BlueCat Linux Kernel Porting Guide

Release 3.0

DOC-0370-00

Contents

Preface

For More Information

For more information about the LynxOS BlueCat product, refer to the following printed and online documentation.

- *BlueCat Release Notes*

  This printed document contains late-breaking information about the BlueCat product release.

- *BlueCat Linux User's Guide*

  This document describes fundamental concepts of device drivers under LynxOS. The following chapters provide numerous examples to help you develop LynxOS device drivers for your LynxOS applications.

- *Online information*

  Commands and utilities, provided online in text format, can be accessed by using the `man` command. For example:

  ```
  man gcc
  ```

  Other documentation listed here may also be online. In case of a discrepancy between the printed and the online version, the online version is correct.

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	Example(s)
Garamond - body text; *italicized* for emphasis, new terms, book titles, and text that would be replaced with a real name or value	Refer to the *LynxOS User's Manual*. `cat` *filename* `mv` *file1 file2*

`Courier New` (used within body text) - commands, options, filenames, pathnames, parameter names, and other computer-generated data	`ls`
	`-l`
	`myprog.c`
	`/dev/null`

`Courier New 7 pts.` - text that appears 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:`

`Courier New Bold` (used within examples) - command lines and options entered on the computer	`login:` **`myname`**
	`cd /usr/home`

VERDANA (all caps) - environment variables	DISPLAY

Font and Description	**Example(s)**
Verdana Bold - keyboard options, button names, and menu sequences	**Enter, Ctrl-C**
Verdana Italics - functions or function names	*getenv()*

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
Phone: (408) 879-3940
Fax: (408) 879-3945

LynuxWorks Europe

Internet: 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

CHAPTER 1

Introduction

Overview

The BlueCat Linux Kernel Porting Guide augments the BlueCat Linux Development System by presenting the requirements and approaches for porting BlueCat Linux to a new hardware environment. BlueCat Linux is derived from standard Linux sources and, in many cases, porting follows the same steps as would any other version of Linux. BlueCat Linux, however, extends the standard Linux implementations through ease of configuration for reduced footprints and installation in ROM. When porting to a new target environment, the primary challenges the developer will face will be in the initialization and startup code, which is often target and/or storage medium specific. This guide provides a detailed description of how BlueCat Linux boots, thus aiding the developer in their porting efforts.

Chapter 2, “x86 Porting” provides a detailed view at how BlueCat Linux is ported to new x86 environments. With the standardization of the x86 platform, ports to new environments are often accomplished without any significant effort. The x86 platform is chosen as a platform for the detailed description simply because the vast majority of the readers are familiar with this environment and they will easily relate to the initialization and boot process.

Chapter 3, “General Porting Guidelines”provides an overview of how to approach porting BlueCat Linux to non-x86 environments. It discusses the various options available to the developer, outlines the ROM loader requirements for BlueCat Linux, and explains how to work with target resident monitors.

Documentation

This guide provides a top-level description of key issues in the porting process. While reading this guide, it is advisable to have the following additional documentation available for reference:

BlueCat Linux User's Guide, LynuxWorks, Inc.

Additionally, access to a computer which has BlueCat Linux installed is necessary, as specific files are referenced throughout this guide.

The “HOWTOs” available at **`http://www.linuxdoc.org`** can often provide valuable insights into specific aspects of a BlueCat Linux port. So readers should familiarize themselves with what HOWTO documents are available.

CHAPTER 2 *x86 Porting*

Overview

This chapter concentrates on BlueCat Linux porting issues specific to x86 target environments. It covers porting and operation in traditional, PC style environments where disk drives are available, as well as custom target environments where BlueCat Linux is stored in some sort of permanent media, such as flash. The goals of this chapter are to present the following key concepts in a concise fashion:

- An overview of the BlueCat Linux target binary image, and how it is built
- A description of the BlueCat Linux boot process for both disk and flash based booting when using a standard BIOS
- Building “headless” platforms which use a serial port for a console, and operate without a graphics card and keyboard
- An overview of installation of custom drivers, and how to approach developing and debugging them
- Requirements for working without a BIOS
- An overview of critical kernel configuration options, and how they are configured prior to building a BlueCat target image

BlueCat Linux is a hosted development environment. All tools, libraries, and source files required to build a target BlueCat image reside on a host development platform. Various Linux- and Windows-based operating systems are supported (consult the B*lueCat Linux User’s Guide* for a current list of available hosts. BlueCat Linux must be installed on an appropriate host prior to reading this chapter and working with the examples contained within.

Two primary target configurations are discussed:

- PC-style target platforms in which disk drives are available
- Embedded target platforms which are diskless, and in which BlueCat Linux is stored in flash

Additionally, discussion will include how to build clean development environments for systems which contain a network interface.

The most important concepts discussed in this chapter focus on the development process: how to build BlueCat target images on the host, and then execute and debug them on the target. To enhance the reader's understanding of the development process, a detailed description of the boot process and an overview of the critical boot related files are provided.

Getting Started

The first step in getting started is through an understanding of how to build one or more of the demonstration examples provided with the BlueCat distribution. This porting guide will work with the demonstration examples shown in the following table.

Table 2-1: Demonstration Examples

Demo	Description
`hello`	Used to confirm basic functionality of the port, and potentially diagnose boot problems
`ping`	Used to verify correct operation of the network interface
`nfsroot`	Used to install BlueCat Linux to the storage medium, as well as provide a reasonable system development environment

These demonstration examples are documented in the *BlueCat Linux User's Guide* and are installed in the `$BLUECAT_PREFIX/demo` directory. The reader is encouraged to become familiar with these demonstrations, and how they are built and installed on a floppy, prior to attempting to bring up new hardware with this guide.

The process of understanding the porting process is centered around the following steps:

1. Build the demonstration program.
2. Install the demonstration BlueCat target image for target execution; for disk based systems, this is installation on a floppy disk drive, and for flash based systems, in flash.
3. Verify correct operation on the target hardware.

Through this, the reader is presented with the following key concepts:

- An understanding of how to build BlueCat Linux target images
- An understanding of how BlueCat Linux target images execute the early and critical, boot code
- An understanding of how to debug BlueCat Linux target images which fail to boot properly

Target Image Format

It is extremely important for developers who will be porting BlueCat Linux to have a solid understanding of the target image. Table 2-2: "Target Image Components" outlines the key components of which a BlueCat Linux target image is comprised.

Table 2-2: Target Image Components

Component	Description
Boot Parameters	Basic boot information, such as disk geometry for systems with a disk, whether a ramdisk is present, whether a command line is present and the root filesystem specification
Boot Block	Used only for systems which boot from a disk, this block is loaded by the target BIOS to bootstrap the boot process
Linux Setup Code	Target-independent setup with the primary purpose of transitioning the processor into protected mode and then jumping to the Linux kernel start-up code
Command Line	Optional, this is a text string which provides additional configuration information to the BlueCat Linux kernel, such as using the serial port as a console, or mounting root filesystem via NFS
BlueCat Linux	Actual Linux kernel image
Root Filesystem	Optional root filesystem for diskless systems

Figure 2-1: "BlueCat Linux Target Image Layout" provides a block diagram of the BlueCat Linux target image.

