

# BlueCat Linux User's Guide

---

BlueCat Linux Release 3.1

DOC-0418-00

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

Copyright ©1987-2001, LynuxWorks, Inc. All rights reserved.  
U.S. Patents 5,469,571; 5,594,903

Printed in the United States of America.

All rights reserved. No part of the *BlueCat Linux User's Guide* may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photographic, magnetic, or otherwise, without the prior written permission of LynuxWorks, Inc.

LynuxWorks, Inc. makes no representations, expressed or implied, with respect to this documentation or the software it describes, including (with no limitation) any implied warranties of utility or fitness for any particular purpose; all such warranties are expressly disclaimed. Neither LynuxWorks, Inc., nor its distributors, nor its dealers shall be liable for any indirect, incidental, or consequential damages under any circumstances.

(The exclusion of implied warranties may not apply in all cases under some statutes, and thus the above exclusion may not apply. This warranty provides the purchaser with specific legal rights. There may be other purchaser rights that vary from state to state within the United States of America.)

---

# *Contents*

---

<b>PREFACE</b>	<b>XIII</b>	
Typographical Conventions	xiii	
Special Notes	xiv	
Technical Support	xv	
LynuxWorks U.S. Headquarters	xv	
LynuxWorks Europe	xv	
World Wide Web	xv	
<b>CHAPTER 1</b>	<b>INSTALLATION</b>	<b>1</b>
BlueCat Linux Distribution Overview		1
System Requirements		2
BlueCat Linux Distribution CD-ROMs		2
Binary Architecture CD-ROM Tree Structure		3
Binary Architecture CD-ROM Directories		4
Binary Architecture CD-ROM Subdirectories		5
Target Support Package CD-ROM Tree Structure		6
Target Support Package CD-ROM Directories		6
Target Support Package CD-ROM Subdirectories		7
Source Architecture CD-ROM Tree Structure		7
Source Architecture CD-ROM Directories		8
Source Architecture CD-ROM Subdirectories		9
Installing BlueCat Linux		9
Installing the Default Configuration		9
Packages in the Default BlueCat Linux Configuration		11
Installing Support for Target Boards		11
Activating Support for a Target Board		12

Installing Optional BlueCat Linux Packages .....	13
Optional BlueCat Linux Packages .....	13
Installing Sources of BlueCat Linux RPM Packages .....	14
Using BlueCat Linux RPM .....	15
BlueCat Linux Directory Structure .....	16
Overview .....	16
BlueCat Linux Components .....	17
Setting Up BlueCat Linux Execution Environment .....	18
For Linux Cross Development Hosts .....	18
Unsetting the BlueCat Linux Environment .....	19
Setting Up Multiple Instances of BlueCat Linux .....	19
Uninstalling BlueCat Linux .....	19
Uninstalling an Entire BlueCat Linux Installation .....	19
Uninstalling BlueCat Linux Support for a Target Board .....	20
Uninstalling a BlueCat Linux Component .....	20

---

**CHAPTER 2**

<b>DEVELOPING BLUECAT LINUX APPLICATIONS .....</b>	<b>21</b>
Overview of BlueCat Linux Directory Structure .....	21
Development Directory Tree Structure .....	21
Kernel Tree .....	22
Kernel Subdirectory .....	22
BlueCat Linux Kernel RPM Packaging .....	23
BlueCat Linux Kernel v/s “Pristine” Linux Kernel .....	24
Target Board Tools and Files .....	24
Cross Development Tools .....	24
Demo Systems .....	24
OS Loader .....	25
BlueCat Linux Embedded System .....	25
Development Process Overview .....	26
Customizing the BlueCat Linux Kernel .....	28
Configuring the Kernel .....	28
Default Kernel Configuration .....	29
Kernel Configuration Procedure Overview .....	29
Kernel Configuration Command Interface .....	29
Building the Kernel .....	30
Building the Kernel for an x86 Target Board .....	30
Managing Multiple Kernel Profiles .....	31
Debugging the Kernel .....	33

Kernel Debugging Requirements .....	33
Building a Kernel for Debug Purposes .....	33
Debugging .....	33
Setting Up Serial Ports .....	34
Starting Kernel Debugging .....	35
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
Building a Root Filesystem .....	40
BlueCat Linux Root Filesystem Utility – mkrootfs .....	40
mkrootfs Specification File .....	41
Images Created by mkrootfs .....	41
Managing Multiple Embedded Applications .....	42
Optimizing Footprint .....	42
Customizing the Kernel for Size .....	42
Using mkrootfs to Build a Minimal Filesystem .....	43
Discarding Symbols from Files .....	43
Getting Necessary Shared Libraries .....	43
Using Static Libraries .....	44
Using the Memory Sizing Benchmark .....	44

---

## CHAPTER 3

<b>DOWNLOADING AND BOOTING BLUECAT LINUX .....</b>	<b>47</b>
BlueCat Linux Boot Procedure Overview .....	47
BlueCat Linux Boot Scenarios .....	48
Booting BlueCat Linux from a Floppy Disk .....	48
Copying BlueCat Linux onto Floppy Disk .....	49
Booting BlueCat Linux on x86 from Floppy Disk .....	50
Booting BlueCat Linux from Hard Disk .....	50
Copying onto Hard Disk from Cross Development Host .....	50
Copying BlueCat Linux onto Hard Disk using OS Loader .....	52
Copying BlueCat Linux with Compressed Root Filesystem to Hard Disk .....	53
Copying to Hard Disk using install Demo System .....	55
Booting from Hard Disk on an x86 Target Board .....	57
Booting from DiskOnChip .....	57

Copying BlueCat Linux to DiskOnChip .....	58
Booting BlueCat Linux from Target ROM/Flash Memory .....	59
Downloading into Target ROM/Flash Memory using Firmware	59
Downloading into Target ROM/Flash Memory using OS Loader	60
Downloading into Target ROM/Flash Memory using install Demo	
System .....	65
Booting from Target ROM/Flash Memory using Firmware ....	69
Booting from Extension BIOS on x86 .....	69
Booting BlueCat Linux over Network or Parallel Port .....	71
Booting over Network using Target Board Firmware .....	71
Booting over Network or Parallel Port using OS Loader .....	72
BlueCat Linux OS Loader Overview .....	73
BlueCat Linux Loader Shell (BLOSH) .....	74
BLOSH Startup Sequence .....	74
BLOSH Environment Variables .....	74
BLOSH Command Reference .....	76
boot – Booting a BlueCat Linux Kernel .....	76
cd – Change Current Working Directory .....	77
exec – Execute a Program .....	77
flash – Program Image into Flash Memory .....	77
help – Print Help Message .....	78
mkboot – Create a Bootable Disk .....	78
mount – Mount a Filesystem .....	78
ntar – Download and Unpack a tar Archive .....	79
read – Download an Arbitrary File .....	79
reset – Reboot the System .....	80
script – Process List of Commands in a File .....	80
set – Show or Modify Environment Variables .....	80
IP Auto-Configuration .....	81
Setting Up a BOOTP Server .....	81
Setting Up a TFTP Server .....	82
Booting Images from a Different Subnet .....	82
Setting Up an NFS Server .....	83
Setting Up a PFTP Server .....	84
Linux .....	84
Windows NT/2000 .....	85
Windows 98 .....	85
Using BlueCat Linux OS Loader in Embedded Systems .....	85
Booting from a TFTP Server .....	85

Booting from an NFS Server .....	85
Mounting a Root Filesystem from NFS .....	86
Booting from a PFTP Server .....	86
Auto-Booting BlueCat Linux .....	86
Target Board-Specific Auto-Boot of BlueCat Linux .....	87
Downloading and Executing Programs .....	88
Configuring BlueCat Linux OS Loader .....	88
Configuring OS Loader as a Demo System .....	88
Configuring Hardware Device Support .....	88
Customizing BlueCat Linux OS Loader .....	89
Rebuilding BLOSH .....	89
Adding New Commands to BLOSH .....	89

**CHAPTER 4**

<b>BLUECAT LINUX DEMO SYSTEMS .....</b>	<b>91</b>
Demo System Conceptual Overview .....	91
Demo Systems .....	92
Demo Systems Location .....	92
Supported Demo Systems .....	92
Demo System Components .....	97
Contents of Demo System Directory .....	98
Configuring a Demo System .....	99
For Hardware Devices .....	99
For the Boot Device .....	99
Building Demo Systems .....	100
Using the Makefile to Rebuild a Demo System .....	100
Running Demo Systems .....	101
Demo Systems Reference .....	102
Demo Systems Requirements .....	102
Simple Systems .....	104
Shells .....	106
Simple Networking .....	111
Utility Systems .....	114
Debuggers .....	117
Disk Operations .....	124
X Window and Friends .....	127
Advanced Networking .....	129
BlueCat Linux Messenger .....	133
Java .....	135

Flash Memory Support and Flash File System .....	136
Advanced Power Management .....	137

---

<b>CHAPTER 5</b>	<b>BLUECAT LINUX ADVANCED POWER MANAGEMENT . 141</b>
General Architecture .....	141
Overview .....	141
APM Modules and Components .....	142
APM Event Queue .....	144
Pre- and Post-Events .....	144
Event Processing .....	145
Power States .....	146
Software Inactivity Timers .....	147
APM Interfaces .....	148
Kernel-Space Client Interface .....	148
PMD Device Drivers Interface .....	149
IOCTL Interface to User Space .....	150
User-Space Components .....	151
APM Daemon .....	151
APM Control Utility .....	152
APM proc File .....	152
CPU Power Management .....	152
Configuring APM in the Kernel .....	153
APM Interfaces Reference .....	154
Common Constants and Data Types .....	154
PMD Handles .....	154
Error Codes .....	154
PMD State Codes .....	155
Event Codes .....	155
Kernel-Mode API .....	156
Client Handles .....	156
Client Callback .....	156
Client Services .....	157
PMD API .....	159
PMD Callback .....	159
PMD Requests .....	160
PMD Services .....	160
IOCTL Commands .....	161
User-Mode Interfaces .....	162
/proc/mapm Directory .....	162

mapmd Command Line Format .....	163
mapmd Configuration File .....	163
Handler Program Command Line Format .....	164
Event Logging .....	164
mapmd Operation .....	164
APM Control Utility .....	164
APM Control Utility Command Line Format .....	164
mapm_ctrl Operation .....	165
Developing APM Drivers .....	165
Sample APM Client .....	165
Registering an APM Client .....	165
Deregistering an APM Client .....	166
Processing APM Events .....	166
Sample PMD Driver .....	167
Registering a PMD Driver .....	167
Deregistering a PMD Driver .....	168
Processing Requests in a PMD Driver .....	168

---

## CHAPTER 6

<b>FLASH SUPPORT AND FLASH FILE SYSTEM .....</b>	<b>169</b>
Flash Support and FFS Architecture .....	169
BlueCat Linux Interfaces to Flash Memory .....	169
The mtdchar Interface .....	170
The mtdblock Interface .....	171
The Flash File System (FFS) Interface .....	172
MTD Interface .....	172
Flash Memory Partitioning .....	174
Partitioning Method .....	174
Flash Memory Entities and Device Nodes .....	174
Partition Configuration .....	176
FFS Internals .....	176
FFS Layout .....	176
Power Loss Recovery .....	181
Wear Leveling .....	182
Wear Leveling and Garbage Collection Algorithm .....	182
Synchronous Operations .....	184
Automatic Bad Block Mapping .....	184
The mtdchar Interface Reference .....	185
FFS IOCTL Command Reference .....	187
MTD Interface Reference .....	188

Flash Memory Management Tools and Mechanisms .....	194
Configuring Flash Memory Partitions .....	194
Configuring Partitions at Build Time .....	194
Configuring Partitions using Kernel Boot-Time Parameters ..	195
Configuring Partitions using flash_fdisk .....	195
Using the /proc/mtd File .....	195
Erasing a Flash Memory Device or Partition .....	196
Writing Raw Data to Flash Memory .....	196
Managing an FFS .....	197
Downloading BlueCat Linux into Flash Memory with Flash Management Tools .....	197
Developing an MTD Driver .....	197
Registering an MTD Driver .....	198
Deregistering the MTD Driver .....	200
Configuring Partitions at Runtime .....	200

---

**APPENDIX A      DEFAULT BLUECAT LINUX PACKAGES 203**

---

**APPENDIX B      OPTIONAL BLUECAT LINUX PACKAGES 227**

---

<b>APPENDIX C      MKROOTFS COMMAND REFERENCE 241</b>	
Specfile Format .....	242
Options .....	244

---

<b>APPENDIX D      MKBOOT COMMAND REFERENCE 247</b>	
Options .....	248
Examples .....	249

---

**APPENDIX E      MKKERNEL COMMAND REFERENCE 251**

---

**APPENDIX F      MAPMD COMMAND REFERENCE 253**

	Options .....	253
<hr/>		
<b>APPENDIX G</b>	<b>MAPM_CTRL COMMAND REFERENCE 255</b>	
	Parameters .....	255
	Examples .....	256
<hr/>		
<b>APPENDIX H</b>	<b>MAPMD.CONF COMMAND REFERENCE 257</b>	
	Example .....	258
<hr/>		
<b>APPENDIX I</b>	<b>FLASH_FDISK COMMAND REFERENCE 259</b>	
	Configuration String Format .....	259
	Example .....	260
<hr/>		
<b>APPENDIX J</b>	<b>FLASH_ERASE COMMAND REFERENCE 261</b>	
	Example .....	261
<hr/>		
<b>APPENDIX K</b>	<b>PFTPD COMMAND REFERENCE 263</b>	
	Parameters .....	263
	Configuration File .....	264
	Configuration File Format .....	264
<hr/>		
<b>INDEX</b>	.....	265



---

# Preface

---

## Typographical Conventions

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

### Font and Description

### Examples

Garamond – body text; *italicized* for emphasis, new terms, book titles, and text that should be replaced by a user-defined real name or value

Refer to the *BlueCat Linux User's Guide*.  
cat *filename*  
mv *file1 file2*

Courier New (used within body text) – commands, options, filenames, pathnames, and other computer-generated data

```
ls  
-l  
myprog.c  
/dev/null
```

Courier New 7 pts. – **blocks of text that appear on the display screen after entering instructions or commands**

```
Loading file /tftpboot/shell.kdi into  
0x4000  
.....  
File loaded. Size is 1314816  
Copyright 2000 LynuxWorks, Inc.  
All rights reserved.  
  
LynxOS (ppc) created Mon Jul 17  
17:50:22 GMT 2000  
user name:
```

Font and Description	Examples
<b>Courier New Bold</b> (used within examples) – command lines and options entered on the computer by the user	login: <b>myname</b> cd /usr/home
VERDANA (all caps) – environment variables	DISPLAY
<b>Verdana Bold</b> – keyboard options, button names, and menu sequences	Enter , Ctrl-C
<i>Verdana Italics</i> – functions or function names	<i>getenv()</i>

---

## 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.*



**WARNING!** *Used for conditions or acts that could seriously injure personnel (death is a remote possibility). For example, electrical shock hazards.*

---

## 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, Frequently Asked Questions (FAQs), and LynuxWorks news groups.

### LynuxWorks U.S. Headquarters

Internet: [support@lynuxworks.com](mailto:support@lynuxworks.com)

Phone: (408) 879-3940

Fax: (408) 879-3945

### LynuxWorks Europe

Internet: [tech\\_europe@lynuxworks.com](mailto:tech_europe@lynuxworks.com)

Phone: (+33) 1 30 85 06 00

Fax: (+33) 1 30 85 06 06

### World Wide Web

<http://www.lynuxworks.com>



This chapter describes the installation procedure for BlueCat Linux.

---

## BlueCat Linux Distribution Overview

The important features of BlueCat Linux distribution are described below:

- BlueCat Linux uses a set of standard CD-ROMs as the distribution medium. These CD-ROMs are used for installing BlueCat Linux.
- BlueCat Linux is a cross development product. It allows for software development on a cross development host and the necessary tools for transferring software to the target board. Therefore, BlueCat Linux must be installed on a cross development host.
- BlueCat Linux is easy to install. The distribution CD-ROMs contain an installation program to install the minimal product configuration onto the cross development host. The installation program uses the Red Hat Package Manager (RPM), which simplifies installing and uninstalling packages.
- BlueCat Linux can be installed and used by any user logged in on the cross development host, provided the user has permission to mount the CD-ROM.
- Multiple independent installations of BlueCat Linux on a single cross development host system are supported.
- BlueCat Linux coexists with native cross development host tools and features. The installed BlueCat Linux is activated by executing a shell script that sets up the BlueCat Linux execution environment.

---

## System Requirements

BlueCat Linux installation is performed on a pre-installed, fully operational cross development host. The cross development host system can either be a Linux host (Intel PC running Red Hat Linux 6 or higher or TurboLinux Workstation 6.0), or a Windows host (Intel PC running Windows 98, Windows NT, or Windows 2000).

---

**NOTE:** *BlueCat Linux installation instructions and examples in this User's Guide are based on x86 target boards. For detailed information regarding non-x86 target boards, please consult the appropriate Target Support Guide (TSG).*

---

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 cross development hosts for supported target boards, as defined in the table below
- CD-ROM drive

Table 1-1: BlueCat Linux Core Components for Supported Target Boards

Host	x86	PowerPC	ARM	SH	MIPS
<b>Linux</b>	650 MB	650 MB	610 MB	550 MB	715 MB
<b>Windows</b>	530 MB	530 MB	490 MB	430 MB	610 MB

---

**NOTE:** *The amount of free disk space shown in the table above may be slightly more or less for a particular installation of BlueCat Linux, depending on the disk space required by its target board-specific components.*

---

---

## BlueCat Linux Distribution CD-ROMs

The BlueCat Linux distribution is contained on two types of CD-ROMs:

1. The *BlueCat Linux Installation CD-ROMs* – Installation of BlueCat Linux support for a particular target board requires

two Installation CD-ROMs: A *Binary Architecture CD-ROM* (for a specific microprocessor family), and a *Target Support Package CD-ROM* (binary and source for a specific target board).

- The *Binary Architecture CD-ROM* contains the common binary files for all supported boards necessary for development on a specific microprocessor family (e.g., the x86 microprocessor family). There is a different Binary Architecture CD-ROM for each family. The *BlueCat Linux User's Guide* can also be found on this CD-ROM in PDF format.
- The *Target Support Package (TSP) CD-ROM* contains both BlueCat Linux binary and source files used to support development on a specific target board. A TSP CD-ROM is installed after installing the core components. This CD-ROM also contains the *BlueCat Linux Target Support Guide* for a given target board.

Use these CD-ROMs to install BlueCat Linux binaries onto the cross development host.

2. The *BlueCat Linux 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.

## Binary Architecture CD-ROM Tree Structure

The Binary Architecture CD-ROM directories are organized as shown in Figure 1-1:

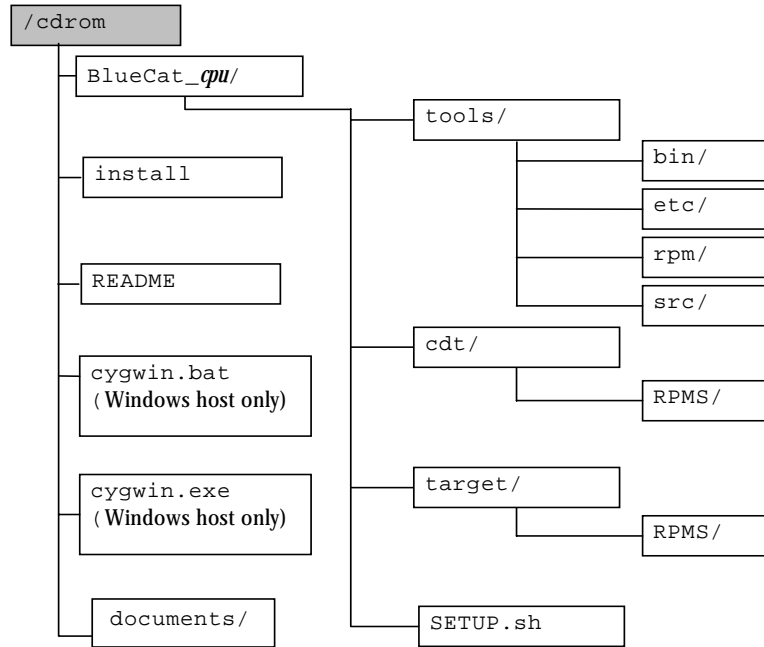


Figure 1-1: Binary Architecture CD-ROM Tree Structure

---

**NOTE:** *In the `Bluecat_cpu` directory, “cpu” implies an architecture family, e.g., `i386` for `x86`.*

---

## Binary Architecture CD-ROM Directories

The BlueCat Linux Binary Architecture CD-ROM has the following main directories:

`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`, which provides user-level package database management and file extraction relative to the installation point.

<code>cdt</code>	Cross Development Tools – directory containing binary packages of the cross development tools that run on the cross development host
<code>target</code>	Target board directory containing binary packages built to run on a target board
<code>documents</code>	Directory containing product documentation as a PDF (Portable Document Format) file.

## Binary Architecture CD-ROM Subdirectories

Table 1-2 briefly describes all the components of the BlueCat Linux Binary Architecture CD-ROM.

Table 1-2: Binary Architecture CD-ROM Subdirectories

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

Table 1-2: Binary Architecture CD-ROM Subdirectories (Continued)

Node	Description
- cygwin.bat	Windows host installation script
- cygwin.exe	Windows host installation script
- documents/	Product documentation

## Target Support Package CD-ROM Tree Structure

The Target Support Package CD-ROM directories are organized as shown in Figure 1-2:

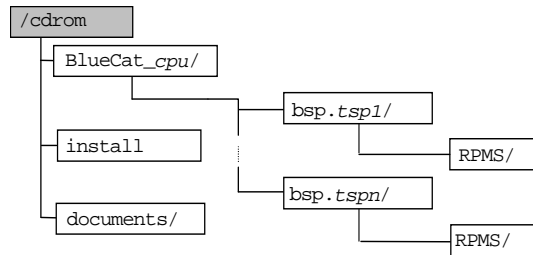


Figure 1-2: Target Support Package CD-ROM Tree Structure

## Target Support Package CD-ROM Directories

The BlueCat Linux Target Support Package CD-ROM has a number (one or more) of directories `bsp.tsp1`, . . . , `bsp.tspn`, each for a specific board. Additionally, there is a directory, `documents`, which contains product documentation specific to the Target Support Packages.

`bsp.tspn`

Target board directory containing target board-specific binary and source packages, i.e., `tspn` (where `tspn` is `cpci_cpn5360` for the x86-based CPN5360 target board)

`documents` Directory containing product documentation specific to each Target Support Package. This documentation is in PDF format.

## Target Support Package CD-ROM Subdirectories

Table 1-3 briefly describes the components of the BlueCat Linux Target Support Package CD-ROM.

Table 1-3: Target Support Package CD-ROM Subdirectories

Node	Description
<code>/mnt/cdrom</code>	Mount point (typical)
<code>- BlueCat_<i>cpu</i>/</code>	BlueCat Linux tools and packages for a specific CPU architecture (e.g., BlueCat_i386 for x86)
<code>-- bsp.<i>tsp</i>&lt;<i>t</i>&gt;/</code>	BlueCat Linux packages for a specific target board, where <i>tsp</i> is its Target Support Package
<code>--- RPMS/</code>	Target board specific packages
<code>- install</code>	Installation program
<code>- documents/</code>	Product documentation

## Source Architecture CD-ROM Tree Structure

The Source Architecture CD-ROM includes directories organized as shown in Figure 1-3:

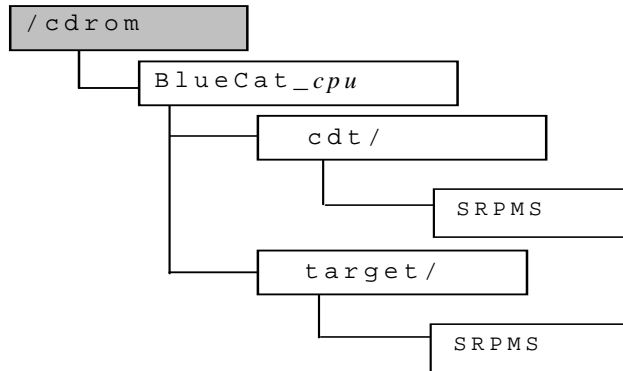


Figure 1-3: Source Architecture CD-ROM Tree Structure

## Source Architecture CD-ROM Directories

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 cross development host
target	Target board directory containing sources of the RPM packages configured to run on a target board

## Source Architecture CD-ROM Subdirectories

Table 1-4 briefly describes the components of the Source Architecture CD-ROM.

Table 1-4: Source Architecture CD-ROM Subdirectories

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

## Installing BlueCat Linux

### 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.

#### - **Linux**

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

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

Please note that some desktops such as Red Hat Linux GNOME auto-mount the CD-ROM upon insertion into a CD-ROM drive. 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 auto-mount feature and mount the CD-ROM manually. Proceed to Step 2.

#### - **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 the Windows host. Upon completion of this script a `bash` window appears. All 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.*

**NOTE:** *The contents of the `documents` directory are not intalled with the above installation procedure.*

---

## Packages in the Default BlueCat Linux Configuration

The default BlueCat Linux configuration includes all the RPM packages installed from the Binary Architecture CD-ROM. Refer to Appendix A, “Default BlueCat Linux Packages” for a detailed description of the packages included in the default BlueCat Linux configuration.

## Installing Support for Target Boards

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 above entitled “**Windows**” under “Installing the Default Configuration” on page 9.)

1. Insert the Target Support Package CD-ROM containing BlueCat Linux for one or more boards in the microprocessor family into the CD-ROM drive and mount it. To mount the CD-ROM on a Linux host, refer to the section above entitled “**Linux**” under “Installing the Default Configuration” on page 9.
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. Run the BlueCat Linux Target Support Package by typing:

```
BlueCat:$ /mnt/cdrom/install tsp
```

where *tsp* is the specific Target Support Package to be installed. (e.g., `cpci_cpn5360` for the x86-based CPN5360 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 TSP please consult the accompanying Target Support Guide.*

---

4. Optionally, repeat the previous step for another board supported in the Target 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 Target Support Package CD-ROM. Only the steps described in the section “Installing Support for Target Boards” have to be performed to install target board support on top of a pre-installed core.

---

**NOTE:** *The contents of the `documents` directory are not intalled with the above installation procedure.*

---

## Activating Support for a Target Board

Installation procedures described above install the BlueCat Linux core, and Target Support Packages for a number of target boards. To activate these packages for a target board, use the following procedure:

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

```
$ . SETUP.sh tsp
```

where *tsp* refers to the Target Support Package for a specific target 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 Optional BlueCat Linux Packages

In addition to the default BlueCat Linux packages, the Binary Architecture CD-ROM contains a number of packages that can be installed on a per-package basis. Mostly, these are source packages of certain BlueCat Linux components that are not needed for most BlueCat Linux development activities.

To install an optional package onto the cross development host, use the following procedure (assuming that the Binary Architecture CD-ROM has been inserted and mounted). For example, the following session demonstrates the installation of a new font package, e.g., the `XFree86_trg-cyrillic-fonts` package:

1. From the top of the directory where BlueCat Linux is installed, set up the BlueCat Linux environment by typing:

```
$ . SETUP.sh
```

2. Extract the package to be installed. For instance, enter:

```
BlueCat[i386_core]:$ rpm -i \  
/mnt/cdrom/BlueCat_i386/target/RPMS/  
XFree86_trg-cyrillic-fonts-3.3.5-\  
1.i386.rpm
```

## Optional BlueCat Linux Packages

Refer to Appendix B, “Optional BlueCat Linux Packages” for a detailed description of the optional BlueCat Linux packages.

## 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 a single Source Architecture CD-ROM per microprocessor family. This CD-ROM contains all 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 the BlueCat Linux environment variables to enable the build procedure for determining the target board that the build has been called for.

