sample code shows how a PMD driver processes APM requests from the APM core:

```c
apm_error_t pmd_callback(apm_pmd_handle_t handle, apm_pmd_request_t request, unsigned int arg_i, void * arg_p, void * user )
{
    switch( request )
    {
    case APM_PMD_REQUEST_ON:
        /* Perform hardware-specific device power-on. */
        ... break;
    case APM_PMD_REQUEST_OFF:
        /* Perform hardware-specific device power-off. */
        ... break;
    default: /* No other states supported by this PMD. */
        return APM_ERROR_NOTIMP;
    }

    return APM_SUCCESS;
}
```
CHAPTER 6  Flash Support and Journalling
Flash File System

This chapter provides a detailed description of flash memory support and the Journalling Flash File System (JFFS) in BlueCat Linux. The JFFS is based on the JFFS initially developed by Axis Communications, which is now present in the 2.4.2 Linux kernel distributions.

Flash Support and JFFS 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 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 6-1: 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/mtdcharx 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 134.

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 as a Journalling Flash File System (JFFS).

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/mtdblockx` 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) Interface**

The Journalling Flash File System (FFS) is a POSIX-compliant file system implemented on flash memory. JFFS 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 or its files and directories goes through the VFS, which directs appropriate tasks to the JFFS layer.

JFFS 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 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 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/mtdblockx) 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, `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, `mtdchar`, or `mtdblock`. 
An interesting illustration of this general concept is the support of flash memory devices composed of sectors of non-uniform 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 in a single flash memory device, or use particular sectors of flash memory as a JFFS 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 via a block device
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 numbers 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 Layout

All data physically stored in JFFS is divided into chunks. Each chunk starts with a control structure called `raw_inode`:

```c
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 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 file name and a piece of file data, according to the values of the `nsize` and `dsize` members of the above structure, respectively. The file name 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 file name and the file data may not be present in a node. There is no need for the file name 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 file name, 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 file name 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, etc.), 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
- **nsise** - 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:

```c
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:

```
aaaaacaaaaaabbaaaaaa
```

where each lower-case character represents 50 of the equivalent uppercase characters. The table below 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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
<th>Flash Memory Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>raw_inode created during open()</td>
<td>Relevant fields are set as follows:</td>
</tr>
<tr>
<td></td>
<td>offset = 0</td>
<td>nsize = 4 (name size)</td>
</tr>
<tr>
<td></td>
<td>dsize = 0</td>
<td>rsize = 0 (removed size)</td>
</tr>
<tr>
<td></td>
<td>version = 1</td>
<td>test - file name</td>
</tr>
<tr>
<td>0x3c</td>
<td>raw_inode created during first write()</td>
<td>Relevant fields are set as follows:</td>
</tr>
<tr>
<td></td>
<td>offset = 0</td>
<td>nsize = 0</td>
</tr>
<tr>
<td></td>
<td>dsize = 1000</td>
<td>rsize = 0</td>
</tr>
<tr>
<td></td>
<td>version = 2</td>
<td>AAA....AAA</td>
</tr>
<tr>
<td>0x40</td>
<td>raw_inode created during second write()</td>
<td>Relevant fields are set as follows:</td>
</tr>
<tr>
<td></td>
<td>offset = 600</td>
<td>nsize = 0</td>
</tr>
<tr>
<td></td>
<td>dsize = 100</td>
<td>rsize = 100</td>
</tr>
<tr>
<td></td>
<td>version = 3</td>
<td>BBB....BBB</td>
</tr>
<tr>
<td>0x7c</td>
<td>raw_inode created during third write()</td>
<td>Relevant fields are set as follows:</td>
</tr>
<tr>
<td></td>
<td>offset = 200</td>
<td>nsize = 0</td>
</tr>
<tr>
<td></td>
<td>dsize = 50</td>
<td>rsize = 50</td>
</tr>
<tr>
<td></td>
<td>version = 4</td>
<td>CCC....CCC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dsize = 0</td>
</tr>
<tr>
<td></td>
<td>flash_offset = 0x40</td>
</tr>
<tr>
<td>2</td>
<td>dsize = 200</td>
</tr>
<tr>
<td></td>
<td>flash_offset = 0x7c</td>
</tr>
<tr>
<td>3</td>
<td>dsize = 50</td>
</tr>
<tr>
<td></td>
<td>flash_offset = 0x540</td>
</tr>
<tr>
<td>4</td>
<td>dsize = 350</td>
</tr>
<tr>
<td></td>
<td>flash_offset = 0x7c + 250 = 0x176</td>
</tr>
<tr>
<td>5</td>
<td>dsize = 100</td>
</tr>
<tr>
<td></td>
<td>flash_offset = 0x4a0</td>
</tr>
<tr>
<td>6</td>
<td>dsize = 300</td>
</tr>
<tr>
<td></td>
<td>flash_offset = 0x7c + 700 = 0x338</td>
</tr>
</tbody>
</table>
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 file name 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 multi-node `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 file name 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 6-2: Data Pointers](image)