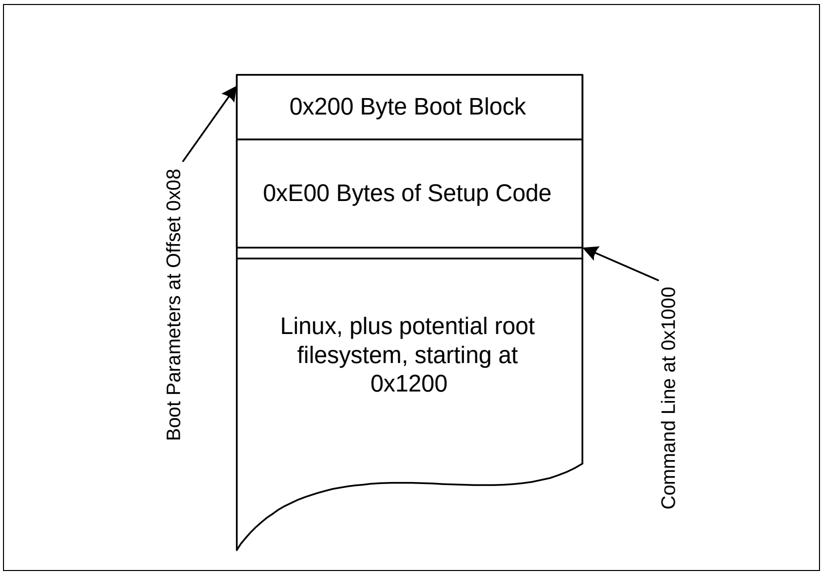
0x200 Byte Boot Block
0xE00 Bytes of Setup Code
Linux, plus potential root filesystem, starting at 0x1200
Boot Parameters at Offset 0x08
Command Line at 0x1000

Figure 2-1: BlueCat Linux Target Image Layout

The target image, whether installed on disk, or in flash, always exists in this format. The image is constructed using the host-based tools. Table 2-3: "Target Image Generation Tools" provides an overview of these tools.

Table 2-3: Target Image Generation Tools

Tool	Description
`mkboot`	Used to install BlueCat Linux on a disk, or build a image suitable for installation into flash
`mkrootfs`	Used to build root filesystems which are stored on disk or in flash, and loaded to RAM for runtime access
`mkkernel`	Used to cause a "Make" down in the Linux Kernel source directory

The *BlueCat Linux User's Guide* provides a detailed description of these tools, and gives numerous examples of how they are used. The reader is encouraged to have read the user's guide prior to working with this porting guide. Additional examples of how to use these tools is contained within this chapter.

Console I/O

In the x86 environment, two options exist for a boot console:

- A standard PC type configuration, with a graphics controller and keyboard
- A "headless" configuration using the serial port; in this configuration, seldom does a graphics controller or keyboard exist

Console I/O through the serial port is native to BlueCat Linux, provided that the serial drivers are present and the kernel is configured to allow console I/O through the serial port. The determination of how BlueCat Linux configures a console at boot time is done through the boot command line.

A boot command line which instructs BlueCat Linux to use a serial port for a console is defined as follows:

```
console=ttyS0,9600n8
```

In this case, BlueCat Linux is configured to use the first Serial Port, with line settings of 9600 baud, no parity and eight-bit characters. To install this command line in the target image, it must be saved in a file, and then installed as part of target image generation using the `mkboot` command.

For disk-based systems, it is installed during the installation of the other BlueCat Linux target image components on the target disk. Installation is typically done as follows:

```
echo “console=ttyS0,9600n8” > cl.txt
mkboot -c cl.txt device
rm cl.txt
```

Installation of a command line for target images being installed in flash is slightly different than disk-based images, as the target image is built in one step (disk images can be built in individual steps, as the disk is a random access device, while flash images are built as one file which is then installed in flash using developer specific tools). Construction of an image containing a command line that is suitable for installation in flash is done as follows:

```
echo “console=ttyS0,9600n8” > cl.txt
mkboot -m -k target.disk -c cl.txt target.kdi
rm cl.txt
```

In this example, `target.disk` is a file which contains the BlueCat Linux setup code and kernel image, and was built earlier in the target make process using `mkkernel`. The output file *target.kdi* is the resultant BlueCat Linux image which is suitable for installation in flash.

Refer to the `mkboot` command reference for further information on this operation. Also note that the demonstrations shipped with BlueCat Linux are built to operate in a system which contains a graphics card and keyboard. If the developers environment is “headless,” changes are required in generation of the final bootable image.

For flash, it is required that the Makefiles be updated in these directories to correctly build an image which uses a Serial Port for Console I/O. For disks, an additional step must be taken with `mkboot` to install the command line.

Demonstration Programs

The demonstration programs are the first step in validating the developer’s target hardware. They were selected for use in this porting guide as they require differing requirements of correct target hardware operation, thus aiding the developer in debugging any potential problems with their target.

Table 2-4: "Demonstration Program Hardware Requirements"outlines what hardware needs to be functional for each of the demonstration programs.

Table 2-4: Demonstration Program Hardware Requirements

Demonstration	Hardware Requirements
`hello`	Correct operation of the console device, memory, storage medium and BlueCat system generation process
`ping`	Verifies that the correct network interface device driver is installed in the kernel, and that the network interface is operational (only for systems with a network)
`nfsroot`	Provides a means of booting a BlueCat system with a root filesystem on the host, and potentially gives the developer a stable development environment for further testing and configuration—both `hello` and `ping` must operate correctly for this to work (only for systems with a network)

NOTE: *Each of the demonstrations shown in the Demonstration Program Hardware Requirements table will execute perfectly on a system capable of hosting either Red Hat or Turbolinux. Developers are strongly encouraged to go through the process of building each demo, installing it on a floppy disk, and booting it on a standard PC platform capable of properly running RedHat or Turbolinux. In so doing, the developer validates the development process, ensuring that images for target hardware are being built correctly.*

Initial Target System Validation

The first step in target environment and target BlueCat Linux port validation is ensuring that target images properly execute the initial boot operation. The most important demonstration example is `hello`.

With systems booting from a disk drive (typically a floppy), this example validates that the BlueCat image is being properly installed on the disk, that the BIOS is able to load the Boot Block from the disk, and that control is being correctly passed to the BlueCat Linux target image. For flash-based systems, this example validates that the BlueCat image is being properly installed in flash, that the target systems flash loader is operating correctly, and that control is being correctly passed to the BlueCat Linux target image that was loaded out of flash and into RAM.

