LogicLoader™ User Guide

(LogicLoader Version 2.5)

Logic PD // Products
Published: July 2011
Last revised: April 2012

Revision History

REV	EDITOR	REVISION DESCRIPTION	LoLo Ver.	APPROVAL	DATE
A	EN	-Initial Release	2.5.0	JCA	07/28/11
B	SO, EN	-Throughout: General updates; Removed information on legacy syntax and config block; -Section 6.1.1: Corrected first command to include *image.elf* file name; -Section 7.3.1: Removed references to */dev/config*; -Section 7.5.2: Removed references to external mode line for ignoring Q key, as this functionality is no longer supported; -Section 9.1: Added information on where CPU programming manuals can be found; -Section 11.2: Added example output for the last command in the section; -Section 12.3: Added note that LogicLoader can read/write FAT16 file systems, but only read FAT32 file systems; Corrected command to create a read-only FATFS; -Section 13.2.1: Corrected length of *boot* partition to 6 blocks and 384kB; Corrected start block of *data* partition to 11	2.5.1	EN, DH	04/03/12

Table of Contents

Table of Figures and Tables

1 Introduction to LogicLoader™

1.1 Overview

LogicLoader™ is a bootloader/monitor program developed by Logic PD that initializes an embedded device and is capable of loading both operating systems and applications. In addition, LogicLoader provides a full suite of commands for hardware configuration, in-field device management, hardware debug, manufacturing, and test.

Customizable and extendable at the user level, LogicLoader is built for multiple processor platforms (ARM, ColdFire, i.MX, XScale), with support for both CompactFlash FAT and YAFFS file systems. LogicLoader contains a fully integrated TCP/IP stack, with DHCP and TFTP support, providing network bootstrap support. Greater customization to your specific needs can be achieved through conditional scripting and the ability for LogicLoader to drive LCD displays to show custom splash screens, making LogicLoader an excellent tool to fast-forward your embedded product design.

1.2 Differences between Versions 2.4 and 2.5

The following are the major differences between LogicLoader v2.4 and v2.5.

- The *lboot.lol* script replaces the config block; see Section 7
- New feature to handle custom video displays; see Section 8.3
- New feature for CPU pin configuration; see Section 9
- New *lboot.sup* setup file; see Section 10
- New Appendix for setup file keys

1.3 Product Features

Operating System (OS) Bootstrap
- Load multiple OSs (Microsoft Windows Embedded CE, Linux, etc.)
- Load an OS from SD/MMC, CompactFlash, resident flash array, serial connection, or Ethernet connection
- Fully configure a hardware platform for the OS
- Activate custom software functions to initialize hardware before the OS starts
- Power on self-test capability

In-field Device Management
- Modify boot actions at run time
- Remote device management eases debugging and upgrading

Hardware Debug
- Link in custom test functions to verify custom hardware
- Use a familiar UNIX-like interface for debugging the device
- Ethernet-based download and debug interface for Windows Embedded CE

Custom Applications
- Use LogicLoader to load, burn, and jump to any custom embedded application

Manufacturing and Test
- Add in custom functional test software for your specific device needs
- Take advantage of the fast Ethernet connectivity to reduce manufacturing test time

Download Formats
- SREC
- ELF
- BIN
- RAW

1.4 Acronyms

API	Application Programming Interface
BIN	Microsoft BIN file format
CPLD	Complex Programmable Logic Device
CF	CompactFlash®
DHCP	Dynamic Host Configuration Protocol
EEPROM	Electrically Erasable Programmable Read-Only Memory
ELF	Executable Linkable Format
FAT	File Allocation Table
FATFS	File Allocation Table File System
GPIO	General Purpose Input Output
GNU	GNU is not UNIX
IO	Input/Output
IP	Internet Protocol
JTAG	Joint Test Action Group
LAN	Local Area Network
LwIP	Lightweight implementation of the TCP/IP protocol stack
OS	Operating System
RAM	Random Access Memory
RAW	RAW file format, e.g., absolute binary
RISC	Reduced Instruction Set Computer
SOC	System on Chip
SOM	System on Module
SRAM	Static Random Access Memory
SREC	Motorola S-Record file format
TCP/IP	Transport Control Protocol/Internet Protocol
TFTP	Trivial File Transfer Protocol
YAFFS	Yet Another Flash File System

1.5 Technical Specifications

Please refer to the component specifications and data sheets applicable to your SOM:

- SOM Hardware Specification
- Applicable Processor Manual

1.6 LogicLoader v2.5 Command Description Manual

For a complete description of the LogicLoader shell's (losh) commands, please see the *LogicLoader v2.5 Command Description Manual*[1] available from Logic PD's website. The *LogicLoader v2.5 Command Description Manual* explains how to use each LogicLoader command.

1.7 LogicLoader User Manual Addendums

Logic PD has written a SOM-specific addendum for each SOM that runs LogicLoader. The *LogicLoader User Manual Addendum* is located under the *User Manuals* heading on Logic PD's *Registered Products* downloads page.[2]

1.8 LogicLoader Labs

Logic PD has written informal labs that provide a step-by-step introduction to basic LogicLoader commands and usage for specific SOM platforms. These labs are available for download under the *User Manuals* heading on Logic PD's *Registered Products* downloads page.