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 multi-threaded 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 re-initialization 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:

```c
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 ecc_type; /* 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:

```c
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:
Chapter 6 - Flash Support and Journalling Flash File System

```c
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:

```c
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:

```c
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 should 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:

```c
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:

```c
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.
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:

   ```c
   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:

```c
struct mtd_info {
    char name[32];
    u_char type;
}
```

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.

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
- MTDUNKNOWN   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 non-zero 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. \texttt{oobsize} and \texttt{oobblock} hold the size of each out-of-band area, and the number of bytes of real memory which each is associated, respectively. For example, NAND flash memory has \texttt{oobblock == 512} and \texttt{oobsize == 16} for 16 bytes of OOB data per 512 bytes of flash memory.

\begin{verbatim}
  u_long oobblock;
  u_long oobsize;
\end{verbatim}

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:

\begin{verbatim}
  u_long ecctype;
\end{verbatim}

The \texttt{ecctype} field is an enumeration and can be one of the following:

- \texttt{MTD_ECC_NONE} No automatic ECC available
- \texttt{MTD_ECC_RS_DiskOnChip} Automatic ECC on DiskOnChip

\texttt{eccsize} holds the size of the blocks on which the hardware can perform automatic ECC.

\begin{verbatim}
  u_long eccsize;
\end{verbatim}

When a driver is a kernel-loadable module, this field is a pointer to the \texttt{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 \texttt{__MOD_INC_USE_COUNT (mtd->module)} in the open routine, for example.

\begin{verbatim}
  struct module *module;
\end{verbatim}

The following routine adds \texttt{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). \texttt{struct erase_info} contains a pointer to a callback function, which is called by the MTD driver when the erase is complete.

\begin{verbatim}
  int (*erase) (struct mtd_info *mtd,
                 struct erase_info *instr);
\end{verbatim}

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.

```c
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).

```c
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.

```c
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:

```c
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.

```c
void (*sync) (struct mtd_info *mtd);
```

The following is used as a pointer to data that is private to the MTD driver:

```c
void *priv;
}; /* end of struct mtd_info */
```

The starting offset of the region relative to the beginning of the partition:

```c
struct mtd_flash_region
{
    loff_t start_offset;
}
```

The size of the region:

```c
loff_t size;
```

The size of sectors within the region:

```c
u_long erasesize;
```

The number of sectors within the region:

```c
int n_sectors;
```

A pointer to the next region:

```c
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:

```c
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.
Chapter 6 - Flash Support and Journalling Flash File System

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 re-established 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 154 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

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.

Once a JFFS 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.

```c
/* 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 */
```
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))
{
Deregistering the MTD Driver

The following sample code shows how an MTD driver can complete its operation:

```c
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.

```c
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;
}
```
This appendix includes man pages describing 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 re-established 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
```
BlueCat Linux Advanced Power Management (APM) control utility

mapm_ctrl [<PMD_handle_or_name> <state> |<timer> <arg>,<arg>...]]

Description
The APM control utility allows the user to send explicit requests to perform a particular action at an individual PMD. When executed with no parameters, mapm_ctrl shows the current APM status.

Parameters
- <state> Can be one of: ON, OFF, STOP, AUTO, SPECIAL
- <timer> If specified, the inactivity timer is primed for the PMD; the timer is specified in milliseconds.
- <arg> State-specific arguments

Examples
To switch the PMD # 0 to the ON state, use the following command:

bash# mapm_ctrl 0 ON

To switch the PMD named HDD to the STOP state after a one-minute inactivity period, use the following command:

bash# mapm_ctrl HDD STOP 60000

To switch the PMD named CPU to the SPECIAL state with an additional argument of 255, use the following command:

bash# mapm_ctrl CPU SPECIAL 0 255
mapmd

BlueCat Linux Advanced Power Management (APM) daemon

mapmd [options]

Description

The MAPM daemon defines how the BlueCat Linux APM software reacts to particular APM events at the user level. By default, the user configuration is read by the daemon at startup from the /etc/mapmd.conf configuration file.