Failure of this example can be traced to a multitude of problems, from incorrect building of the BlueCat Linux target image, incorrect installation on the disk or into flash, or failure of the BIOS or flash loader. Most important is to understand that if the `hello` example does not work, it is extremely unlikely that any of the other demonstrations will boot and operate correctly. Diagnosis of the exact problem requires an in-depth understanding of how the BlueCat Linux boot process operates. This is discussed in detail in the next section of this chapter.

If the developer installs this image on a floppy disk, and is unsuccessful in booting it on a standard PC platform capable of running Red Hat or Turbolinux, then the most likely cause of the problem is incorrect floppy disk installation. This is why it is extremely important to validate the build process on known working hardware before moving to new and untested target environments—*validate the build process and your understanding of how the demonstration programs operate.*

Full Target System Validation

The demonstration images help the developer validate the basic operation of the target system. Systems which contain proprietary hardware or specialized peripherals will require validation of both the hardware and the associated device driver. This is often an iterative process based on a process of “test, rebuild, and test again.” The best way to approach this is to configure a solid target test runtime environment that allows easy testing of new BlueCat kernels and associated device drivers.

For systems with a standard Ethernet interface, the demonstration program `nfsroot` can be built upon to form an extremely powerful and easy-to-use test environment. The foundation for a clean test environment is that a standardized boot environment can be installed, such as the `nfsroot` demo in either flash or on a boot disk. Updated kernels or application code are simply loaded and installed over the network (from the NFS root directory on the host). For flash-based systems, this can be a significant advantage, as it takes the need to reprogram flash with each test out of the equation.

NOTE: *Developers whose final product does not contain an Ethernet interface are highly encouraged to install one for use during application software and kernel development. For standard PC platforms, this is often simple installation of a supported NIC card. For proprietary target platforms, one approach is to allow a standard network interface device to be populated on the target hardware during application software development, but not installed during production.*

Additionally, an alternative to using the `nfsroot` demonstration as a starting point for a development baseline, developers may install the BlueCat loader. The BlueCat loader allows target kernels to be booted over the network, which aids in development and debugging by avoiding the need to reprogram flash media or reinstall on a disk with each sysgen cycle.

Regardless of the development environment, ensuring that the `hello` demonstration is stable is a solid starting point in incrementally building BlueCat Linux target images for testing. Each of the demonstration systems shipped with BlueCat Linux provides additional approaches to target hardware validation. Developers are encouraged to start with these basic environments and build on top of them, adding their specific required functionality.

Understanding the Boot Process

Several key actions are taken during the BlueCat Linux boot process:

- The BlueCat Linux is loaded from the storage media, whether it be a disk drive or flash, into RAM.
- Key kernel boot parameters are placed in known locations in RAM.
- The processor is placed in protected mode and control is passed to the BlueCat Linux kernel primary start-up code.

In diagnosing any potential problems during these first three key steps of BlueCat Linux boot, it is essential that the developer have a detailed understanding of the following three areas:

- Key files associated with system boot, and their functionality in the system
- Layout of low memory, and how it is used during the boot process
- Overall flow of the boot process

Key Boot Files

The following table lists each key boot support file, and provides a brief description of its operation.

Table 2-5: Key Boot Support Files

File	Description
`bcboot.S`	Used only for disk based images. Contains the assembly language instructions used for construction of the boot sector installed by the utility `mkboot`.
`bootsect.S`	Contains the standard boot sector linked with traditional Linux images. This is part of the standard Linux distribution; used during the BlueCat Linux build process in both disk and flash-based images. For disk-based images, replaced by the code in `bcboot.S` by the `mkboot` utility. Remains part of the system simply to allow the standard Linux makefiles to be used in the kernel directories without significant changes in structure.
`setup.S`	Contains the next step in booting BlueCat Linux after the boot block image executes. Used both in disk- and flash-based images.
`video.S`	Used during startup to determine what type display adapter is installed, and then doing configuration of the system for it. Used both in disk- and flash-based images.
`rom.S`	Flash loader used to boot BlueCat Linux out of flash and into RAM. Used only with flash-based images.

BlueCat Linux boot code is located under the following directory:

```
$BLUECAT_PREFIX/usr/src/linux-2.2.12/arch/i386/boot
```

All x86 specific boot files are located either in this directory, or in directories contained within this directory.

Low-Memory Usage during Boot

The next table describes the key memory locations used in low memory during the boot process.

Table 2-6: Key Locations in Low Memory

Component	Address	Description
Boot Block	`0x07C0:0x0000`	Used only in disk-based systems. The BIOS loads the first 512 bytes of the target image into this location and passes control to it.
Scratchpad	`0x1000:0x0000`	Used by the code in the boot block or the flash loader to store temporary 64k blocks of kernel code for transfer to high memory.
Flash Loader	`0x8000:0x0000`	Used only in flash-based systems. The flash loader copies itself out of ROM into this location when the BIOS calls its extension block entry point during system boot.
Setup	`0x9000:0x0200`	The code contained in the file `setup.S` is loaded to this location by code in the boot block or the flash loader.
Kernel Parameters	`0x9000:0x01F1`	The Linux kernel expects various parameters, such as the root filesystem device, to be located starting at this offset. Placed here by the code in `setup.S`.
Setup	`0x9000:0x0200`	The code contained in the file `setup.S` is loaded to this location by code in the boot block or the flash loader.
Command Line	`0x9000:0x1000`	The command line installed by the `mkboot` utility. If a command line is supplied, the kernel expects it to be in this location.

Figure 2-2: "Low-Memory Layout" shows a block diagram of the key blocks of low memory.

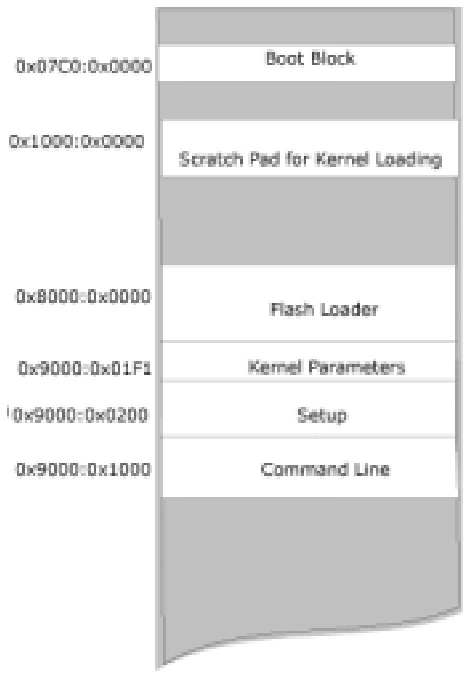

Figure 2-2: Low-Memory Layout

Booting from Disk

When booting from disk, the BIOS loads the first 512 bytes from the boot disk into memory, performs some minor validation, and then passes control to the code located at the start of this 512-byte block, which is called the boot block.