The following example demonstrates rebuilding the sources for the line editor (`ed`) RPM package. Assuming that the Source Architecture CD-ROM is mounted at `/mnt/cdrom`, type the following to install the sources onto the cross development host:

```
BlueCat:$ rpm -i /mnt/cdrom/BlueCat_cpu\
/target/SRPMs/ed_trg-0.2-1.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 `ed` RPM package (`ed-0.2.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 `ed` RPM package does not patch the tar file, there are no patch files in the `SOURCES` directory.

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

Successful completion of the command creates a tree of source files for the `ed` package in the directory

`$BLUECAT_PREFIX/cdt/src/bluecat/BUILD`. Use this tree to rebuild the package from the sources. For instance:

```
BlueCat:$ cd \
$BLUECAT_PREFIX/cdt/src/bluecat/SPECS
BlueCat:$ rpm -ba ed_trg.spec
```

This places the rebuilt RPM package (`ed_trg-0.2-1.cpu.rpm`) into the `$BLUECAT_PREFIX/cdt/src/bluecat/RPMS/cpu` directory.

Reinstall this package into the BlueCat Linux execution environment with:

```
BlueCat:$ rpm -i --force \
$BLUECAT_PREFIX/cdt/src/bluecat/RPMS/cpu/\
ed_trg-0.2-1.cpu.rpm
```

---

**NOTE:** *Rebuild of certain RPM packages requires installation of appropriate optional BlueCat Linux packages. Refer to “Installing Optional BlueCat Linux Packages” on page 13 for details.*

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

---

## Using BlueCat Linux RPM

The BlueCat Linux installation procedure is based on the BlueCat Linux `rpm` utility included on the distribution CD-ROM. This version of `rpm` provides the standard functionality of the Red Hat 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 cross development 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 Figure 1-4:

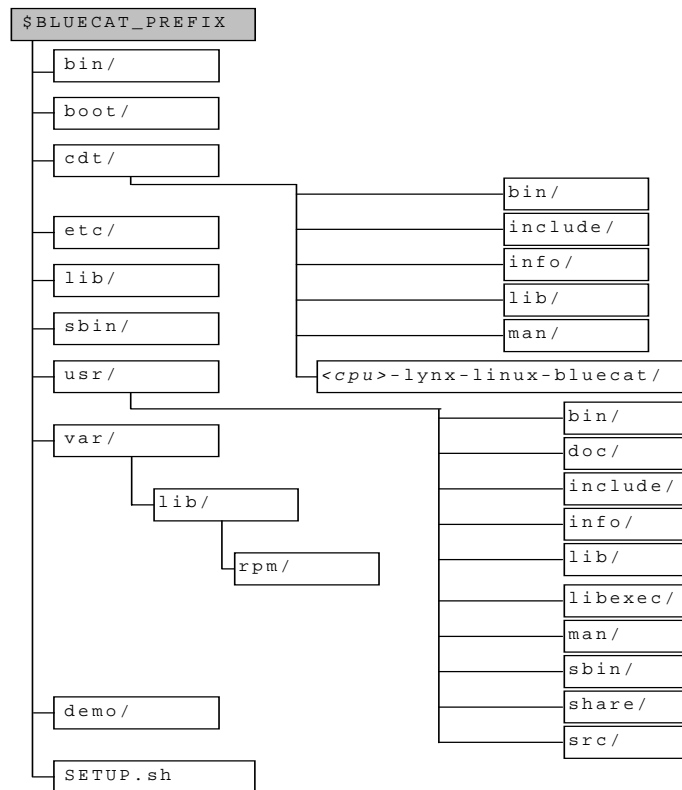


Figure 1-4: BlueCat Linux Directory Structure

## BlueCat Linux Components

Table 1-5 briefly describes all the key components of the BlueCat Linux directory structure.

Table 1-5: BlueCat Linux Directory Structure Components

Node	Description
<code>\$BlueCat_PREFIX</code>	Installation directory
<code>- bin/</code>	Target board binaries
<code>- boot/</code>	Target board <code>boot</code> directory
<code>- cdt/</code>	Cross development tools
<code>-- bin/</code>	Cross development binaries
<code>-- include/</code>	Cross development <code>include</code> files
<code>-- info/</code>	GNU <code>info</code> files
<code>-- lib/</code>	Cross development libraries
<code>-- man/</code>	Cross development tool manpages
<code>--<i>cpu</i>-lynx-linux-bluecat/</code>	Cross development binaries
<code>- demo/</code>	BlueCat Linux demo system configurations
<code>- etc/</code>	Target board configuration files
<code>- lib/</code>	Target board libraries
<code>- sbin/</code>	Target board system binaries
<code>- usr/</code>	Target board top-level <code>usr</code> directory
<code>-- bin/</code>	More target board binaries
<code>-- doc/</code>	Various documentation files
<code>-- include/</code>	Target board <code>include</code> files
<code>-- info/</code>	More GNU <code>info</code> files
<code>-- lib/</code>	More target board libraries
<code>-- libexec/</code>	Auxiliary files
<code>-- man/</code>	Main manpages
<code>-- sbin/</code>	More target board system binaries

Table 1-5: BlueCat Linux Directory Structure Components (Continued)

Node	Description
-- share/	Various target board shared files
-- src/	Main source tree
- var/	Top-level target board var directory
-- lib/	Libraries
-- rpm/	RPM database
- SETUP.sh	Shell script for setting up cross development environment

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

---

## Setting Up BlueCat Linux Execution Environment

### For Linux Cross Development Hosts

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

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

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

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

```
$ . SETUP.sh [tsp]
```

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 `tsp` option is specified, the `BLUECAT_TARGET_TSP` variable is set to activate the Target 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).

## Setting Up Multiple Instances of BlueCat Linux

As already noted, 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 BlueCat Linux Installation

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

```
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. Also, close the current shell to clean up all previous environment settings.

## Uninstalling BlueCat Linux 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] [tsp1...tsp2...tspn]
```

If the `-f` option is not specified, the script prompts the user before starting uninstallation. Otherwise, the 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 value of *package\_name*, use the `-qa` argument with the `rpm` command. This displays a list of RPM packages installed.

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.

# *Developing BlueCat Linux Applications*

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

---

## Overview of BlueCat Linux Directory Structure

### Development Directory Tree Structure

Installing BlueCat Linux results in a development directory tree, which is structured as follows:

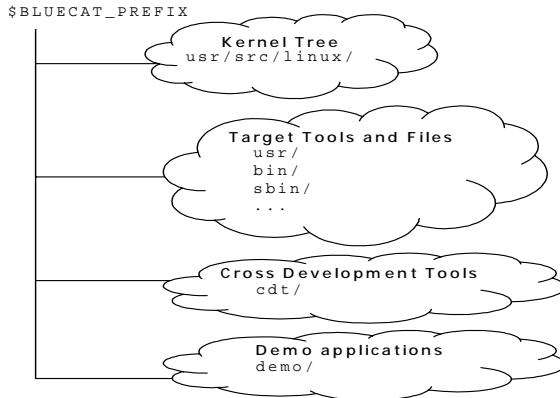


Figure 2-1: BlueCat Linux Development Directory Tree Structure

## Kernel Tree

This section contains a description of the contents of the kernel tree.

### Kernel Subdirectory

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

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

The structure of the BlueCat Linux kernel subdirectory is the same as that of the “standard” Linux kernel, version 2.2. The location of the contents is similar to that of the Linux kernel subdirectory. As a reminder, Table 2-1 shows important areas of the Linux kernel subdirectory:

Table 2-1: Kernel Subdirectory Contents

Kernel Subdirectory	Contents
linux/init	Functions needed to start the kernel
linux/kernel	Kernel core and system calls
linux/mm	Memory management
linux/fs	Implementations of various filesystems supported by Linux kernel
linux/ipc	Sources for System V IPC, such as semaphores, shared memory, and message queues
linux/net	Implementation of various network protocols (TCP/IP, ARP, and so on)
linux/lib	Some standard C library functions
linux/modules	The directory in which runtime modules are held
linux/include	Kernel-specific header files
linux/drivers	Device drivers for hardware components, divided into subdirectories according to the device type
linux/arch/i386	Architecture-dependent code for Intel 386 and compatible processors
linux/arch/ppc	Architecture-dependent code for PowerPC processors

Table 2-1: Kernel Subdirectory Contents (Continued)

Kernel Subdirectory	Contents
linux/arch/arm	Architecture-dependent code for ARM processors
linux/arch/sh	Architecture-dependent code for SH processors
linux/arch/mips	Architecture-dependent code for MIPS processors

## BlueCat Linux Kernel RPM Packaging

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

Table 2-2: Kernel Subdirectory Packages

Package	Description
kernel_trg-headers-2.2.12-1	C header files for the BlueCat Linux kernel
kernel_trg-source-2.2.12-1	Source code files for the BlueCat Linux kernel and kernel object files built for the default configuration of the kernel
kernel_trg- <del>target</del> -2.2.12-1	Linux kernel binary built for the default configuration of the BlueCat Linux kernel
kernel_trg-doc-2.2.12-1	References to options that can be passed to the BlueCat Linux kernel at load time
kernel_trg-bcboot-2.2.12-1	BlueCat Linux boot record template used for copying BlueCat Linux on a hard disk or a floppy
kernel_trg-pcmcia-cs-2.2.12-1	BlueCat Linux loadable kernel modules, the daemon and device drivers for PCMCIA adapters (x86 target board only)

## BlueCat Linux Kernel v/s “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 package. These are available on the BlueCat Linux Source Architecture CD-ROM in `BlueCat_target-qpu/target/SRPMS`.

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

SRPMS File	<code>kernel_trg-2.2.12-1.src.rpm</code>
Patch	<code>linux-bc.patch.gz</code>
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 Board 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 embedded target boards by including them in the target board filesystem.

## 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 embedded target boards.

## Demo Systems

The BlueCat Linux distribution package contains a number of prebuilt, ready-to-run BlueCat Linux 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 filesystem 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 subdirectory corresponds to a specific demo system. Each demo system demonstrates a particular feature of BlueCat Linux. Refer to Chapter 4, “Demo Systems” for a detailed description of all BlueCat Linux demo systems included in the distribution.

---

*NOTE: Documented demo systems may be board specific, and may not be included in all Target Support Packages (TSPs). For details regarding demo systems supported on a specific target board, please refer to the relevant Target Support Guide (TSG).*

---

## OS Loader

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

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

---

# BlueCat Linux Embedded System

A typical BlueCat Linux embedded system has two components:

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

The BlueCat Linux development tools enable the creation of kernel and filesystem images suitable for booting and downloading onto an embedded target board, and the creation of application programs for BlueCat Linux.

## Development Process Overview

The flow chart in Figure 2-2 shows the major steps of the BlueCat Linux development process.

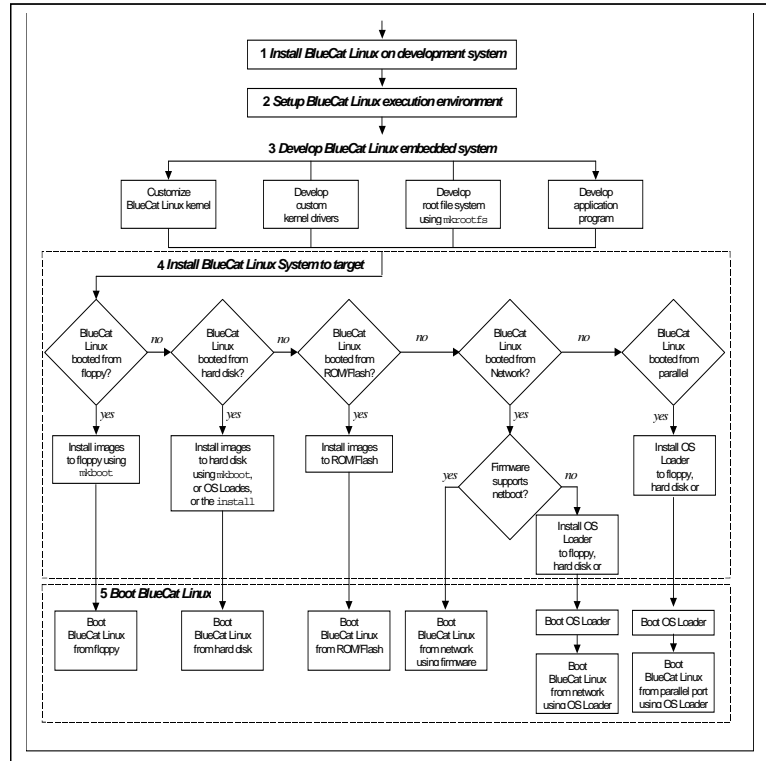


Figure 2-2: BlueCat Linux Development Flow Chart

As shown in the flow chart, the procedure for developing an embedded BlueCat Linux system is:

1. Install BlueCat Linux on the cross development host. The installation procedure is described in detail in Chapter 1, “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. For additional details, refer to Chapter 1, “Installation.”

3. Develop a custom BlueCat Linux system for the embedded target board using the cross development tools. The actual development may involve any of the following activities:
  - Customizing the BlueCat Linux kernel configuration
  - Developing custom kernel drivers and features
  - Developing custom application programs
  - Creating the root filesystem 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 embedded 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 download procedure is described in Chapter 3.

5. Boot BlueCat Linux on the target board. Depending on the boot device on 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

boots the BlueCat system images from the same or alternative boot device.

Refer to in Chapter 3 for a detailed description of the BlueCat Linux boot procedure.

---

## Customizing the BlueCat Linux Kernel

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

### Configuring the Kernel

The need to configure the kernel depends on the nature of the user's embedded system and application programs. The BlueCat Linux installation includes a complete, fully prebuilt kernel that corresponds to the default set of kernel features.

The default configuration of the BlueCat Linux kernel may be appropriate for the user's embedded system. Users should refer to the *Target 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 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

## Default Kernel Configuration

Refer to the appropriate *Target Support Guide* for a complete description of the default configuration of the BlueCat Linux kernel.

## Kernel Configuration Procedure Overview

The BlueCat Linux kernel can be configured in the same way that the “standard” Linux kernel v. 2.2 is configured. The configuration procedure consists of the following steps:

1. The configuration takes as input the file `.config` located in `$BLUECAT_PREFIX/usr/src/linux` directory and interrogates it to see which components are to be included in the kernel. The `.config` file holds the definition of the kernel configuration options and current assignments.
2. The configuration initiates an interactive interface that leads the user through the kernel configuration options and allows changes to current settings.
3. Updated kernel settings are stored back in the `.config` file in the `$BLUECAT_PREFIX/usr/src/linux` directory. A number of kernel header files are updated to reflect the changes in the kernel configuration.

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

As mentioned earlier, the BlueCat Linux distribution 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 (such as a hardware device driver) has been added.

## Building the Kernel for an x86 Target Board

A BlueCat Linux kernel can be built in three different ways:

- It can be built to boot from a floppy disk or hard disk.

- It can be built for downloading directly to the target board using OS Loader.
- It can be built to boot from target ROM/flash memory.

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 *Target Support Guide*

Building the Kernel to Boot from Floppy Disk 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 `$BLUECAT_PREFIX/usr/src/linux` directory and leaves the new `bzImage` file in the `$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 to a target board, which, therefore, does not contain a boot sector.

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

The `bvmlinux.out` file is then downloaded directly to the target board, uncompressed, and booted.

Building the Kernel to Boot from Target ROM/Flash Memory

The same `bvmlinux.out` image is used to create an image programmable 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 the 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.

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

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. Please refer to Appendix E, “`mkkernel` Command Reference.”

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

## Debugging the Kernel

GDB is used to debug BlueCat Linux device drivers and kernel code from the BlueCat Linux cross development host.

### Kernel Debugging Requirements

To use the cross GDB for kernel debugging purposes, the following items are required:

- The target board BlueCat Linux with the kernel debugger enabled
- The BlueCat Linux cross development host
- A serial line connection between the BlueCat Linux target board and the cross development host

### Building a Kernel for Debug Purposes

To build a BlueCat Linux kernel for debugging purposes 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.

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.

### Debugging

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. The kernel debugger is configured using the `CONFIG_BLUECAT_KDBG` option. To build a BlueCat Linux kernel with the kernel debugger, turn this option on and then rebuild the kernel.

---

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

## 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.



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

---

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

---

## Starting Kernel Debugging

As with other components of BlueCat Linux, it is important to set up the BlueCat Linux environment before calling the cross 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:$ 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 instance:

```
(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 the cross GDB and the kernel debugger.

The cross GDB can be run both prior to or after loading a target board kernel. When running prior to the kernel, GDB waits for the synchronization signal being sent by the target board BlueCat Linux kernel debugger at the earlier stages of initialization. If the debugging session is begun after the target board kernel is up, the cross development 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 (i.e., 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.

## 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 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.

Table 2-3 describes additional GDB commands:

Table 2-3: Supported GDB Commands

Command Format	Example	Description
<code>info proc [addr   pid]</code>	<code>info proc 5</code>	Displays contents of a <code>task_struct</code> kernel structure for task ID <i>pid</i> , address <i>addr</i> , or current task
<code>info inode addr</code>	<code>info inode 0xc0240000</code>	Displays contents of an <code>inode</code> structure at address <i>addr</i>
<code>info block addr</code>	<code>info block 0xc0240000</code>	Displays contents of a <code>buffer_head</code> structure at address <i>addr</i>
<code>info sfiles addr</code>	<code>info sfiles 0xc0240000</code>	Displays contents of a <code>files_struct</code> structure at address <i>addr</i>
<code>info file addr</code>	<code>info file 0xc0240000</code>	Displays contents of a <code>file</code> structure at address <i>addr</i>
<code>info fifo addr</code>	<code>info fifo 0xc0240000</code>	Displays contents of a <code>pipe_inode_info</code> structure at address <i>addr</i>
<code>info sblock addr</code>	<code>info sblock 0xc0240000</code>	Displays contents of a <code>super_block</code> structure at address <i>addr</i>

---

## 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 embedded target boards. The cross development tools ensure that the correct libraries included in 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 call 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.

This setup involves two components of the GDB debugger:

- The cross GDB, configured as a GDB client, runs on the cross development host.
- The `gdbserver` program, conveniently named `gdbserver`, runs on the embedded 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 `$BLUECAT_PREFIX/usr/bin`. The cross GDB binary (`gdb`) resides in `$BLUECAT_PREFIX/cdt/bin`.

Keep a copy of the application program to be debugged on 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.

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

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

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

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

For example, to use a network connection instead of a serial line, enter:

```
bash# gdbserver target_IP:port_number test_prog \  
foo.txt
```

The only difference between the two is that the first argument specifies that the user is communicating with the cross development host GDB via network. The `target_IP:port_number` argument means that `gdbserver` is to expect a network connection from the machine `target_IP` to the local network port, `port_number`. (Currently, the `target_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.

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. 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
```

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 embedded target board 1.0.0.1. 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.

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 Filesystem

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

### BlueCat Linux Root Filesystem Utility – `mkrootfs`

`mkrootfs` is a cross development tool that creates a single file, the image containing a root filesystem 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 filesystem image that can be directly booted onto the target board. When the kernel is booted on the target board, it automatically unpacks the filesystem 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 Flash File System (FFS) image that can be downloaded into target flash memory and then used to boot the system.

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

## mkrootfs Specification File

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

The `mkrootfs` specification file syntax includes a comprehensive set of commands to create a minimal filesystem 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 filesystem 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 filesystem.

If the `-l` option is specified in the call to `mkrootfs`, `mkrootfs` automatically includes all the shared libraries required for all the tools and programs included in an embedded system’s root filesystem. This is done by scanning each executable and including in the filesystem all the shared libraries on which it depends. For more information, refer to Appendix C, “mkrootfs Command Reference.”

## Images Created by mkrootfs

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

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

If neither the `-T` or `-J` option is specified, `mkrootfs` creates a compressed root filesystem 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 filesystem image to RAM and mounts it as the root filesystem.

---

## 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 filesystem containing all the tools and custom programs used by the embedded application; the root filesystem 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 filesystem images.

---

## Optimizing Footprint

### Customizing the Kernel for Size

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

## Using mkrootfs to Build a Minimal Filesystem

The `mkrootfs` tool lets the user handpick files and directories included in the target board filesystem used by the embedded BlueCat Linux system. If the size of the target board filesystem 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 filesystem and there are files in that directory that are not needed for the application, use the `rm` command to remove them. Alternatively, use the `cp` command to 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 filesystem. 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 filesystem. With this option, each executable is scanned and all the shared libraries it depends upon are added to the target board filesystem. Furthermore, after the target board filesystem is built, `mkrootfs` runs the `ldconfig` utility to create the cache and all appropriate symbolic links.

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

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

## Using Static Libraries

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

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

## Using the Memory Sizing Benchmark

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

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

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

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

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

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

`/proc/memstattotal` contains the following fields:

Total	Total amount of RAM controlled by BlueCat Linux
Used	Memory allocated by the kernel for internal data, processes data, caches, and others
Required	Memory allocated by the kernel excluding memory allocated for the buffer and page caches (Minimum recommended size for each cache is also displayed.)
Buffers	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
VMAlloc	Kernel memory allocated via <code>vmalloc</code>
KMAlloc	Kernel memory allocated via <code>kmalloc</code>

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 (i.e., process memory that cannot be swapped out to disk)

Mapped	Size of memory regions mapped from files (including the binary image and libraries)
Shared	Memory shared with other processes

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

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

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

```
nnnnnnnn 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.*

---

# *Downloading and Booting BlueCat Linux*

This chapter explains the downloading and booting of BlueCat Linux on embedded systems. The BlueCat Linux OS Loader is discussed in detail.

---

## BlueCat Linux Boot Procedure Overview

BlueCat Linux can be booted on the target board using one of the following boot scenarios:

- Copy BlueCat Linux onto a floppy disk and then boot BlueCat Linux onto the embedded 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 embedded target board from a network using the BlueCat Linux OS Loader.
- Boot BlueCat Linux onto the embedded target board from a network using the target board firmware.
- Boot BlueCat Linux onto the embedded target board from a network or parallel port using the BlueCat Linux OS Loader.

---

## mkboot Cross Development Tool

The `mkboot` cross development tool can be used to copy 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 filesystem image to the media; the compressed filesystem is loaded by the kernel into RAM and mounted as the root filesystem. The filesystem image is stored on the media following the compressed kernel image
- Defining for the kernel the root filesystem that must be mounted on the boot

Additionally, `mkboot` can be used to create a BlueCat Linux image composed of a Linux kernel and a compressed filesystem. 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 Appendix D, “mkboot Command Reference.”

---

## BlueCat Linux Boot Scenarios

### Booting BlueCat Linux from a Floppy Disk

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

## Copying BlueCat Linux 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:

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

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

```
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:

<code>fd0</code>	Floppy disk; for Windows host, replace with <b>A:</b>
<code>-b</code>	Copies the BlueCat Linux boot sector
<code>-k</code>	Copies the compressed kernel to the medium
<code>-f</code>	Copies the compressed root filesystem image to the medium
<code>-r</code>	Sets the device node on the target board to mount as the root filesystem or uncompress the filesystem image

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

## Booting BlueCat Linux on x86 from Floppy Disk

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

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 filesystem is mounted from a hard disk, a Flash File System (FFS), a network filesystem using NFS, or the compressed root filesystem image on the floppy disk is decompressed into a RAM disk and mounted as the root filesystem.

## Booting BlueCat Linux from Hard Disk

This section explains how to boot BlueCat Linux onto the embedded target board from a hard disk.

### Copying onto Hard Disk from Cross Development Host

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

1. Attach the hard disk 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. For example:

```
# 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 filesystem, the copy onto the hard disk is exactly the same as the for floppy disk, except that the hard disk device node for x86 must be used instead of the floppy device node in all the commands. Refer to “Copying BlueCat Linux onto Floppy Disk” on page 49 for details.

For this boot scenario this step completes the copy.

4. If the filesystem is mounted from a partition on the hard disk, create the partition using `fdisk`.
5. Create the filesystem on the newly made partition.

```
# mke2fs /dev/hda2
```

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

```
# mount /dev/hda2 /mnt1
# cd /mnt1
# tar xvf osloader.tar \
  user's/BlueCat/Linux/installation/point/demo/osloader/
```

7. Unmount the hard disk:

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

8. 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.

- **On an x86:**

```
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 filesystem.

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

## Copying BlueCat Linux onto Hard Disk using OS Loader

To copy BlueCat Linux onto a hard disk attached to the target board using the OS Loader, use the following procedure:

1. Attach the hard disk to the target board.
2. Boot the OS Loader on the target board from any of the boot devices supported on it. The OS Loader demo system can be found in `$BLUECAT_PREFIX/demo/osloader`.

---

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

---

3. A successful boot brings up the BLOSH 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/disk.disk  
> set FILE tftp /tftpboot/disk.tar
```

4. Create a partition to hold the kernel. For example:

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

5. 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 filesystem using `fdisk`.
7. Create a filesystem 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 filesystem and copy it from the TFTP server:

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

10. Copy the kernel image, specifying the root filesystem.

- **On an x86:**

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

11. Remove the floppy disk and reset the target board:

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

## Copying BlueCat Linux with Compressed Root Filesystem to Hard Disk

The procedure in “Copying BlueCat Linux onto Hard Disk using OS Loader” describes copying a root filesystem into a partition on the hard disk.

Alternatively, BlueCat Linux can be booted using a compressed root filesystem image copied onto a hard disk.

To copy BlueCat Linux with a compressed root filesystem onto a hard disk attached to the target board using the OS Loader, use the following procedure:

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

---

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

---

3. A successful boot displays the BLOSH 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/disk.disk
> set RFS tftp /tftpboot/disk.rfs
```

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

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

5. 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 filesystem images on the hard disk. For example:

- **On an x86:**

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

7. Remove the floppy disk and reset the target board:

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

## Copying to Hard Disk using install Demo System

To copy BlueCat Linux on a hard disk attached to the target board with the `install` demo system, use the following procedure:

1. Attach the hard disk to the target board.
2. Boot the `install` demo system on the target board from any of the boot devices supported on it, for example, a floppy disk. The `install` demo system can be found in the `BLUECAT_PREFIX/demo/install` directory.

---

**NOTE:** *Make sure that support for the hard disk used is configured in the `install` demo system. If necessary, reconfigure `install` to add support for the hard disk.*

---

3. A successful boot brings up the shell prompt on the target board console.
4. Create a boot partition to hold the kernel. For example:

```
# 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.*

---

5. Create a second partition to hold the root filesystem using `fdisk`.
6. Create a filesystem on the newly made partition:

```
# mke2fs /dev/hda2
```

7. Mount the partition on the hard disk:

```
# mount /dev/hda2 /mnt
# cd /mnt
```

8. Configure the network. For instance:

```
# ifconfig eth0 172.16.1.54
```

9. Untar the root filesystem created by `mkrootfs -T` by copying it from the TFTP server. For instance, the following command copies the root filesystem of the `xclock` demo system:

```
# tftp_i 172.16.1.2 \
/tftpboot/xclock.tar | tar xvf -
```

10. Copy the kernel image, copying it to a file in the newly created filesystem on the hard disk. For instance, the following command Copies the compressed kernel image for the `xclock` demo system:

```
# tftp_i 172.16.1.2 \
/tftpboot/xclock.disk > /mnt/xclock.disk
```

11. Copy the BlueCat Linux boot sector and kernel to the hard disk using the `mkboot` tool.

- **On an x86:**

```
# mkboot -b /dev/hda
# mkboot -k /mnt/xclock.disk /dev/hda
# mkboot -r /dev/hda2 /dev/hda
```

12. Remove the floppy disk and reboot the target board:

```
# sync
# sync
# reboot -f
```

The entire `install` demo system is available in full source in the BlueCat Linux installation. It can be used as a starting point for customizing the installation procedure. The `install` demo system includes the full source of the modified non-interactive `tftp` utility (`tftp_i`), which sends the file being received to the standard output.

## Booting from Hard Disk on an x86 Target Board

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 filesystem is mounted from a hard disk or a network filesystem using NFS, or the compressed root filesystem image on the hard disk is decompressed into a RAM disk and mounted as the root filesystem.

## 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 driver for Linux 2.2.x i386 v1.21 from the M-Systems Web site ([www.m-sys.com](http://www.m-sys.com)).
2. Unpack the driver as described in the DiskOnChip documentation from the top of the BlueCat Linux installation:

```
BlueCat:$ cd $BLUECAT_PREFIX
BlueCat:$ tar xzf \
user's/archive/location/driver.tgz
```

3. In the BlueCat Linux embedded system, turn on the DiskOnChip kernel configuration option (`CONFIG_BLK_DEV_GENERIC_FLASH_DOC`). For example, run `make xconfig` on the kernel configuration and enable the `DiskOnChip support` option in the Block Devices submenu.
4. Build the driver:

```
BlueCat:$ cd $BLUECAT_PREFIX/usr/src \
/linux/drivers/block/flash_doc
BlueCat:$ make
```

5. 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 62 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 filesystem. 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 57) DiskOnChip support in the `osloader` or `install` demo systems can be enabled 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 provided by M-Systems to ensure a correct search sequence for the disk boot devices on the target board, as dictated by the existing application. These

issues are described in detail in the documentation that comes with the DiskOnChip hardware.

---

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

---

## Booting BlueCat Linux from Target ROM/Flash Memory

This section explains how to boot BlueCat Linux onto the embedded target board from target ROM/flash memory.

### Downloading into Target ROM/Flash Memory using Firmware

This boot scenario assumes that the firmware has a user command or an equivalent feature for passing control to target ROM/flash memory. BlueCat Linux can be downloaded into target ROM/flash memory using either target board firmware, OS Loader, or the `install` demo system.

To download BlueCat Linux into 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 filesystem. For instance, the following command creates a BlueCat Linux image for the `Hello, world` demo system:

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

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

## Downloading into Target ROM/Flash Memory using OS Loader

This section explains downloading BlueCat Linux into target ROM/flash memory using the OS Loader.

### Downloading Kernel and Filesystem Image using OS Loader

This section explains how to download a BlueCat Linux image composed of a *compressed kernel image* and a *compressed filesystem* into target ROM/flash memory. Such an image can be created on the cross development host using the `mkboot -m` command. For instance, the following command creates the BlueCat Linux image for the `Hello, world` demo system:

```
BlueCat:$ mkboot -m -k hello.kernel -f \  
hello.rfs hello.kdi
```

The `demo` area of the BlueCat Linux distribution includes prebuilt BlueCat Linux images composed of a kernel image and a filesystem. These can be tried for downloading BlueCat Linux images into target ROM/flash memory, without the need to rebuild them on the cross development host.

To download a *composite BlueCat Linux image* into 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.
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 about to be downloaded. For example:

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

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

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

5. Reset the target board:

```
> reset
```

### Downloading a Kernel and FFS-Based Root Filesystem using OS Loader

The procedure in “Downloading Kernel and Filesystem Image using OS Loader” shows the use of a compressed root filesystem contained in a composite BlueCat Linux image. Alternatively, a root filesystem can be downloaded to a Flash File System (FFS), and the kernel then made to mount it at boot.