Options

-f <file> Specifies alternate configuration file for mapmd operation.

See also the man page for the mapmd configuration file: mapmd.conf(1)

mapmd.conf

BlueCat Linux Advanced Power Management (APM) configuration file

Description

mapmd.conf is a configuration file for the mapmd daemon. Each line describes a configuration of a particular Power Managed Device (PMD). Lines starting with “#” are considered to be comments and are ignored by mapmd.

The format of a configuration line is as follows:

[<PMD_handle_or_PMD_name>] <command>[,<command>...]

The first word of a line specifies a PMD name or a PMD handle (as a hexadecimal number). The following comma-separated words specify the list of programs that are called in response to an APM event.

mapmd can be forced to re-read the configuration by sending the HUP signal to the mapmd process.
Example
When switching a PMD named HDD from one state to another, an event-describing message is printed onto the system console, and a record is added to the log file:

[HDD] /etc/mapm_log.sh, /etc/mapm_msg.sh

mkboot

Creates a bootable disk (floppy, hard disk or an image) with BlueCat Linux embedded system.

mkboot [options] device|stdout

Description
The mkboot utility is capable of performing the following tasks:

- Installing a BlueCat Linux boot sector
- Installing a compressed BlueCat Linux kernel
- Installing a 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 for programming into target ROM/flash memory or downloading over a network by the target board firmware

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, re-installing the boot sector invalidates all the booting options set by previous calls to mkboot for this disk.
Appendix A - Command Reference

- **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.

- **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*`.

- **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.
Examples

To copy the `showcase` demo system onto a floppy:

- On the Linux host:
  ```bash
  BlueCat:$ cd $BLUECAT_PREFIX/demo/showcase
  BlueCat:$ mkboot -b -k showcase.disk -f showcase.rfs
       -r /dev/fd0 /dev/fd0
  ```

- On a Windows host:
  ```bash
  BlueCat:$ cd $BLUECAT_PREFIX/demo/showcase
  BlueCat:$ mkboot -b -k showcase.disk -f showcase.rfs
       -r /dev/fd0 a:
  ```

To copy the `showcase` demo system onto an IDE hard disk for x86 target board:

```bash
BlueCat:$ cd $BLUECAT_PREFIX/demo/showcase
BlueCat:$ mkboot -b -k showcase.disk -f showcase.rfs -r
       /dev/hda /dev/hda
```

To copy the `showcase` demo system on a floppy and set the kernel command line, use one of the following sequences:

```bash
BlueCat:$ cd $BLUECAT_PREFIX/demo/showcase
BlueCat:$ echo console=441 >cl.txt
BlueCat:$ echo mem=4M >>cl.txt
BlueCat:$ mkboot -b /dev/fd0
BlueCat:$ mkboot -k showcase.disk -f showcase.rfs -r
       /dev/fd0 /dev/fd0
BlueCat:$ mkboot -c cl.txt /dev/fd0
```

or

```bash
BlueCat:$ cd $BLUECAT_PREFIX/demo/showcase
BlueCat:$ mkboot -b /dev/fd0
BlueCat:$ mkboot -k showcase.disk -f showcase.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 firmware-downloadable or flash memory-programmable image containing the showcase demo system:

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

---

**mkkernel**

Builds the BlueCat Linux kernel

```bash
mkkernel config_file kernel1 [kernel2]
```

**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*. 
Builds a target board file system image according to a specification file

**Description**

mkrootfs [-DilLTv] [-N inodes] [-r blocks] <spec_file> \<output_file>
mkrootfs [-iIlLv] -J [chunk size] <spec_file> \<output_file>
makerootfs -t [-iIlLv] <spec_file>

This utility creates a gzipped target board file system image file or a tar file (if the -T option is specified), or a Flash File System image (if the -J option is specified), according to a given specification file. It can be used to create a minimal root file system for the target board.

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 169.

Currently, mkrootfs supports two target board file system types, the Linux ext2 file system (for compressed images) and the BlueCat Linux Journalling Flash File System (JFFS).

**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, etc.) 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:

```bash
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.