There are three key addresses contained in the boot block, shown in the Table 2-7: "Boot Block Key Addresses" After it is loaded into memory, the

BIOS validates the boot block signature, and if it is valid, passes control to offset 0x00 in the block.

Table 2-7: Boot Block Key Addresses

Offset	Description
`0x000`	Entry Point - After loading the Boot Block into memory, the BIOS passes control to this offset.
`0x1BE`	Partition Table - Not used by BlueCat Linux or the BIOS
`0x1FE`	Boot Block Signature - Must contain 0xAA55. The BIOS uses this to validate that the Boot Block contains a valid image.

Referring back toTable 2-6: "Key Locations in Low Memory" on page 13 and Figure 2-2: "Low-Memory Layout" on page 14, which describes low memory usage during boot, it is seen that the boot block is loaded at offset `0x07C0:0x0000` in memory.

BIOS Boot Actions

Standard x86 PC Platform BIOS all boot using the same process, performing the following key steps and functions:

1. Initialization of the System Hardware - The BIOS is responsible for correctly configuring all of the systems hardware devices to a basic functionality state. For example, memory is configured and tested. Additionally, the standard x86 peripherals used at boot, such as the keyboard, display, floppy, and IDE peripherals are brought to an operational state. For systems with a PCI bus, these PCI interfaces are initialized to an operational state.
2. Searching for BIOS extensions, and calling (executing) any of these extensions to do any board specific initialization. For example, a SCSI Interface may have some BIOS extensions which allow standard BIOS calls to take advantage of the SCSI interface for Disk I/O.
3. Passing control to the BIOS resident boot loader by trapping through `int 19`.

4. The BIOS boot loader loads the first 512 bytes from the media (the boot block) into memory at `0x07C0:0x0000`, and if the signature offset `0x1FE` is valid, passes control to the code located at offset `0x0000`. At this point, the BIOS has completed its boot functionality (but still is called by the BlueCat Linux start-up code to further the boot process).

NOTE: *The BIOS boot process is an extremely important concept and step in booting a standard x86 PC type-platform. The BIOS does not care if it is loading DOS, Windows, Linux, or any other operating system. It simply attempts to load the first block off of the media to physical location* `0x07C0:0000`*, validate its signature, and then pass control to the code that is located at offset* `0x0000` *in this block.*

Boot Block Boot Actions

Once control has been passed from the BIOS to the code in the boot block, this code furthers the boot process by loading the logic from `setup.S` into memory, as well as loading the kernel itself into memory. The following steps are taken:

1. A stack pointer is loaded at `0x07C0:0x1000`.
2. The message `BlueCat Boot` is sent to the console supported by the BIOS.
3. The code contained in `setup.S` is loaded as a 4k block at location `0x0920:0x0000.`
4. The kernel is loaded in 64k blocks to `0x1000:0x0000` and transferred to high memory at `0x0010:0x0000` using BIOS calls.
5. Parameters to the kernel are taken from the parameters stored at the start of the boot block and saved at the offsets starting at `0x9000:0x01F1`.
6. A jump instruction is executed to offset `0x00` in the setup block.

Setup Boot Actions

Once control has been passed to the code in `setup.S`, the following occurs:

1. Performs some PC platform-specific configuration updates, such as evaluating the state of the disk system, configuring the keyboard, calling logic contained in `video.S` to determine the video capability of the system and reprogramming the 8259 interrupt controllers
2. Places the processor in protected mode and jumps to the BlueCat Linux Kernel entry point.

Diagnosing Boot Problems

Figure 2-3: "Primary Steps in Booting from Disk" provides a flow chart of boot operation with a disk. Each key segment of the boot process can cause problems. When evaluating potential problems with boot, the best approach is to try to determine how far into these key steps booting occurs.

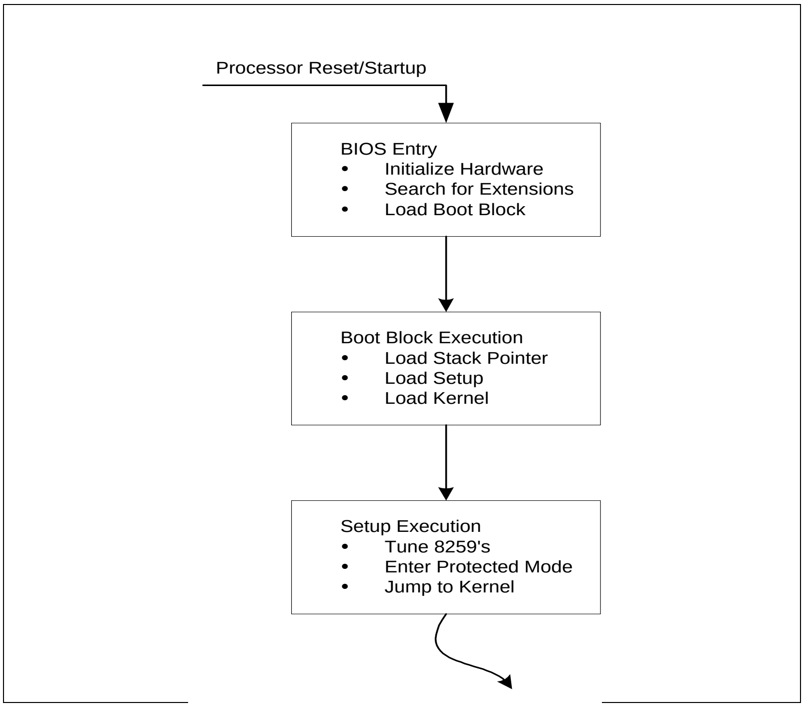

Figure 2-3: Primary Steps in Booting from Disk

The first critical operation is getting a correct boot block into memory and having control passed to it. Figure 2-4: “Boot Block Details” provides a more detailed view of what a correct boot block looks like in the BlueCat Linux environment.

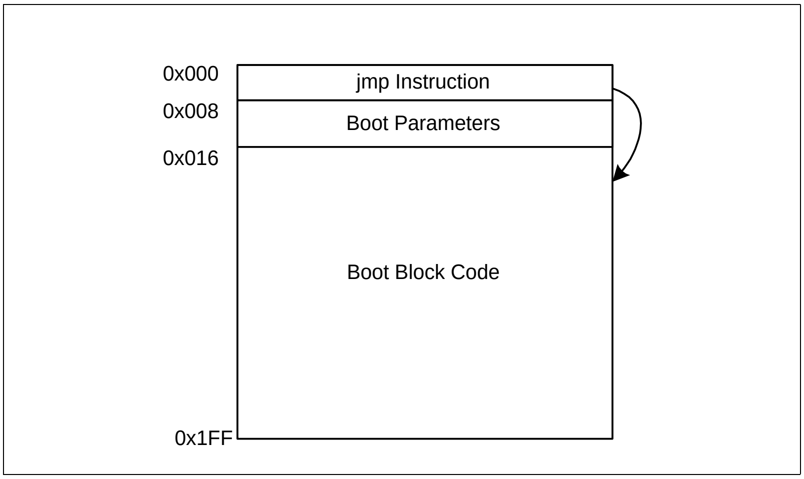

Figure 2-4: Boot Block Details

The boot parameters that are contained starting at offset `0x08` are installed by the `mkboot` utility when building the bootable BlueCat Linux target image. Table 2-8: “Boot Parameters in Boot Block” provides a description of each of the boot parameters contained in the Boot Block.