[1] http://support.logicpd.com/downloads/1440/
[2] http://support.logicpd.com/auth/

2 LogicLoader

2.1 LogicLoader Overview

LogicLoader is a bootloader/firmware monitor program developed by Logic PD. LogicLoader is designed to initialize an embedded device, load and bootstrap an operating system, and provide a low-level firmware monitor with debugging functionality.

2.2 LogicLoader Basics

Most OSs rely on an underlying bootloader to initialize a device from its reset condition. In general, OSs are designed with the assumption that the system will be in a specific, pre-defined state before the OS is started. Some example assumptions might be that system RAM has been initialized and cleared, processor interrupts are disabled, and a timer has been initialized to provide a system tick for the OS. The LogicLoader program initializes Logic PD's SOM platforms and prepares them for use by an OS.

Another basic function of LogicLoader is the capability to upgrade device software (flash memory, CPLD firmware, serial EEPROM contents) after deployment. This in-field upgrade ability requires a bootloader program that is capable of loading software images from various sources, as well as committing loaded images to non-volatile memory. LogicLoader implements this by enabling the system to load system software from flash memory, a CompactFlash storage card, a Local Area Network, or a device attached to the system's serial port. LogicLoader also has the ability to upgrade an existing OS residing in system flash.

LogicLoader was developed to fulfill the need for an OS- and processor-independent bootloader that can interface with a variety of hardware transports. The GNU development tool chain used to build LogicLoader is cross-platform capable.

2.3 Using LogicLoader for Debugging

LogicLoader implements a feature-rich firmware monitor, including the LogicLoader shell (losh). Losh is a command interpreter providing control over system state prior to loading an OS image. It has features such as command recall, command line editing, automated control via scripting, and diagnostic routines.

Losh includes many commands designed specifically to help software and hardware engineers debug low-level interfaces. Some examples include:

- Read and write any arbitrary memory address using the *x* and *w* commands.
- Read and write any arbitrary register in a peripheral using *x* and *w* specifying a device in the *filename* argument.
- Automatic LogicLoader runtime integrity check. When idle, LogicLoader will continually perform a checksum on itself to test for any corruption. If any corruption is detected, a warning will be printed, and the shell variable *SYS_INTEGRITY_FAIL* will be incremented for each failure. *SYS_INTEGRITY_PASS* is incremented for each correct checksum calculated.
- Memory detection. Memory detection is done at LogicLoader boot time. The detected memories will be indicated in the *MEM_xxxxxx* shell variables.
- Manufacturing ID information. LogicLoader will read the contents of the ID ROM on the SOM and populate the shell variables *ID_xxxxxx*. Also, *info id* can be used to read the ID information.
- Device type and state information. Using *info device*, the user can view the type of device installed, as well as the state and capabilities of that device.

- Memory layout information. Using *info mem,* the user can view the memory map location of every memory device. *Info mem* provides memory geometry information, including bad block information of NAND flash devices. *Info mem* also indicates LogicLoader's memory usage.
- Display and modify any CPU pin that LogicLoader uses. Any CPU pin that LogicLoader uses can be inhibited or redirected to a different pin (within the capabilities of the CPU). See the *info pin* and *pin* commands.

All commands return a value to the command line that can be used to conditionally evaluate the command result. Refer to the *LogicLoader v2.5 Command Description Manual* for a complete description of all available commands.

Developers may code their own test programs using the provided GNU development tool chain and use LogicLoader to load and run their software. This provides the ability to verify and debug hardware interfaces without the overhead of building, downloading, and running large OS images.

2.4 Manufacturing Advantages with LogicLoader

LogicLoader can be used with a desktop software utility to load a device's system software on the manufacturing line. This utility is customizable to suit your desired transfer mechanism and additional needs. LogicLoader can also be augmented with functional test software to completely verify a device before it leaves the manufacturing line. For example, LogicLoader could launch a device's final functional test at the end of a manufacturing line, and then load the device's final software image before packaging. Contact Logic PD[3] for more information on using LogicLoader to streamline manufacturing.

[3] http://support.logicpd.com/support/askaquestion.php

3 The LogicLoader Shell

3.1 LogicLoader Shell Overview

Losh is a command interpreter similar to those found in Unix environments. Losh implements a rudimentary network and file system command set, enhanced with custom diagnostic and memory manipulation commands for debugging hardware.

Developers familiar with a Unix-like command line interface should find the losh implementation familiar and easy to work with. Many of losh's commands are patterned after their Unix counterparts and share the same syntax.

3.2 LogicLoader Shell Basics

Losh uses a standard output stream (stdout). By default, stdout refers to a SOM's debug serial port. The output of any command that displays information to stdout (e.g., the 'cat' command) can be viewed using the terminal emulation program connected to the SOM's debug serial port. Likewise, the standard input stream (stdin) also refers to the SOM's debug serial port by default.

Losh includes a virtual file system that uses standard Unix path names. The highest-level (or root) directory is designated by the identifier /. A special sub-directory of the root with the name *dev* is used to enumerate and interact with the system's various peripherals and their associated device drivers.