### Options

- `-D` Does not create a `lost+found` directory on the target board file system.
- `-i` Ignores non-fatal 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.
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.
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

```plaintext
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:
Appendix B - BLOSH Commands

> 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

```bash
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

```bash
echo [ -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.

```bash
> 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, uncompressed 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 file name `<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.
driver event codes 116
logging 124
notification 116
post-event codes 116
Pre and Post 104
pre-event codes 115
processing 105, 126
application programs
building 38
debugging 38
procedure 39
requirements 38
developing 21, 38
downloading and executing 79
stripping 39, 40
symbol table 39
architecture, JFFS 129
architecture-dependent power state 107
archive, downloading and unpacking 83
Autobooting BlueCat Linux 76
target board-specific 77
autonomous power management capabilities 107

B
bad block mapping 132, 144, 147
Binary Architecture CD-ROM 3, 8
  Directories and components 4, 5
  structure 4
block-device interface to flash memory 129, 131
BLOSH 49, 62, 95, 96
  adding new commands 80
  boot command 81, 175
  cd command 81, 176
  Commands for netbooting 77
  Commands Reference 81
  configuration file 50
  environment variables 50, 60, 62, 68, 69, 71, 180
  exec command 82, 176
  flash command 82, 177
  help command 85, 177
  mkboot command 83, 177
  mount command 81, 178
  ntar command 83, 178
  read command 84, 179
  Rebuilding 85
reset command 85, 179
script command 84, 179
set command 81, 180
Start-up Sequence 49
BLOSH Commands 175–180
BlueCat Linux
  Advanced Power Management 101
  Application development 21
  boot devices supported 27
  Components 15
  composite image 68
  default configuration 8
  demo systems 90
  development directory 21
  Development Process 26
  Directory Structure 14
  execution environment 16, 17
  installing 8–18
  Interfaces to Flash Memory 129
  kernel debugger 33
  optimizing footprint 43
  OS Loader 90, 96
  uninstalling 17
  uninstalling components 18
BlueCat Linux Loader Shell (BLOSH) 49
boot devices
  configuring demos for 91
  demo systems 93
  supported by BlueCat Linux 27
boot mechanism
  extension BIOS 73
  floppy disk 57
  hard disk 65
  OS Loader 74
  ROM/flash memory 67
Booting
  composite image 68
  from a Floppy Disk 56
  from DiskOnChip 63
  from extension BIOS 72
  from Hard Disk 58
  using Network and osloader 60
  from RAM using an NFS Server 78
  from RAM using TFTP 78
  from ROM/flash 66, 67
  kernel and file system image 67
  kernel and JFFS-based root file system 69
  over a Network or Parallel Port 74
  using firmware 74
Index

D

data chunks, JFFS 137, 138, 141
Debugging
  Application Programs 38
  finishing 36
  interrupting kernel 36
  Kernel 32
  requirements 33
  setting breakpoints 35
  setting up serial ports 34
  starting 34
  synchronization signals 35
default configuration
  packages 10
  TSP 28
Demo System 24
  Boot Devices 93
  Boot Procedure 93
  Components 88
  Configuring 91
  Configuring for boot device 91
  Configuring for hardware device 91
  developer 98
  Directory contents 89
  List 90
  Location 88
  Makefile 89
  Memory Size Options 95
  osloader 96
  Overview 87
  Requirements 90, 94
  Running 93
  showcase 97
  Storage Size Options 94
Deregistering
  APM Client 125
  MTD Driver 159
  PMD driver 127
developer demo system 90, 98
Developing
  APM Drivers 125
  Application Programs 38
  BlueCat Linux Applications 21–46
  MTD Drivers 157
  development directory, BlueCat
  cross development tools 24
development directory, BlueCat
  demo systems 24
  osloader 25
  Kernel tree 22
    location 22
  RPM packages 23
  target tools and files 24
Development Process, BlueCat Linux 21
Device node
  flash memory 135
  floppy disk driver 56
  hard disk 58
  mtdblock 132
  mtdchar 131
Directories
  Binary Architecture CD-ROM 4
  Source Architecture CD-ROM 7
  Target Support Package CD-ROM 6
directory structure, BlueCat Linux 14
diskOnChip 61
  adding support for 63
  booting from 63
  Copying onto 65
  ECC 150
Distribution media 3
Documentation CD-ROM 3
Documents
  man pages location 15
Downloading
  application programs 79
  BlueCat Linux, methods for 27
  into Flash Memory using Flash
    Management Tools 156
  into ROM/flash memory using
    Firmware 66
  OS Loader 74
  downloading and Booting BlueCat Linux 47–80