### Downloading an FFS Image Built on the Cross Development Host using OS Loader

To download BlueCat Linux with an FFS root filesystem 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 filesystem. Instead, specify the device node number of the target flash memory partition into which the FFS root filesystem is to be downloaded. The following example assumes that the root filesystem is to be downloaded in the second partition:

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

2. Prepare the FFS image of the root filesystem using the `mkrootfs` utility. For example:

```
BlueCat:$ mkrootfs -lvJ hello.spec \  
hello.jffs
```

3. Boot the OS Loader on the target board. The boot can be from a floppy disk, network, or any other boot device.
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 FFS root filesystem image. Make sure that the sizes of the first and the second partition are large enough to hold the kernel image and the FFS 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 image. For example:

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

6. Download the BlueCat Linux kernel image, specifying the target 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 `FILE` environment variable to point to the FFS image to be downloaded into the second flash memory partition:

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

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

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

9. Reset the target board:

```
> reset
```

## Creating a Root Filesystem in FFS on the Target Board using OS Loader

The procedure in “Downloading an FFS Image Built on the Cross Development Host using OS Loader” demonstrates downloading a root filesystem into target flash memory as a prebuilt Flash File System image. Alternatively, a root filesystem can be downloaded into target flash memory using the `tar` archive of the filesystem and runtime target board FFS management tools.

To download BlueCat Linux into target flash memory with an FFS root filesystem 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 filesystem. Instead, specify the device node number for the flash memory partition on which the FFS root filesystem is to be downloaded. The following example assumes that the root filesystem is to be downloaded in the second flash memory partition:

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

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

```
BlueCat:$ mkrootfs -lvT hello.spec \  
hello.tar
```

3. Boot the OS Loader on the target board. The boot can be from a floppy disk, network, or any other boot device.
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 filesystem image. Make sure that the size of the first and second partitions is large enough to hold the BlueCat Linux kernel image and the tar root filesystem 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/hello.kdi
```

6. Download the BlueCat Linux kernel image, specifying the target flash memory partition created it. For example, 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 on which the root filesystem 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 filesystem. For example:

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

9. Mount the target flash memory partition created for the root filesystem as an FFS. For example:

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

10. Untar the root filesystem into the FFS copying it from the TFTP server:

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

11. Unmount the target flash memory partition:

```
> cd /
> umount /mnt
```

12. Reset the target board:

```
> reset
```

## Downloading into Target ROM/Flash Memory using install Demo System

This section explains downloading BlueCat Linux into target ROM/flash memory using the `install demo` system.

### Downloading Kernel and Filesystem Image using install Demo System

To download a composite BlueCat Linux image into target ROM/flash memory using the `install demo` system, use the following procedure:

1. Boot the `install demo` system on the target board. A successful boot brings up the shell command prompt on the target board console.
2. Partition the target flash memory device using the `flash_fdisk` utility. This requires the creation of a partition that resides at the beginning of flash memory and large enough to hold the BlueCat Linux image. The precise geometry of the flash memory 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.
3. 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 flash memory and is intended to keep a BlueCat Linux image of up to 640 KB:

```
# flash_fdisk /dev/mtdchar0 0-9:10-15
```

4. Empty the partition into which the BlueCat Linux image is to be downloaded. For example:

```
# flash_erase /dev/mtdchar1
```

5. Configure the network. For instance:

```
# ifconfig eth0 172.16.1.54
```

6. Download the BlueCat Linux image, copying it to the device file corresponding to the appropriate target flash memory partition. For example:

```
# tftp_i 172.16.1.2 /tftpboot/hello.kdi \  
> /dev/mtdchar1
```

## 7. Reset the target board:

```
# reboot -f
```

Downloading Kernel and FFS-Based Root Filesystem using install Demo System

The procedure in “Downloading Kernel and Filesystem Image using install Demo System” shows the use of a compressed root filesystem contained in a composite BlueCat Linux image. Alternatively, a root filesystem can be downloaded to a Flash File System (FFS), and the kernel then made to mount it at boot.

Downloading FFS Image Built on Cross Development Host using install Demo System

To download BlueCat Linux using an FFS root filesystem image built on the cross development host, use the following procedure:

1. Prepare a BlueCat Linux image composed of a compressed kernel image, but not including a compressed filesystem image. Instead, specify the device node number for the target flash memory partition into which the FFS root filesystem is to be downloaded. The following example assumes that the root filesystem is to be downloaded in the second partition:

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

2. Prepare the FFS image of the root filesystem using the `mkrootfs` utility. For example:

```
BlueCat:$ mkrootfs -lvJ hello.spec \  
hello.jffs
```

3. Boot the `install` demo system on the target board. A successful boot brings up the shell command prompt on the target board console.
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 FFS root filesystem image. Make sure that the size of the first and second partitions is large enough to hold the BlueCat Linux image and the FFS images, respectively. For example:

```
# flash_fdisk /dev/mtdchar0 0-4:5-10
```

5. Empty the partitions into which the BlueCat Linux kernel and root filesystem images is to be downloaded. For example:

```
# flash_erase /dev/mtdchar1
# flash_erase /dev/mtdchar2
```

6. Configure the network. For instance:

```
# ifconfig eth0 172.16.1.54
```

7. Download the BlueCat Linux kernel image copying it to the device file corresponding to the appropriate partition. For example:

```
# tftp_i 172.16.1.2 /tftpboot/hello.kdi \  
> /dev/mtdchar1
```

8. Download the FFS image copying it to the device file corresponding to the appropriate partition. For example:

```
# tftp_i 172.16.1.2 /tftpboot/\  
hello.jffs > /dev/mtdchar2
```

9. Reset the target board:

```
# reboot -f
```

### Creating a Root Filesystem in FFS on the Target Board using install Demo System

The procedure in “Downloading FFS Image Built on Cross Development Host using install Demo System” shows the downloading of a root filesystem into target flash memory as a prebuilt Flash File System image. Alternatively, a root filesystem can be downloaded into target flash memory using the tar archive of the filesystem and runtime target board FFS management tools.

To download BlueCat Linux into target flash memory with an FFS root filesystem created from the `tar` archive on the target board, use the following procedure:

1. Prepare a BlueCat Linux image composed of a compressed kernel image, but not including a compressed filesystem image. Instead, specify the device node number for the target flash memory partition into which the FFS root filesystem is

to be downloaded. The following example assumes that the root filesystem is to be downloaded in the second partition:

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

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

```
BlueCat:$ mkrootfs -lvT hello.spec \  
hello.tar
```

3. Boot the `install` demo system on the target board. A successful boot brings up the shell command prompt on the target board console.
4. Partition the target flash memory device using the `flash_fdisk` utility. Create at least two partitions: one for the BlueCat Linux kernel image, another for the root filesystem image. For example:

```
# flash_fdisk /dev/mtdchar0 0-4:5-10
```

5. Empty the partitions into which the BlueCat Linux kernel and root filesystem images is to be downloaded. For example:

```
# flash_erase /dev/mtdchar1  
# flash_erase /dev/mtdchar2
```

6. Configure the network. For instance:

```
# ifconfig eth0 172.16.1.54
```

7. Download the BlueCat Linux kernel image, copying it to the device file corresponding to the appropriate partition. For example:

```
# tftp_i 172.16.1.2 /tftpboot/hello.kdi \  
> /dev/mtdchar1
```

8. Mount the target flash memory partition into which the root filesystem is to be downloaded:

```
# mount /dev/mtdblock2 /mnt -t jffs  
# cd /mnt
```

9. Untar the root filesystem created by `mkrootfs -T` by copying it from the TFTP server. For instance, the following

command downloads the root filesystem of the `hello` demo system:

```
# tftp_i 172.16.1.2 /tftpboot/hello.tar \  
| tar xvf -
```

10. Unmount the target flash memory partition:

```
# cd /  
# umount /mnt
```

11. Reset the target board:

```
# reboot -f
```

## Booting from Target ROM/Flash Memory using Firmware

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

Refer to the appropriate *Target Support Guide* for the embedded 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 into 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 target flash memory into RAM and begins the OS bootstrapping process.
4. If a root filesystem image is programmed into target flash memory, the kernel decompresses it into a RAM disk and mounts it as the root filesystem. Otherwise, a root filesystem is mounted from a hard disk or an NFS cross development host.

## Booting from Extension BIOS on x86

This section explains booting BlueCat Linux on the x86 target board from an extension BIOS.

## Downloading BlueCat Linux into Extension BIOS

To download BlueCat Linux into 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.

---

**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 one of the reference target boards based on the x86 architecture (PC-680 Mobile Industrial Computer™ of Octagon Systems Corporation). It may be necessary to make changes in the Boot BIOS code to support specifics of custom target board hardware. Refer to the comments in the source files of the ROM Boot BIOS residing in the `$BLUECAT_PREFIX/usr/src/linux/arch/i386/boot/romboot` directory.*

---

2. Create a BlueCat Linux image composed of a compressed kernel and a compressed filesystem. To do this, use the `mkboot -m` command on the cross development host. For instance:

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

creates an image, `hello.kdi`, that can be programmed into target ROM/flash memory.
3. Program the image created in the previous step into target flash memory using the flash memory and BIOS management tools provided with target board hardware.
4. Enable support for extension BIOS on the target board, either in BIOS or using on-board jumpers, depending on target board hardware.
5. Reset the target board. BlueCat Linux boots from target ROM/flash memory.

## Booting from Extension BIOS

On a target Intel based PC 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 filesystem image in the RAM is decompressed into a RAM disk, and is mounted as the root filesystem.

## Booting BlueCat Linux over Network or Parallel Port

This section explains booting BlueCat Linux on an embedded target board from a network or a parallel port.

### Booting over Network using Target Board Firmware

This section explains booting BlueCat Linux onto the embedded target board over a network using the target board firmware. This boot scenario assumes that the firmware has a command or an equivalent feature that allows booting of an image from a TFTP cross development host.

### Creating an Image for Booting from a Network 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 filesystem suitable for booting onto a PowerPC target board from a network using firmware. For instance:

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

creates an image (`hello.kdi`) suitable for booting from a network using firmware.

### Using Firmware to Boot BlueCat Linux over a Network

To boot BlueCat Linux over a network using target board firmware, use an appropriate firmware command. Refer to the appropriate *Target Support Guide* for a detailed description of the network boot firmware commands on a specific target board.

### Bootting over Network or Parallel Port using OS Loader

This section explains booting BlueCat Linux onto an embedded 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.

#### Using OS Loader to Boot BlueCat Linux

The BlueCat Linux OS Loader provides a command interface to boot an embedded target board with BlueCat Linux. Refer to the section entitled “BlueCat Linux OS Loader Overview” 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 filesystem image into the target board memory.
2. OS Loader shuts down its kernel.
3. OS Loader moves the loaded kernel and filesystem images into 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 filesystem image in RAM is decompressed into a RAM disk, and is mounted as the root filesystem.

---

*NOTE: The procedure for booting BlueCat Linux on a target board using OS Loader is also applicable to the `i_osloader` demo system.*

---

---

## BlueCat Linux OS Loader Overview

The BlueCat Linux OS Loader is an embedded BlueCat Linux system enhanced with the ability to boot another BlueCat Linux system on the target board. The OS Loader is optimized for a small memory footprint, while keeping the ability to support advanced booting of BlueCat Linux.

The OS Loader is capable of booting BlueCat Linux over a network, using such protocols as BOOTP, TFTP and NFS. 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—since the user does not need to download BlueCat Linux onto the target board every time the system or user applications are updated.

Once system development is complete, the OS Loader supports download of a BlueCat Linux system on the hard disk or flash memory of the embedded target board. Once BlueCat Linux is downloaded on the target

board hard disk or flash memory, the OS Loader can be removed and the final system can boot up directly from the disk.

Structurally, the OS Loader is a combination of the BlueCat Linux kernel and the BLOSH command interpreter. BLOSH is started as an `init` process and provides a command interface implementing booting facilities.

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 implements the user interface to BlueCat Linux booting facilities.

### 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 kernel has mounted the root filesystem in the RAM disk.

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

Having completed the processing of `/etc/blosh.rc`, BLOSH enters interactive mode and prompts the user for command input. 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 BlueCat Linux kernel or explicitly defined by the user via the BLOSH `set` command.

The following is a list of the environment variables 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 filesystem by the <code>read</code> command—This variable has the same format as the KERNEL environment variable. If IP auto-configuration is enabled in the BlueCat Linux OS Loader, this variable is set automatically.
HOME	Default working directory
HOST	IP address of the network cross development host from which BLOSH downloads BlueCat Linux images—If IP auto-configuration 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 auto-configuration is enabled in the BlueCat Linux OS Loader, this variable is set automatically.
KERNEL	BlueCat Linux kernel image to be loaded by the <code>boot</code> command—Format of this variable is as follows:  <i>boot_type type_specific_parameters</i>  The following <i>boot_types</i> are supported:
<i>file filename</i>	Boots image from the specified file in the local filesystem
<i>tftp filename</i>	Boots image from the specified file on the TFTP server
<i>nfs directory filename</i>	Boots image from the specified file in a specified directory on the NFS server
<i>pftp filename</i>	Boots image from the specified file on the PFTP server

<i>PPORT</i>	Name of a character device that represents a parallel port to load BlueCat Linux images from—In the standard configuration this device is <code>/dev/bpar0</code> .
<i>RFS</i>	BlueCat Linux root filesystem image loaded by the <code>boot</code> command—This variable has the same format as the <code>KERNEL</code> environment variable.

## 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. For example, `boot` can be abbreviated as `b`, `bo`, or `boo`.

### `boot` – Booting a BlueCat Linux Kernel

`boot`

The `boot` command boots a BlueCat Linux system. The location of the kernel image and optional root filesystem image, as well as the kernel boot parameters, are specified by their respective environment variables.

If the root filesystem is specified, the booted kernel loads the filesystem image into RAM and mounts it as a root filesystem. If the root filesystem 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 filesystem. This can be, for instance, a filesystem on a local disk, or an NFS-based filesystem.

If booting from the network, the networking-related environment variables must be set to appropriate values. Also, the network server machine (either 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 to the target board.

The following command sequence demonstrates booting a BlueCat Linux system from a TFTP server. Both kernel and root filesystem images are specified:

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set KERNEL tftp /tftpboot/kernel
> set RFS tftp /tftpboot/rootfs.gz
> boot
```

## 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.

## exec – Execute a Program

```
exec [-r] program [params]
```

The `exec` command executes the specified program found in the BlueCat Linux root filesystem as a new process. If the `-r` flag is specified, the new program completely replaces BLOSH in RAM. The `params` string, if provided, is passed to the process as the parameters.

For instance, the following command shows the contents of the BlueCat Linux OS Loader `root` directory.

```
> exec /bin/ls -lt /
```

(This example assumes that the `ls` utility is contained in the `/bin` directory, which is not the default case. However, arbitrary utilities and files can be added to the BlueCat Linux OS Loader filesystem. See “Configuring OS Loader as a Demo System” under the section “Configuring BlueCat Linux OS Loader” for details.)

## flash – Program 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 – 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.

## mkboot – Create 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 filesystem 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 filesystem image to be downloaded
CMD	Specifies the kernel command line

The following command sequence shows the downloading of a BlueCat Linux kernel and root filesystem onto a hard disk for an x86 target board. The kernel boots from a hard disk, uncompresses the filesystem in the RAM, and mounts it as the root filesystem.

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set KERNEL tftp /tftpboot/disk.disk
> set RFS tftp /tftpboot/disk.rfs
> mkboot -b -r /dev/hda /dev/hda
```

## mount – Mount a Filesystem

```
mount device directory
```

The `mount` command mounts a filesystem at the specified directory.

## ntar – Download and Unpack 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 filesystem 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 – 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 filesystem under the filename *file*.

This command is used to download a BLOSH script file. Alternatively, the `read` command can be used to copy an executable file to the BlueCat Linux OS Loader root filesystem.

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 – Reboot the System

`reset`

The `reset` command unconditionally shuts down the BlueCat Linux OS Loader and performs a hardware reset.

## 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, 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 – Show or Modify 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.

## IP Auto-Configuration

The BlueCat Linux kernel supports automated configuration of the station IP address and routing tables using BOOTP or RARP protocols. The BlueCat Linux OS Loader supports IP auto-configuration to set the environment variables necessary for network booting using TFTP or NFS. When IP auto-configuration succeeds, the following BLOSH environment variables are set at startup:

IP	Set to the station IP address acquired via BOOTP
HOST	Set to the IP address of the server that answered the BOOTP request
FILE	Set to the boot filename acquired via BOOTP

To enable IP auto-configuration in the OS Loader kernel, the `CONFIG_IP_PNP` and `CONFIG_IP_PNP_BOOTP` configuration options must be enabled. Note that IP auto-configuration without a properly set up BOOTP server pauses kernel loading for about one minute and no BLOSH variables are set.

## Setting Up a BOOTP Server

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

1. Information about the IP and hardware addresses of the stations to be auto-configured using the BOOTP protocol must be added to the `/etc/bootptab` file. Each station is described by a line in the following format:

```
# 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 advanced information on the `bootptab` file format, refer to the `bootptab(5)` manual page.

2. If the `bootpd` 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
```

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 `inet.conf(5)` manual page.

## Setting Up a TFTP Server

To set up the TFTP daemon on a Linux server, uncomment or add the following line to the `/etc/inetd.conf` file. This allows TFTP download from the `/tftpboot` directory:

```
tftp dgram udp wait \  
root/usr/sbin/tcpd in.tftpd /tftpboot
```

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

For detailed information on the `/etc/inetd.conf` file format, refer to the `inetd.conf(5)` manual page.

## Booting Images from a Different Subnet

To enable the BlueCat Linux OS Loader to download the kernel and root filesystem images from an NFS or TFTP server on a different subnet, use the following procedure:

1. Turn the `CONFIG_IP_PNP` kernel configuration option on in the `osloader.config` file. This is done by running `make xconfig` in the OS Loader 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:gwip: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
```

A command line is passed to the BlueCat Linux kernel using the `-c` option of the `mkboot` utility. Refer to the `mkboot(1)` manual 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 an NFS Server

To enable the BlueCat Linux OS Loader to download kernel and root filesystem images from an NFS server, the directory that the images reside in must be exported from the NFS server. To export a directory from a Linux server, a line in the following format must be added to the `/etc/exports` file:

```
/nfsboot 1.0.3.2(r,no_root_squash)
```

---

**NOTE:** *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 manual pages.

## 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. This section explains starting the PFTP server on the different cross development hosts. Please also refer to Appendix K, “pftpd Command Reference.”

To start the PFTP server, execute the following command in the BlueCat Linux environment:

```
BlueCat:~# pftpd start
```

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

```
BlueCat:~# pftpd stop
```

Additional steps are required depending on the cross development host operating system.

### Linux

Before running the PFTP server on the Linux host the low-level driver must be loaded to allow the daemon to access the parallel port.

To load the low-level driver, execute the following commands under a superuser (root) account:

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

To start the server the user must log in as a system administrator.

## Windows 98

No additional action is required to run the server.

---

**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.*

---

## Using BlueCat Linux OS Loader in Embedded Systems

This section shows some simple examples of how the BlueCat Linux OS Loader can be used in embedded systems. All examples assume that the BlueCat Linux OS Loader is already downloaded onto the target board.

### Booting from 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 cross development host. Both the kernel image as well as the root filesystem image are downloaded:

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set KERNEL tftp /tftpboot/nfsroot.kernel
> set RFS tftp /tftpboot/nfsroot.rfs
> boot
```

### Booting from 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 cross development host. Both the kernel image as well as the root filesystem image are downloaded:

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set KERNEL nfs /nfsboot nfsroot.kernel
> set RFS nfs /nfsboot nfsroot.rfs
> boot
```

On a Linux NFS server, the following line must be present in the `/etc/exports` file:

```
/nfsboot 1.0.3.2(r,no_root_squash)
```

## Mounting a Root Filesystem 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 filesystem as the root filesystem. The example assumes that the BlueCat Linux kernel is configured to mount an NFS-based filesystem (vs. mounting the RAM disk-based filesystem downloaded in the image):

```
> set IP 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set KERNEL tftp /tftpboot/nfsroot.kernel
> set CMD root=/dev/nfs \
nfsroot=1.0.3.1:/nfsrootip=1.0.3.2:1.0.3.1::::
> boot
```

## Booting from a PFTP Server

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

```
> set PPORT /dev/bpar0
> set KERNEL pftp /pftpboot/rootfs.kernel
> set RFS pftp /pftpboot/rootfs.gz
> boot
```

## Auto-Booting BlueCat Linux

This example shows the contents of a sample `/etc/blosh.rc` used to auto-boot the system from the TFTP server without any manual intervention:

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

set IP 1.0.3.2
set HOST 1.0.3.1
set IF eth0
set KERNEL tftp /tftpboot/rootfs.kernel
set RFS tftp /tftpboot/rootfs.gz
boot
```

## Target Board-Specific Auto-Boot of BlueCat Linux

This example shows how a combination of the IP auto-configuration 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 auto-configuration process sends a BOOTP request to the network.
2. The BOOTP cross development host replies with the IP addresses of the target board, the TFTP cross development host, and the target board-specific file stored in the cross development host filesystem. 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 cross development 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.

## 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 on the target board. Note that 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 1.0.3.2
> set HOST 1.0.3.1
> set IF eth0
> set FILE tftp /tftpboot/program
> read /program
> exec chmod +x /program
> exec /program
```

## Configuring BlueCat Linux OS Loader

### Configuring OS Loader as a Demo System

The BlueCat Linux OS Loader is a BlueCat Linux image composed of a BlueCat Linux kernel and a root filesystem containing the BLOSH command interpreter.

### Configuring Hardware Device Support

The BlueCat Linux OS Loader kernel must be configured with support for devices and media from which the kernel and root filesystem 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 filesystem 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

## Customizing BlueCat Linux OS Loader

### Rebuilding BLOSH

To rebuild BLOSH, execute **make** in the BLOSH source directory.

### Adding New Commands to BLOSH

Command names and functions that implement them are listed in the table contained in the `blosh_cmd.c` file in the BLOSH source directory.

Each table entry has the following structure, defined in the `blosh_cmd.h` file:

```
typedef struct blosh_cmd_entry_s
{
    const char*      name;
    blosh_cmd_result_t (*do_cmd)(int argc, char** argv);
    const char*      usage;
    const char*      description;
    char*            (*support_level)(char* buf, size_t size);
}
blosh_cmd_entry_t;
```

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

This chapter describes BlueCat Linux demo systems. They can be used to jump-start the user's own development of custom systems and applications for embedded target boards, as the source is included in the demo system.

---

**NOTE:** *The list of demos documented in Chapter 4 are supported on x86-based target boards. To determine the demos supported on specific platforms, consult the relevant Target Support Guide.*

---

---

## Demo System Conceptual Overview

Once BlueCat Linux is installed onto the cross development host, a number of prebuilt, ready-to-run demo systems that can be booted on an embedded target board become available. Each demo system displays a particular feature of BlueCat Linux.

Chapter 4 provides a detailed description of each demo system. LynuxWorks, Inc. recommends using these demo systems in order to become familiar with BlueCat Linux without having to learn the development environment in detail.

Another feature of the demo systems that is useful for developers is that each demo 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 system, or a set of demo systems that is the 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.

Since the demo systems included in BlueCat Linux span a wide range of features, each user might find a different starting point suitable to his/her custom embedded system.

The demo systems range from a simple `Hello World` single-process demo system, an advanced-networking configuration GUI-based demo system, to complex multicomputing configurations based on BlueCat Linux extensions such as `Messenger`.

---

## Demo Systems

### Demo Systems 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.

### Supported Demo Systems

The `$BLUECAT_PREFIX/demo` directory contains the subdirectories listed in Table 4-1:

Table 4-1: Demo System Requirements

Subdirectory	Demo System	Requirements
<code>caffeine</code>	Embedded CaffeineMark demo system	Storage: Medium RAM: Large Network: Yes Disk: None Special: None
<code>default</code>	Simple shell and filesystem demo system with the default kernel configuration	Storage: Small RAM: Small Network: None Disk: None Special: None

Table 4-1: Demo System Requirements (Continued)

Subdirectory	Demo System	Requirements
developer	The shell, ping, ftp, gdb, and vl_demo demo systems	Storage: Medium RAM: Large Network: Yes Disk: None Special: Host and target machines must be connected by a serial line
diskboot	Booting BlueCat Linux from a hard disk	Storage: Small RAM: Small Network: None Disk: Yes Special: None
ftp	FTP demonstration	Storage: Small RAM: Small Network: Yes Disk: None Special: None
ffs	Flash memory support and Flash File System (FFS) demonstration	Storage: Tiny RAM: Small Network: None Disk: None Special: Demo system updates target flash memory
gdb	GDB with gdbserver usage demonstration	Storage: Tiny RAM: Small Network: Yes Disk: None Special: Host and target machines must be connected by a serial line to use remote debugging via serial line
gnutar	tar demonstration	Storage: Small RAM: Small Network: None Disk: Yes Special: A partition on the disk must contain a tar file

Table 4-1: Demo System Requirements (Continued)

Subdirectory	Demo System	Requirements
hello	Hello World demo system	Storage: Tiny RAM: Tiny Network: None Disk: None Special: None
install	Installing BlueCat Linux onto a hard disk	Storage: Small RAM: Small Network: Yes Disk: Yes Special: None
kdbg	BlueCat Linux kernel debugger demonstration	Storage: Small RAM: Small Network: None Disk: None Special: Host and target machines should be connected by the serial line to use remote debugging via serial line
loadkeys	loadkeys utility demonstration	Storage: Small RAM: Small Network: None Disk: None Special: PC keyboard and display
mapm	BlueCat Linux Advanced Power Management (APM) demo system	Storage: Tiny RAM: Tiny Network: None Disk: None Special: None
memsize	Memory sizing benchmark demo system	Storage: Tiny RAM: Small Network: None Disk: None Special: None
modular	Kernel modules usage demo system	Storage: Tiny RAM: Small Network: None Disk: None Special: None

Table 4-1: Demo System Requirements (Continued)

Subdirectory	Demo System	Requirements
<code>msng_exmpl</code>	Messenger example demo system	Storage: Tiny RAM: Medium Network: None Disk: None Special: None
<code>msng_minet</code>	Messenger Network Interface demo system	Storage: Small RAM: Medium Network: None Disk: No Special: Multinode cPCI systems (CPV5350 or MCP750 board as a cPCI host; one or more MCPN750 boards as I/O controllers)
<code>multi_user</code>	Multi-user environment demonstration	Storage: Small RAM: Medium Network: Yes Disk: None Special: PC keyboard and display
<code>multi_user_net</code>	Multi-user networking environment demonstration	Storage: Medium RAM: Medium Network: Yes Disk: None Special: PC keyboard and display
<code>nfsroot</code>	Kernel image configured to mount root filesystem from NFS server	Storage: Tiny RAM: Tiny Network: Yes Disk: None Special: NFS server in the network
<code>osloader</code>	BlueCat Linux OS Loader	Storage: Tiny RAM: Tiny Network: Yes Disk: None Special: None
<code>ping</code>	<code>ping</code> demonstration	Storage: Small RAM: Small Network: Yes Disk: None Special: None

Table 4-1: Demo System Requirements (Continued)

Subdirectory	Demo System	Requirements
rcp	Remote copy demo system	Storage: Small RAM: Small Network: Yes Disk: None Special: None
rlogin	Remote log in demonstration	Storage: Small RAM: Small Network: Yes Disk: None Special: None
rootfs	Kernel image configured to mount root filesystem from disk	Storage: Tiny RAM: Tiny Network: None Disk: Yes Special: Hard disk must have a pre-installed root filesystem
shell	Shell and filesystem demonstration	Storage: Small RAM: Small Network: None Disk: None Special: None
showcase	Configures an Apache Web Server	Storage: Small RAM: Small Network: Yes Disk: None Special: None
tcl	Tcl shell demonstration	Storage: Small RAM: Medium Network: None Disk: None Special: None
tcpdump	tcpdump Ethernet snooping demo system	Storage: Small RAM: Medium Network: Yes DISK: None Special: None

Table 4-1: Demo System Requirements (Continued)

Subdirectory	Demo System	Requirements
tutorial	Process/thread tutorial	Storage: Small RAM: Small Network: None Disk: None Special: None
vl_demo	VisualLynux program demonstration	Storage: Medium RAM: Large Network: Yes Disk: None Special: The host and target machines must be connected by a serial cable to use remote debugging via serial line
xclock	X Windows demo system	Storage: Medium RAM: Large Network: None Disk: No Special: Any VGA-compatible video card, PS/2 mouse
xdemo1	X Windows color demonstration	Storage: Medium RAM: Large Network: None Disk: No Special: Any VGA-compatible video card, PS/2 mouse
xdemo2	Tcl/Tk color demonstration	Storage: Medium RAM: Large Network: None Disk: No Special: Any VGA-compatible video card, PS/2 mouse

## Demo System Components

A demo system directory contains all the files required to build the demo system. A demo system is composed of the following:

- A BlueCat Linux kernel customized for the embedded system
  - This is 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 filesystem containing the tools and custom programs required for the demo system to bootstrap and run – This is represented by a prebuilt, compressed root filesystem image and a `mkrootfs` specification file that can be used to rebuild the root filesystem. Use the `mkrootfs` tool to build the root filesystem image from the `.spec` file.
- Optionally, a demo system contains custom applications and files such as sample application programs, developed specially for the demo system.