Table 2-8: Boot Parameters in Boot Block

Name	Offset	Description
`boot_magic`	`0x08`	Validates that at least one or more boot parameters has been installed
`boot_device`	`0x0A`	BIOS ID of disk
`boot_sectors`	`0x0B`	Number of sectors on the disk
`boot_heads`	`0x0C`	Number of heads on the disk
`boot_cl`	`0x0D`	A flag to indicate that a boot command line is installed at offset `0x1200` of the disk image
`boot_size`	`0x0E`	Number of sectors holding the kernel image
`boot_rdonly`	`0x10`	Flag to indicate that the kernel root filesystem should be mounted read only
`boot_ramdisk`	`0x12`	Flags used when the target image contains a ramdisk
`boot_root`	`0x14`	Initial root filesystem device

When building a disk image, the `mkboot` utility will install and update these parameters based on any invoked operation. For example, if a command line is installed with `mkboot`, the command line is placed at offset `0x1200` in the file, and the parameter `boot_cl` is updated to reflect that a command line has been installed.

The boot block itself is placed on the disk through the following command:

```
mkboot -b /dev/fd0
```

This example installs the default boot block on a floppy disk. The boot block is installed from the following file:

```
$(BLUECAT_PREFIX)/boot/bcboot.img
```

Notable about this is that when the BlueCat Linux kernel is built, the following file is linked into the beginning of the Linux kernel:

```
$(BLUECAT_PREFIX)/usr/src/linux/arch/i386/boot/\
bootsect.S
```

The `mkboot` utility replaces this boot block with the BlueCat Boot Block when building executable images.

If the developer is encountering problems in diagnosing boot problems, he or she may desire to install his or her own modified version of `bcboot.img` in the boot sector. This is possible, but needs to be accomplished using native Linux tools. It should be noted that the file

```
$(BLUECAT_PREFIX)/usr/src/linux/arch/i386/boot/\
bcboot.S
```

is assembled and linked into

```
$(BLUECAT_PREFIX)/usr/src/linux/arch/i386/boot/\
bcboot.img
```

with each make. In this, `bcboot.S` may be modified before a `make`, and then installed by hand prior to testing the new image. The following steps must be taken to install a custom boot block:

1. Modify the `bcboot.S` source file, and install your test code.
2. Perform a `make` of the BlueCat Linux image as you normally would.
3. Install the BlueCat Linux system on the media following the steps you normally would take.

Once these steps have been followed, the modified version of `bcboot.img` may be installed on the media. When doing this, it is extremely important to remember the following:

NOTE: *The* `mkboot` *utility installed the boot parameters with a length of 14 bytes starting at offset 8 on the media. These boot parameters must remain intact when installing a new boot block.*

To install the new boot block without corrupting the boot parameters, the linux command `dd` must be used to copy the block onto the media, while skipping the first 22 bytes. This is done as follows:

```
dd if=bcboot.img of=/dev/fd0 bs=1 skip=22 seek=22
```

Refer to the manpage on `dd` for detailed information. The summary of this command is that the file `bcboot.img` was installed on the floppy disk, skipping the first 22 bytes (which leaves the boot parameters installed by `mkboot` intact).

Booting from Flash

Booting from flash differs from booting from floppy, mainly through the installation of a custom boot loader, used by the BIOS, to load BlueCat Linux from flash to RAM. This custom BIOS is an extension to the standard PC BIOS, and understands the target hardware configuration, memory map and flash operation.

This custom loader becomes an extension to the native target BIOS. This is accomplished by the native BIOS during start-up, when it scans the memory region between `0xE0000` and `0xFFFF0` on 16-byte boundaries. This scan operation is used to determine if there are any BIOS extensions. An example of how this operates would be installation of a standard, off-the-shelf SCSI card. Cards such as this contain card-specific BIOS support code that is installed for normal operation by the native BIOS.

The native BIOS determines if there is an extension by looking for a certain signature. For standard BIOS extensions, this is a word containing `0xAA55` at the start of the extension, followed by a byte that contains the size of the extension in 1k blocks. The last byte of the BIOS extension contains a checksum value. Figure 2-5: "BIOS Extension Block" is a block diagram of a BIOS extension of this nature.

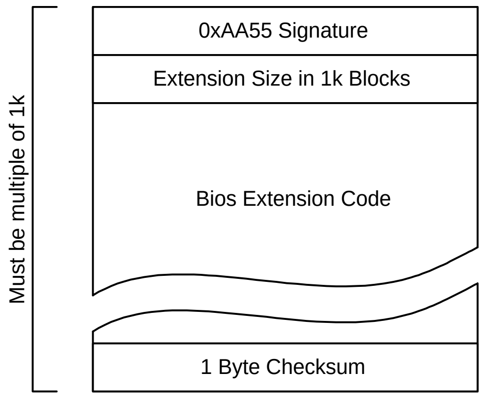

Figure 2-5: BIOS Extension Block

If the BIOS, when scanning the region between `0xE0000` and `0xFFFF0` on 16 byte boundaries, finds a valid extension, a far call is made to offset `0x03` within it. The extension then takes action specific to the function it supports.

Typically, this is installation of itself to replace a native BIOS support function such as disk I/O.

In the case of BlueCat Linux and booting from flash, this extension serves as a loader capable of loading BlueCat Linux from the target flash. It replaces the standard BIOS loader (which has an entry point at `int 19`).

BIOS extensions to load BlueCat Linux are very target specific, as each target will have flash devices accessed through target-specific needs. An example of a BlueCat Linux boot loader BIOS extension is located in the following file:

```
$(BLUECAT_PREFIX)/usr/src/linux/arch/i386/boot/\
romboot/rom.S
```

This extension follows the rules outlined above. When invoked, it replaces the native BIOS `int 19` loader with a loader which understands the target flash environment. When the native BIOS reaches the point during normal system startup where it invokes the boot loader through the `int 19` call, this BlueCat specific loader is invoked.

The typical operation of the BlueCat flash loader at this point is to copy the *entire* BlueCat Linux image out of the target flash, load it to high memory and pass control to it. This differs from the native BIOS floppy loader, which only loads the first 512 bytes of the image before passing control to it.