E

embedded system
  components 24
  definition 24
  Kernel 24
  root file system 24
environment variables
  BLOSH 50, 60, 62, 68, 69, 71, 180
  BLUECAT_PREFIX 16
  BLUECAT_TARGET_TSP 16
  PATH 16
**BlueCat Linux User’s Guide**

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>185</td>
</tr>
</tbody>
</table>

**Erasing a flash memory Device or Partition** 155
**Error Codes, APM service** 114
**error correction capabilities** 150
**event queue, APM** 104
**event record, APM** 102, 111

**Event, APM**
- descriptor 109, 114
- driver event codes 116
- notification 116
- post-event codes 116
- Pre and Post 104
- pre-event codes 115
- processing 105, 126

**execution environment**
- setting 16
- setting up for Linux hosts 16
- unsetting 17

**exportfs utility** 53

**extension BIOS** 72
- boot mechanism 57
- booting from 56
- copying onto 56

**F**

**fdisk** 58, 61, 63, 65

**file system**
- ext2 169
- JFFS 129, 132

**files, demo system** 89

**Finishing Kernel Debugging** 36

**Firmware** 67, 74, 93
- autoboot feature 67
- using for downloading 66

**Flash Memory**
- adding support for 148
- configuring partitions 146, 153
- /proc/mtd file 154
- device-independent management 133
- Entities and Device Nodes 135
- erasing 130, 145, 155
- major and minor numbers 135
- Management Tools and Mechanisms 153
- mounting 134
- partitioning 68, 71, 134, 161
- Build-time parameters 153
- Configuring 136
- physical operation 133, 143

**power loss during update** 141
**programming image into** 82, 177
**runtime partitioning** 136
**several partitions** 134
**support** 129
**writing raw data to** 155
**writing to** 142, 143

**synchronous operation** 143

**Flash Memory Support and Journaling Flash File System** 129–160

**flash_disk** 136
**flash_erase** 130, 161
**flash_fdisk** 68, 69, 71, 154, 161
**syntax** 154

**floppy disk**
- boot mechanism 57
- boot sector 57
- booting from 56
- copying onto 56

**garbage collection** 142, 143
- command 147
- criteria evaluation 143
- kernel thread 143
- rules 143
- single iteration 143

**GDB** 32, 34, 35, 40
- additional commands 37
- Commands 37
- extensions 37
- features 32
- quitting 36
- gdbserver 38, 99

**globbing** 170

**GNU Cross Compiler (gcc)** 44

**GNU tools** 38

**GnuPro Debugger (GDB)** 32
**goals, makefile** 92

**H**

**hard disk**
- boot mechanism 65
- boot partition 58
booting from 58
compressed root file system copied to 62
 copying onto 58
device node 58
kernel command line 65
master boot record 65
rfs mounted from partition on 59
root file system copied to partition on 60
setup code 65
Hardware device, configuring demos for 91
hosts supported by BlueCat Linux 2

i
i_osloader 75, 96
ifconfig 97, 98, 99
inactivity period parameter 108
inactivity timer 104, 108, 110, 117, 124
inetd process 52
in-RAM pointers to flash data 142
install demo system 10, 65
Installation CD-ROMs 3
Installation, BlueCat Linux 8–18
Installing
default configuration 8
 on Linux hosts 8
 on Windows hosts 9
 on NFS-mounted disk 10
Sources of BlueCat Linux RPM
 Packages 12
target board support 10
Interface
APM 108
 IOCTL 108, 111
 Kernel-space 103, 108, 116
 PMD device drivers 108, 110, 119
 user-mode 122
Kernel configuration 29
MTD 129, 133, 148
to Flash Memory 129
 block-device interface 129, 131
cache and buffering mechanism 132
character-device interface 129, 130,
 145, 154
JFFS file system 129, 132
 Interrupting the Kernel 36
 IOCTL commands, APM 121
 IOCTL interface to user space 108, 111
 IP autoconfiguration, OS Loader 76