3.2.1 Using the LogicLoader Shell

Losh includes both a basic command line editing feature and a command history feature. This provides you with a quick way to repeat commands. Using the up and down arrow keys, you can scroll through the list of previously executed commands. When a desired command is displayed, press RETURN to repeat the command. The right and left arrow keys move the cursor anywhere within the current line. This allows you to modify, delete, or insert text anywhere in the current line without having to backspace the entire line and re-type commands.

Losh includes a user help feature through the *help* command. Typing *help* followed by any command name at the losh prompt will display the command's syntax, usage, and an example. This may be especially helpful to users who are just becoming familiar with losh.

Commands may be run in the background by adding an '*&*' suffix.

3.2.1.1 Understanding the *ls* Command

The *ls* command lists the contents of the current directory. A sample terminal output that results from running the *ls* command is shown in Figure 3.1 below.

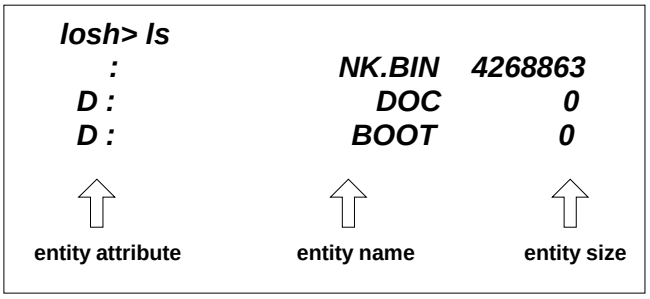

Figure 3.1: ls Command Columns

The first column, *entity attribute*, can be blank or contain a *D*, *S*, *R*, *r,* or *H*. A blank field indicates a normal attribute, a *D* indicates a directory attribute, an *S* indicates a device driver attribute, an *R*

indicates a read-only attribute, an *r* indicates that reserved bits are set, and an *H* indicates a hidden attribute.

The second column, *entity name*, is the name of the entity as it exists on the file system. This name should be used, with attention to case, in any commands referencing the entity.

The third column, *entity size*, indicates the size (in bytes) of the entity on the storage device.

4 Flash Devices and LogicLoader

LogicLoader supports both NOR and NAND flash devices; however, the usage is entirely dependent upon the available flash type(s) on your SOM (i.e., some Logic PD SOMs only have NAND, some only have NOR, and some have both NAND and NOR). NOR flash devices are linear, memory-mapped devices that can be read in a similar manner to any RAM device. Programming NOR devices requires a programming algorithm. LogicLoader supports NOR flash devices conforming to the Common Flash Interface (CFI) specification, which includes most NOR flash devices used today. NAND flash devices are block devices that require read and write algorithms. Currently, there is no common algorithm used to read or write to NAND flash devices; every manufacturer requires a unique algorithm.

4.1 NOR Addressing

Reading from a NOR device occurs in a similar manner to that of any RAM device. Writing to a NOR device, however, is a little more complicated. The default state for NOR flash is that each bit is set at 1. Halfwords can be used to set bits from 1 to 0; however, writing to a NOR device can only set bits from 1 to 0. In order to set a bit from 0 to 1, the entire block containing that bit has to be erased (i.e., all bits in that block are returned to their default state of 1).

Despite the similar addressing scheme between NOR flash and RAM devices, NOR flash cannot be used as a RAM device because NOR is block-organized to allow for erasing. The fact that NOR can be read as RAM is only used at boot time, when it can be used as a permanent byte addressed storage device. When NOR is used as a file system device, block addressing is used.

4.2 Booting from NOR

When LogicLoader is stored in NOR flash, it relocates itself at boot time from flash memory to system SDRAM and then spends the remainder of its run time executing out of system SDRAM.

4.3 Booting from NAND

When LogicLoader is stored in NAND flash, it requires a pre-loader called NoLo. NoLo is responsible for locating LogicLoader in NAND flash and then copying LogicLoader to system SDRAM. For platforms that boot from NAND, LogicLoader is located in a flash file system on the NAND device and is named *lboot.elf*. Once LogicLoader is in system SDRAM, it spends the remainder of its run time executing out of system SDRAM.

4.4 Booting from SD/MMC

When LogicLoader is stored on an SD/MMC card, LogicLoader requires the NoLo pre-loader to boot. The exact file name of the NoLo file is platform dependent and is dictated by the CPU boot ROM. NoLo is responsible for locating LogicLoader on the SD/MMC card and then copying LogicLoader to system SDRAM. For platforms that boot from an SD/MMC card, LogicLoader is located in a file system on the SD/MMC device and is named *lboot.elf*. Once LogicLoader is in system SDRAM, it spends the remainder of its run time executing out of system SDRAM.

4.5 NAND Addressing

NAND devices use an addressing scheme of block, page, and sector. A block is the smallest erasable chunk of memory, whereas pages and sectors are merely mechanisms that describe the addressing hierarchy (blocks are made up of pages; pages are made up of sectors). The number of blocks, pages, and sectors will be unique for each particular NAND flash device. Some NAND devices may not have any sectors, in which case addressing is performed using only blocks and pages.