## Contents of Demo System Directory

A typical demo system directory contains the files and subdirectories in Table 4-2:

Table 4-2: Demo System Files and Subdirectories

File/ Subdirectory	Description
<i>dema.config</i>	Kernel <code>.config</code> configuration file
<i>dema.spec</i>	<code>mkrootfs</code> specification file
<i>dema.kernel</i>	Prebuilt, compressed kernel image suitable for booting onto a target board from the network using the BlueCat Linux OS Loader
<i>dema.disk</i>	Prebuilt, compressed kernel image suitable for copying on a floppy or hard disk
<i>dema.rfs</i>	Prebuilt, compressed RAM disk root filesystem image suitable for booting onto a target board from a network using the BlueCat Linux OS Loader, or for loading from a floppy disk or hard disk
<i>dema.tar</i>	<code>tar</code> image of the root filesystem suitable for copying on a partition on a hard disk or NFS-mounting
<i>dema.kdi</i>	Image composed of the compressed kernel image ( <code>.disk</code> ) and optional compressed RAM disk root filesystem ( <code>.rfs</code> ) suitable for booting onto a target board from a network using firmware, or programming into target ROM/flash memory

Table 4-2: Demo System Files and Subdirectories (Continued)

File/ Subdirectory	Description
Makefile	Makefile to build the demo system
src	Source files of the custom programs used in the demo system
local	Configuration files specific to the demo system

## Configuring a Demo System

Each demo system has its own kernel configuration. See the relevant *demo.config* for a complete specification of the demo system kernel configuration.

### For Hardware Devices

Each demo system is configured to support the feature displayed in the kernel (e.g., `ftpd`, etc.). The user might still need to reconfigure the kernel in case the target board hardware is different from that on which the demo system 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.

Refer to descriptions of specific demo systems in Chapter 4, and the *Target Support Guide* for a BlueCat Linux target board for the specifications of the hardware supported by the prebuilt demo system kernel.

### For the Boot Device

Depending on the nature of a demo system and the boot options supported by a BlueCat Linux target board, different demo systems have support for different boot devices configured in the kernel. For instance, if the BlueCat Linux distribution is for an x86-based target board then those demo systems that can fit onto a floppy disk (1.44 MB) have support for the floppy 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/she will have to first reconfigure the kernel to add support for the hard disk.

To support booting from a floppy or a hard disk, the user may need to configure the hardware device driver for the floppy or hard disk, respectively. Refer to the description of specific demo systems in Chapter 4, and the *BlueCat Linux Target Support Guide* document for a target board for specifications of boot options supported by the demo system by default.

---

## Building Demo Systems

### 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 in Table 4-3:

Table 4-3: Typical Makefile Targets

Makefile Target	Description
<code>kernel</code>	Builds <i>demo.kernel</i> and <i>demo.disk</i>
<code>rootfs</code>	Builds <i>demo.rfs</i> and <i>demo.tar</i>
<code>kdi</code>	Builds <i>demo.kdi</i>
<code>this</code>	Builds custom programs in <code>src/</code>
<code>all</code>	Builds all of the above
<code>xconfig</code>	<ul style="list-style-type: none"><li>• Copies <i>demo.config</i> to the kernel <code>.config</code></li><li>• Calls <code>make xconfig</code></li><li>• Copies the updated kernel <code>.config</code> into <i>demo.config</i></li></ul>
<code>clean</code>	Removes all prebuilt binaries

## Running Demo Systems

A demo system on the target board can be booted from one of the devices in Table 4-4, depending on the boot options supported by the target board:

Table 4-4: Demo System Boot Devices

Boot Device	Boot Procedure	Detailed Description
<b>Floppy Disk</b>	<ul style="list-style-type: none"> <li>• Copy demo system onto a floppy from cross development host using <code>mkboot</code></li> <li>• Boot target board from floppy disk</li> </ul>	See “Booting BlueCat Linux from a Floppy Disk” on page 48 in Chapter 3.
<b>Hard Disk</b>	<ul style="list-style-type: none"> <li>• Copy demo system on a hard disk either from the cross development host using <code>mkboot</code>, or from the target board using the <code>install</code> demo system or the OS Loader</li> <li>• Boot the target board from a hard disk</li> </ul>	See “Booting BlueCat Linux from Hard Disk” on page 50 in Chapter 3.
<b>ROM/flash memory</b>	<ul style="list-style-type: none"> <li>• Download a demo system into target ROM/flash memory using firmware, external device-specific tools, or from the target board using the <code>install</code> demo system or the OS Loader.</li> <li>• Boot target board from ROM/flash memory</li> </ul>	See “Booting BlueCat Linux from Target ROM/Flash Memory” on page 59 in Chapter 3.

Table 4-4: Demo System Boot Devices (Continued)

Boot Device	Boot Procedure	Detailed Description
<b>Network or Parallel Port using OS Loader</b>	<ul style="list-style-type: none"><li>• Download the OS Loader on a floppy disk, hard disk, or into target ROM/flash memory</li><li>• Boot the OS Loader on target board</li><li>• Boot demo system from the network (TFTP or NFS) or the parallel port (PFTP) using the OS Loader</li></ul>	See “Booting over Network or Parallel Port using OS Loader” on page 72 in Chapter 3.
<b>Network using Firmware</b>	<ul style="list-style-type: none"><li>• Boot a demo system from the network using the firmware <code>netboot</code> option</li></ul>	See “Booting over Network using Target Board Firmware” on page 71 in Chapter 3.

---

## Demo Systems Reference

This section contains a detailed description of all the demo systems included in BlueCat Linux. For each system, a description and basic requirements for the target board hardware and demo system environment are provided.

---

**NOTE:** *Not all demos listed may be supported on a specific target board. Consult the relevant Target Support Guide for this information.*

---

## Demo Systems Requirements

This section contains a reference of the 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. It can be any of the following:

Table 4-5: Storage Size Options

Target Board	Tiny	Small	Medium	Large	Huge
ARM	2 MB	4 MB	8 MB	16 MB	>16 MB
x86	1 MB	2 MB	4 MB	8 MB	>8 MB
PowerPC	2 MB	4 MB	8 MB	16 MB	>16 MB
SH	1 MB	2 MB	4 MB	8 MB	>8 MB
MIPS	2 MB	4 MB	8 MB	16 MB	>16 MB

The “RAM” entry under each demo system description shows minimal requirements for system memory on the target board. It can be any of the following:

Table 4-6: Memory Size Options

Target Board	Tiny	Small	Medium	Large
ARM	4 MB	8 MB	16 MB	>16 MB
x86	4 MB	8 MB	16 MB	>16 MB
PowerPC	4 MB	8 MB	16 MB	>16 MB
SH	4 MB	8 MB	16 MB	>16 MB
MIPS	4 MB	8 MB	16 MB	>16 MB

If a demo system requires a network, it means that the target board must have an Ethernet connection to the local area network.

If a demo system requires a disk, it means that the target board must have a hard disk connected to it.

If a demo system requires a specific kernel option, 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 filesystem, `mkboot -c` must be used to pass a kernel option to the kernel command line. Refer to the manpage for `mkboot(1)` for details.

---

**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 filesystem is built to include a correct kernel command line in the image.*

---

## Simple Systems

hello

### DEMO

Hello World demo system

### SYNOPSIS

This demo system keeps printing `hello, world` on the console until the user either resets or powers down.

### DESCRIPTION

This demo system starts immediately upon booting and runs as an `init` process. The demo system contains only one executable that is statically linked, so no libraries are needed. `/sbin/init` is a symbolic link to the `hello_world` executable.

### REQUIREMENTS

Storage	Tiny
RAM	Tiny
Network	None
Disk	None
Special	None

## tutorial

## DEMO

Process/Thread tutorial

## SYNOPSIS

This demo system contains an interactive demonstration of processes and threads in the Linux environment.

## REQUIREMENTS

Storage	Small
RAM	Small
Network	None
Disk	None
Special	None

## DESCRIPTION

The system boots up in the single-user mode and opens a `bash` window. Use the command keys shown in the demo system window to start processes and threads. Use `q` to exit the demo system. Type `demo` to restart it.

## modular

## DEMO

Kernel modules usage demo system

## SYNOPSIS

This demonstrates dynamic loading and unloading of kernel modules.

## REQUIREMENTS

Storage	Tiny
RAM	Small
Network	None
Disk	None
Special	None

## DESCRIPTION

The system boots up in single-user mode. `init` starts `bash` without a login prompt. To load an example module `hello_world.o` use the `insmod` command. Once the module is loaded in the kernel, it prints the `Hello, world.` message on the system console. In order to unload the module use the `rmmod` command. Right before the module is unloaded, it prints `Goodbye!` on the system console. For example:

```
bash# insmod hello_world.o
Hello, world.
bash# rmmod hello_world
Goodbye!
bash#
```

## Shells

shell

## DEMO

Shell and filesystem demo system

## SYNOPSIS

This demonstrates the Bourne-Again Shell (`bash`), which provides a simple environment without networking support. The filesystem includes `ls`, `ps`, `reboot` and `shutdown` commands.

---

## REQUIREMENTS

Storage	Small
RAM	Small
Network	None
Disk	None
Special	None

## DESCRIPTION

This demo system runs `init` on normal boot. It executes `mount` to mount the `proc` filesystem needed for `ps` and then executes `mingetty` on the first virtual console. Log in as `root` with a blank or null password and then test included commands.

default

## DEMO

Shell and filesystem demo system with default kernel configuration

## SYNOPSIS

This demonstrates the Bourne-Again Shell (`bash`) in the context of the default kernel configuration. The filesystem includes the `ls`, `ps`, `reboot`, and `shutdown` commands.

## REQUIREMENTS

Storage	Small
RAM	Small
Network	None
Disk	None
Special	None

## DESCRIPTION

This demo system runs `init` on normal boot. It executes `mount` to mount the `proc` filesystem needed for `ps` and then executes `mingetty` on the first virtual console. Log in as a

`root` with a blank or null password and then test the included commands.

`tcl`

## DEMO

Tcl shell demonstration

## SYNOPSIS

The demo system contains the `tcl` (Tool Command Language) shell.

## REQUIREMENTS

Storage	Small
RAM	Medium
Network	None
Disk	None
Special	None

## DESCRIPTION

The demo system boots up in single-user mode. `init` starts `bash` without a login prompt. Type `tcl` to experiment with this shell.

`multi_user`

## DEMO

multi-user environment demo system

## SYNOPSIS

This is a simple multi-user environment demonstration, which can be used as a template for building multi-user oriented target boards.

## REQUIREMENTS

Storage	Small
RAM	Medium
Network	Yes
Disk	None
Special	PC keyboard and display.
Kernel Option	<code>ramdisk_size=8192</code>

## DESCRIPTION

The system boots in multi-user mode. There are three virtual consoles available. To switch between them use the usual sequences `<Alt>-F1 (2,3)`. Log in as `guest` with the password `guest`. The superuser account is available as the login `root` with the password `root`. When the `bash` prompt appears, use the basic commands (`ls`, `ps`, `mount`) to browse and control the system.

`multi_user_net`

## DEMO

multi-user networking environment demo system

## SYNOPSIS

This is a simple multi-user networking environment demonstration, which can be used as a template for building network-oriented target boards.

## REQUIREMENTS

Storage	Medium
RAM	Medium
Network	Yes
Disk	None
Special	PC keyboard and display
Kernel Option	<code>ramdisk_size=8192</code>

## DESCRIPTION

The system boots in multi-user mode. There are three virtual consoles available. To switch between them use the usual sequences `<Alt>-F1(2,3)`. Log in as `guest` with the password `guest`. The superuser account is available as the login `root` with the password `root`. When the `bash` prompt appears, use the `ifconfig` command to configure the network, for instance:

```
bash# /sbin/ifconfig eth0 173.16.1.62
```

Then use the basic commands to operate the system, with the addition of network-oriented commands (such as `rlogin`, `telnet`, `ftp`). Also, the major network daemons (`in.rlogind`, `in.telnetd`) are available via `inetd`.

loadkeys

## DEMO

Kernel keymap loading demo system

## SYNOPSIS

This demo system shows how the kernel keymap loading facility is used.

## REQUIREMENTS

Storage	Small
RAM	Small
Network	None
Disk	None
Special	PC keyboard and display
Kernel Option	<code>ramdisk_size=8192</code>

## DESCRIPTION

The system boots in multi-user mode with `mingetty` spawned on `tty1`. To log in, type `root` with an empty password. After that, use the `loadkeys` command to load different keyboard maps. For instance:

```
bash# qwerty<Ctrl+C>
bash# loadkeys fr
Loading /usr/lib/kbd/keymaps/i386/\
azerty/ fr.kmap.gz
bash# azerty<Ctrl+C>
```

This example demonstrates loading a French keyboard layout. To switch back to the initial `qwerty` keymap, use the `loadkeys defkeymap` command.

## Simple Networking

ping

### DEMO

```
ping demo system
```

### SYNOPSIS

Simple networking demonstration

### REQUIREMENTS

Storage	Small
RAM	Small
Network	Yes
Disk	None
Special	None

### DESCRIPTION

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, 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
```

rcp

## DEMO

rcp demo system

## SYNOPSIS

remote copy demo system

## REQUIREMENTS

Storage	Small
RAM	Small
Network	Yes
Disk	None
Special	None

## DESCRIPTION

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 172.16.1.62
bash# route add default gw 172.16.0.1
```

The user can `rcp` from another machine on the network:

```
bash# rcp 172.16.1.2:/some_dir/some_file\
/tmp
```

The user can `rcp` to the target board from another machine on the network:

```
[user@host user]$ rcp /some_dir/\
some_file root@172.16.1.62:/tmp
```

rlogin

## DEMO

This demonstrates the use of `rlogin`.

## SYNOPSIS

Remote log in demo system

## REQUIREMENTS

Storage	Small
RAM	Small
Network	Yes
Disk	None
Special	None

## DESCRIPTION

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 172.16.1.62
```

The user can `rlogin` to another machine on the network:

```
bash# rlogin 195.239.208.81
```

The user can `rlogin` to the target board from another machine on the network:

```
[user@host user]$ rlogin -l root \  
172.16.1.62
```

`ftp`

## DEMO

An FTP demonstration

## SYNOPSIS

This demonstrates the use of the `ftp` client.

## REQUIREMENTS

Storage	Small
RAM	Small
Network	Yes
Disk	None
Special	None
Kernel Option	<code>ramdisk_size=8192</code>

## DESCRIPTION

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 172.16.1.62
bash# route add default gw 172.16.0.1
```

The user can `ftp` to another machine on the network:

```
bash# ftp 195.239.208.81
```

The user can `ftp` to the target board from another machine on the network:

```
[user@host user]$ ftp 172.16.1.62
```

## Utility Systems

`install`

### DEMO

Installing a BlueCat Linux system onto hard disk

### SYNOPSIS

This demo system installs a BlueCat Linux system onto a hard disk.

## REQUIREMENTS

Storage	Small
RAM	Small
Network	Yes
Disk	Yes
Special	None

## DESCRIPTION

The system boots in single-user mode. `init` starts `bash`.

1. Bring up the network interface(s) manually using the `ifconfig` command and optionally, set up the kernel routing table using the `route` command.
2. Create a partition on the hard disk using `fdisk` and install the BlueCat Linux filesystem into the newly created partition using a combination of `ttfcp_i` and `tar`.
3. Install the BlueCat Linux compressed kernel image on the hard disk and make the hard disk bootable using `mkboot`.

Refer to Chapter 1, “Installation” for a detailed description of the installation procedure.

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 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`.

`memsize`

## DEMO

Memory sizing benchmark demo system

## SYNOPSIS

This demonstrates the BlueCat Linux memory sizing benchmark.

## REQUIREMENTS

Storage	Tiny
RAM	Small
Network	None
Disk	None
Special	None

## DESCRIPTION

The system boots in single-user mode. `init` mounts the `proc` filesystem and starts `bash`. Run any of the utilities present in the filesystem to let the benchmark collect process specific statistics. (The RFS contains `ls`, `ps`, and `bash` binaries). For instance:

```
bash# ps
bash# cd /; ls -laR
```

The statistics are available in the following `/proc` files:

- `/proc/memstattotal`

System-wide memory usage statistics

- `/proc/memstatproc`

Process-specific statistics for currently running processes

- `/proc/memstathist`

Process-specific statistics for finished processes

- `/proc/memstattracepage`

Traceback of kernel components that have allocated significant amount of RAM

For instance:

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

## Debuggers

`gdb`

### DEMO

Remote GDB demonstration

### SYNOPSIS

This demonstrates the debugging of an application program on the target board from remote GDB connected through a serial line or network.

## REQUIREMENTS

Storage	Tiny
RAM	Small
Network	Yes
Disk	None
Special	The host and target machines must be connected through a serial line to use remote debugging via a serial line. Serial <code>tty</code> devices (nodes) available in the <code>rootfs</code> image are <code>/dev/ttyS0</code> and <code>/dev/ttyS1</code> .

## DESCRIPTION

The system boots in single-user mode. `init` starts `bash` without the login prompt. `gdb` on the cross development host can be connected to `gdbserver` on the target board either through a network or a serial line.

1. On the target board, to connect `gdb` to `gdbserver` via network use the `ifconfig` command to bring up a network interface(s) and optionally, set up the kernel routing table using the `route` command. For instance:

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

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

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

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

```
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`). Start `gdb` specifying the program name as a parameter:

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

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

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

or use the standard GDB commands to debug `test_prog`:

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

---

**NOTE:** *Use COM2 for the Windows host.*

---

kdbg

## DEMO

Kernel debugger

## SYNOPSIS

This demonstrates debugging the BlueCat Linux kernel.

## REQUIREMENTS

Storage:	Small
RAM:	Small
Network:	None
Disk:	None
Special:	The host and target machines must be connected by a serial line to use remote debugging from the host GDB. The serial <code>tty</code> devices (nodes) available in the <code>rootfs</code> image are <code>/dev/ttyS0</code> and <code>/dev/ttyS1</code> .

## DESCRIPTION

The BlueCat Linux kernel debugger stops the kernel on the target board at an early stage of the kernel initialization process. This creates a synchronization point for the cross GDB and the kernel debugger.

As with any other component of BlueCat Linux, it is important to set up the BlueCat Linux environment before calling the cross GDB.

1. On the cross development host, go to the kernel directory (`$BLUECAT_PREFIX/usr/src/linux`) and start `gdb` by specifying the kernel image as the parameter:

```
BlueCat: bash# gdb -nw vmlinux
```

2. Use the GDB `target remote` command to connect to the kernel debugger on the target board:

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

---

**NOTE:** *Use COM2 for the Windows host.*

---

```
Remote debugging using /dev/ttyS1
0xc0123456 in kdbg breakpoint ()
(gdb)
```

3. Once communication is established, `gdb` reports the location at which the kernel was interrupted. Let the kernel continue using the GDB `continue` command:

```
(gdb) continue
Continuing.
```

4. Wait until the kernel boots in single-user mode. Then interrupt it from the cross development host using the `Ctrl-C` break symbol. The console displays:

```
Program received signal SIGTRAP, 2
Trace/breakpoint trap.
0xc0106279 in cpu_idle ()2
(gdb)
```

The kernel stops and displays an exception number:

```
bash#
Entered SKDB: exception 3
```

5. Set a breakpoint in the `sys_execve()` function (entry point in the `execve` system call implementation routine) and let the kernel continue:

```
(gdb) br sys_execve
Breakpoint 1 at 0xc0106951
(gdb) continue
Continuing.
```

6. On the target board, run the `ls` utility:

```
bash# ls
Entered KDBG: exception 3
```

7. Check that the breakpoint has been hit and remove it:

```
Breakpoint 1, 0xc0106951 in sys_execve ()
(gdb) delete 1
(gdb)
```

8. Try using the informational commands of the debugger (`info proc`, for instance) and let the kernel continue:

```
(gdb) info proc
...
(gdb) continue
Continuing.
```

9. On the target board, load the `hello_world` module using the option `-m` with the `insmod` utility to get module symbol information:

```
bash# insmod -m hello_world.o
Hello, world.
Sections:
Size      Address  Align
.this    0000004c c2800000 2**2
.text    0000002a c280004c 2**2
.rodata  0000001f c2800076 2**0
.data    00000000 c2800098 2**2
.kstrtab 0000000c c2800098 2**0
.bss     00000000 c28000a4 2**2
__ksymtab 00000000 c28000a4 2**2
Symbols:
c2800000 d __this_module
c280004c t gcc2_compiled.
c280004c t .text
c280004c t init_module
c2800064 t cleanup_module
c2800076 r .rodata
c2800098 d .data
c28000a4 d .bss
bash#
```

10. Interrupt the kernel again by typing CTRL-C and manually add the symbol information to the GDB symbol database. Note that the user should use the address of the `.text` section within the module, and not the start address of the module:

```
Program received signal SIGTRAP, Trace/breakpoint\
trap. 0xc0106279 in cpu_idle ()
(gdb) add-symbol-file hello_world.o\
0xc280004c
```

```
add symbol table from file hello_world.o
at text_addr = 0xc280004c?
(y or n) y
Reading symbols from hello_world.o...done.
(gdb)
```

11. Set a breakpoint inside the `cleanup_module` routine. Check that it has been hit and let the kernel continue:

```
(gdb) break cleanup_module
Breakpoint 1 at 0xc2800067
(gdb) c
Continuing.
```

12. On the target board run the `rmmod` utility to unload module from the running kernel:

```
bash# rmmod hello_world
```

13. Check that the breakpoint has been hit and try single-stepping the kernel:

```
Breakpoint 1, 0xc2800067 in
cleanup_module ()
(gdb) stepi
0xc280006c in cleanup_module ()
(gdb)
```

14. Now detach from the kernel. This detaches the cross development host GDB from the kernel debugger and allows the kernel to continue:

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

vl\_demo

## DEMO

VisualLynux program demonstration

## SYNOPSIS

This demo configures the target board for the required RPC 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 KDI (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.

#### REQUIREMENTS

Storage	Medium
RAM	Large
Network	Yes
Disk	None
Special	The host and target machines must be connected by a serial cable to use remote debugging via serial line.
Kernel Option	<code>ramdisk_size=28472</code>

#### DESCRIPTION

The system boots up in multi-user mode starting the network and the routed daemon. The IP address is set to `172.17.1.217`. The files `etc/resolv.conf` and `.rhosts` can be changed to adapt the target system to the local network settings.

1. On the cross development host, set up the BlueCat Linux environment by sourcing the `SETUP.sh` script from the top of the BlueCat Linux development directory tree.
2. Start VisualLynux from the installed BlueCat Linux environment and create a new project with an empty debugging environment. (Refer to the *VisualLynux User's Guide* for details about the VisualLynux program.)
3. Add new files to the project from the demo system source tree, `$BLUECAT_PREFIX/demo/vl_demo/src`.
4. Rebuild the project.
5. Set up the target system in VisualLynux and start a debugging session with the remote target. Perform debugging actions

such as single stepping, setting breakpoints, examining memory/register contents using VisualLynux.

## Disk Operations

disk

### DEMO

Disk utilities

### SYNOPSIS

This demo system contains formatting and partitioning tools.

### DESCRIPTION

The only supported filesystem type is `ext2`. The system boots in single-user mode. `init` starts `bash` without a login prompt. Use `fdisk` to manage partitions, `mkfs` to create filesystems on partitions and `fsck` to check the integrity of the filesystem.

### REQUIREMENTS

Storage	Small
RAM	Small
Network	None
Disk	Yes
Special	None

gnutar

### DEMO

A `tar` demonstration

### SYNOPSIS

This demo system contains the `mount` and `tar` utilities for mounting local disks and extracting gzipped tar images from them.

## REQUIREMENTS

Storage	Small
RAM	Small
Network	None
Disk	Yes
Special	A partition on the disk must contain a <code>tar</code> file.

## DESCRIPTION

The only supported filesystem type is `ext2`. The system boots in single-user mode. `init` starts `bash` without a login prompt. Mount a filesystem that has gzipped tar images and use `tar` to extract them.

`rootfs`

## DEMO

Root filesystem mount demo system

## SYNOPSIS

The kernel image is configured to mount the root filesystem from a disk.

## REQUIREMENTS

Storage	Tiny
RAM	Tiny
Network	None
Disk	Yes
Special	The hard disk must have a pre-installed root filesystem.
Kernel Option	<code>root=/dev/hda1</code> (for root filesystem on the first partition of the first IDE disk)

## DESCRIPTION

This demo system lets the user boot the target board and mount the root filesystem from the local disk. This demo system does not contain a RAM disk filesystem, but instead boots directly from the hard disk.

The only supported filesystem type is `ext2`. The kernel automatically mounts the root filesystem from the hard disk and the boot sequence proceeds from there.

`diskboot`

## DEMO

`diskboot demo system`

## SYNOPSIS

This demonstrates booting from a hard disk.

## REQUIREMENTS

Storage	Small
RAM	Small
Network	None
Disk	Yes
Special	None

## DESCRIPTION

This demonstrates a BlueCat Linux system that can be copied onto and booted from a hard disk.

Copy this demo system onto a hard disk and boot it from there. The filesystem includes the `ls`, `ps`, `reboot`, and `shutdown` commands.

## X Window and Friends

`xclock`

### DEMO

X Windows demo system

### SYNOPSIS

This demo system opens an X11 window with a continuously updated graphical clock.

### REQUIREMENTS

Storage	Medium
RAM	Large
Network	None
Disk	No
Special	Any VGA-compatible video card, PS/2 mouse
Kernel Option	<code>ramdisk_size=10240</code>

### DESCRIPTION

The X-server and the `xclock` program are started (according to `.bashrc` and `.xinitrc`) upon booting.

`xdemo1`

### DEMO

X Window color demonstration

### SYNOPSIS

This demo system opens an X11 window with the BlueCat Linux logo moving on the screen.

**REQUIREMENTS**

Storage	Medium
RAM	Large
Network	None
Disk	No
Special	Any VGA-compatible video card, PS/2 mouse
Kernel Option	ramdisk_size=10240

**DESCRIPTION**

No special startup actions are required. The X-server and the `xdemo1` programs are started (according to `.bashrc` and `.xinitrc`) on boot. Use the mouse buttons to switch between demo system screens.

`xdemo2`

**DEMO**

Tcl/Tk color demonstration

**SYNOPSIS**

This demo system opens an X11 window with the BlueCat Linux logo moving on the screen.

**REQUIREMENTS**

Storage	Medium
RAM	Large
Network	None
Disk	No
Special	Any VGA-compatible video card, PS/2 mouse
Kernel Option	ramdisk_size=10240

**DESCRIPTION**

No special startup actions are required. The X-server and the `xdemo2` programs are started (according to `.bashrc` and

`.xinitrc`) on boot. Use the mouse buttons to switch between demo system screens.

## Advanced Networking

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	<code>ramdisk_size=4096</code>

### 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. The user should bring up the network interface(s) manually using the `ifconfig` command and optionally, should 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.

developer

## DEMO

developer demo system

## SYNOPSIS

The `shell`, `ping`, `ftp`, `gdb`, and `v1_demo` demo system functionalities are combined in one BlueCat Linux system.

## DESCRIPTION

Refer to description of the `shell`, `ping`, `ftp`, `gdb`, and `v1_demo` demo systems for a list of features and capabilities supported by this demo system.

## REQUIREMENTS

Storage	Medium
RAM	Large
Network	Yes
Disk	None
Special	The host and target machines must be connected by a serial line to use remote debugging via a serial line. Serial <code>tty</code> devices (nodes) available in the <code>rootfs</code> image are <code>/dev/ttyS0</code> and <code>/dev/ttyS1</code> .
Kernel Option	<code>ramdisk_size=28472</code>

tcpdump

## DEMO

This demonstrates the use of the `tcpdump` utility.

## SYNOPSIS

Ethernet snooping demo system

## REQUIREMENTS

Storage	Small
RAM	Medium
Network	Yes
Disk	None
Special	None

## DESCRIPTION

The system boots in single-user mode. `init` starts `bash` without the 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. Then `tcpdump` can be used. For instance:

```
bash# ifconfig eth0 172.16.1.62
bash# route add default gw 172.16.0.1
bash# tcpdump -c 10
```

```
nfsroot
```

## DEMO

The kernel image is configured to mount the root filesystem from the NFS server.

## SYNOPSIS

This demo system lets the user boot the target board and mount the root filesystem from an NFS server. It does not contain a RAM disk filesystem, but instead boots up from a network.

## REQUIREMENTS

Storage	Tiny
RAM	Tiny
Network	Yes
Disk	None
Special	NFS server in the network
Kernel Option	<code>root=/dev/nfs</code>

## DESCRIPTION

The kernel uses the command line to determine the target machine IP address, the NFS server IP address, and the boot directory on the NFS server. The kernel automatically mounts a filesystem from an NFS server and the boot sequence proceeds from there.

Use the following procedure to configure the NFS server:

1. Create an empty directory `/nfsroot`.
2. Untar `nfsroot.tar` included in the demo system to the `/nfsroot` directory.
3. Edit `/etc/exports` on the NFS server.  
Add the following line:

```
/nfsroot target_machine_name (rw,  
no_root_squash)
```

For a detailed explanation of how to configure the NFS server, refer to the `exports(5)` man page.

4. Restart the `nfs` daemon on the NFS server:

```
bash# /etc/rc.d/init.d/nfs restart
```

`nfsroot.tar` contains the same filesystem as the one included in the `tcl` demo system. When the target board boots and the `nfsroot` filesystem is mounted the user can use the Tcl shell described in the `tcl` demo system.

`nfsroot` system is built to use a BOOTP server for the IP-autoconfiguration of the BlueCat Linux target board. The

user must set up a BOOTP server on the network before running the demo system.

## BlueCat Linux Messenger

`msng_exmpl`

### DEMO

Messenger demo system

### SYNOPSIS

This demonstrates how two user-space applications communicate via the Messenger Application Programming Interfaces (APIs).

### REQUIREMENTS

Storage	Tiny
RAM	Medium
Network	None
Disk	None
Special	None

### DESCRIPTION

The system boots in single-user mode. `init` starts `bash` without the login prompt. Run the Messenger applications manually, first with the receiver in the background and then with the transmitter. For instance:

```
bash# msng_exmpl -s &  
bash# msng_exmpl -c 10
```

When the transmitter is executed, the receiver prints the received data packet onto the console.

`msng_minet`

### DEMO

A Messenger Network Interface (MINET) demo system

## SYNOPSIS

This demonstrates TCP/IP connectivity of cPCI intelligent cards via the Messenger Network Interface (MINET).

## REQUIREMENTS

Storage	Small
RAM	Medium
Network	None
Disk	No
Special	Multinode cPCI systems (CPV5350 or MCP750 board as a cPCI host; one or more MCPN750 boards as I/O controllers)
Kernel Option	<code>ramdisk_size=8192</code>

## DESCRIPTION

The Input Output Processor (IOP) boards are booted over the cPCI backplane from the BlueCat Linux cross development host. The IOP boards do not have to be set up by the user. The system boots in single-user mode. `init` starts `bash` without the login prompt. At this point, the MINET interface can be initialized on the cross development host controller system.

1. To initialize the MINET interface on the cross development host controller system, call the `ifconfig` command. For instance:

```
bash# ifconfig minet0 1.0.3.1
```

2. Then, start `rarp`:

```
bash# rarp -f
```

3. To complete initialization start the IOP Manager:

```
bash# iopmand
```

Connect the serial COM1 of the non-system controller board via virtual terminal. This command boots BlueCat Linux onto the non-system controller boards.

4. To monitor the state of the backplane, use `iopmanc -i`. This command should show a separate entry for each non-system controller board installed in the chassis. When a board is fully booted, it is in the `BOOTED` state.
5. At this point, the MINET interface is ready to be tested. On a non-system board run `ifconfig minet0` (with no IP address). This command should print the IP address assigned by `rarp` on the cross development host to the board. Use this address to `ping` the board from the cross development host. For instance:

```
bash# ping -c 5 1.0.3.14
```

`ping` can of course, be run from any board in the system:  
`ping`-ing between non-system controller boards is supported.

## Java

caffeine

### DEMO

The embedded CaffeineMark demo system

### SYNOPSIS

This demo system measures the performance of the Java Virtual Machine (JVM) supported via the `kaffe` package.

### REQUIREMENTS

Storage	Medium
RAM	Large
Network	Yes
Disk	None
Special	None
Kernel Option	<code>ramdisk_size=32768</code>

### DESCRIPTION

The system boots in single-user mode. To start the Embedded CaffeineMark benchmark use the following command:

```
bash# caffeine
```

Successful execution of the benchmark displays the benchmarking results on the system console.

## Flash Memory Support and Flash File System

```
ffs
```

### DEMO

Flash memory support and Flash File System (FFS) demo system

### SYNOPSIS

This demonstrates the use of the BlueCat Linux flash memory management tools on the embedded target board.

### REQUIREMENTS

Storage	Tiny
RAM	Small
Network	None
Disk	None
Special	Demo system updates target flash memory

### DESCRIPTION

The system boots in the single-user mode. `init` mounts the `proc` filesystem and starts `bash`.

1. Use the `flash_fdisk` utility to partition the flash memory device. For instance, the following command creates three partitions in flash memory:

```
bash# flash_fdisk /dev/mtdchar0 \  
0-2:3:4-10
```

2. The registered MTD device drivers can be examined at any time using the `/proc/mtd` file:

```
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 of screen output corresponds to the entire flash memory device. Lines 2-4 correspond to the partitions created by the previous command.