J
JFFS
architecture 129
bad block mapping 132, 144
 block table 147
cache and buffering mechanism 132
creating on flash device 156
data chunks 137, 141
 length 138
file modification 138
in-RAM list of pointers to file nodes 139
Internals 137
 IOCTL Command Reference 147
 Layout 137
 log structure 138
 managing 156
 mounting 141
 POSIX I/O call 141
 number storage 141
 power loss recovery 132
 read operation 144
 scan algorithm 141
 supported calls 132
 wear leveling 132, 142
writing to flash memory 143
JFFS file system interface to flash memory 129,
 132, 133
JFFS Image Built on the Host 69
JFFS root file system 69, 70

K
Kernel
 .config file 29, 88
 building 30
bvmlinux.out file 31
 compressed image 67, 72, 75
 Configuration Interface 29
 Configuration Procedure overview 29
 customizing 28
 debugging 32
 GDB extensions 33
 header files 29
image size 43
interrupting 36
mkkernel utility 18, 32
Multiple profiles 31
optimizing footprint 43
reconfiguring 28
kernel command line, hard disk 65
Kernel Debugger 33, 35
default settings 34
kernel-space client, APM 101, 102, 108
callback 109, 110, 116
client services 117
interface 103, 108
opaque client handle 109, 116

L
ldconfig 44
Libraries
shared 43
static 44
Linux
cache and buffering mechanism 132
installing BlueCat Linux on 8
memory page caching 144
pristine kernel 23
release version 1, 22, 29
Linux host
setting execution environment 16
Location
cygwin.bat 5
cygwin32.exe 5
Demo Systems 88
man pages 15
SETUP.sh 5, 16
log file, event record 102, 111
log structure, JFFS 138
low-level device drivers, PMDs 101, 102, 110
LynuxWorks, contacting xii

M
major and minor numbers, flash entities 135
make bzImage 31
make config 29, 33
make menuconfig 29
make oldconfig 30, 32
make xconfig 30
Makefile
Goals 92
rebuilding a demo system 92
man pages location 15
Managing
JFFS 156
Multiple Embedded Applications 42
Multiple Kernel Profiles 31
tools for flash memory 153
mapm_ctrl 102, 103, 111, 112, 124, 125
command line format 124
Operation 125
mapmd 101, 104, 111
Command Line Format 123
Configuration File 123
Operation 124
options 164
mapmd.conf file 164
mapmd_ctrl 104
master boot record, hard disk 65
memory requirements
embedded system 44
Memory Sizing benchmark 44
/proc file system 45
Memory Technology Device interface 129, 133
Memory Technology Device interface
reference 148
memory tracking facility 46
memory usage statistics 44
Minimal Filesystem 43
mkboot command, BLOSH 83, 177
mkboot utility 48, 54, 59, 61, 63, 66, 67, 69, 72,
93, 165
functionality 48
options 165
mkconfig utility 32, 42, 88, 168
mkrootfs utility 41, 42, 44, 69, 70, 88, 169
building a minimal file system 43
file systems supported 169
image types 41, 42
options 172
Specification File 41, 169
stripping symbols 43
modules, APM 103
MTD 129, 133, 148
electromagnetic interference capabilities 150
memory technology type 148
module types 148
Index

registration service 134
MTD driver
access routine 133, 148
adding 133
configuring 153
configuring partitions at runtime 160
data structure 133
deregistering 159
developing 157
entry point 133
registering 133, 148, 157
write() callback 143, 144
writing 148
mtd_info structure 133, 134, 148
mtdblock 129, 131, 133, 135, 141
device node 132
ioctl() call 132
mount command 131
POSIX operations 131
mtchar 129, 130, 131, 133, 154, 155
device node 131
Interface Reference 145
ioctl() call 130
POSIX system calls 130
Multiple
embedded applications 42
instances of BlueCat Linux 17
kernel profiles 31
multi-user environment 17

N

network, booting over 74
using firmware 74
using OS Loader 74
Networking
booting images from different subnet 54
enabling on host 51
NFS 78
mounting a root file system from 78
NFS server, setting up 53, 54

O

Optimizing Footprint 43
customizing kernel 43
discarding symbols from files 43
going shared libraries 43
memory sizing benchmark 44
minimal file system 43
using static libraries 44
Options
mapmd 164
mkboot 95, 165
mkrootfs 172
OS Loader 47, 48, 56, 66, 68, 69, 71, 74, 93
boot mechanism 74
Customizing 80
downloading 74
Hardware support for net booting 75
IP autoconfiguration 76
location 25, 48
network protocols supported 49
osloader demo system 90, 96