NAND devices currently come in two types where addressing is concerned: small page and large page. Small-page devices have a page size of 512 bytes; large-page NAND devices have a page size of 2048 bytes. Larger page sizes tend to offer higher densities of NAND flash.

Whether the smallest chunk of data is addressed using a page or a sector, there is a spare area associated with that smallest chunk. This spare area will be 16 bytes for small-page devices and 64 bytes for large-page devices. The spare area is used by software to manage:

- Error correction codes to correct single-bit errors and to identify two or more bit errors.
- Manufacturer bad block identification.
- Flash file system metadata. The specific metadata will be unique to the particular flash file system used. LogicLoader dedicates a portion of the NAND spare area to YAFFS.

4.6 NAND Bad Blocks

NAND devices can develop bad blocks over time, as well as contain bad blocks when shipped from the manufacture. Bad blocks are defined as having two or more bit errors within the block. Single-bit errors need to be corrected with software using an ECC algorithm. Most NAND blocks can be erased and rewritten on the order of 100,000 cycles before potentially going bad. NAND manufacturers state that the device integrity decays only with erase/program cycles. However, some third-party studies indicate that data integrity may decay with a large number of read cycles as well. LogicLoader and YAFFS assume data integrity does not decay with reads. YAFFS assumes writes may lose integrity over time, so NAND writes are all verified and two or more bit errors will result in YAFFS marking the block bad.

Unlike NAND flash, NOR flash devices do not have bad blocks.

4.7 NAND Programming

NAND devices are programmed by sending commands to the device. Similar to that of NOR devices, programming of NAND devices consists of an erase phase that fills the entire block with 1s and a program phase that writes 0s to the device. Since NAND is a block device, a flash file system is needed to manage where data is read from and written to in order to avoid bad blocks on the device.

4.7.1 Skip Bad Block Method

A common algorithm used to program flash devices in production is the *skip bad block method*. This is a flash file system in its simplest form. As the name implies, data is written contiguously on the device from low numbered blocks to higher numbered blocks, while skipping any bad blocks marked by the manufacturer. This algorithm works well for programming a NAND device once, but is not capable of removing and rewriting portions of the written image.

4.7.2 YAFFS Overview

The YAFFS file system has been optimized for NAND use. YAFFS is able to:

- Identify and avoid bad blocks using an ECC algorithm.
- Use load leveling, where erasing and writing is averaged out among all the blocks of the device, and no one block is erased and written repeatedly.
- Manage metadata, such as directories and links.

YAFFS comes in two types: YAFFS1 and YAFFS2. YAFFS1 is the first incarnation of the YAFFS file system and only supports small-page NAND flash devices. YAFFS2 is an improved version that supports both small-page and large-page NAND flash devices. The losh environment makes

no distinction between the two types and refers to both YAFFS1 and YAFFS2 as *YAFFS*. For the remainder of this document, any reference to *YAFFS* is applicable to both YAFFS1 and YAFFS2.

More information regarding how YAFFS operates in LogicLoader can be found in Section 13 of this document.

5 Block Devices

Within LogicLoader, a block device is any device that only contains pages and sectors that can be read from or written to (this includes devices that require a block erase before a write). Examples of block devices are: ATA, NOR flash, and NAND flash. Losh supports all block devices with the same command set. More detailed information for each command can be found in the *LogicLoader v2.5 Command Description Manual*.

5.1 Using Block Reference

In the losh command set, there are two different methods to directly reference a block on a device. Some commands require a *B* to be placed in front of the block number. For example:

```
losh> erase /dev/nand0 B9 B500
```

In this example, omitting the *B* would indicate flat memory addressing. Use of flat memory addressing is discouraged and should be replaced with block-aligned memory addressing.

Other commands require the block address to be written without a *B* in front of the block number. For example:

```
losh> part-add /dev/nand0 a 1 1024
```

This command creates a partition *a* in the NAND device. The partition starts at block 1 and is 1024 blocks long. (Partitions are discussed in detail in Section 11 of this document.)

Because the specific command dictates the proper method to reference a block, it is important to understand the requirements of that specific command. The *help* feature may be useful in determining which method should be used.

5.2 burn

The *burn* command works with any block device. It burns the loaded image to the device from a given block offset. For example:

```
losh> burn /dev/flash0 5
```

This command will burn the loaded image to flash starting at block 5.

For burning to a NAND device with the skip bad algorithm, use the *dd* command as described below.

5.3 dd

The *dd* command copies blocks from a source device to a destination device. The command can use the skip bad block algorithm and it can be turned on with a flag. Use of the *dd* command is not limited to block devices; it can be used with whatever device you want. However, the device used with the command does determine how to specify block size. For NOR flash, the *dd* command requires data to be aligned to the width of the device; for ATA or NAND flash, the command requires you to specify the exact read/write page size. For ATA, the page size is 512; for NAND flash, the page size is 512, 1024, 2048, or 4096 (518, 1056, 2112, or 4224 when

including spare area). The *info mem* command can be used to determine the correct page size for your NAND device, where the page is denoted by the more general term *chunk*.

The *dd* command can also be used to provide an image that is file-system independent to program multiple NAND flash devices in manufacturing. Please contact Logic PD for more information on how to use the *dd* command to create an image suitable for a manufacturing environment.