3. Use the `flash_erase` utility to empty a flash memory partition or the entire flash memory device. For instance, the following command erases all sectors on the third partition:

```
bash# flash_erase /dev/mtdchar3
```

The device has 1 region with sectors of the same size. The total size of the device is 0xE0000 bytes.

```
Erasing..... done.
```

4. Mount the Flash File System and use the `cp`, `mv`, `ln` commands to work with it. For instance:

```
bash# mount /dev/mtdblock3 /mnt -t jffs
bash# cp /test_file /mnt
bash# cat /mnt/test_file
Flash test has passed OK!
```

5. Unmount the Flash File System using the `umount` command:

```
bash# umount /mnt
```

6. Reboot the system:

```
bash# /sbin/reboot -f
```

## Advanced Power Management

mapm

### DEMO

The BlueCat Linux Advanced Power Management (APM) demo system

## SYNOPSIS

This demo system demonstrates basic features and capabilities of the BlueCat Linux APM software. A special pseudo-device PMD (Power Managed Device) driver is used to simulate various aspects of a power-managed device functionality.

## REQUIREMENTS

Storage	Tiny
RAM	Tiny
Network	None
Disk	None
Special	None

## DESCRIPTION

The BlueCat Linux kernel is built to enable support for the APM core module, the APM user-space interface driver, and the simulated PMD test device driver. The simulated PMD registers itself with the PMD core and reacts to any APM requests by printing an informational message on the system console. In case the PMD has transitioned to the `STOP` state, the PMD device driver simulates occurrence of a wakeup event at the PMD in precisely 1 minute from the transition to the `STOP` state.

The `mapmd` daemon is configured to call an event handling program that reacts to transitioning to the `OFF` event by printing a message on the system console.

1. Read the file `/proc/mapm/pmds` to get the status information on the registered PMD devices:

```
bash# cat /proc/mapm/pmds
0 [MAPM test driver] ON 0 0 NONE
```

2. Use the `mapm_ctrl` utility to transition the simulated PMD to the `STOP` state:

```
bash# mapm_ctrl 0 STOP 0
```

The following message appears on the system console:

```
MAPM test driver: switch to STOP state
```

3. Wait 1 minute for the PMD to switch back to the `ON` state on the wakeup event:

```
MAPM test driver: switch to ON state
```

4. Use the inactivity timer parameter to test conditional transition to a lower-power state. For instance:

```
bash# mapm_ctrl 0 OFF 30000
```

In 30 seconds the PMD switches to the `OFF` state. The following message is printed on the system console by the PMD driver:

```
MAPM test driver: switch to OFF state
```

Additionally, the user-space event handler prints the following message onto the system console:

```
Simulated PMD event handler: switch to  
OFF state
```



# *BlueCat Linux Advanced Power Management*

This chapter describes Advanced Power Management (APM) support implemented in BlueCat Linux. BlueCat Linux APM support is defined in a generic manner, which allows support of diverse target board architectures while using the same software interfaces.

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, or any combination of these.

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, as well as to provide user-readable APM status information.

All interactions with clients of the APM software are 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 at 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 BlueCat Linux 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

Figure 5-1 shows the layout of the BlueCat Linux APM modules and its components:

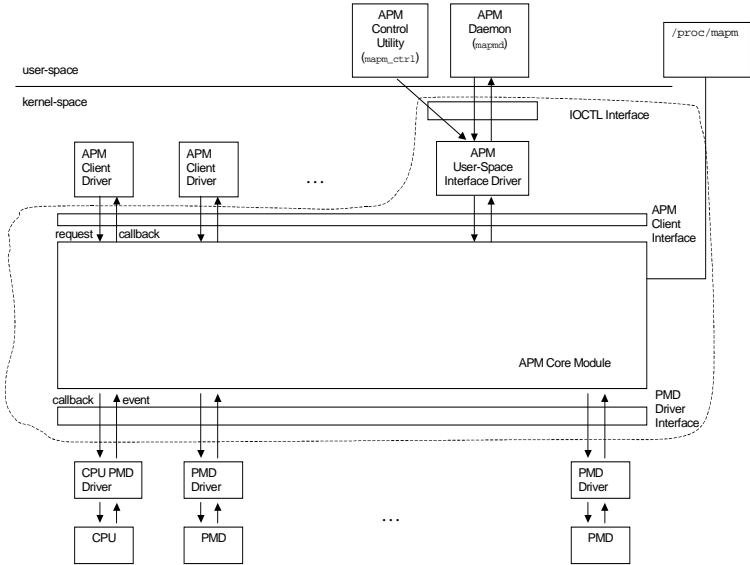


Figure 5-1: BlueCat Linux APM Modules and Components

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 `mapmd` daemon and the `mapmd_ctrl` control utility. The IOCTL interface is used by the `mapmd`

daemon to access APM events, to allow reaction at the user-space level. The IOCTL interface is used by the `mapmd_ctrl` control utility to explicitly control the APM and power-managed devices.

Another user interface is through the APM `proc` file (`/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. The event queue is used to sequentially process APM events. Events are processed on a first-come-first-processed basis. Processing of an event does not commence until the previous event in the queue has been processed completely.

## Pre- and Post-Events

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

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

Events are placed onto the event queue by the APM core module for one of the following reasons:

- In reaction to the 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 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.)

- As a result of a PMD driver reporting a hardware-driven state transition at the PMD
- As a result of an inactivity timer expiration

## Event Processing

The event queue is processed by a special *event processing* kernel task maintained by the APM core module. The event processing task processes events in the queue 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 the APM events does not result in blocking interrupt handlers or other critical kernel tasks.

Processing of an event is done in the following manner: For each kernel client that has 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, but eventually it has to return either a success or a failure. 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.

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.

All registered client callbacks are called sequentially. Once the last callback returns, event processing is complete for a post-event. For a pre-event, event processing continues in the following manner:

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.

If all the clients have indicated their 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, and then updates its internal tables to reflect the change at the PMD. As a final step in event processing, an appropriate post-event is placed in the event queue.

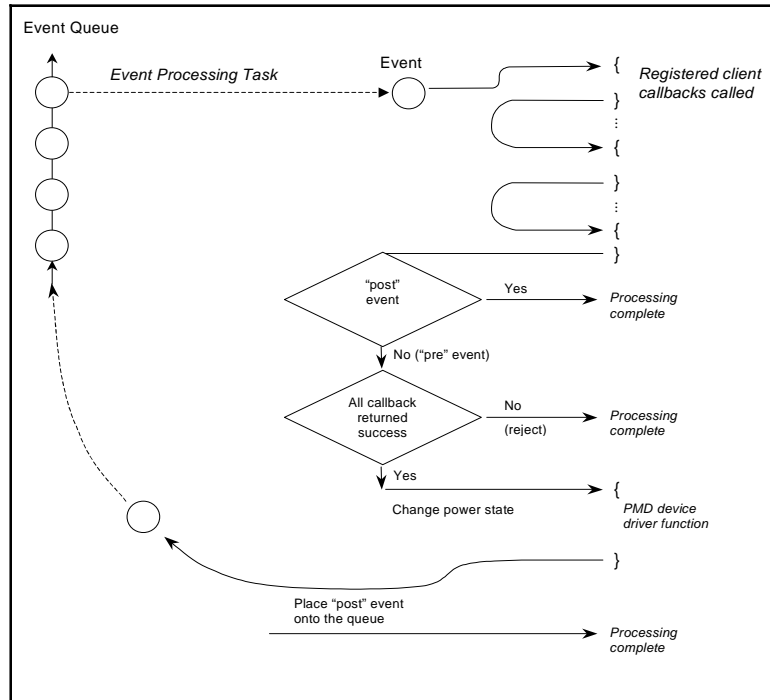


Figure 5-2: Event Processing

## Power States

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 state machine:

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.

However, it is possible 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 from occurring, 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 attention and/or reaction from the client. 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 PMD 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, which the client wishes to be notified of. 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 between different events. A callback uses the PMD handle to determine the PMD that the event has occurred at.

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, 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 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 these

Refer to Appendix F “`mapmd` 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:

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. Refer to Appendix G, “`mapm_ctrl` Command Reference” for details.

## 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`. The `mapm_cpu_suspendable` 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 (for instance, a system clock tick).

## 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 to save on kernel code size and runtime memory requirements. When disabled, APM has no impact on kernel size.

Table 5-1: Configuring APM in the Kernel

Option	Description
CONFIG_BLUECAT_APM	This option controls the presence of the APM core module in the kernel. Disabling the option automatically turns off all other APM components.
CONFIG_BLUECAT_APM_USER	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.
CONFIG_BLUECAT_APM_TEST	A test PMD device driver.
CONFIG_BLUECAT_APM_CPU	A CPU PMD skeleton device driver.

**NOTE:** *The APM core uses the standard kernel option CONFIG\_PROC\_FS (kernel support for the /proc filesystem) 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:

Table 5-2: Error Codes

Error Code	Description
APM_SUCCESS	Service successful
APM_ERROR_INVALID	Invalid argument
APM_ERROR_REJECT	Request rejected
APM_ERROR_RETRY	Out of resources
APM_ERROR_SYSTEM	OS error
APM_ERROR_NOTIMP	Not implemented
APM_ERROR_GENERAL	General error
APM_ERROR_NOPMD	No PMD with such handle
APM_ERROR_PMD	PMD internal error
APM_ERROR_NOCLIENT	No client with such handle

**Table 5-2: Error Codes (Continued)**

Error Code	Description
APM_ERROR_BUSY	Resource busy
APM_ERROR_MEM	Out of memory

## PMD State Codes

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

**Table 5-3: PMD States**

Code	Description
APM_STATE_ON	On state
APM_STATE_OFF	OFF state
APM_STATE_STOP	STOP state
APM_STATE_AUTO	AUTO state
APM_STATE_SPECIAL	SPECIAL state

## Event Codes

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

**Table 5-4: Event Codes**

Pre-Event Code	Description
APM_EVENT_REQUEST_ON	Request to switch to ON state
APM_EVENT_REQUEST_OFF	Request to switch to OFF state
APM_EVENT_REQUEST_STOP	Request to switch to STOP state
APM_EVENT_REQUEST_AUTO	Request to switch to AUTO state
APM_EVENT_REQUEST_SPECIAL	Request to switch to SPECIAL state

Table 5-5: Post-Event Codes

Post-Event Code	Description
APM_EVENT_ON	Transition to ON state
APM_EVENT_OFF	Transition to OFF state
APM_EVENT_STOP	Transition to STOP state
APM_EVENT_AUTO	Transition to AUTO state
APM_EVENT_SPECIAL	Transition to SPECIAL state

Table 5-6: PMD Driver Event Codes

PMD Driver Event Code	Description
APM_EVENT_NEW	New PMD driver registers
APM_EVENT_DELETE	Existing PMD driver unregisters

## 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:

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

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 `apm_client_info_t` structure is defined as follows:

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

APM unregisters the client specified by the `handle`.

```
typedef struct apm_client_info_s
{
    apm_client_callback_t callback;
    char * name;
}
apm_client_info_t;
apm_error_t apm_client_unregister
(apm_client_handle_t handle );
```

APM sets the event mask. The client receives event notifications from the specified PMD as defined by the `mask` argument. The mask is bitwise *or* of event codes (`APM_EVENT_*`).

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

APM makes a 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. The request is negotiated with other clients registered to receive notifications of the state transitions at the PMD. If the `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.

```
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.

```
apm_error_t apm_info( 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:

```
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:

Table 5-7: Possible PMD Request Code Values

Request Code	Description
<code>APM_PMD_REQUEST_ON</code>	Switch PMD to ON state
<code>APM_PMD_REQUEST_OFF</code>	Switch PMD to OFF state
<code>APM_PMD_REQUEST_STOP</code>	Switch PMD to STOP state
<code>APM_PMD_REQUEST_AUTO</code>	Switch PMD to AUTO state
<code>APM_PMD_REQUEST_STATE</code>	Report current device state
<code>APM_PMD_REQUEST_SPECIAL</code>	Switch to the SPECIAL state

## 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:

```
apm_pmd_handle_t * handle,  
void * user,  
apm_state_t state,  
unsigned int arg_i,  
void * arg_p);
```

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

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

```
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.

```
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:

```
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.

```
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:

```
handle \[ name\] state timer client event
```

<i>handle</i>	Hexadecimal value of the PMD handle
<i>name</i>	Name provided by the PMD at registration
<i>state</i>	The current PMD state—it is one of the following: ON, OFF, STOP, AUTO or SPECIAL.
<i>timer</i>	The value of the inactivity timer, if set for the PMD—else, this field is 0.
<i>client</i>	If an inactivity timer has been set for the PMD, this is the hexadecimal value of the client handle that has primed the timer. Else, this field is 0.
<i>event</i>	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.

Please refer to Appendix F, “`mapmd` Command Reference” for details.

## 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:

```
\[ PMD handle or PMD name \] command [ , command . . . ]
```

[ *PMD handle* or *PMD 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.

Please refer to Appendix H, “mapmd.conf Command Reference” for details.

## Handler Program Command Line Format

When `mapmd` calls program in response to `mapmd` event, it provides command line parameters in the following format:

```
user_program PMD_handle_or_name state arg[ , arg..]
```

<i>PMD_handle_or_name</i>	Specifies the PMD (in the same form as the configuration file for this PMD).
<i>state</i>	Specifies the state the PMD has been switched to, and is one of the following: ON, OFF, AUTO, STOP or SPECIAL.
<i>arg[ , arg..]</i>	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. Please refer to Appendix G, “`mapm_ctrl` Command Reference” for details.

---

# 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:

```
/* 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:

```
/* Client driver deinitialization. */
void client_term( void )
{
    ...
    /* We do not use the device anymore.
     * If nobody is using it, turn the device off.
     */
    apm_client_request( handle, CLIENT_OUR_PMD,\
        APM_STATE_OFF,
                        0, NULL, NULL );
    /* 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:

```
/* 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:
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:

```
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;
}
```

# *Flash Support and Flash File System*

This chapter provides a detailed description of flash memory support and the Flash File System (FFS) in BlueCat Linux.

---

## Flash Support and FFS Architecture

This section provides a general overview of the BlueCat Linux flash memory support architecture. General concepts introduced here are described in detail later in Chapter 6.

### 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
- FFS filesystem

Regardless of the interface is used to access flash memory, access to an actual flash memory device is 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 Figure 6-1.

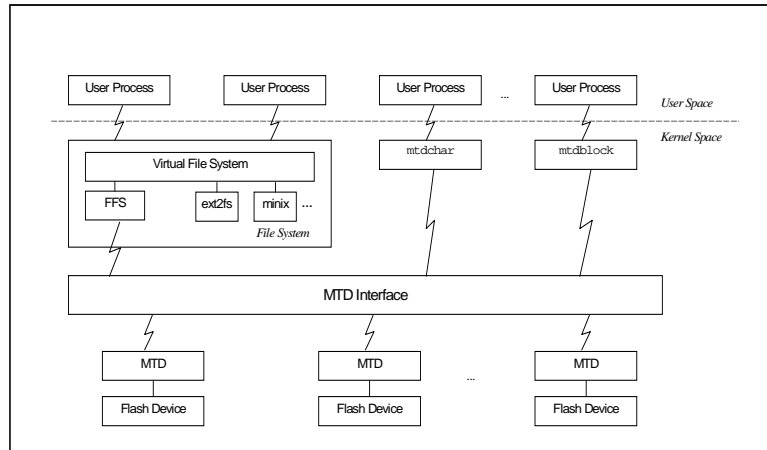


Figure 6-1: Flash Memory Support Architecture

## 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, etc. 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. Check Appendix J, “`flash_erase` Command Reference.”

---

**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 using 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 174.

## 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 filesystem. 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 Flash File System (FFS).

Like the `mtdchar` interface, `mtdblock` can be accessed using the standard POSIX operations, all of which have their standard interpretations. However, unlike `mtdchar`, `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 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.

## The Flash File System (FFS) Interface

The Flash File System (FFS) is a POSIX-compliant filesystem implemented on flash memory. FFS resides underneath the Virtual File System (VFS) layer. This means that, as with any other type of filesystem supported by Linux, any system call on an FFS or its files and directories goes through the VFS, which directs appropriate tasks to the FFS layer.

FFS is designed for the efficient use of flash memory devices. It has wear leveling, power loss recovery, and bad block mapping features built in.

FFS supports the following types of files defined by the POSIX.1 standard:

- Regular files
- Directories
- Special device files
- Symbolic links
- Named pipes

The following filesystem-specific calls defined by the POSIX.1 standard are supported by the FFS 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 FFS. 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 an FFS 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 FFS, `mtdchar`, and `mtdblock` all use the MTD interface, rather than directly implement 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.

There are a few entry points defined by the MTD interface that must be implemented by an MTD driver to successfully support a flash memory device. 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 only call an MTD driver's entry point when an operation on the physical flash memory is required. An MTD driver functions independently of the reason a particular low-level operation is needed or whether it has been initiated at FFS, `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 to ensure 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 Flash File System in a single flash memory device, or use particular sectors of flash memory as an FFS 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.

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, regardless of whether it corresponds to an entire flash memory device or just 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 an FFS via a block device node (`/dev/mtdblockx`) or raw-accessed via a character device node (`/dev/mtdcharx`).

### Flash Memory Entities and Device Nodes

A user process specifies a particular flash memory device or partition by using an appropriate device node. There is a one-to-one relationship

between flash memory device nodes and flash memory entities maintained by BlueCat Linux.

To elaborate further, a flash memory entity (a device or a partition) has one device node corresponding to it within a device group (character devices or block devices). This means that each flash memory entity can, in fact, have two device nodes corresponding to it—one for the character interface (`mtdchar`), and the other for the block interface (`mtdblock`).

A flash memory character device node (`/dev/mtdcharx`) always has a major number of `MTD_CHAR_MAJOR (90)`. A flash memory block device node (`/dev/mtdblockx`) always has a major number of `MTD_BLOCK_MAJOR (31)`.

The minor number is used to distinguish between flash memory devices, and then between partitions within a device. The function used to determine the minor number for a concrete flash memory entity (a device or a partition) is very simple: the minor number of the entity is equal to the number of flash memory entities that have registered at the MTD interface before that entity.

Consider, for instance, three MTDs:

- MTD1 creates 3 partitions
- MTD2 does not create any partitions
- MTD3 creates 2 partitions.

Provided the MTDs register in the order they are defined above, MTD1 performs 4 registrations: 1 for the entire device and 3 for the 3 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. The minor number it gets is 4. MTD2 registers 3 times and gets the minor number of 5 for the entire device and the minor numbers of 6 and 7 for the 2 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.

However BlueCat Linux 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

of 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 the order in which the entities have registered themselves with the MTD interface.

## Partition Configuration

The user can use any of the following means 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 Chapter 6 for a detailed description of the flash memory management tools and mechanisms.

---

## FFS Internals

### FFS Layout

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

```
struct jffs_raw_inode
{
    __u32 magic;          /* A constant magic number. */
    __u32 ino;           /* Inode number. */
    __u32 pino;          /* Parent's inode number. */
    __u32 version;       /* Version number. */
    __u32 mode;          /* The file's type or mode. */
    __u16 uid;           /* The file's owner. */
    __u16 gid;           /* The file's group. */
    __u32 atime;         /* Last access time. */
    __u32 mtime;         /* Last modification time. */
    __u32 ctime;         /* Creation time. */
    __u32 offset;        /* Where to begin to write. */
    __u32 dsize;         /* Size of the node's data. */
    __u32 rsize;         /* How much are going to be replaced? */
    __u8 nsize;          /* Name length. */
    __u8 nlink;          /* Number of links. */
    __u8 spare : 6;     /* For future use. */
}
```

```

    __u8 rename : 1; /* Is this a special rename? */
    __u8 deleted : 1; /* Has this file been deleted? */
    __u8 accurate; /* The inode is obsolete if accurate ==\
0.*/
    __u64 call_num /* write() call number */
    __u32 num_nodes /* Number of the nodes written within a\
call
                    */
    __u32 alignment; /* This makes an alignment of fields\
predictable
                    */
    __u32 dchksum; /* Checksum for the data. */
    __u16 nchksum; /* Checksum for the name. */
    __u16 chksum; /* Checksum for the raw inode. */
};

```

`raw_inode` is followed by a filename and a piece of file data according to the values of the `nsize` and `dsize` members of the above structure, respectively. The filename stored with `raw_inode` is not absolute, but relative to the parent directory. The `pino` field of the `raw_inode` structure is used to maintain a complete directory tree.

A directory is represented in FFS 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 (parent inode number) pointing to the `raw_inode` of the directory.

Both the filename and the file data may not be present in a node. There is no need for the filename to be present in each `raw_inode` associated with the file because all `raw_inodes` for a file have the same value of the `ino` field.

When the file is renamed, a new `raw_inode` with the same `ino` field, the new filename, but no file data is created. If only the `mode` or access permissions, or the modification time of the file is changed, then a new `raw_inode` with the appropriate `mode`, `mtime` and `atime` fields, but without the filename and data is created.

There is a limitation on the length of a chunk, depending on the sector size of the underlying flash memory parts. A chunk cannot be longer than half the largest flash memory sector.

When a file is stored in the filesystem, 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 filesystem is modified (renamed, deleted, 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 FFS is that the FFS 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:

<code>ino</code>	Inode number associated with the file
<code>pino</code>	Inode number of the parent directory
<code>version</code>	Highest version of all <code>raw_inodes</code> of the file previously written + 1
<code>offset</code>	Position in the file at which <code>write()</code> is performed
<code>dsize</code>	Size of data appended to the file
<code>rsize</code>	0
<code>nsiz</code>	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 (re-written). The relevant fields are set as follows:

<code>ino</code>	Inode number associated with the file
<code>pino</code>	Inode number of the parent directory
<code>version</code>	Highest version of all <code>raw_inodes</code> of the file previously written + 1
<code>offset</code>	Position in the file at which <code>write()</code> is performed
<code>dsize</code>	Size of data rewritten in the file
<code>rsize</code>	Size of data rewritten in the file
<code>nsiz</code>	0

The FFS maintains an in-RAM list of pointers to all the actual nodes of each file. Hence, when a file is being read, the system looks through the list to find where the data requested is located in flash memory.

Consider the following code fragment, which writes data to a file and then overwrites two portions of it:

```
char msg1[1000];
char msg2[100];
char msg3[50];

int hdl = open ("test",O_CREAT+O_RDWR);
memset(msg1, 'A', sizeof(msg1));
write(hdl, msg1, sizeof(msg1));

lseek(hdl, 600, SEEK_SET);
memset(msg2, 'B', sizeof(msg2));
write(hdl, msg2, sizeof(msg2));

lseek(hdl, 200, SEEK_SET);
memset(msg3, 'C', sizeof(msg3));
write(hdl, msg3, sizeof(msg3));
```

Thus after three writes the file should contain a pattern like:

```
aaaacaaaaaaaaabbaaaaaa
```

where each lower-case character represents 50 of the equivalent uppercase characters. Table 7-1 shows the contents of the FFS and the in-RAM list after the sample code above has completed:

Table 7-1: FFS and in RAM List After Completed Sample Code

Flash Memory Contents		In RAM List	
Offset	Description	Node	Contents
0x0	raw_inode created during <i>open()</i> Relevant fields are set as follows: offset = 0	1	dsize = 0 flash_offset = 0x40
0x3c	nsize = 4 (name size) dsize = 0 (data size) rsize = 0 (removed size) version = 1 test - filename	-	
0x40	raw_inode created during first <i>write()</i> Relevant fields are set as follows: offset = 0	2	dsize = 200 flash_offset = 0x7c
0x7c	nsize = 0 dsize = 1000 rsize ≠ 0 version = 2 AAA...AAA	3	dsize = 50 flash_offset = 0x540
0x464	raw_inode created during second <i>write()</i> Relevant fields are set as follows: offset = 600	4	dsize = 350 flash_offset = 0x7c + 250 = 0x176
0x4a0	nsize ≠ 0 dsize = 100 rsize = 100 version = 3 BBB...BBB	5	dsize = 100 flash_offset = 0x4a0
0x504	raw_inpde created during third <i>write()</i> Relevant fields are set as follows: offset = 200	6	dsize = 300 flash_offset = 0x7c + 700 = 0x338
0x540	nsize = 0 dsize = 50 rsize = 50 version = 4 CCC...CCC		

An FFS is mounted using the standard `mount` command. The `-t jffs` flag is passed directly to the kernel to specify that an FFS is being mounted.

An `mtdblock` special device node is used as a parameter to `mount` to specify a flash memory device on which the filesystem is mounted. BlueCat Linux allows for mounting a filesystem on a partition, as well as on an entire device.

When FFS is being mounted, flash memory is scanned and an in-memory representation of the filesystem 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 the correct node. 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 previous node.

## Power Loss Recovery

When scanning flash memory for `raw_inodes`, the FFS scan algorithm described in “FFS Layout” searches for the `raw_inode` number to identify each chunk. When a chunk is found, the `checksums` for `raw_inode` and, if present, the filename and chunk data are calculated and compared with the values stored in the `raw_inode` structure in flash memory. If a power loss has occurred during a write operation then the `checksums` are incorrect and the node is rejected. This means that if a less recent node for this part of the file is present in flash memory, it is used instead of the reject. The stock result is as if no operation that caused the rejected node to be created has occurred.

Each POSIX I/O call that leads to a flash memory update is enumerated and its number is stored in all the `raw_inodes` to be written during call processing. The number of nodes written by a call is stored along with the call number. So if the power fails during the multi-node `write()` call, the successfully written nodes of the partially complete write are found and rejected by the scan procedure during the filesystem mount. The scan procedure also finds the largest call number stored in flash memory and initializes the call counter appropriately.

The BlueCat Linux FFS therefore guarantees that if a power loss occurs at a flash memory update, then the FFS is restored to its previous state upon reboot, as if the failing POSIX I/O call never occurred.

64 bits are used to store the call numbers. Supposing there are 32 MB of flash memory, and each write stores one `raw_inode` (about 70 bytes) without the filename and file data: It takes less than  $2^{19}$  calls to fill the flash memory completely; 64 bits can hold the value of  $2^{64}$ . Hence, many holes are burned in flash memory before the call counter overflows.

## Wear Leveling

### Wear Leveling and Garbage Collection Algorithm

Wear leveling is maintained by the following algorithm implemented in the FFS. 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.



Writing to flash memory is always performed at the tail. This means that even if a file is being deleted from the FFS, no actual deletions are performed immediately in flash memory. Instead, metadata 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 1/3 the flash memory size, or the free space is below 5%. These garbage collection criteria are evaluated when sending a signal to the garbage collection thread.