The reason for this difference is based on the functionality of the normal boot block code, which loads setup and the kernel from the disk it originated (making BIOS calls). The user-supplied flash loader must load the entire BlueCat Linux image, set the kernel parameters correctly, and pass control to the entry point of the system that would normally be entered after the image was fully loaded from flash.

BIOS Flash Boot Actions

Standard x86 PC platform BIOS all boot using the same process, performing the following key steps and functions:

1. Initialization of the System Hardware - The BIOS is responsible for correctly configuring all of the systems hardware devices to a basic functionality state. For example, memory is configured and tested. Additionally, the standard x86 peripherals used at boot, such as the keyboard, display,

floppy, and IDE peripherals are brought to an operational state. For systems with a PCI bus, these PCI interfaces are initialized to an operational state.

2. Searching for BIOS extensions and calling (executing) any of these extensions to do any board-specific initialization. For example, a SCSI interface may have some BIOS extensions which allow standard BIOS calls to take advantage of the SCSI interface for disk I/O. In the case of BlueCat Linux, the developer will have installed a custom loader as a BIOS extension; the BIOS finds this during startup and its installation entry is called.
3. The extension copies itself out of its flash location into RAM location `0x8000:0x0000`, installs itself as the default boot loader invoked through `int 19`. The normal BIOS loader entry point is saved at `int 18`. The extension then returns to the BIOS for it to continue normal start-up.
4. When the BIOS has finished any other start-up sequences, it passes control to the new flash loader by trapping to it through `int 19`.

Flash Loader Operation

Once the flash loader is invoked, it takes the following steps:

1. A stack pointer is installed at `0x8000:0x8000.`
2. The message `BlueCat Linux ROM-Boot:` is printed through the BIOS console.
3. Loads nine logical “sectors” from flash into RAM at location `0x9000:0x0000`, which effectively moves the code from `setup.S` into RAM, as well as the command line
4. The kernel is loaded in 64k blocks to `0x1000:0x0000`, and transferred to high memory at `0x0010:0x0000` using BIOS calls.
5. If a root filesystem is contained in flash, it is loaded in the same fashion as the kernel to high memory; small blocks transfer to low memory, with BIOS calls to move the blocks to high memory.

6. Parameters to the kernel are taken from the parameters stored at the start of the boot block and saved at the offsets starting at `0x9000:0x01F1`.
7. A jump instruction is executed to offset 0x00 in the setup block.

Setup Boot Actions

Once control has been passed to the code in `setup.S`, the following occurs:

1. Performs some PC platform-specific configuration updates, such as evaluating the state of the disk system, configuring the keyboard, calling logic contained in `video.S` to determine the video capability of the system and reprogramming the 8259 interrupt controllers.
2. Places the processor in protected mode and jumps to the BlueCat Linux Kernel entry point.

Diagnosing Boot Problems

Figure 2-6: "Primary Steps in Booting from Flash" is a flow chart of boot operation from flash. Each key segment of the boot process can cause problems. When evaluating potential problems with boot, the best approach is to try to determine how far into these key steps booting occurs.

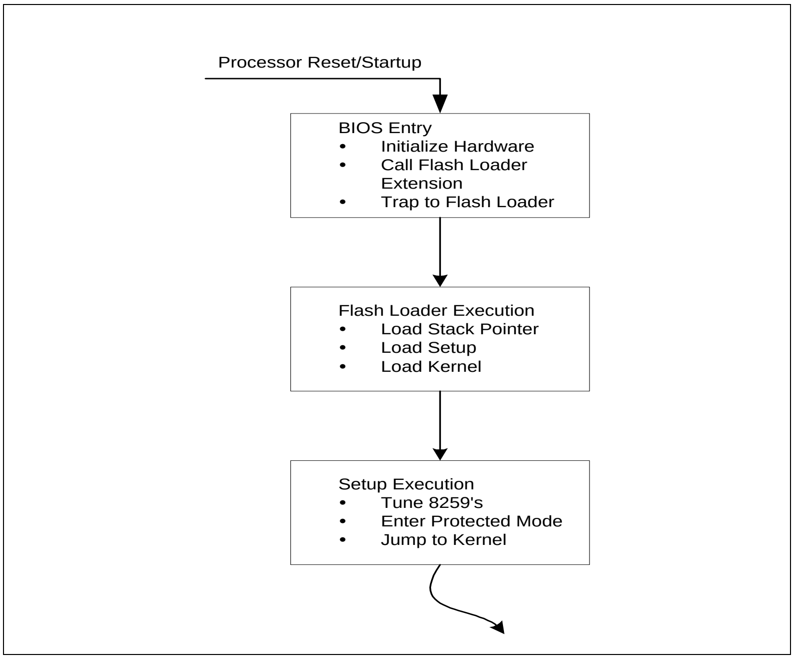

Figure 2-6: Primary Steps in Booting from Flash

When diagnosing problems with booting from flash, the following key boot steps must be verified:

1. That the BIOS correctly found the BIOS Extension, stored between `0xE0000` and `0xFFFF0`, and called the extension
2. That the BIOS Extension correctly installed the new boot loader in `int 19`
3. That the new boot loader was invoked through `int 19` by the BIOS

These key steps validate that a BIOS start-up and load procedure is successfully being accomplished, and that any further problems lay in the boot loader or the BlueCat Linux image itself. Without the ability to load the Linux image (or a diagnostic image) from flash into RAM, no further development can occur.

When trying to evaluate problems with the loader, keep a few things in mind:

- The loader determines how many bytes to load by evaluating the fields in the boot parameters, contained at offset 8 of the flash image.
- The loader does not care what sort of binary it is loading; it loads based on number of blocks specified in the boot parameters, and then passes control to what normally is code from `setup.S`.
- Developers may install low-level diagnostics in place of the `setup.S` code, code which can report that it is in fact executing after the load, and then diagnose the state of the rest of the load.

Custom Driver Overview

This section provides an outline of custom driver requirements which may be encountered during a port. Since it is beyond the scope of this document to fully describe and discuss driver construction or operation, this discussion is limited to an overview designed to guide the developer in the correct direction.

Excellent reference on drivers can be found in *Linux Device Drivers* by Alessandro Rubini (O'Reilly). Additional information may be found through the Linux "HOWTO" documents which are located at the following web site:

`http://www.linuxdoc.org`

General Notes on Drivers

Drivers are installed in the kernel through two approaches:

- Linkage with the kernel during sysgen
- Dynamically loaded as a module

Kernel resident drivers are linked during sysgen, and cannot be removed without rebuilding the kernel. Drivers installed as modules may be installed and un-installed at any time during normal system operation.

Kernel Resident Drivers

Drivers linked with the kernel reside in a subdirectory under the following directory:

```
$BLUECAT_PREFIX/usr/src/linux-2.2.12/drivers/*
```

There are several subdirectories, each specific to the type of driver. For example, the directory `net` contains network interface drivers, while the directory `pci` contains PCI-specific drivers. Each subdirectory contains a `Makefile`, as does the `drivers` directory.