For example:

```
losh> dd if:/load of:/dev/nand0 count:16 ibs:512 obs:512 os:16
skip_bad:1
```

This example command will copy the contents of *load* into the NAND data area, skip spare area, and use the skip bad block algorithm. For command argument details, please see the *LogicLoader v2.5 Command Description Manual*.

5.4 *erase*

The *erase* command can be used with any device. However, extra caution must be taken when erasing NAND and NOR blocks so as not to erase a YAFFS partition or any LogicLoader files. An attempt to erase these files or partitions will require confirmation before the erase command continues; this will prevent mistakenly erasing files required by the system to boot. NAND blocks with bad block markers will not be erased. For example:

```
losh> erase /dev/nand0 B10 B502
```

This command will erase 502 blocks of the device starting at block 10.

There is an optional *force* argument that can be used with the *erase* command. The *force* argument will force the *erase* command to erase all blocks in the specified address range even if they have been marked bad. Without the *force* argument, the *erase* command will skip bad blocks in an effort to preserve bad block information. Extreme caution must be used when using the *force* argument. If a block has been marked bad by the NAND manufacturer, and the block is erased with the *force* argument, there is no way to ever recover the bad block information. For example:

```
losh> erase /dev/nand0 B10 B502 force
```

NOTE: The legacy syntax `erase <offset> <length> <device>` is only supported for compatibility reasons.

5.5 *info*

The *info* command can be used to return specific information about the NAND and NOR devices, as well as information about any YAFFS boot partitions. This information is returned by using the *info mem* and *info YAFFS* arguments. The *info mem* command includes geometry data for NAND and NOR flash devices. The geometry information includes:

- Base address (unique to NOR devices)
- Number of blocks

- Bytes per block
- Is chunk device (if *is_chunk_device* equals 0, then the following information is not relevant and, therefore, is not printed)
- Number of chunks
- Chunk size
- Bad block list
- Bytes per chunk
- Bytes per spare

5.6 update

The *update* command is used to load and install an update image; it also includes support for updating LogicLoader in the YAFFS partition. When update files are sent to the SOM using the *update* command, LogicLoader will identify the update as LogicLoader and then program the NAND part as needed.

6 Program Loading

Using LogicLoader to download any application, operating system, or update to a device requires an understanding of the interaction between the *load*, *burn*, *jump*, and *exec* commands. The purpose of this section is to describe each individual command and explain the interaction between these commands.

6.1 Understanding the *load* Command

The purpose of the *load* command is to transfer a binary image to a device. The image must be in one of the following supported formats: ELF, SREC, RAW, or BIN. The *load* command uses information inherent to the supported formats (or as entered as part of the command for RAW format) to determine where the downloaded image should be stored in the device's memory. The *load* command stores the destination address of the downloaded image for later use by the *burn* command, and stores the program start address for later use by the *jump* or *exec* commands. For RAW format, the *load* command will store the destination address as the program start address. The image must be destined to reside in either flash memory, system RAM, or on-chip SRAM.

The *load* command also creates a file in the root of the file system called */load*. This file can be used by any other file system commands; a common use of the */load* file is to copy the loaded image into a YAFFS partition.

If an image is destined for system RAM or on-chip SRAM, the *load* command stores the image directly to its run-time location. Refer to Figure 6.1 for a graphical representation of this process.