Criteria evaluation occurs at two points during FFS 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 filesystem mount. The second is when processing the `write_super()` call issued by VFS, which occurs about every 5 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, as the multi-threaded nature of the garbage collection is not utilized because of the sequential, blocking call to the garbage collection system.

Upon reboot, when mounting the filesystem, 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 FFS, 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, i.e., the callback does not return until the write to flash memory is actually completed. The users do not need to force synchronicity by using the `O_SYNC` flag when opening a file.

There is no buffering for writing. The VFS filesystem may (and FFS does) use caching for reading data. FFS uses caching for the read operation because it supplies the Linux *generic\_file\_read()* function in `struct file_operations` when registering the filesystem 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 filesystem) 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 BlueCat Linux FFS 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 FFS layer, which handles the code and maintains the table of bad blocks.

---

## The mtdchar Interface Reference

The `mtdchar` character device interface implements flash memory-specific IOCTL commands defined below. These commands can be used if the `linux/mtd/mtd.h` file is included.

### MEMGETINFO

This IOCTL command copies the MTD driver information to the user space. It is returned in the `mtd_info_user` structure pointed to by the `arg` parameter of the *ioctl()* call.

The `mtd_info_user` structure is defined as follows:

```
struct mtd_info_user
{
    u_char type;          /* Type of memory technology */
    u_long flags;        /* Device capabilities */
    u_long size;         /* Size of the device in bytes */
    u_long n_regions;    /* Number of Flash regions */
    u_long oobblock;     /* Size of block that has out-of-band data
                        */
    u_long oobsize;     /* Size of each out-of-band area */
    u_long ecctype;     /* Error correction type */
    u_long eccsize;     /* Size of blocks for automatic */
                        /* error correction
                        */
};
```

## MEMGETREGIONS

This IOCTL command copies information about the flash memory regions defined by the MTD driver to the user space. It is returned in an array of the `mtd_flash_region_user` structures pointed to by the `arg` parameter of the `ioctl()` call. The `mtd_flash_region_user` structure is defined as follows:

```
struct mtd_flash_region_user
{
    loff_t start_offset; /* Region starting offset */
    loff_t size;         /* Region size */
    u_long erasesize;   /* Size of the sectors */
    int   _sectors;     /* Number of sectors */
};
```

## MEMERASE

This IOCTL command erases a specified area in flash memory. This area is specified with the `erase_info_user` structure pointed to by the `arg` parameter of the `ioctl()` call. The `erase_info_user` structure is defined as follows:

```
struct erase_info_user
{
    unsigned long start; /* Offset to start erase from */
    unsigned long length; /* The length of the area to be\
erased */
};
```

The values of the `start` and `length` fields of the above structure must be aligned to a sector boundary. Also, the area defined by the structure must be a part of a single flash memory region.

## MEMWRITEOOB

This IOCTL command writes out-of-band data specified with the `mtd_oob_buf` structure pointed to by the `arg` parameter of the `ioctl()` call. The `mtd_oob_buf` structure is defined as follows:

```
struct mtd_oob_buf
{
    loff_t start; /* Starting offset of the oob_area */
    ssize_t length; /* The length of data to be written */
    unsigned char *ptr; /* Pointer to the data to be written */
};
```

### MEMREADOOB

This IOCTL command reads out-of-band data to the `mtd_oob_buf` structure pointed to by the `arg` parameter of the `ioctl()` call. The `mtd_oob_buf` structure is defined as follows:

```
struct mtd_oob_buf
{
    loff_t start;          /* Starting offset of the oob_area */
    ssize_t length; /* The length of data to be read */
    unsigned char *ptr; /* Pointer to the memory where data
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:

```
struct mtd_partition_conf
{
    char * conf;          /* Configuration string */
    int size;            /* Size of the configuration string */
};
```

---

## FFS 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 FFS.

### JFFS\_GET\_BAD\_TABLE

This IOCTL command provides user-space programs with access to the table of bad blocks.

The semantics of the command is as follows. The user allocates memory to hold the bad-block table. The user then supplies

`struct jffs_bad_table` with the `num_sectors` field containing the number of bytes already allocated for the bad block table and with the `bad_block_table` field pointing to the allocated space. Information about each sector in the table takes one byte. If the byte is not “0,” it means

that the corresponding sector is marked as bad. If the number of sectors associated with the partition is less than `num_sectors`, then on return `num_sectors` is set to the actual number of sectors.

The `jffs_bad_table` structure is defined as follows:

```
struct jffs_bad_table
{
    char * bad_block_table; /* The array representing device\
blocks */
    int   num_sectors;     /* The size of the array */
};
```

`JFFS_GARBAGE_COLLECT`

The IOCTL command initiates a garbage collection procedure. The `arg` parameter of the `ioctl()` call must have one of the following values:

```
JFFS_GC_TRIGGER
JFFS_GC_ONCE
JFFS_GC_COMPLETE
```

If `JFFS_GC_ONCE` is specified, then the GC 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 1/3 the flash memory entity size or the amount of free space is less than 5% of the total partition size.

If `JFFS_GC_COMPLETE` is specified, then the GC procedure is run when the amount of dirty space is larger than a smallest flash memory sector of the partition.

---

## MTD Interface Reference

This section describes the general structure of the Memory Technology Device (MTD) subsystem and its interfaces. The MTD system is divided into two types of modules: *users* and *drivers*. *Drivers* are kernel modules that provide raw read/write/erase access to the physical memory devices. *Users* are kernel modules that use the MTD drivers and provide a higher-level interface to the user space. The term “module” does not automatically imply Linux-loadable modules; MTD modules can be linked statically to the kernel.

The idea here is simple: the MTD interface provides for an open-interface, extensible approach that makes it easy to add support for flash memory

devices (and other types of memory devices) without the need to update any of the user interfaces, such as FFS, `mtdchar`, or `mtdblock`.

Writing an MTD driver is simple:

1. Allocate and populate `struct mtd_info` with information about the supported device and pointers to the driver's access routines.
2. Register it at the MTD interface by calling:

```
int add_mtd_device(struct mtd_info *mtd)
```

Access routines implemented by an MTD driver must conform to the rules specified by the MTD interface.

What follows is a description of `struct mtd_info` that provides the interface to access an MTD driver from MTD users:

```
struct mtd_info {
    char name[32];
```

The name of the device is rarely used, but presented to the user via the `/proc/mtd` interface. When the `proc` filesystem support is compiled into the kernel and the filesystem mounted, one can inspect the MTD drivers registered within the system by looking through the `/proc/mtd` file.

```
u_char type;
```

The type of memory technology used in this device may be one of the following:

<code>MTD_ABSENT</code>	No technology
<code>MTD_RAM</code>	RAM
<code>MTD_ROM</code>	ROM
<code>MTD_NORFLASH</code>	NOR flash memory
<code>MTD_NANDFLASH</code>	NAND flash memory
<code>MTD_PEROM</code>	EPROM
<code>MTD_OTHER</code>	Other
<code>MTD_UNKNOWN</code>	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 – it 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. `oobsize` and `oobblock` hold the size of each out-of-band area, and the number of bytes of real memory with which each is associated, respectively. For example, NAND flash memory has `oobblock == 512` and `oobsize == 16` for 16 bytes of OOB data per 512 bytes of flash memory.

```
u_long oobblock;  
u_long oobsize;
```

Some types of hardware not only allow access to flash memory or similar devices, but also have ECC (error correction) capabilities built-in to the interface:

```
u_long ecctype;
```

The `ecctype` field is an enumeration. It can be one of the following:

```
MTD_ECC_NONE      No automatic ECC available
```

```
MTD_ECC_RS_DiskOnChip Automatic ECC on DiskOnChip
```

`eccsize` holds the size of the blocks on which the hardware can perform automatic ECC.

```
u_long eccsize;
```

When a driver is a kernel-loadable module, this field is a pointer to the `struct module` of the module. It is used to increase and decrease the module's usage count as appropriate. The user modules are responsible for increasing and decreasing the usage count of the driver as appropriate, for example, by calling `__MOD_INC_USE_COUNT (mtd->module)` in the open routine.

```
struct module *module;
```

The following routine adds `struct erase_info` to the erase queue for the device. The routine may sleep until the erase is complete, or it may simply queue the request and return immediately. The exact behavior of the routine is defined by the particular MTD driver implementation. Currently, the MTD interface does not provide means to instruct the driver for a specific type of operation (sleep or return). `struct erase_info` contains a pointer to a callback function, which is called by the MTD driver when the erase is complete.

```
int (*erase) (struct mtd_info *mtd,
             struct erase_info *instr);
```

For devices that are entirely memory-mapped, and which can be mapped directly into user-space page tables, support for execute-in-place (XIP) mapping of data on flash memory may be possible. The precise semantics of this are yet to be defined, so it is probably best not to implement or attempt to use these two functions at the moment. Currently, in BlueCat Linux there is no support for the execute-in-place feature.

```
int (*point) (struct mtd_info *mtd,
             loff_t from,
             size_t len,
```

```
        u_char **mtdbuf);  
void (*unpoint) (struct mtd_info *mtd,  
                u_char * addr);
```

The following exemplifies *read()* and *write()* functions for the memory device. These may sleep, and should not be called from the IRQ context or with locks held. The *buf* argument is assumed to be in the kernel space. Currently, the MTD interface does not provide means to instruct the driver for a specific type of operation (sleep or return).

```
int (*read) (struct mtd_info *mtd,  
            loff_t from,  
            size_t len,  
            u_char *buf);  
  
int (*write) (struct mtd_info *mtd,  
             loff_t to,  
             size_t len,  
             const u_char *buf);
```

For the devices that support automatic ECC generation or checking, these routines below behave exactly as the *read/write()* functions above. Additionally, the *write\_ecc()* function places the generated ECC data into *eccbuf*, and the *read\_ecc()* function verifies the ECC data and attempts to correct any errors it detects.

```
int (*read_ecc) (struct mtd_info *mtd,  
               loff_t from,  
               size_t len,  
               u_char *buf,  
               u_char *eccbuf);  
  
int (*write_ecc) (struct mtd_info *mtd,  
                loff_t to,  
                size_t len,  
                const u_char *buf,  
                u_char *eccbuf);
```

These functions provide access to out-of-band data for devices that have it:

```
int (*read_oob) (struct mtd_info *mtd,  
               loff_t from,  
               size_t len,  
               u_char *buf);  
  
int (*write_oob) (struct mtd_info *mtd,  
                loff_t to,
```

```

        size_t len,
        const u_char *buf);

```

The following routine sleeps until all pending flash memory operations are complete. This call is not used currently by any existing MTD drivers. Instead, their *write()* and *erase()* calls are implemented in the blocking manner.

```

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

```

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

```

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

```

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

```

struct mtd_flash_region
{
    loff_t start_offset;

```

The size of the region:

```

    loff_t size;

```

The size of sectors within the region:

```

    u_long erasesize;

```

The number of sectors within the region:

```

    int n_sectors;

```

A pointer to the next region:

```

    struct mtd_flash_region * next;
}; /* end of struct mtd_flash_region */

```

The starting offset of the partition relative to the beginning of the physical device:

```

struct mtd_partition
{
    loff_t start_offset;
    loff_t start_offset;

```

Size of the partition:

```

    loff_t size;

```

A pointer to the next partition:

```
    struct mtd_partition * next;  
} /* end of struct mtd_partition */
```

---

## Flash Memory Management Tools and Mechanisms

This section explains in detail the control of flash memory devices in embedded applications.

### Configuring Flash Memory Partitions

The first step is to configure flash memory partitions. This step is optional, as it can well be that an embedded application requires the use of an entire flash memory device as a single entity, for instance to hold an FFS.

#### 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 4. Build-time parameters have the format shown below:

```
CONFIG_MTD_driver_PART="0,4:1-3:5-34"
```

The numbers in quotation marks correspond to the sectors that are allocated to particular partitions. Colons separate configuration input for the partitions. In the example above, sectors 0 and 4 are allocated to the first partition, sectors 1, 2, 3 to the second partition, and sectors 5 to 34 to the third one.

## Configuring Partitions using Kernel Boot-Time Parameters

The second method is a boot-time configuration through kernel command-line parameters. The MTD drivers included in the user's distribution are written to recognize boot-time parameters in the format shown below:

```
driver_part_conf="0,4:1-3:5-34"
```

The numbers in quotation marks have the same interpretation as for the build-time configuration parameters.

## Configuring Partitions using flash\_fdisk

The third method runs the runtime configuration utility `flash_fdisk(1)` to configure flash memory partitions. `flash_fdisk` performs `ioctl()` calls to the `mtdchar` interface specifying the command `MEMDEFPARTTABLE` and supplying configuration information for a partition. `mtdchar`, in turn, calls the underlying MTD driver's callback defined in `modify_part_table` of `struct mtd_info`. The MTD driver calls `del_mtd_device()` to unregister previously registered partitions (if any) and then calls `add_mtd_device()` to register the newly created partition. `flash_fdisk` has the following syntax:

```
flash_fdisk mtdchar_node configuration_string
```

The configuration string has the same format as in the first two configuration methods.

Please refer to Appendix I, "flash\_fdisk Command Reference."

The following example command creates the same partitioning 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:

```
mtdminor: size id_string
```

For example, the following snapshot shows output for the configuration that has one flash memory device with 3 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 an FFS 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`” 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 Appendix J, “`flash_erase` 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 an FFS

To create an FFS in a flash memory device or partition, use the `mount` command with the flag `-t jffs`. For instance, the following command creates an FFS in the second flash memory partition:

```
# mount /dev/mtdblock2 /mnt -t jffs
```

The same command mounts an already existing FFS in a flash memory entity specified by the block device node parameter.

Once an FFS is mounted, it can be used in the same manner as a filesystem of any other type is used. For instance:

```
# cd /mnt
# mkdir tmp
# cd tmp
# cp /tmp/test_file .
# ls -lR /mnt
```

When done with using an FFS, unmount it:

```
# cd /
# umount /mnt
```

A clean unmount of an FFS calls the garbage collector. This ensures that no time is spent on garbage collection the next time the user mounts the FFS.

## 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 embedded target board.

---

## Developing an MTD Driver

This section shows the skeleton of a sample MTD driver.

## Registering an MTD Driver

The following sample code shows the registration of a sample MTD driver.

```

/* Service function that parses configuration string
 * and calls add_mtd_device() as appropriate.
 */
extern int mtd_create_partitions(const char      * layout,
                               struct mtd_info ** part_mtds,
                               int              max_num,
                               struct mtd_info * driver);

/* mtd_info structures of the configured partitions.
 */
static struct mtd_info * mtd_part_mtds[4];

/* The number of the configured partitions.
 */
static int                configured_parts;

/* Partitions configuration string. Initialized during
 * kernel build-time configuration. May be changed at boot
 * time
 * during the kernel command line parsing in init/main.c
 */
char * mtd_sample_part_conf = CONFIG_MTD_SAMPLE_PART;

/* mtd_info structure corresponding to the whole device.
 */
static struct mtd_info  mymtd;

    /* Callbacks declarations.
    */
static int sample_erase(struct mtd_info *mtd,
                       struct erase_info *instr);
static int sample_read(struct mtd_info *mtd,
                      loff_t from,
                      size_t len,
                      size_t *retlen,
                      u_char *buf);
static int sample_write(struct mtd_info *mtd,
                      loff_t to,
                      size_t len,
                      size_t *retlen, const
                      u_char *buf);
static int sample_runtime_part_conf(char* str);

int init_mtdsample(void)
{
    int                res;
    struct mtd_flash_region * region;

    /* Allocate memory for the Flash regions info.
     * This example assumes that the device has 4
     * regions
     * with sectors of the same size.
     */
    mtd_info->flash_regions = (struct mtd_flash_region *)
kmalloc(sizeof(struct mtd_flash_region) * 4, GFP_KERNEL);
    if (mtd_info->flash_regions == 0)
    {
        res = -ENOMEM;
        goto Done;
    }

    region = mtd_info->flash_regions;

    /* One 16K sector.
     */
    region->size                = 16 * 1024;

```

```

        region->erasesize      = 16 * 1024;
        region->n_sectors      = 1;
        region->start_offset   = 0;
        region->next           = region + 1;
        region->next->start_offset = region->start_offset +
region->size;
        region++;

        /* Two 8K sectors.
        */
        region->size           = 8 * 1024 * 2;
        region->erasesize      = 8 * 1024;
        region->n_sectors      = 2;
        region->next           = region + 1;
        region->next->start_offset = region->start_offset +
region->size;
        region++;

        /* One 32K sector.
        */
        region->size           = 32 * 1024;
        region->erasesize      = 32 * 1024;
        region->n_sectors      = 1;
        region->next           = region + 1;
        region->next->start_offset = region->start_offset +
region->size;
        region++;

        /* Thirty one 64K sectors.
        */
        region->size           = 64 * 1024 * 31;
        region->erasesize      = 64 * 1024;
        region->n_sectors      = 31;
        region->next           = 0;

        /* Setup the MTD structure.
        */
        mymtd->name = "MTD sample device";
        mymtd_info->size = 2048*1024;
        mymtd->n_regions = 4;
        mymtd_info->erase = sample_erase;
        mymtd->read = sample_read;
        mymtd->write = sample_write;
        mymtd->modify_part_table = sample_runtime_part_conf;

        if (add_mtd_device(mymtd))
        {
            /* Registration failed.
            */
            ...
        }

        /* Optionally configure partitions.
        */
        configured_parts = mtd_create_partitions(mtd_sample_part_conf,
                                                mtd_part_mtds,
                                                4,
                                                mymtd);

        if (configured_parts < 0)
        {
            /* Partitions configuration failed.
            */
            ...
        }

        res = 0;
        Done:
        return res;
    }

```

## Deregistering the MTD Driver

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

```
void cleanup_mtdsample(void)
{
    int i;

    /* First unregister configured partitions.
    */
    for (i = 0; i < configured_parts; i++)
    {
        if (mtd_part_mtds[i])
        {
            del_mtd_device(mtd_part_mtds[i]);
            kfree(mtd_part_mtds[i]->part);
            kfree(mtd_part_mtds[i]->flash_regions);
            kfree(mtd_part_mtds[i]);
        }
    }

    /* Unregister the MTD driver.
    */
    del_mtd_device(mtd_info);
}
}
```

## Configuring Partitions at Runtime

The following sample code shows how to write an MTD driver callback for runtime partition configuration.

```
static int sample_runtime_part_conf(char* str)
{
    int res = 0;

    /* First, unregister previously created partitions, if\
    any.
    */
    while (configured_parts > 0)
    {
        if (mtd_part_mtds[configured_parts - 1])
        {
            if (del_mtd_device(mtd_part_mtds[configured_parts \
-1]))
            {
                res = -EBUSY;
                goto Done;
            }
            kfree(mtd_part_mtds[configured_parts - 1]->part);
            kfree(mtd_part_mtds[configured_parts - 1]
->flash_regions);
            kfree(mtd_part_mtds[--configured_parts]);
        }
    }

    /* Register new partitions.
    */
    configured_parts = mtd_create_partitions(str,
mtds,
mtd_part\
4,
my_mtd);

    if (configured_parts < 0)
    {
```

```
/* Partitions configuration failed.
 */
    ...
    res = -EINVAL;
    goto Done;
}