P

parallel port 54, 55, 79, 93
booting from 74
Partition
Flash Memory
Build-time parameters 153
configuring 136, 153
MTD Driver 160
Partitioning
Flash Memory 68, 71, 134, 161
runtime 136, 161
PATH 16
PFTP daemon 173
PFTP server, setting up 54
pftp 173
PMD 101
API 104, 119
callback 119
callback parameters 119
conditional switch 108
CPU driver 112
deregistering driver 127
device drivers interface 108, 110
driver event codes 116
event codes 115
handles 114
inactivity period parameter 108
inactivity timer 108
post-event codes 116
power state identifier 109
pre-event codes 115
processing requests in a driver 128
registering driver 127
registration service 110
Requests 120
sample driver 127
Services 120
State Codes 115
status data 112
unregistration 110
POSIX I/O call, JFFS 141
Power Loss Recovery 132, 141
power management
autonomous 107
CPU 112
custom 107
power state 107
architecture-dependent 107
CPU 102
negotiating transition 108, 117
state identifier 109
status data 112
switching 107
Power-Managed Device (PMD) 101
pristine Linux kernel 23

R

raw_inode structure 137, 141
checksums 141
file name 137
file node number 138, 139
parent inode number 137, 139
Rebuilding
BLOSH 85
Demo System 92
reconfiguring kernel 28
Red Hat Linux GNOME desktop 8
Red Hat Package Manager (RPM) 2
Registering
APM Client 125
MTD Driver 157
PMD Driver 127
requirements
cross development host 2
demo systems 94

ROM Boot BIOS 72, 73
ROM/flash memory
boot mechanism 67
booting from 66
downloading onto using firmware 66
root file system
.spec file 88
building 41
compressed image 67
copied to hard disk 62
copied to hard disk partition 60
creating on JFFS partition on target 70
discarding symbols 43
minimal 43
mkrootfs utility 41
mounted from partition on hard disk 59
mounting from NFS 78
RPM packages
installing sources of 12
Rebuilding 13
rpm utility 4, 18
using 13
Running Demo Systems 93

S

Sample
APM Client 125
PMD Driver 127
serial ports for debugging 34
Setting Up
BlueCat Linux Execution Environment 16
BOOTP Server 51
Multiple Instances of BlueCat Linux 17
NFS Server 53, 54
PFTP Server 54
TFTP Server 52
setup code 57
hard disk 65
SETUP.sh 10, 11, 12, 13, 16, 18, 27, 56
location 5, 16
shared libraries 43
showcase demo system 90, 97
single iteration of garbage collection 143
Source Architecture CD-ROM 3, 12, 23
Directories 7
structure 7
sources of RPM packages 12
Index

Special Note formats  xii
Specification file, mkrootfs  169
Starting Kernel Debugging  34
static libraries  44
strip on/off command  43
structure, md_info  133, 134
structure, raw_inode  137
subdirectories, demo system  89
Subnet  54
superuser  8, 55, 56
support for target boards
activating  11
installing  10
uninstalling  17
symbol table, application program  39

T

target board
activating support  11
creating root file system on JFFS partition  70
Firmware  66, 74, 93
autoboot feature  67
installing support  10
Tools and Files  24
uninstalling support  17
target remote command  34, 40, 99
Target Support Package CD-ROM  3, 10, 11
Directories and components  6
Technical Support  xii
telnet, port number  39
TFTP server, setting up  52
Tk interpreter  30
tools
BlueCat Linux  1
Flash memory management  153
tracebacks for kernel calls  45
TSP, default kernel configuration  28
Typographical Conventions  xi

U

Uninstalling
a BlueCat Linux Component  18
BlueCat Linux  17

Support for a BlueCat Linux installation  17
Support for a target board  17
Unsetting the BlueCat Linux Environment  17
user-space client  102
user-space control utility  102, 111, 112, 124
user-space daemon  101, 111
user-space interface driver  103, 111, 112
Using
/proc/mtd file  154
Makefile to Rebuild a Demo System  92
Memory Sizing Benchmark  44
mkrootfs to Build a Minimal File System  43
Static Libraries  44

V

VFS  132, 143, 144
VFS commands  138
Virtual File System (VFS) layer  132

W

wakeup event  107
wakeup logic  107
wear leveling  132, 142
Windows
installing BlueCat Linux on  9
installing CYGWIN  9
Writing Raw Data to Flash Memory  155
writing to flash memory  142, 143

X

XIP  149, 150