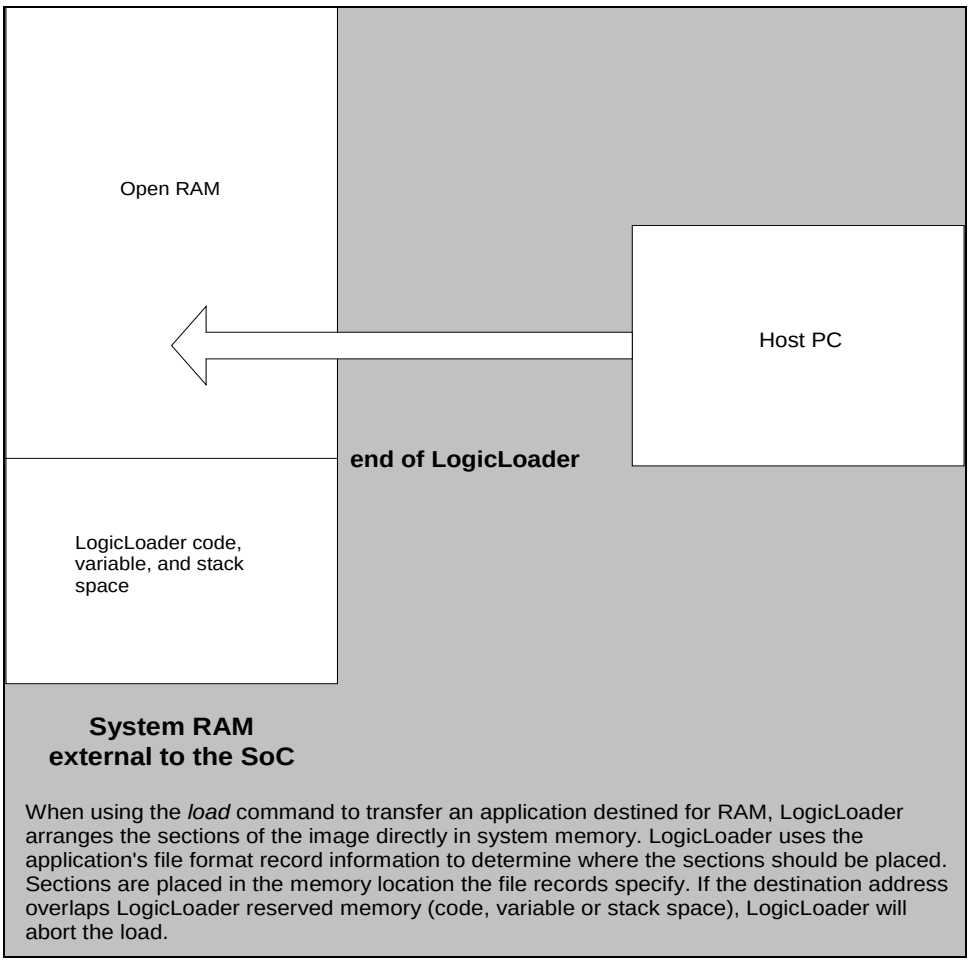

Figure 6.1: Downloading to RAM

If a downloaded application is destined for flash memory, the *load* command transfers the file into a temporary RAM buffer on the device. The transferred image may be programmed into flash using the *burn* command after the transfer is complete. Refer to Figure 6.2 for a graphical representation of this process.

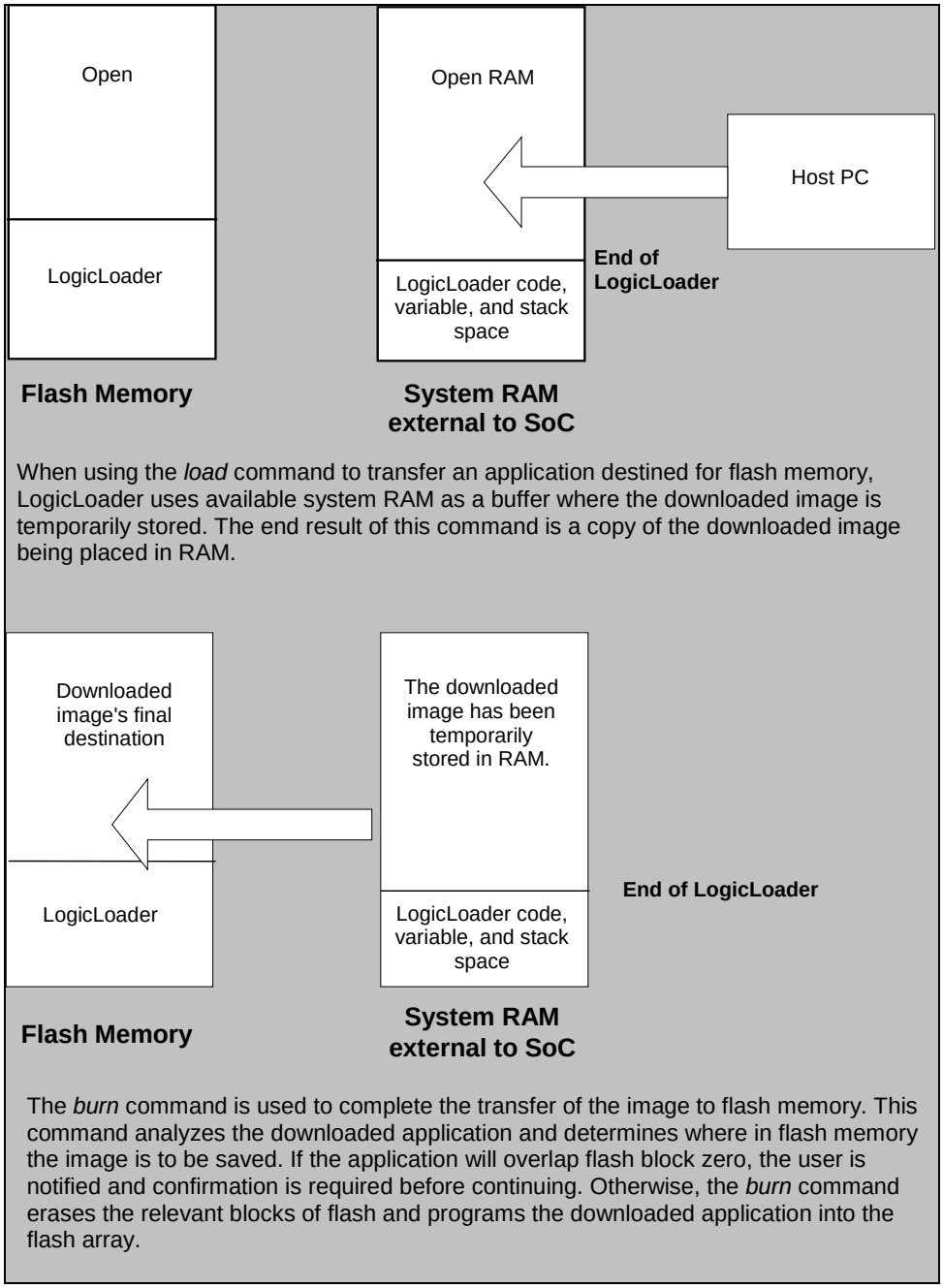

Figure 6.2: Downloading to Flash

6.1.1 Using TFTP as a Source

A file located on a TFTP server can be used as the source for the following commands: *load*, *cat*, *hd*, *md5sum*, and *cp*.