Done:
return res;
}
```



# Default BlueCat Linux Packages

This appendix describes all the packages included in the BlueCat Linux default configuration. The default configuration installed on the cross development host by the `install` program contains all the packages necessary for the use of BlueCat Linux. Specifically, the following packages are installed on the cross development host.

Table A-1: Packages in the Default BlueCat Linux Configuration

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
<b>Archiving</b>								
	<code>cpio</code>	<code>cpio</code> utility	target	Y	Y	Y	Y	Y
	<code>cpio</code>	<code>cpio</code> utility	cdt	Y	Y	Y	Y	Y
	<code>ncompress</code>	File compression	target	Y	Y	Y	Y	Y
	<code>rmt</code>	Remote access for tape drives	target	Y	Y	Y	Y	Y
	<code>sharutils</code>	Encodes and decodes shell archive ( <code>shar</code> ) files	target	Y	Y	Y	Y	Y
	<code>tar</code>	GNU file archiving program	target	Y	Y	Y	Y	Y
<b>Publishing</b>								
	<code>groff</code>	GROFF documentation system	target	Y	Y	Y	Y	Y
<b>Application/System</b>								

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	arpwatch	Network monitoring	target	Y	Y	Y	Y	Y
	bind-utils	Queries DNS name servers for Internet Host information	target	Y	Y	Y	Y	Y
	console-tools	Controls console fonts and keyboard	target	Y	Y	Y	Y	Y
	hdparm	Set sIDE hard drive parameters	target	Y	Y	Y	Y	Y
	knfsd-clients	Shows clients mounting an NFS cross development host	target	Y	Y	Y	Y	Y
	mkxauth	Manipulates X authentication databases	target	Y	Y	Y	Y	Y
	mttools	Accesses MS-DOS files	target	Y	Y	Y	Y	Y
	mt-st	mt and st tape drive management	target	Y	Y	Y	Y	Y
	netcfg	GUI for setting up network configuration	target	Y	Y	Y	Y	Y
	pciutils	Inspects and sets devices on the PCI bus	target	Y	Y	Y	Y	Y
	procinfo	Displays kernel and filesystem information	target	Y	Y	Y	Y	Y
	procps	Utilities for monitoring system	target	Y	Y	Y	Y	Y
	psmisc	Utilities for managing processes	target	Y	Y	Y	Y	Y

**Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)**

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	rdate	Gets and sets date remotely	target	Y	Y	Y	Y	Y
	rdist	Maintains identical files on multiple hosts	target	Y	Y	Y	Y	Y
	samba_common	Files used by both Samba server and clients	target	Y	Y	Y	Y	Y
	screen	Multiple logins from one tty line	target	Y	Y	Y	Y	Y
	setconsole	Utility to add new console device	target	Y	Y	Y	Y	Y
	setserial	Sets and gets serial port information	target	Y	Y	Y	Y	Y
	time	GNU utility for monitoring a program's use of system resources	target	Y	Y	Y	Y	Y
	timeconfig	Sets time configuration parameters and HW clock	target	Y	Y	Y	Y	Y
	ucd-snmp-utils	Support for SNMP protocol	target	Y	Y	Y	Y	Y
	usernet	GUIs for manipulating network interfaces	target	Y	Y	Y	Y	Y
	watchdog	Watchdog timer support	Y	Y	Y	Y	Y	Y
	which	Shows full path of a file	target	Y	Y	Y	Y	Y
	xcpustate	Horizontal bar style CPU monitor	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	xosview	Graphical system usage display	target	Y	Y	Y	Y	Y
	xsyinfo	Graphical system usage display	target	Y	Y	Y	Y	Y
	xtoolwait	Speeds up start of X sessions	target	Y	Y	Y	Y	Y
<b>Text</b>								
	diffutils	diff and cmp utilities	target	Y	Y	Y	Y	Y
	diffutils	diff and cmp utilities	cdt	Y	Y	Y	Y	Y
	ed	Line-oriented editor	target	Y	Y	Y	Y	Y
	ed	Line-oriented editor	cdt	Y	Y	Y	Y	Y
	gawk	GNU version of the awk text processing utility	target	Y	Y	Y	Y	Y
	gawk	GNU version of the awk text processing utility	cdt	Y	Y	Y	Y	Y
	grep	grep text search utility	target	Y	Y	Y	Y	Y
	grep	grep text search utility	cdt	Y	Y	Y	Y	Y
	less	Text file browser	target	Y	Y	Y	Y	Y
	less	Text file browser	cdt	Y	Y	Y	Y	Y
	m4	UNIX macro processor	cdt	Y	Y	Y	Y	Y
	sed	Text stream editor	target	Y	Y	Y	Y	Y
	sed	Text stream editor	cdt	Y	Y	Y	Y	Y
	textutils	GNU text file modifying utilities	target	Y	Y	Y	Y	Y

**Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)**

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	textutils	GNU text file modifying utilities	cdt	Y	Y	Y	Y	Y
<b>Communications</b>								
	getty_ps	login support	target	Y	Y	Y	Y	Y
	minicom	Terminal emulator	target	Y	Y	Y	Y	Y
	sliplogin	Dialing SLIP connections	target	Y	Y	Y	Y	Y
<b>Editors</b>								
	vim-common	Common files needed by any version of VIM editor	target	Y	Y	Y	Y	Y
	vim-minimal	Minimal version of VIM editor	target	Y	Y	Y	Y	Y
<b>File</b>								
	file	Identifies file types	target	Y	Y	Y	Y	Y
	file	Identifies file types	cdt	Y	Y	Y	Y	Y
	fileutils	GNU versions of common file management utilities	target	Y	Y	Y	Y	Y
	findutils	find and xargs	target	Y	Y	Y	Y	Y
	gzip	GNU data compression program	target	Y	Y	Y	Y	Y
	slocate	Helps find files	target	Y	Y	Y	Y	Y
	stat	Detailed file information	target	Y	Y	Y	Y	Y
<b>Internet</b>								
	finger	Information on system users	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	ftp	FTP client (file transfers)	target	Y	Y	Y	Y	Y
	fwwhois	Provides information on system users	target	Y	Y	Y	Y	Y
	lynx	Text-based HTML browser	target	Y	Y	Y	Y	Y
	netscape-common	Needed for Netscape Communicator and Navigator	target	Y	N	N	N	N
	netscape-communicator	Netscape Communicator web client	target	Y	N	N	N	N
	netscape-navigator	Netscape Navigator web browser	target	Y	N	N	N	N
	netscape	Netscape tools, including web browser, news reader and e-mail client	target	N	Y	N	N	N
	rsh	Clients and servers for remote access commands (rsh, rlogin, rcp)	target	Y	Y	Y	Y	Y
	tcpdump	Network traffic monitoring tool	target	Y	Y	Y	Y	Y
	telnet	Telnet program	target	Y	Y	Y	Y	Y
	traceroute	Displays IP route information	target	Y	Y	Y	Y	Y
<b>Multimedia</b>								
	libgr-progs	Scripts for graphics manipulation	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
<b>Debuggers</b>								
	ddd	A GUI for several command-line debuggers	cdt	Y	Y	Y	Y	Y
	gdb	GNU debugger	cdt	Y	Y	Y	Y	Y
	gdbserver	Server for GNU source-level debugger for C, C++, and Fortran	target	Y	Y	Y	Y	Y
	strace	Records systems calls and arguments	target	Y	Y	Y	Y	Y
<b>Languages</b>								
	cpp	GNU C-Compatible Compiler Preprocessor	cdt	Y	Y	Y	Y	Y
	egcs	GNU C compiler	cdt	Y	Y	Y	Y	Y
	egcs-c++	GNU C++ compiler	cdt	Y	Y	Y	Y	Y
	expect	Tcl extension for automating interactive programs	cdt	Y	Y	Y	Y	Y
	expect	Tcl extension for automating interactive programs	target	Y	Y	Y	Y	Y
	kaffe	Virtual machine for executing Java Byte Code	target	Y	Y	Y	Y	Y
	kaffe	Virtual machine for executing Java Byte Code	cdt	Y	Y	Y	Y	Y
	perl	Support for the PERL programming language	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	tcl	Tcl scripting language	cdt	Y	Y	Y	Y	Y
	tcl	Tcl scripting language	target	Y	Y	Y	Y	Y
	tclx	Tcl/Tk extensions for POSIX systems	cdt	Y	Y	Y	Y	Y
	tix	Set of capable widgets for Tk	cdt	Y	Y	Y	Y	Y
	tix	Set of capable widgets for Tk	target	Y	Y	Y	Y	Y
	tk	Tk GNI toolkit for Tcl	cdt	Y	Y	Y	Y	Y
	tk	Tk GNI toolkit for Tcl	target	Y	Y	Y	Y	Y
<b>Libraries</b>								
	glibc-devel	Header and object files for development using standard C libraries	target	Y	Y	Y	Y	Y
	glibc-devel	Header and object files for development using standard C libraries	cdt	Y	Y	Y	Y	Y
	glibc-profile	GNU libc libraries, including support for gprof profiling	target	Y	Y	Y	Y	Y
	glib-devel	GIMP ToolKit (GTK+) and GIMP Drawing Kit (GDK) support library	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	glib-devel	GIMP ToolKit (GTK+) and GIMP Drawing Kit (GDK) support library	cdt	Y	Y	Y	Y	Y
	lesstif	LessTif Releasemeister	cdt	Y	Y	Y	Y	Y
	lesstif-devel	Static library and header files for LessTif/Motif-2.0 development	cdt	Y	Y	Y	Y	Y
	libtermcap-devel	Library to access the termcap database	target	Y	Y	Y	Y	Y
	ncurses-devel	Needed for development applications that use ncurses	target	Y	Y	Y	Y	Y
	popt	C library for parsing command line parameters	target	Y	Y	Y	Y	Y
	Xfree86-devel	X Window libraries	cdt	Y	Y	Y	Y	Y
	Xfree86-devel	X Window libraries	target	Y	Y	Y	Y	Y
	xpm-devel	Tools for developing applications	cdt	Y	Y	Y	Y	Y
	xpm-devel	Tools for developing applications	target	Y	Y	Y	Y	Y
	zlib-devel	zlib compression/decompression	cdt	Y	Y	Y	Y	Y
<b>System</b>								
	kernel-headers	C header files for the BlueCat Linux kernel	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	kernel-source	Source code files for BlueCat Linux kernel and kernel object files built for kernel default configuration	target	Y	Y	Y	Y	Y
<b>Tools</b>								
	autoconf	GNU tool for building applications with automatic configuration	Y	Y	Y	Y	Y	Y
	automake	Experimental Makefile generator	Y	Y	Y	Y	Y	Y
	binutils	Binary utilities (ar, nm, objcopy, objdump, ranlib, size, strings, strip, c++filt, addr2line, nlmconv)	target	Y	Y	Y	Y	Y
	binutils	Binary utilities (ar, nm, objcopy, objdump, ranlib, size, strings, strip, c++filt, addr2line, nlmconv)	cdt	Y	Y	Y	Y	Y
	dev86	Assembler and linker for real mode 80x86 instructions	cdt	Y	N	N	N	N
	gettext	Experimental Makefile generator	cdt	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	libtool	Simplifies process of using shared libraries in programs	cdt	Y	Y	Y	Y	Y
	make	make utility for generating executables	cdt	Y	Y	Y	Y	Y
	patch	Apply diff files to original files	cdt	Y	Y	Y	Y	Y
	rcs	Revision control	cdt	Y	Y	Y	Y	Y
<b>Documentation Packages</b>								
	faq	Linux FAQ	target	Y	Y	Y	Y	Y
	helptool	GUI for searching through help resources	cdt	Y	Y	Y	Y	Y
	howto	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-html	Information on configuring and using Linux in HTML format	target	Y	Y	Y	Y	Y
	howto-sgml	Information on configuring and using Linux in SGML format	target	Y	Y	Y	Y	Y
	indexhtml	HTML page with links to Red Hat	target	Y	Y	Y	Y	Y
	install-guide	The Linux Documentation Project <i>Getting Started Guide</i>	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	kernel-doc	Reference to options that can be passed to the BlueCat Linux kernel at load time	target	Y	Y	Y	Y	Y
	man-pages	Manual pages for Linux	target	Y	Y	Y	Y	Y
	nag	<i>Network Administration Guide</i>	target	Y	Y	Y	Y	Y
	sag	Linux <i>System Administration Guide</i> in HTML format	target	Y	Y	Y	Y	Y
	specspo	Portable object catalogs used for internationalization	target	Y	Y	Y	Y	Y
<b>Base</b>								
	adjtimex	Corrects drift in system clock	target	Y	Y	Y	Y	Y
	authconfig	Terminal mode program for setting up NIS (Network Information Service)	target	Y	Y	Y	Y	Y
	chkconfig	Updates run-level information	target	Y	Y	Y	Y	Y
	crontabs	root cron tables	target	Y	Y	Y	Y	Y
	e2fsprogs	Utilities for creating, checking, modifying and correcting inconsistencies in second extended (ext2) filesystems	target	Y	Y	Y	Y	Y

**Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)**

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	e2fsprogs	Utilities for creating, checking, modifying and correcting inconsistencies in second extended (ext2) filesystems	cdt	Y	Y	Y	Y	Y
	eject	Ejects removable media	target	Y	Y	Y	Y	Y
	etcskel	User directory skeleton	target	Y	Y	Y	Y	Y
	genromfs	Runtime support for lightweight read-only filesystems and tools for building them	target	Y	Y	Y	Y	Y
	info	Utility for reading GNU project textinfo files	cdt	Y	Y	Y	Y	Y
	initscripts	Basic system scripts to boot BlueCat Linux system, change run levels, and shut the system down	target	Y	Y	Y	Y	Y
	ipchains	Update of Linux firewalling code	target	Y	Y	Y	Y	Y
	isapnptools	ISA Plug-and-Play support	target	Y	N	N	N	N
	kbdconfig	Setting keyboard map	target	Y	Y	Y	Y	Y
	ld.so	Shared library configuration tool	target	Y	Y	N	N	N
	ldconfig	Determines runtime link bindings for the dynamic loader	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	ldconfig	Determines runtime link bindings for the dynamic loader	cdt	Y	Y	Y	Y	Y
	lilo	The Linux boot loader	target	Y	N	Y	N	N
	logrotate	Simplifies the administration of log files	target	Y	Y	Y	Y	Y
	man	Manual pages for Linux	target	Y	Y	Y	Y	Y
	mingetty	getty program for virtual consoles	target	Y	Y	Y	Y	Y
	mgetty	Smart getty for logins on ttys	target	Y	Y	Y	Y	Y
	mktemp	Creates a temporary filename	target	Y	Y	Y	Y	Y
	mktemp	Creates a temporary filename	cdt	Y	Y	Y	Y	Y
	mount	Mounts and unmounts filesystems, disables and enables swapping	target	Y	Y	Y	Y	Y
	net-tools	Tools for setting up networking such as: arp, rarp, ifconfig, netstat...	target	Y	Y	Y	Y	Y
	ntsysv	Updates and configures run levels	target	Y	Y	Y	Y	Y
	pam	Pluggable Authentication Modules (PAM) support	target	Y	Y	Y	Y	Y

**Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)**

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	passwd	Sets user passwords	target	Y	Y	Y	Y	Y
	pwdb	Password database library	target	Y	Y	Y	Y	Y
	rpm	Red Hat Package Management system	target	Y	Y	Y	Y	Y
	rpm	Red Hat Package Management system	cdt	Y	Y	Y	Y	Y
	setup	System configuration files	target	Y	Y	Y	Y	Y
	shadow-utils	Utilities for managing shadow password files and user/group accounts	target	Y	Y	Y	Y	Y
	shapecfg	Adjusts network traffic shaper bandwidth limiters	target	Y	Y	Y	Y	Y
	SysVinit	Programs that control basic system processes	target	Y	Y	Y	Y	Y
	termcap	Terminal feature database used by certain applications	target	Y	Y	Y	Y	Y
	tmpwatch	Removes files that have not been accessed in a specified period of time	target	Y	Y	Y	Y	Y
	utempter	Privileged helper for utmp/wtmp updates	target	Y	Y	Y	Y	Y
	util-linux	Basic system utilities	target	Y	Y	Y	Y	Y
	vixie-cron	A cron with better security	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	yp-tools	Network Information Service (NIS) client	target	Y	Y	Y	Y	Y
<b>Daemons</b>								
	apache	Apache Web server	target	Y	Y	Y	Y	Y
	at	run commands at certain times	target	Y	Y	Y	Y	Y
	bdflush	Disk cache synchronizer	target	Y	Y	Y	Y	Y
	bind	Named IP name server	target	Y	Y	Y	Y	Y
	bootparamd	Server process needed by some diskless boot clients	target	Y	Y	Y	Y	Y
	dhcp	Server for the Dynamic Host Configuration Protocol	target	Y	Y	Y	Y	Y
	gated	Support for routing protocols: RIP 1 & 2, DCN, HELLO, OSPF 2, EGP 2, and BGP 2 & 4	target	Y	Y	Y	Y	Y
	gpm	Mouse support for text-based applications	target	Y	Y	Y	Y	Y
	knfsd	High performance kernel-level NFS server	target	Y	Y	Y	Y	Y
	lpr	Manages printing services	target	Y	Y	Y	Y	Y
	netkit-base	ping and inetd networking programs	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	portmap	Manages RPC connections; includes security for NIS & NFS information	target	Y	Y	Y	Y	Y
	ppp	PPP daemon	target	Y	Y	Y	Y	Y
	pump	DHCP and BOOTP client daemons in one	target	Y	Y	Y	Y	Y
	routed	Routing daemon handles incoming RIP traffic	target	Y	Y	Y	Y	Y
	samba	SMB server	target	Y	Y	Y	Y	Y
	sendmail	Mail transport agent	target	Y	Y	Y	Y	Y
	syslogd	syslogd and klogd daemons for system logging	target	Y	Y	Y	Y	Y
	tcp_wrappers	Security tool that acts as a wrapper for TCP daemons	target	Y	Y	Y	Y	Y
	tftp	Trivial File Transfer Protocol	target	Y	Y	Y	Y	Y
	timed	Needed for keeping network computers' times synchronized	target	Y	Y	Y	Y	Y
	ucd-snmp	SNMP support	target	Y	Y	Y	Y	Y
	wu-ftpd	FTP server daemon	target	Y	Y	Y	Y	Y
	Xfree86-xfs	Font server for the X Window System	target	Y	Y	Y	Y	Y
	xntp3	NTP network time synchronization	target	Y	Y	Y	Y	Y
	ypbind	NIS client daemon	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	ypserv	NIS server daemon	target	Y	Y	Y	Y	Y
<b>Kernel</b>								
	kernel	Linux kernel binary built for the BlueCat Linux kernel default configuration	target	Y	Y	Y	Y	Y
	kernel-bcboot	The BlueCat Linux boot sector	target	Y	Y	Y	Y	Y
	kernel-pcmcia-cs	Loadable kernel modules to support PCMCIA cards	target	Y	N	N	N	N
	modutils	Kernel daemon and kernel module utilities	target	Y	Y	Y	Y	Y
<b>Shared Libraries</b>								
	cracklib	Password-checking library	target	Y	Y	Y	Y	Y
	cracklib-dicts	Dictionaries for cracklib	target	Y	Y	Y	Y	Y
	glib	Library of handy utility functions	target	Y	Y	Y	Y	Y
	glib	Library of handy utility functions	cdt	Y	Y	Y	Y	Y
	glib10	More useful C functions	target	Y	Y	Y	Y	Y
	glibc	GNU libc libraries	target	Y	Y	Y	Y	Y
	glibc	GNU libc libraries	cdt	Y	Y	Y	Y	Y
	gmp	Arbitrary precision math	target	Y	Y	Y	Y	Y
	gtk+	GIMP toolkit	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	gtk+10	X libraries for GIMP and other programs	target	Y	Y	Y	Y	Y
	gtk-engines	Graphical engines for the GTK+ toolkit themes	target	Y	Y	Y	Y	Y
	imlib	Image loading and rendering library for X	target	Y	Y	Y	Y	Y
	libelf	Access to elf object files	target	Y	Y	Y	Y	Y
	libghttp	Library of routines for handling HTTP 1.1 requests	target	Y	Y	Y	Y	Y
	libgr	Library for handling various graphics file formats including pixmaps	target	Y	Y	Y	Y	Y
	libjpeg	Functions for manipulating JPEG images	target	Y	Y	Y	Y	Y
	libpng	Library for manipulating PNG graphics image format files	target	Y	Y	Y	Y	Y
	libstdc++	Standard C++ library	target	Y	Y	Y	Y	Y
	libtermcap	Access to the termcap database	target	Y	Y	Y	Y	Y
	libtiff	Manipulates TIFF image files	target	Y	Y	Y	Y	Y
	libungif	Loading and saving GIF format files	target	Y	Y	Y	Y	Y
	libxml	Manipulates XML files	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	ncurses	Terminal-independent method of updating character screens	target	Y	Y	Y	Y	Y
	ncurses	Terminal-independent method of updating character screens	cdt	Y	Y	Y	Y	Y
	newt	Supports color text mode user interfaces	target	Y	Y	Y	Y	Y
	readline	Library for reading and returning lines from a terminal	target	Y	Y	Y	Y	Y
	slang	Shared library for the S-Lang extension languages	target	Y	Y	Y	Y	Y
	svgalib	Low-level SVGA graphics library	target	Y	N	N	N	N
	Xfree86-libs	Shared libraries needed by the X Window System version 11 release 6	cdt	Y	Y	Y	Y	Y
	Xfree86-libs	Shared libraries needed by the X Window System version 11 release 6	target	Y	Y	Y	Y	Y
	xpm	Pixmap library for the X Window system	cdt	Y	Y	Y	Y	Y
	xpm	Pixmap library for the X Window system	target	Y	Y	Y	Y	Y
	zlib	In-memory compression support	target	Y	Y	Y	Y	Y

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	zlib	In-memory compression support	cdt	Y	Y	Y	Y	Y
<b>Shells</b>								
	ash	Smaller version of the Bourne shell	target	Y	Y	Y	Y	Y
	bash	GNU Bourne Again shell (bash)	target	Y	Y	Y	Y	Y
	sash	Small shell that needs no shared libraries	target	Y	Y	Y	Y	Y
	sh-utils	GNU utilities commonly used in shell scripts	target	Y	Y	Y	Y	Y
	tcsh	C shell clone	target	Y	Y	Y	Y	Y
<b>Hardware Support</b>								
	Xconfigurator	Menu driven program for configuring an X server	target	Y	Y	Y	Y	Y
	XFree86-SVGA	XFree86 server for most simple framebuffer SVGA devices	target	Y	N	N	N	N
	XFree86-VGA16	Generic XFree86 server for VGA16 boards	target	Y	N	N	N	N
	XFree-86-XF86Setup	GUI for setting up and configuring X servers	target	Y	N	N	N	N
<b>X</b>								

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	kterm	Terminal emulator for the Japanese character set	target	Y	Y	Y	Y	Y
	rxvt	Color VT102 emulator	target	Y	Y	Y	Y	Y
	X11R6-contrib	Many useful X programs	target	Y	Y	Y	Y	Y
	XFree86	Basic fonts, programs and docs for an X workstation	cdt	Y	Y	Y	Y	Y
	XFree86	Basic fonts, programs and docs for an X workstation	target	Y	Y	Y	Y	Y
	XFree86-ISO8859-2-Type-1-fonts	Extra Central European fonts	target	Y	Y	Y	Y	Y
	XFree86-75dpi-fonts	Medium resolution fonts	target	Y	Y	Y	Y	Y
	XFree86-ISO8859-2-100dpi-fonts	High resolution Central European fonts	target	Y	Y	Y	Y	Y
	XFree86-ISO8859-2-75dpi-fonts	Medium resolution Central European fonts	target	Y	Y	Y	Y	Y
	XFree86-ISO8859-9-100dpi-fonts	High resolution Turkish fonts	target	Y	Y	Y	Y	Y
	XFree86-ISO8859-9-75dpi-fonts	Medium resolution Turkish fonts	target	Y	Y	Y	Y	Y
	xinitrc	Starts window managers	target	Y	Y	Y	Y	Y
<b>BlueCat Linux Packages</b>								

Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	bc_misc	Miscellaneous BlueCat Linux utilities	cdt	Y	Y	Y	Y	Y
	demo	BlueCat Linux demo system configuration	target	Y	Y	Y	Y	Y
	ffs	Flash File System	target	Y	Y	Y	Y	Y
	irshd	A modified version of the standard rshd server	target	Y	Y	Y	Y	Y
	lddyn_fix	A package to make cross tools work in a cross development environment	cdt	Y	Y	Y	Y	Y
	mapm	Advanced Power Management	target	Y	Y	Y	Y	Y
	mkboot	Creates a bootable disk (floppy or hard disk) with the BlueCat Linux system	target	Y	Y	Y	Y	Y
	mkboot	Creates a bootable disk (floppy or hard disk) with the BlueCat Linux system	cdt	Y	Y	Y	Y	Y
	mkrootfs	BlueCat Linux utilities for maintaining multiple embedded systems (mkimage, etc.)	cdt	Y	Y	Y	Y	Y
	mkrootfs_wrappers	Set of utilities providing open-source “wrapper” functionality to mkrootfs	cdt	Y	Y	Y	Y	Y

**Table A-1: Packages in the Default BlueCat Linux Configuration (Continued)**

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	msng	LynuxWorks Messenger	target	Y	Y	N	Y	Y
	pcdsrvr	Contains PosixWorks Cross Development Server	target	Y	Y	Y	Y	Y
	pftpd	Cross development host side daemon that allows booting BlueCat Linux target boards from a parallel port	cdt	Y	Y	Y	Y	Y
	sm	Linux Startup Monitor and BlueCat Linux utility for changing it	cdt	N	Y	N	N	N

# Optional BlueCat Linux Packages

This appendix describes optional packages available on the BlueCat Linux distribution CD-ROM. The following table shows optional packages available on the BlueCat Linux distribution CD-ROM.

Table B-1: Optional BlueCat Linux Packages

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
<b>Archiving</b>								
	dump	dump and restore backup	target	Y	Y	Y	Y	Y
<b>Publishing</b>								
	freetype-utils	Several utilities to manipulate and examine TrueType fonts	target	Y	Y	Y	Y	Y
	ghostscript	PostScript and PDF viewer	target	Y	Y	Y	Y	Y
	ghostscript-fonts	Fonts for ghostscript	target	Y	Y	Y	Y	Y
	groff-gxditview	X display for groff target board	target	Y	Y	Y	Y	Y
	gv	User interface for ghostscript	target	Y	Y	Y	Y	Y
	texinfo	Creates on-line documents from TeX source	target	Y	Y	Y	Y	Y

Table B-1: Optional BlueCat Linux Packages (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	texinfo	Creates on-line documents from TeX source	cdt	Y	Y	Y	Y	Y
	sgml-tools	SGML formatting package	target	Y	Y	Y	Y	Y
<b>Application/System</b>								
	procps-X11	Backwards compatible wrapper for xconsole	target	Y	Y	Y	Y	Y
	samba_client	Samba (SMB) client programs	target	Y	Y	Y	Y	Y
	setuptools	Text-mode utility	target	Y	Y	Y	Y	Y
	statserial	Shows serial port handshaking lines	target	Y	Y	Y	Y	Y
	vlock	Locks console access	target	Y	Y	Y	Y	Y
<b>Communications</b>								
	mgetty-sendfax	Sends faxes	target	Y	Y	Y	Y	Y
	mgetty-viewfax	X11 utility to view faxes	target	Y	Y	Y	Y	Y
	mgetty-voice	Voice modem support	target	Y	Y	Y	Y	Y
<b>Editors</b>								
	vim-enhanced	VIM version of the vi editor, which includes recent enhancement	target	Y	Y	Y	Y	Y
	vim-X11	VIM version of the vi editor for the X Window system	target	Y	Y	Y	Y	Y
<b>Engineering</b>								

**Table B-1: Optional BlueCat Linux Packages (Continued)**

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	bc	Calculator and arithmetic language (bc & dc)	target	Y	Y	Y	Y	Y
<b>Multimedia</b>								
	libungif-progs	Programs for manipulating GIF images	target	Y	Y	Y	Y	Y
	mpg123	MPEG audio player	target	Y	Y	Y	Y	Y
	multimedia	Multimedia X utilities	target	Y	Y	Y	Y	Y
<b>Languages</b>								
	itcl	Object mega widgets for Tcl	cdt	Y	Y	Y	Y	Y
	itcl	Object mega widgets for Tcl	target	Y	Y	Y	Y	Y
	python	Python programming language	cdt	Y	Y	Y	Y	Y
	python	Python programming language	target	Y	Y	Y	Y	Y
	tclx	Tcl/Tk extensions for POSIX systems	target	Y	Y	Y	Y	Y
	tkinter	Tk interface for Python	cdt	Y	Y	Y	Y	Y
	tkinter	Tk interface for Python	target	Y	Y	Y	Y	Y
<b>Libraries</b>								
	apache-devel	Needed to develop additional modules for the Apache web server	target	Y	Y	Y	Y	Y

Table B-1: Optional BlueCat Linux Packages (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	audiofile-devel	Needed to develop audiofile applications	target	Y	Y	Y	Y	Y
	bind-devel	DNS development for bind service	target	Y	Y	Y	Y	Y
	e2fsprogs-devel	ext2 filesystem specific static libraries and headers	target	Y	Y	Y	Y	Y
	e2fsprogs-devel	ext2 filesystem specific static libraries and headers	cdt	Y	Y	Y	Y	Y
	esound-devel	Used to develop Esound applications	target	Y	Y	Y	Y	Y
	fnlib-devel	Headers, libraries, and documentation for Fnlib	target	Y	Y	Y	Y	Y
	freetype-devel	Needed to develop and compile applications that use the FreeType library	target	Y	Y	Y	Y	Y
	gdbm-devel	GNU database system library	target	Y	Y	Y	Y	Y
	gd-devel	Needed for the gd graphics library	target	Y	Y	Y	Y	Y
	glibc-profile	GNU libc libraries, including support for gprof profiling	cdt	Y	Y	Y	Y	Y
	gmp-devel	GNU MP arbitrary precision library	target	Y	Y	Y	Y	Y

Table B-1: Optional BlueCat Linux Packages (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	gnome-libs-devel	Used to develop GNOME applications	target	Y	Y	Y	Y	Y
	gpm-devel	Used to develop text-mode programs that use the mouse	target	Y	Y	Y	Y	Y
	gtk+-devel	Static libraries and header files needed for developing GIMP ToolKit (GTK+) applications	target	Y	Y	Y	Y	Y
	imlib-devel	Image loading and rendering library for X11R6	target	Y	Y	Y	Y	Y
	libghttp-devel	Needed for developing libghttp	target	Y	Y	Y	Y	Y
	libgr-devel	Needed for developing programs that handle the graphics file formats supported by libgr	target	Y	Y	Y	Y	Y
	libjpeg-devel	Needed to develop applications that use the libjpeg library	target	Y	Y	Y	Y	Y
	libpcap	Low-level network traffic monitory library	target	Y	Y	Y	Y	Y

Table B-1: Optional BlueCat Linux Packages (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	libpng-devel	Library to manipulate PNG (Portable Network Graphics) format files	target	Y	Y	Y	Y	Y
	libtiff-devel	Library to manipulate TIFF format files	target	Y	Y	Y	Y	Y
	libungif-devel	Library to load and save GIF image files	target	Y	Y	Y	Y	Y
	libxml-devel	Needed for development of libxml applications	target	Y	Y	Y	Y	Y
	ncurses-devel	Needed for development applications that use ncurses	cdt	Y	Y	Y	Y	Y
	newt-devel	Needed for development applications that use Newt – a text-mode user interface library based on slang	target	Y	Y	Y	Y	Y
	ORBit-devel	ORBit CORBA ORB	target	Y	Y	Y	Y	Y
	pciutils-devel	Inspecting and setting devices connected to the PCI bus	target	Y	Y	Y	Y	Y
	python-devel	Library to add Python extensions	cdt	Y	Y	Y	Y	Y
	python-devel	Library to add Python extensions	target	Y	Y	Y	Y	Y

Table B-1: Optional BlueCat Linux Packages (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	readline-devel	Development files for programs that use the readline library	target	Y	Y	Y	Y	Y
	rpm-devel	Used for creating tools that need an intimate knowledge of RPM	target	Y	Y	Y	Y	Y
	rpm-devel	Used for creating tools that need an intimate knowledge of RPM	cdt	Y	Y	Y	Y	Y
	slang-devel	Static library and header files for development using S-Lang	target	Y	Y	Y	Y	Y
	svgalib-devel	Needed for developing applications that use the SVGAlib low- level graphics library	target	Y	N	N	N	N
	ucd-snmp-devel	Libraries and header files for use with the UCD-SNMP tools	target	Y	Y	Y	Y	Y
	Xaw3d-devel	3-D dimensional look for the Athena widgets for X	target	Y	Y	Y	Y	Y
	zlib-devel	zlib compression/ decompression	target	Y	Y	Y	Y	Y
<b>Tools</b>								

Table B-1: Optional BlueCat Linux Packages (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	ElectricFence	Checks for <i>malloc()</i> violations	cdt	Y	Y	Y	Y	Y
	flex	Tool for creating scanners (text pattern recognizers)	target	Y	Y	Y	Y	Y
	python-tools	Python programming language tools	cdt	Y	Y	Y	Y	Y
	python-tools	Python programming language tools	target	Y	Y	Y	Y	Y
<b>Documentation Packages</b>								
	howto-chinese	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-croatian	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-french	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-german	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-greek	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-indonesian	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-italian	Information on configuring and using Linux	target	Y	Y	Y	Y	Y

**Table B-1: Optional BlueCat Linux Packages (Continued)**

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	howto-japanese	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-korean	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-polish	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-russian	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-serbian	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-slovenian	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-spanish	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-swedish	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	howto-turkish	Information on configuring and using Linux	target	Y	Y	Y	Y	Y
	lpg	Guide to programming on Linux systems in HTML format	target	Y	Y	Y	Y	Y
	python-docs	Documentation for Python language	cdt	Y	Y	Y	Y	Y
	python-docs	Documentation for Python language	target	Y	Y	Y	Y	Y

Table B-1: Optional BlueCat Linux Packages (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	sendmail-doc	Documentation for the send mail transport agent (MTA)	target	Y	Y	Y	Y	Y
	XFree86-doc	Documentation on various X11 programming interfaces	target	Y	Y	Y	Y	Y
<b>Base</b>								
	chkfontpath	Adds and removes X font path information	target	Y	Y	Y	Y	Y
	genromfs	Runtime support for lightweight read-only filesystems and tools for building them	cdt	Y	Y	Y	Y	Y
	info	Utility for reading GNU project textinfo files	target	Y	Y	Y	Y	Y
	losetup	Loop block device for setting up virtual filesystems	target	Y	Y	Y	Y	Y
	mailcap	Describes how to deal with received non-text mail messages	target	Y	Y	Y	Y	Y
	mouseconfig	Sets up and configures the mouse	target	Y	Y	Y	Y	Y
	quota	Monitors and limits disk usage	target	Y	Y	Y	Y	Y
	raidtools	Software raid support	target	Y	Y	Y	Y	Y

**Table B-1: Optional BlueCat Linux Packages (Continued)**

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	rootfiles	root account default files	target	Y	Y	Y	Y	Y
<b>Daemons</b>								
	am-utils	BSD automounter	target	Y	Y	Y	Y	Y
	esound	Allows multiple applications to use the sound card	target	Y	Y	Y	Y	Y
	ORBit	CORBA ORB	target	Y	Y	Y	Y	Y
	sendmail-cf	Needed for reconfig using the sendmail.cf file	target	Y	Y	Y	Y	Y
<b>Kernel</b>								
	kernel-smp	SMP version of Linux kernel	target	Y	Y	N	N	N
<b>Shared Libraries</b>								
	audiofile	Handles audio sound format files	target	Y	Y	Y	Y	Y
	cracklib	Password-checking library	cdt	Y	Y	Y	Y	Y
	cracklib-dicts	Dictionaries for cracklib	cdt	Y	Y	Y	Y	Y
	fnlib	24-bit font rendering for X	target	Y	Y	Y	Y	Y
	freetype	TrueType font rendering	target	Y	Y	Y	Y	Y
	gd	Graphics library for handling GIF files	target	Y	Y	Y	Y	Y
	gdbm	GNU database indexing library	target	Y	Y	Y	Y	Y

Table B-1: Optional BlueCat Linux Packages (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	gnome-libs	Libraries needed for GNOME	target	Y	Y	Y	Y	Y
	imlib-cfgeditor	Configuration editor for imlib	target	Y	Y	Y	Y	Y
	Xaw3d	Athena Widget enhancements for X	target	Y	Y	Y	Y	Y
<b>Desktop</b>								
	lesstif-mwm	A LessTif/Motif(R) window manager based on fvwm	cdt	Y	Y	Y	Y	Y
<b>Hardware Support</b>								
	XFree86-3DLabs	XFree86 server for 3DLabs video cards	target	Y	N	N	N	N
	XFree86-8514	XFree86 server program for older IBM 8514 or compatible video cards	target	Y	N	N	N	N
	XFree86-AGX	XFree86 server for AGX-based video cards	target	Y	N	N	N	N
	XFree86-FBDev	X server for the generic frame buffer device on some machines	target	Y	Y	Y	Y	Y
	XFree86-I128	The XFree86 server for Number Nine Imagine 128 video cards	target	Y	N	N	N	N
	XFree86-Mach32	XFree86 server for Mach32-based video cards	target	Y	N	N	N	N

Table B-1: Optional BlueCat Linux Packages (Continued)

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	XFree86-Mach64	XFree86 server for Mach64-based video cards	target	Y	N	N	N	N
	XFree86-Mach8	XFree86 server for Mach8 video cards	target	Y	N	N	N	N
	XFree86-Mono	Generic XFree86 monochrome server for VGA cards	target	Y	N	N	N	N
	XFree86-P9000	XFree86 server for P9000 cards	target	Y	N	N	N	N
	XFree86-S3	XFree86 server for video cards based on the S3 chip	target	Y	N	N	N	N
	XFree86-S3V	XFree86 server for video cards based on the S3 Virge chip	target	Y	N	N	N	N
	XFree86-W32	XFree86 server for video cards based on ET4000/W32 chips	target	Y	N	N	N	N
	XFree86-Xnest	Nested window server	target	Y	Y	Y	Y	Y
	XFree86-Xvfb	Virtual framebuffer X Window System server for XFree86	target	Y	Y	Y	Y	Y
<b>X</b>								
	lesstif-clients	UIL and xmbind, two separate LessTif add-ons	cdt	Y	Y	Y	Y	Y
	urw-fonts	PostScript fonts	target	Y	Y	Y	Y	Y
	XFree86-100dpi-fonts	X Window System 100dpi fonts	target	Y	Y	Y	Y	Y

**Table B-1: Optional BlueCat Linux Packages (Continued)**

Category	Package Name	Description	cdt/ target	x86	PPC	ARM	SH	MIPS
	XFree86-cyrillic-fonts	Cyrillic fonts for X	target	Y	Y	Y	Y	Y
	XFree86-ISO8859-2	Central European font support	target	Y	Y	Y	Y	Y
	XFree86-ISO8859-9	Turkish fonts	target	Y	Y	Y	Y	Y

This appendix includes a man page describing the `mkrootfs` utility.

## UTILITY

`mkrootfs`

## SYNOPSIS

Builds a target board filesystem image according to a specification file.

## DESCRIPTION

```
mkrootfs [-DilLTv] [-N inodes] [-r \
blocks] spec_file output_file
mkrootfs [-ilLv] -J [chunk size] spec_file \
output_file
makerootfs -t [-ilLv] spec_file
```

This utility creates a gzipped target board filesystem 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 filesystem for the target board.

A *specfile* describes the target board filesystem contents and configuration. Its syntax is similar to a shell script and is explained in “Specfile Format”.

Currently `mkrootfs` supports two target board filesystem types – the Linux `ext2` filesystem (for compressed images) and the BlueCat Linux Flash File System (FFS).

## Specfile Format

*Specfile* 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 board filesystem paths (affects all subsequent relative target board filesystem 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 board filesystem (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 filesystem.

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 filesystem.

`rm dst... dst`

Removes files/directories from target board filesystem.

Allows any wildcards. Globbing is performed just like in `sh`. Removes directories recursively.

`ln [-s] src/path dst/path`

Creates target board filesystem 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 filesystem.

With `-p`, `mkdir` creates intermediate directories if required. The `-m` option allows the user to specify permissions for the `dir` 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 filesystem 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, and they 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/dir* owner.

It works recursively in the same manner as `chmod`.

`mkrootfs` supports a simple `if-else-endif` construct. It has the following syntax:

```
if arg1 = arg2
...
[else]
...
endif
```

This can be used to arrange for conditional parsing of the specification file. For example:

```
if $BLUECAT_TARGET_CPU = ppc
... # PowerPC part
else
... # non-PowerPC part
endif
```

Note that variable substitution and argument parsing is performed as with any other commands.

---

## Options

<code>-D</code>	Does not create a <code>lost+found</code> directory on the target board filesystem.
-----------------	---

- 
- `-i` Ignores non-fatal errors. By default, *mkrootfs* fails on errors that can result in target board filesystem 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 filesystem. Furthermore, after building the target board filesystem, *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 filesystem instead of the default value, which is calculated based on the filesystem size and can be incorrect for the user's embedded system.
- `-r freespace` Reserves specified free space on the target board filesystem. The argument must be a numeric value indicating the number of free 1 KB blocks that should be allocated.
- `-t` Test mode – No image is created; all required actions are printed instead.
- `-T` Creates a `tar` file on output. When this option is specified, no image file is created, but the `output_file` argument is used as a filename for the resulting `tar` archive.
- `-J [chunk_size]` Creates a Flash File System (FFS) image. This image can be downloaded into flash memory and then mounted as a root

filesystem when BlueCat Linux boots on the target board. The `chunk_size` argument is optional and can be used to specify the maximum size of a data chunk for the resulting image. By default this value is 4096 bytes, which is suitable in most cases.

-v

Verbose mode – Produces verbose output.

This appendix includes a man page describing the `mkboot` utility.

## UTILITY

`mkboot`

## SYNOPSIS

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 filesystem 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 filesystem, 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 on a hard disk removes any boot loader present on the disk. The user will not be able to boot operating systems other than BlueCat Linux from the disk he has updated using the `mkboot` utility. Also, reinstalling the boot sector invalidates all the booting options set by previous calls to `mkboot` for this disk.
- 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 filesystem image to the media. *none* removes compressed root filesystem. *stdin* takes the

---

	image of the root filesystem from the standard input.
<code>-r <i>xxxx/device</i></code>	Sets the device node on the target board to mount as the root filesystem or uncompress the filesystem image from. For instance, 200 corresponds to <code>/dev/fd0</code> , and 801 corresponds to <code>/dev/sda1</code> . Instead of the major/minor number, the root filesystem can be specified as the standard name of the device node. For example: <code>/dev/hd**</code> , <code>/dev/sd**</code> , <code>/dev/fd**</code> , <code>/dev/tffs*</code> .
<code>-m</code>	Tells <code>mkboot</code> to create an image composed of a BlueCat Linux kernel (specified by the <code>-k</code> flag) and a compressed filesystem (specified by the <code>-f</code> flag) suitable for programming into target ROM/flash memory or downloading over a network by the target board firmware.
<code>-i</code>	Does not automatically install target board-specific parameters into the kernel command line.
<code>-q</code>	Quiet mode – Only error messages are printed on a console.

---

## Examples

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

- On the Linux host:

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/hello
BlueCat:$ mkboot -b -k hello.disk -f \
hello.rfs -r /dev/fd0 /dev/fd0
```

- On a Windows host:

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/hello
BlueCat:$ mkboot -b -k hello.disk -f \
hello.rfs -r /dev/fd0 a:
```

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

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

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

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

or

```
BlueCat:$ cd $BLUECAT_PREFIX/demo/hello
BlueCat:$ mkboot -b /dev/fd0
BlueCat:$ mkboot -k hello.disk -f hello.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 filesystem contained in `/dev/sda1` as the root filesystem 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 `hello` demo system:

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

This appendix includes a man page describing the `mkkernel` utility.