Each driver installed down in a `drivers` subdirectory must have, at a minimum, an entry in the associated `Makefile`. Since drivers may be enabled or disabled through the configuration tools, configuration files must also have entries for these drivers.

NOTE: *Developers installing their own custom drivers down in the* `drivers` *directory as part of a port, have the responsibility for learning and understanding how their drivers should be installed. Failure to take these steps could result in BlueCat Linux operation failure.*

Dynamically Loaded Modules

Dynamically loaded modules are drivers which may be loaded and unloaded at any time during normal Linux operation. When porting BlueCat linux to x86 hardware platforms which have proprietary or specialized hardware extensions, the developer will be responsible for implementing and verifying operation of any custom drivers. Working with loadable modules is often the best approach for the following reasons:

- A stable BlueCat Linux kernel can be booted prior to new driver testing.
- New drivers can be removed from the system at runtime if they do not behave as desired.
- Installation of new modules does not require that the kernel be rebuilt and reinstalled.

In certain cases, such as with a network interface driver, it may be required that the driver be fully developed, tested, and installed as part of the kernel before a full installation of a port can be completed. An example of this would be a system with a custom network interface that also is capable of a full BlueCat Linux installation over the network.

One possible approach to performing this would be in taking one of the demonstration programs and enhancing it to support the development efforts. For example, the `ping` demonstration is an extremely scaled down version of the BlueCat environment designed to demonstrate network functionality.

If the developer porting BlueCat Linux must develop a new, custom network interface driver, the `ping` demonstration would be an excellent starting point for a low-level development environment. Basic tools could be installed in the root filesystem as part of the build process, as could the module under development. The new driver could be loaded as a module during these development phases, yet installed as a resident kernel-linked driver during full installation.

Working without a BIOS

Working without a BIOS presents many challenges to the developer porting BlueCat Linux. The biggest challenge is getting the developer's hardware initialized and running, as this is the primary function of the BIOS. This is a very specialized and environment-specific requirement.

Developers who work without a BIOS are generally those with a proprietary hardware environment. They are faced with the task of understanding all of their low-level interfaces, and properly initializing and testing them. All of this must be done prior to installation of BlueCat Linux.

When developing low-level drivers to support proprietary platforms, the ideal approach is to provide a BIOS-like interface for Linux. The most important aspect in this is having PC-compliant peripherals, and then the native Linux environment does not need to be modified or enhanced.

For example, the x86 version of Linux understands standard PC floppy disk controllers. If the user were to install a non-PC-compliant floppy disk controller, he or she would be required to update the floppy disk drivers resident in the `drivers/block` directory. This would be an extensive and problem-ridden project.

In addition to ensuring that PC-compliant peripherals are present on the target hardware, having the low-level video interface calls through `int 10` would ensure that files down in the `boot` directory would not require modification.

NOTE: *If the developer starts with hardware that is not PC-compliant, he or she will be required to modify Linux-specific driver and kernel files. This can be an extremely difficult process. LynuxWorks can provide professional services to aid clients in this process.*

The next chapter contains an overview which aids in porting BlueCat Linux to environments which do not contain a BIOS.

CHAPTER 3

General Porting Guidelines

Overview

Chapter 2, “x86 Porting” provided a detailed explanation of porting in the x86 environment. This chapter provides an introduction to concepts for porting to other processors, such PPC or ARM. The developer must have previously read Chapter 2, “x86 Porting” to better understand the information provided in this chapter.

The goals of this chapter are to introduce the developers to the following:

- The start-up sequences of a target processor, and BlueCat Linux
- The ROM-type storage requirements when working with BlueCat Linux
- The support code they will be responsible for developing prior to attempting a port of BlueCat Linux to a target.

With the diverse configuration and design options used by developers when working with proprietary (and even commercial) target hardware, it is difficult to provide specific details on how to support these differing configurations in one chapter. The primary focus of this chapter is to ensure that the reader is aware of all of the fundamental approaches to working with various targets, and the foundations required to install BlueCat Linux on these targets.

Target Processor Initialization

In porting BlueCat Linux to a proprietary target environments, the most important fundamental concept for developers to understand is that they, as developers, are completely responsible for board initialization. BlueCat Linux, like all other strains of Linux, has limited ties to the underlying hardware environment, and thus, is not capable of any low-level target initialization.

Proprietary Hardware Environments

Developers working with proprietary hardware are required to develop their own initialization code. Code of this nature often pertains to the following key items:

- Memory Controller Initialization - Configuration of any hardware devices which control the specific memory and memory bus subsystems.
- Diagnostic Console Initialization - Typically, a serial port used to output diagnostic information during the target processor boot sequence. Optionally, support may be provided to interrupt the boot sequence and drop to a diagnostic monitor of some fashion.
- Bus Controller Initialization - Configuration of control hardware for target-specific buses. For example, if the target contains a PCI bus, it may require low-level configuration before the BlueCat Linux PCI interfaces may properly communicate with PCI peripherals.
- Exception Handling Initialization - Processors handle exceptions, such as memory faults (access to invalid memory) in different ways. Linux, once running, properly handles target processor Exceptions. However, during boot, the initialization code must handle exceptions such as memory faults.

This code resides in some sort of ROM, such as flash, and executes immediately upon the processor leaving the reset state (such as after a reset, or during power-on). The architecture of this code rarely has to include considerations for BlueCat Linux or any other operating system, as it never makes “call outs” to operating systems which are loaded later in the boot sequence. Often, this code contains diagnostic logic, such as memory tests or manufacturing tests.

BlueCat Linux, like other strains of Linux, is capable of controlling the target hardware after it has been loaded by the initialization code and passed control to. Some examples:

- A network interface that resides on the PCI bus. The native BlueCat Linux PCI driver scans the PCI bus during BlueCat Linux startup, and if it locates a network interface that has an associated network driver installed in the kernel or available as a module, BlueCat Linux is able to bring that interface up.
- An IDE disk controller that resides on the PCI bus. Like the case of the network interface, the BlueCat Linux driver scans this during BlueCat Linux startup, and if it finds a device supported by a BlueCat Linux driver, BlueCat Linux is able to access the IDE disks.

In both these cases, *the PCI bus was operational before BlueCat Linux was given control*. The developer is responsible for this low-level initialization.

A good approach, when trying to understand these concepts, is to think about the function of the BIOS on a standard x86 PC style platform. The BIOS is given control as the processor comes out of reset, and configures all of the underlying hardware before attempting to load an operating system (whether this operating system is Linux, Windows or another popular operating system). The loaded operating system is given control of a target processor which has all basic underlying functions operational. In addition, the BIOS provides a set of system services (through traps) to support any basic I/O requirements.

NOTE: *It is extremely important for developers installing BlueCat Linux on proprietary hardware to understand that they are responsible for developing their own BIOS level initialization modules prior to attempting to load and run BlueCat Linux.*

LynuxWorks provides professional services for developers who need help in developing a board level support package such as this.