The general form for a TFTP file is */tftp/<server>:<filename>:[port],* where <*server*> is the IP address of the server, <*filename*> is the name of the file on the TFTP server (including

subdirectory identifiers), and *[port]* is the optional port number the TFTP server is listening to. If nothing is specified for the port, it is assumed the TFTP server is using the standard port 69.

For example, to load the ELF file *image.elf* from a TFTP server accessible at IP address 192.168.3.6 that is listening on the standard port, the following command would be used:

```
losh> load elf /tftp/192.168.3.6:image.elf
```

Another example would be to load the Platform Builder file *NK.bin* from the TFTP server at IP address 10.1.240.10 listening on port 3001:

```
losh> load bin /tftp/10.1.240.10:NK.bin:3001
```

6.2 Understanding the *burn* Command

The *burn* command should only be used following the successful download of a binary image destined for flash. If the *load* command is used to download a flash image, the image is temporarily stored in a reserved section of system RAM. The *burn* command is responsible for actually erasing the necessary blocks and programming the downloaded image into flash at the destination address. Refer to Figure 6.2 for more information.

6.3 Understanding the *jump* and *exec* Commands

LogicLoader provides two different ways to transfer execution to your application. The *jump* command is more useful for launching and debugging an application that will be relying on LogicLoader or an OS to setup the run-time environment. The *exec* command is more useful for launching an application, such as an OS that will take over total control of the hardware and the environment. The differences between the *jump* and *exec* command are that only *exec* can pass a command line argument to the program being executed and that *exec* disables interrupts, the cache, and the MMU (if present).

6.3.1 The *jump* Command

The *jump* command is an assembly-level jump to the starting instruction of a program. If *jump* is executed without a parameter, LogicLoader will jump to the program start address of the last program loaded to system RAM (if any). If an address is passed in, the *jump* command will jump to the specified address. After a *jump* command is performed, LogicLoader continues to execute in the background. LogicLoader does not set up a run-time environment for a program; instead the program inherits LogicLoader's current environment. It is the software engineer's responsibility to ensure that the hardware is set up in the desired manner.

This example may be used when writing a function that LogicLoader will jump to:

```
losh> int my_jump_function(void);
```

6.3.2 The *exec* Command

The *exec* command is an assembly-level jump to the starting instruction of a program that will pass in three arguments. If *exec* is executed without a parameter, LogicLoader will jump to the program start address of the last program loaded to system RAM (if any) and pass in a pointer to an empty string. If both an address and command line are specified, the *exec* command will jump to the specified address and pass a pointer to the command line provided. The *exec* command will disable interrupts, the cache, and the MMU (if present) prior to executing the jump.

The *exec* command passes the command line argument via a pointer to memory that has been allocated from LogicLoader's heap. Any application or OS code must preserve the command line, or finish using the command line arguments, before reclaiming LogicLoader's memory space for its own use. Because the *exec* command shuts off the MMU, the image must have a virtual address that maps directly to its physical address since the entry address that *exec* jumps to will always be a physical address.

This example may be used when writing a function that LogicLoader will *exec* to:

```
losh> int my_exec_function(unsigned int arg1, unsigned int arg2, char
*cmd_string);
```

The first two arguments, arg1 and arg2, have different values depending on the flags given to *exec*. The third argument will be a pointer to the command line, as described above.

To boot an ARM Linux kernel, use the *-t* argument with the *exec* command. This causes arg1 to become zero; arg2 is then the architecture ID, and arg3 is a pointer to an ATAG structure that contains, among other things, a pointer to the *cmd_string*.

6.3.3 Command Example Using *load* and *burn* with *jump* or *exec*

An application program that is written for the Zoom Development Kit can be linked to reside in flash or RAM.

First, let's assume that we have built an application for flash. To properly store this program in flash, issue the *load* command followed by the *burn* command. Make note of the program start address (for example: 0x400d0100) so that you can jump to the program after a reset. Once the image has been burned to flash, you may enter the *jump* or *exec* command specifying 0x400d0100 as the argument at any time. However, you can take a shortcut if you have not reset the board, since the *load* command will store the program start address. A valid sequence would be as follows:

```
losh> load elf
```

This transfers the image to the device.

```
losh> burn
```

This programs the image into flash at the destination address stored by the *load* command.

```
losh> jump or exec
```

This will work because the *load* command saved the program's flash start address. Both the burn destination address and the program start address will be valid until the next reset or the next use of the *load* command.