## UTILITY

`mkkernel`

## SYNOPSIS

Builds the BlueCat Linux kernel.

```
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`.



# *mapmd Command Reference*

This appendix includes a man page describing the `mapmd` utility.

## UTILITY

`mapmd`

## SYNOPSIS

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

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

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



---

## *mapm\_ctrl* Command Reference

This appendix includes a man page describing the `mapm_ctrl` utility.

### UTILITY

`mapm_ctrl`

### SYNOPSIS

BlueCat Linux Advanced Power Management (APM) control utility

```
mapm_ctrl [ PMD_handle_or_name state [ timer \  
          arg1, 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

<i>state</i>	Can be one of the following: ON, OFF, STOP, AUTO, SPECIAL
<i>timer</i>	If specified, the inactivity timer is primed for the PMD. The timer is specified in milliseconds.
<i>arg</i>	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 1 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.conf Command Reference*

This appendix includes a man page describing the `mapmd.conf` utility.

### UTILITY

`mapmd.conf`

### SYNOPSIS

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
```

## *flash\_fdisk Command Reference*

This appendix includes a man page describing the `flash_fdisk` utility.

### UTILITY

`flash_fdisk`

### SYNOPSIS

Partitions a flash memory device at runtime.

### DESCRIPTION

`flash_fdisk device_node configuration_string`

The `flash_fdisk` utility modifies the partition configuration of a specified flash memory device at runtime. The device node specified as a parameter must correspond to a flash character device node for an entire flash memory device.

---

**NOTE:** *Please bear in mind that the partition information is not written to flash memory, and needs to be reestablished at every boot.*

---

---

## Configuration String Format

The configuration string has the following format:

`single_part_conf[:single_part_conf]...`

where

`single_part_conf={number[ , |-number]}`

*number* is a decimal number.

Numbers in the configuration string correspond to the sectors allocated to the particular partition. Configuration for different partitions is separated by colons (:).

---

## 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
```

## *flash\_erase Command Reference*

This appendix includes a man page describing the `flash_erase` utility.

### UTILITY

`flash_erase`

### SYNOPSIS

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 3 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.



## *pftpd Command Reference*

This appendix includes a man page describing the `pftpd` utility.

### UTILITY

`pftpd`

### SYNOPSIS

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

<code>start</code>	Starts the <code>pftpd</code> daemon.
<code>stop</code>	Stops all running <code>pftpd</code> daemon processes.
<code>--help</code>	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 of the following format:

<code>pftproot = value</code>	Defines the root directory from which the file searching starts.
<code>allow = value</code>	Defines subdirectories of the root directory to which client access is enabled; specifies an empty value to allow access to all files in the <code>pftproot</code> directory.
<code># comment</code>	The comment text is ignored.

---

**LINUX NOTE:** *In order to enable the daemon, the system administrator must perform installation of a low-level driver.*

---

---

# Index

---

## Symbols

\$BLUECAT\_PREFIX/demo directory 92  
.config file 98  
.spec file 98  
/proc/mapm directory 162  
/proc/mapm file 151, 152  
/proc/mapm/pmds file 162  
/proc/mtd file 195

---

## A

action identifier 150  
Activating Support for a Target Board 12  
Adding New Commands to BLOSH 89  
Advanced Power Management 137, 141  
algorithm  
    garbage collection 182  
    wear leveling 182  
API 142  
APM 141, 253, 255  
    callback interface 143  
    client callback 145  
    client deregistration 166  
    client registration 165  
    Common Constants and Data Types 154  
    configuring 153  
    Control Utility 164  
        Command Line Format 164  
    core module 143  
    daemon 151  
    drivers, developing 165  
    error codes 154  
    event 148  
    event descriptor 148  
    event processing 166  
    Event Queue 144  
    General Architecture 141  
    interactions with client 142  
    Interfaces 148  
    Interfaces Reference 154  
    IOCTL interface 148  
    Kernel-space client interface 148  
    kernel-space interface 143  
    Modules and Components 142  
    PMD device drivers interface 148  
    proc file 151, 152  
    user-space interface driver 143, 150  
APM control utility 151, 152  
APM core module 152  
    event processing kernel task 145  
application programs  
    building 38  
    debugging 38  
    developing 21, 38  
arbitrary file download 79  
arbitrary storage hierarchy 174  
architecture, FFS 169  
architecture-independent interface 141  
archive, downloading and unpacking 79  
Auto-booting BlueCat Linux 86  
    target board-specific 87  
auto-configuration, IP 81  
Automatic Bad Block Mapping 184  
autonomous power management capabilities 146

**B**

- Bad block mapping 184
- benchmark, memory sizing 44
- Binary Architecture CD-ROM 3
  - Directories 4
  - documents directory 5
  - Subdirectories 5
  - Tree structure 3
- BIOS 69
- block device
  - access mechanism 171
  - interface 169
  - IOCTL commands 171
  - node 174
- BLOSH 74
  - boot types supported 75
  - Command Reference 76
  - Environment Variables 74
  - new commands to 89
  - rebuilding 89
  - Startup Sequence 74
- BlueCat Linux
  - Advanced Power Management 141–168
  - Binary Architecture CD-ROM 3
  - Boot Procedure Overview 47
  - Components 17
  - copying on floppy 49
  - Core Components for Supported Target Boards 2
  - default configuration 9
  - Development Process 26
  - Directory Structure 16
  - Distribution Overview 1
  - Installation CD-ROMs for a microprocessor family 2
  - installing 1, 9
  - Interfaces to Flash Memory 169
  - kernel debugger 34
  - Kernel RPM Packaging 23
  - Kernel vs. “Pristine” Linux Kernel 24
  - Messenger 133
  - OS Loader 73
  - Root Filesystem Utility – mkrootfs 40
  - RPM 15
  - Source Architecture CD-ROM 3, 24
  - Target Support Package CD-ROM 3

- uninstalling 19, 20
- BlueCat Linux Embedded System 25
- BlueCat Linux Loader Shell (BLOSH) 74
- BlueCat Linux Target Support Guide 3
- BlueCat Linux User’s Guide 3
- BLUECAT\_PREFIX 18
- boot command 76
- Booting
  - BlueCat Linux Kernel 76
    - from a Hard Disk 57
    - from DiskOnChip 57
    - from NFS Server 85
    - from TFTP Server 85
  - Images from a Different Subnet 82
  - over a Network or Parallel Port 71
- Booting BlueCat Linux 47–90
  - from Hard Disk 50
  - from PFTP Server 86
  - from target ROM/flash Memory 59
  - from target ROM/flash memory using Firmware 69
  - over a Network Using Target Board Firmware 71
  - over Network or Parallel Port, using OS Loader 72
- BOOTP server, setting up 81
- Building
  - Application Programs 38
  - Demo Systems 100
  - Kernel 30
    - for Debug Purposes 33
    - for x86 30
  - Root Filesystem 40
- Build-time parameters 194

---

**C**

- caffeine subdirectory 92
- callback parameters 159
- callback return code 145
- cd command 77
- CD-ROM
  - Binary Architecture 3
  - distribution media 1, 2
  - Source Architecture 3
  - Target Support Package 3
- Change Current Working Directory 77
- character device node 174

- character-device interface 169
  - Client 148
  - client callback 145, 156
    - callback return code 145
  - client handle 156, 163
  - client services 157
  - command line format, APM control utility 164
  - command reference
    - FFS IOCTL 187
    - flash\_erase 261
    - mapm\_ctrl 255
    - mapmd 253
    - mkboot 247
    - mkrootfs 241
    - pfptd 263
  - components
    - APM 142
    - user-space 151
  - Compressed Root Filesystem
    - Copying from on Hard Disk 53
  - Configuration, default 9
  - Configuring
    - APM in the Kernel 153
    - BlueCat Linux OS Loader 88
    - Demo System 99
    - Demos for Hardware Devices 99
    - Demos for the Boot Device 99
    - Flash Memory Partitions 194
    - Flash memory partitions at build time 194
    - Hardware Device Support 88
    - Kernel 28
    - OS Loader as a demo system 88
    - Partitions at Runtime 200
    - Partitions using flash\_fdisk 195
    - Partitions using kernel boot-time parameters 195
  - control utility, APM 164
  - Copying BlueCat Linux
    - onto Hard Disk from cross development host 50
    - to a DiskOnChip Device 58
    - to a Floppy Disk 49
    - to Hard Disk using install Demo System 55
    - to Hard Disk using OS Loader 52
    - with Compressed Root Filesystem to Hard Disk 53
  - Core Components for Supported Target Boards 2
  - CPU
    - PMD driver 142
    - Power Management 152
    - power state 142
  - Creating
    - Bootable Disk 78
    - Image for Booting from Network using firmware 72
    - Root filesystem in FFS on Target Board using install Demo System 67
    - Root Filesystem in FFS on Target Board using OS Loader 63
  - cross development host
    - Linux 2, 18
    - system requirements 2
    - Windows 2
  - cross development tools 24, 48
  - custom power management 147
  - Customizing
    - BlueCat Linux OS Loader 89
    - Kernel 28
    - the Kernel for Size 42
  - Cywin environment, installing on Windows host 10
- 
- D**
- data chunks 176
  - Debugger Demos 117
  - Debugging
    - Application Programs 38
    - finishing 36
    - Kernel 33
    - requirements 33
    - starting 35
  - Default
    - BlueCat Linux configuration 9
    - BlueCat Linux Packages 203–226
    - Kernel Configuration 29
  - default subdirectory 92
  - Demo System 24, 92
    - Boot Devices 101
    - building 100
    - caffeine 135
    - Components 97
    - Conceptual Overview 91

- Configuring 99
- configuring for boot device 99
- configuring for hardware device 99
- default 107
- developer 130
- directory contents 98
- disk 124
- diskboot 126
- ffs 136
- Files and Subdirectories 98
- ftp 113
- gdb 117
- gnutar 124
- hello 104
- install 114
- kdbg 119
- loadkeys 110
- Location 92
- mapm 137
- Memory Size Options 103
- memsize 116
- modular 105
- msgng\_exmpl 133
- msgng\_minet 133
- multi\_user 108
- multi\_user\_net 109
- nfsroot 131
- osloader 115
- ping 111
- rcp 112
- Reference 102
- Requirements 92, 102
- rlogin 112
- rootfs 125
- running 101
- shell 106
- showcase 129
- Storage Size Options 103
- tcl 108
- tcpdump 130
- tutorial 105
- vl\_demo 122
- xclock 127
- xdemo1 127
- xdemo2 128
- demo system configuration, BlueCat Linux OS Loader 88
- Deregistering
  - APM Client 166
  - MTD Driver 200
  - PMD driver 168
- desktop, Red Hat Linux GNOME 9
- developer subdirectory 93
- Developing
  - APM Drivers 165
  - Application Programs 38
  - BlueCat Linux Applications 21–46
  - MTD Drivers 197
- Development Directory Tree Structure 21
- Development Process 26
- device nodes 174
- Directories
  - Binary Architecture CD-ROM 4
  - Source Architecture CD-ROM 8
  - Target Support Package CD-ROM 6
- directory
  - Cross Development Tools 24
  - Demo Systems 24
  - structure, BlueCat Linux 16, 21
  - Target Board Tools and Files 24
- directory structure components, BlueCat Linux 17
- Discarding Symbols from Files 43
- Disk Operations Demos 124
- diskboot subdirectory 93
- DiskOnChip
  - booting from 57
  - Copying on 58
- Distribution
  - BlueCat Linux 1
  - CD-ROM 2
- Documentation 3
- Download an arbitrary file 79
- Download and Unpack a tar Archive 79
- Downloading
  - BlueCat Linux Kernel and FFS-Based Root filesystem using OS Loader 61
  - BlueCatLinux Kernel and FFS-Based Root filesystem using install demo system 66
  - FFS Image Built on Cross Development Host using install Demo System 66
  - Image Composed of Kernel and FileSystem using install Demo System 65
  - Kernel and FileSystem Image using OS Loader 60
  - OS Loader 72

- Downloading and Booting BlueCat Linux 47
  - Downloading and Executing Programs 88
  - Downloading BlueCat Linux
    - into Flash Memory using Flash Management Tools 197
    - into Target ROM/Flash Memory using Firmware 59
    - into target ROM/Flash memory using install Demo System 65
    - into Target ROM/Flash using OS Loader 60
  - Driver Event Code 156
- 
- E**
- embedded system
    - components 25
    - definition 25
    - Kernel 25
    - root filesystem 25
  - environment variables, show or modify 80
  - Erasing a flash memory Device or Partition 196
  - Error Codes 154
  - Event codes 155
  - event descriptor 148
  - Event Logging 164
  - event notification 156
  - Event Processing 145
  - exec command 77
  - Execute a Program 77
  - execution environment
    - setting up 18
    - setting up for Linux cross development host 18
    - unsettling 19
- 
- F**
- FFS
    - architecture 169
    - creating root filesystem in 63, 67
    - data chunks 176
    - image built on host 61
    - Interface 172
    - Internals 176
    - Layout 176
    - managing 197
  - FFS image built on cross development host 66
  - FFS IOCTL Command Reference 187
  - ffs subdirectory 93
  - FFS-based root filesystem 61
    - downloading 66
  - files, demo system 98
  - Filesystem image 60, 65
  - Finishing Kernel Debugging 36
  - Firmware 69, 71
    - using for downloading 59
    - using to create image 72
  - flash command 77
  - Flash File System (FFS) Interface 172
  - Flash memory
    - configuring partitions 194
    - Entities and Device Nodes 174
    - erasing 196
    - Management Tools and Mechanisms 194
    - partition 60, 61, 63, 65, 66, 68, 259
    - Partitioning 174
    - support 169
    - writing raw data 196
  - flash memory device or partition, erasing 170
  - Flash Memory Support and Flash File System 136
  - Flash Support and Flash File System 169–201
  - flash\_disk utility 176
  - flash\_erase 170
  - flash\_erase Command Reference 261
  - flash\_fdisk Command Reference 259–260
  - flash\_fdisk utility 60, 62, 63, 65, 66, 68, 195
  - floppy disk, copying on 49
  - ftp subdirectory 93
- 
- G**
- garbage collection 182, 183
    - criteria evaluation 183
    - single iteration 183
  - GDB 35
  - GDB Commands 37
  - gdb subdirectory 93
  - Getting Necessary Shared Libraries 43
  - gnutar subdirectory 93

---

## H

- Handler Program Command Line Format 164
- hard disk 55
  - booting from 50
  - copying BlueCat Linux on from cross development host 50
- Hardware device
  - configuring demos for 99
  - support 88
- hello subdirectory 94
- help command 78

---

## I

- i\_osloader 73
- Images Created by mkrootfs 41
- images, booting 82
- inactivity period parameter 147
- inactivity timer 147, 162, 165, 255
- install demo system 55, 59, 65, 66, 68
- install subdirectory 94
- Installation 1–20
- Installation CD-ROMs, BlueCat Linux 2
- Installing
  - default configuration 9
  - FFS Image Built on Cross Development Host using OS Loader 61
  - Optional BlueCat Linux Packages 13
  - Sources of BlueCat Linux RPM Packages 14
  - support for target boards 11
- Installing BlueCat Linux 9
- Interface
  - APM 148
  - APM reference 154
  - block device 169
  - character device 169
  - FFS 172
  - IOCTL 148, 150
  - Kernel configuration command 29
  - Kernel-space 148
  - MTD 169, 172
  - PMD device drivers 148

- to Flash Memory 169
- user mode 162

Interface reference, MTD 188

Interrupting the Kernel 36

IOCTL

- Commands 161
- interface to the user space 148
- interface to user space 150

IP auto-configuration 81

---

## J

- Java 135
- JFFS\_GARBAGE\_COLLECT 188
- JFFS\_GET\_BAD\_TABLE 187

---

## K

- kdbg subdirectory 94
- Kernel
  - boot-time parameters 195
  - building 30
    - for debug purposes 33
    - for x86 30
  - Configuration Procedure 29
  - customizing 28
  - customizing for size 42
  - debugger 34
  - Debugger Extensions 37
  - Debugging Requirements 33
  - Default configuration 29
  - Downloading using demo system 66
  - image 60, 65
  - image size 42
  - interrupting 36
  - pristine 24
  - reconfiguring 28
  - RPM packaging 23
  - subdirectory 22
  - Tree 22
- Kernel Configuration Command Interface 29
- kernel scheduler 152
- Kernel-Mode API 156
- Kernel-space client interface 148

---

**L**

ldconfig 43  
 Linux 84  
 Linux host, installing default configuration  
     on 9  
 Linux target board tools 196  
 loadkeys subdirectory 94  
 Location, Demo System 92  
 log-structure 177  
 low-level device drivers 141  
 LynuxWorks, Inc  
     technical support xv  
 LynuxWorks, Inc.  
     Web site xv

---

**M**

make config 29  
 make menuconfig 30  
 make oldconfig 30  
 make xconfig 30  
 Managing  
     FFS 197  
     Multiple Embedded Applications 42  
     Multiple Kernel Profiles 31  
 mapm subdirectory 94  
 mapm\_ctrl 142, 143, 151, 165  
     Operation 165  
 mapm\_ctrl Command Reference 255–256  
 mapmd 151  
     Command Line Format 163  
     Configuration File 163  
     Operation 164  
 mapmd Command Reference 253  
 mapmd.conf 164  
 mapmd.conf Command Reference 257–258  
 media, distribution 1  
 MEMDEFPARTTABLE 187  
 MEMERASE 186  
 MEMGETINFO 185  
 MEMGETREGIONS 186  
 Memory Sizing benchmark 44  
     /proc file 45  
 Memory Technology Device interface 169  
 memory tracking facility 46  
 MEMREADOOB 187  
 memsize subdirectory 94  
 MEMWRITEOOB 186  
 Minimal Filesystem 43  
 mkboot  
     Cross Development Tool 48  
     options 248  
 mkboot command 78  
 mkboot Command Reference 247–250  
 mkkernel 32, 42, 98  
 mkkernel Command Reference 251  
 mkrootfs 40, 42, 43, 63, 66, 98  
     building a minimal filesystem 43  
     options 244  
     specfile 242  
     Specification File 41  
 mkrootfs Command Reference 241–246  
 modular subdirectory 94  
 modules, APM 142  
 Mount a Filesystem 78  
 mount command 78  
 Mounting a Root filesystem from NFS 86  
 msgn\_exmpl subdirectory 95  
 msgn\_minet subdirectory 95  
 MTD 169  
     deregistering driver 200  
     developing a driver 197  
     driver 173, 174  
     Interface 172  
     module types 188  
     registering a driver 198  
     registration service 174  
 MTD Interface Reference 188  
 mtdblock 169, 171, 173, 175, 181  
 mtdblock Interface 171  
 mtdchar 169, 171, 173, 175, 196  
     device node 171  
     Interface 170  
     Interface Reference 185  
 multi\_user subdirectory 95  
 multi\_user\_net subdirectory 95  
 multi-node write() call 181  
 Multiple  
     embedded applications, managing 42  
     independent installations 1  
     instances, setting up 19  
     kernel profiles 31  
     kernel profiles, managing 31  
     multi-user environment 19

---

## N

- native cross development host tools 1
- network
  - protocols 73
- network, booting over 71, 72
- NFS server
  - booting from 85
  - setting up 83
- nfsroot subdirectory 95
- ntar command 79

---

## O

- opaque client handle 148
- opaque handle 154
- Optimizing Footprint 42
- Optional BlueCat Linux Packages 227–240
  - installing 13
- options
  - mkboot 248
  - mkrootfs 244
- OS Loader 25, 52, 59, 60, 61, 63, 72, 73
  - as a demo system configuration 88
  - booting BlueCat Linux 72
  - configuring 88
  - Customizing 89
  - downloading 72
  - in embedded systems 85
- osloader subdirectory 95
- Overview of the BlueCat Linux Directory Structure 21

---

## P

- packaging, RPM 23
- parallel port, booting from 71, 72
- parameters, build-time 194
- parent inode number 177
- Partition
  - configuring 176, 200
  - configuring using flash\_fdisk 195
  - configuring using kernel boot-time parameters 195

- Partitioning Flash Memory 60, 63, 65, 66, 174
- PFTP server
  - booting from 86
  - setting up 84
- pfptd Command Reference 263–264
- ping subdirectory 95
- PMD 141
  - API 159
  - callback 150, 159
  - deregistering driver 168
  - device driver 150
  - device drivers interface 148, 149
  - Driver event code 156
  - event codes 155
  - handle 154, 162
  - inactivity period parameter 147
  - inactivity timers 147
  - name or handle 257
  - Post-event code 156
  - power state 144
  - processing driver requests 168
  - registering driver 167
  - registration service 150
  - Request Code Values 160
  - Requests 160
  - Services 160
  - State Codes 155
- POSIX I/O call 181
- post-event 145
- Post-Event Code 156
- Power Loss Recovery 181
- power management, CPU 152
- Power States 146
  - switches 147
- Power-Managed Device (PMD) 141
- Pre- and Post-Events 144
- Pre-Event Code 155
- Print Help Message 78
- pristine Linux Kernel 24
- Process List of Commands in a File 80
- Processing
  - APM Events 166
  - Requests in a PMD Driver 168
- Program Image into Flash Memory 77
- Programs
  - downloading and executing 88
  - Handler 164

---

## R

- raw\_inode 176, 177, 181
- raw-access 174
- rcp subdirectory 96
- read command 79
- Reboot the System 80
- Rebuilding BLOSH 89
- reconfiguring kernel 28
- Red Hat Linux
  - GNOME desktop 9
  - host 10
  - version 6 or higher 2
- Red Hat Package Manager (RPM) 1
- Registering
  - APM Client 165
  - MTD Driver 198
  - PMD Driver 167
- registration service 150
- Request code values 160
- reset command 80
- rlogin subdirectory 96
- ROM/flash memory
  - booting from 59, 69
  - downloading into
    - using Firmware 59
  - downloading using demo system 65
- root filesystem
  - building 40
  - in FFS 63
  - mounting 86
- rootfs subdirectory 96
- RPM
  - packages, installing sources of 14
  - packaging 23
  - using 15
- Running Demo Systems 101

---

## S

- Sample APM Client 165
- Sample PMD Driver 167
- script command 80
- serial ports, setting up 34
- set command 80
- Setting Up

- a PFTP Server 84
- BlueCat Linux Execution Environment 18
- BOOTP Server 81
- Multiple Instances of BlueCat Linux 19
- NFS Server 83
- Serial Ports 34
- TFTP Server 82

---

- SETUP.sh 5
- shared libraries, getting 43
- shell subdirectory 96
- Shells 106
- Show or Modify Environment Variables 80
- showcase subdirectory 96
- Simple Demo Systems 104
- Simple Networking Demos 111
- single iteration of garbage collection 183
- Software Inactivity Timers 147
- Source Architecture CD-ROM 3, 24
  - Directories 8
  - Subdirectories 9
  - Tree structure 7
- Specfile, mkrootfs 242
- Starting Kernel Debugging 35
- static libraries, using 44
- Subdirectories
  - Binary Architecture CD-ROM 5
  - demo system 98
  - Source Architecture CD-ROM 9
  - Target Support Package CD-ROM 7
- subdirectory, kernel 22
- superuser 9, 49, 84
- support for target boards
  - activating 12
  - installing 11
- Supported Demo Systems 92
- Synchronous Operations 184
- system rebooting 80
- System requirements 2

---

## T

- target board
  - activating support 12
  - firmware 59, 71
  - installing support 11
  - Tools and Files 24
  - uninstalling support 20

- Target Board-Specific Auto-Boot of BlueCat Linux 87
  - Target Support Package CD-ROM 3
    - Directories 6
      - documents directory 7
      - Subdirectories 7
      - Tree structure 6
  - tcl subdirectory 96
  - tcpdump 96
  - Technical Support xv
  - TFTP server
    - booting from 85
    - setting up 82
  - Tk interpreter 30
  - tools, Flash memory management 194
  - Tree structure
    - Binary Architecture CD-ROM 3
    - Source Architecture CD-ROM 7
    - Target Support Package CD-ROM 6
  - TurboLinux Workstation 6.0 2
  - tutorial subdirectory 97
  - Typical Makefile Goals 100
  - Typographical Conventions xiii
- 
- U**
- Uninstalling
    - a BlueCat Linux Component 20
    - BlueCat Linux 19
    - Support for a BlueCat Linux installation 19
    - Support for a target board 20
  - UNIX commands 44
  - Unsetting the BlueCat Linux Environment 19
  - User-Mode Interfaces 162
  - User-Space Components 151
  - user-space control utility (mapm\_ctrl) 142
  - Using
    - /proc/mtd file 195
    - BlueCat Linux OS Loader in
      - Embedded Systems 85
    - BlueCat Linux RPM 15
    - Firmware to Boot BlueCat Linux over a Network 72
    - Makefile to Rebuild a Demo System 100
    - Memory Sizing Benchmark 44
    - mkrootfs to Build a Minimal Filesystem 43
    - OS Loader to Boot BlueCat Linux 72
    - Static Libraries 44
  - Utility Systems Demos 114
- 
- V**
- VFS 172, 177, 183, 184
  - Virtual File System (VFS) layer 172
  - vl\_demo subdirectory 97
- 
- W**
- wakeup event 146
  - Wear Leveling 182
  - Wear Leveling algorithm 183
  - Web site, Lynuxworks xv
  - Windows 98 Host 85
  - Windows host, installing default configuration on 10
  - Windows NT/2000 85
  - write() call 172
  - Writing Raw Data to Flash Memory 196
- 
- X**
- x86
    - building kernel for 30
  - xclock subdirectory 97
  - xdemo1 subdirectory 97
  - xdemo2 subdirectory 97