Commercial Hardware Environments

Commercial hardware environments, such as Motorola boards, are often shipped with a ROM resident debugger and program loader. This “monitor” is essentially a BIOS: it executes immediately upon the processor leaving the reset state, initializes the target processor hardware to a operational state, and then either loads an operating system out of flash or over a network or halts at a console prompt.

In an environment such as this, BlueCat Linux can almost always be executed on top of this resident monitor, as the monitor takes care of the low-level target configuration. Additionally, if this monitor is capable of loading BlueCat Linux in some fashion, such as out of flash or over the network, then the developer will often be able to simply use the monitor for the target initialization.

However, in certain cases, monitor programs shipped with commercial hardware may not prove robust enough for production deployment. For example, in high availability applications, the start-up code may be required to handle exceptions reliably. Most monitors shipped with commercial hardware simply report exception conditions on the console during boot and halt the boot process.

Loading Options during Boot

There are several options for loading BlueCat Linux during the boot process. The exact means of loading is dependent upon the capabilities of the underlying hardware, as well as the needs of the developer. For example, the underlying hardware may support flash for booting, yet during application development, the developer may wish to boot BlueCat Linux over a network interface.

The BlueCat Loader (BLOSH)

LynuxWorks ships BlueCat Linux with a loader based on a scaled down Linux kernel. This loader may be installed in flash or boot off of a disk drive. It provides the developer with the following capabilities:

- Boots full BlueCat Linux and application code from disk, out of flash or over a network

- Executes script files during the boot sequence to control system startup
- In network-based environments, does auto IP configuration using `bootp`

Commercial Monitor Loading

The functionality of monitors shipped with commercial hardware will vary from manufacturer to manufacturer. However, they often have some similar characteristics, such as:

- The ability to load some sort of image into RAM, from either local flash, a disk or over a network. Most often, these images are raw binaries or S-record based.
- They have no knowledge of the type image they are loading, and often simply load the image to a predefined location which the developer cannot control.

An important concept to keep in mind when working with commercial monitors is that they probably will not load BlueCat Linux to the location at which it is built to run via the Make process. In this instance, the start-up code that executes first and that the BlueCat Linux image has passed control to may have to first move the image to the correct location (using position-independent code).

User-Developed Loaders

Developers working with proprietary hardware will most often be developing the target initialization code. When doing this, it may be desirable to create a micro-monitor type of environment to aid in porting BlueCat Linux to the target.

For example, the BlueCat Linux Loader (BLOSH) may be suitable for the developer to work with once the target environment is stable. However, BLOSH is built with a scaled down BlueCat Linux kernel, so in order for it to be installed on a target, BlueCat Linux needs to be ported and operational on the target. It may be desirable for developers to build a micro-monitor which allows loading the BlueCat image over the network or off of a floppy disk. This approach provides a stepping stone to getting a full load and execution environment going.

General Notes on Loaders

Loaders often have to fulfill two needs:

- Loading of kernels and applications under development, in the lab
- Booting the system in a production configuration which is shipped to the field

Development loading often does not require a great deal of error recovery capability, as the user is able to monitor the boot and load process. Often, manual control of exception conditions is desirable in these cases.

Production booting and loading does require that the loader have the ability to handle problems (exceptions) encountered during the boot sequence and recover from them gracefully. The requirements for a production loader will vary from application to application, dependent on how the application is deployed (e.g., a flight control system has different requirements from a voice mail system).

Regardless, it is extremely important to understand that booting of the target and loading of a BlueCat image is always a two-step process:

1. Primary, low-level initialization that is done prior to initiating loading of BlueCat Linux. This level of code is responsible for handling all aspects of the start-up sequence, regardless of whether BlueCat Linux or some other operating system is being loaded.
2. Loading an operating system boot, such as loading BlueCat Linux and the associated application for execution. This loading may occur over a network, from a disk or out of flash.

Once control is passed to BlueCat Linux, then it will be able to service exception conditions and start the developer’s application. Prior to that, the loading environment must manage the target processor to the extent that the developer’s requirements dictate. For example, prior to loading BlueCat Linux and passing control to it, how does the developer require memory faults to be handled? These issues are extremely developer specific, and thus cannot be covered in detail in this guide.

ROM Storage Requirements and Operation

Many embedded systems which execute BlueCat Linux have the capability to store BlueCat Linux (and potentially any associated applications) in ROM or flash, and properly boot it when the target processor comes out of reset. The BlueCat Linux environment does not have any specific formats for ROM-based storage, and thus is able to be installed in a variety of user target configurations. It is up to developers to design, implement, and install a low-level ROM storage support environments for their specific target environments.

A typical target ROM environment has the following characteristics:

- The ability to be integrated into the BlueCat build (`Make`) environment, allowing the ROM image to be created using the standard BlueCat Makefiles
- The ability to have control passed to a user-developed “loader” to load BlueCat Linux from ROM to RAM, and pass control to it
- Potentially have diagnostic capability to inform the developer of problems during boot, or handle catastrophic boot failures such as CRC errors of the BlueCat Linux image.

BlueCat Linux, for each supported target, is shipped with example ROM-based loaders which are integrated into the build environment. They are located in the following directory, or in a subdirectory contained within this directory:

`$(BLUECAT_PREFIX)/usr/src/linux/arch/`*target*`/boot`

The following steps form the foundations for booting BlueCat Linux from ROM:

1. Coming out of reset, the target processor executes initialization code that brings the hardware to an operational state.
2. Control is passed to a BlueCat Linux ROM based loader.
3. The loader transfers BlueCat Linux out of ROM, potentially decompresses it, and passes control to the entry point.

The developer is responsible for management of his or her ROM system. For example, systems with flash must have a means of programming the flash. This may be through monitor resident tools, through flash programmers or through Background Debug Mode-type emulators. The

BlueCat Linux environment does not provide any native support for installation of the BlueCat image through writing to flash.

Summary

The basic infrastructure for porting BlueCat Linux to new target environments is contained in the release for all BlueCat Linux supported processors. This infrastructure forms a good foundation for the developer to port to new environments. For example, a flash loader that is integrated into the BlueCat Linux build environment exists in every release of each supported processor.

The demonstration programs provide an excellent starting point for configuring a new target environment. They are contained in the following directory:

```
$(BLUECAT_PREFIX)/demo
```

An example of how these might be applied would be in starting with the demonstration `hello`. This demonstration does not require much infrastructure support, yet a build of it will generate images suitable for installing in flash or booting over the network. Once this basic type configuration is operational on the developer's hardware, other more robust demos, such as `ping` or `nfsroot` can be installed.

Additionally, LynuxWorks provides a full set of professional services to aid the developer with a port. These services include training, on-site consulting, or complete porting. Contact LynuxWorks for additional information on these services (see “Preface” section).

Index