After a reset, the program may be launched at any time using the jump or exec commands with a specific destination address:

```
losh> jump 0x400d0100
```

or

```
losh> exec 0x400d0100 –
```

Next, let's assume that we have built an application for RAM. To properly load and execute an application out of RAM, issue the *load* command followed by the *jump* or *exec* command. A valid sequence would be as follows:

```
losh> load elf
```

This transfers the image to the device.

```
losh> jump or exec
```

This will work because the *load* command stored the program start address. The program start address will be valid for this program until the next reset, or the next use of the *load* command.

Keep in mind that the option of specifying the program start address, as shown in the flash example, is also available.

6.4 Understanding the *update* Command

LogicLoader deploys software or firmware updates in the form of update files (*.upd* extension). To deploy an update file, use the *update* command. If a filename/path parameter is not passed to the *update* command, the system will assume that *stdin* is being used to send the update file to the system. When the *update* command is activated after the system has received the *.upd* file, it automatically launches the file and performs the actions required.

Update files are comprised of self-extracting applications that, once activated by the update command, run and perform whatever function the application was coded to carry out. This allows a single *update* command to perform a variety of different actions from a self-contained file with minimal user interaction.

The procedure to update LogicLoader with the *update* command differs from the *load/burn* procedure in that only one command implements the entire update process without any user interaction or confirmation.

7 Scripting

7.1 Scripting Overview

Scripts can be used to automate any commands or command sequences entered on the command line. Scripts are comprised of a simple text file with a listing of commands that the user wishes to automatically execute in sequence.

7.1.1 Scripting Rules

Basic scripting rules are as follows:

- Enter commands into the script file with the same syntax used on the losh command line
- Separate commands with a semi-colon or a new line
- End the script with a \n (this tells the parser to stop parsing the file and instructs the command interpreter to start executing the script)
- Use the command exit to end the script (this tells the command interpreter to stop executing the script)

7.2 Launching Scripts

The process of launching a script manually, or post-boot time, uses the *source* command. For example, the command **`source /cf_card/myscript.txt`** will execute the script stored in the file *myscript.txt* on a mounted CompactFlash card. For more information on the *source* command, please refer to the *LogicLoader v2.5 Command Description Manual* document.

The process of auto-launching scripts on startup is referred to as *boot-time scripting*. Boot-time scripts are the primary mechanism used for automatically launching an OS or application when deploying a product to the field. Their capability is the same as other scripts, except that they can be automatically run at startup. You can think of a boot-time script fulfilling the same role as an *autoexec.bat* file commonly found on desktop operating systems. Boot-time script usage is described more thoroughly below.

A third way to launch a script is to send it to the system while LogicLoader is waiting at the losh prompt. If the script file is sent over the terminal emulator connection to the losh shell, the script will be entered on the command line as if typed in by the user. If the script being sent incorporates a carriage return at the end of the script, the command line will launch the script when it receives the carriage return. This type of script launching is primarily used during development when the developer wishes to send a number of development commands to LogicLoader in sequence. For example, if a command sequence initializes the Ethernet interface, downloads a Windows CE OS image, and then launches the OS image with a specific command line.

7.3 Persistent Script Storage

In order for a script to persist across power cycles, the script must be stored in a local, non-volatile memory device on the system. There are a number of different persistent storage locations that can be used to store a script. The primary storage mechanisms supported by LogicLoader are the serial EEPROM, the resident flash array (YAFFS), and the SD/MMC interface. Because different SOMs may not have one or more of these interfaces available in hardware, please refer to the individual SOM's *LogicLoader User Manual Addendum* document for specific persistent storage interface support.

7.3.1 Persisting Scripts with the *echo* Command

The *echo* command can be used to store a script in the serial EEPROM. To include a new line in the first argument to echo, it is necessary to enclose the whole argument in double-quotes. Remember to end the script by inserting *n* before the end quotes to instruct the parser to stop parsing the file. Since scripts stored in the serial EEPROM are not stored as actual files, it is important that any previous information in the serial EEPROM is not interpreted as part of the script. Check the contents of the serial EEPROM with *cat* or *hd* to verify that the contents are as expected. If not, the *erase* command should be used to erase any previous information before the *echo* command is executed.

7.3.2 Serial EEPROM Scripts

For those SOMs that contain EEPROM, the system's serial EEPROM is one persistent storage area that supports the storage and execution of scripts. The serial EEPROM is the primary boot-time script storage location. Boot-time scripts stored to the serial EEPROM are typically short and may redirect to a secondary script on an interface capable of larger storage capacity.

To store a script to the serial EEPROM interface *(/dev/serial_eeprom*), use the *echo* command. An example of using the *echo* command to store information to the serial EEPROM is shown below:

```
losh> echo "LOLOmount fatfs /dev/ata0a /cf; source /cf/B.BAT; exit;\n"
/dev/serial_eeprom
```

7.4 Settings that Affect Scripts

The *set* command can be used to modify several internal variables affecting script execution. These function similarly to the Unix shell scripting analog, where a '-' causes the flags that follow to be set, and a '+' causes them to be unset. It is highly recommended during development to set the *-w* flag to receive warnings about common scripting errors.

The flags available are:

- e Exit script execution immediately when commands fail
- n Read commands, but do not execute; ignored by interactive shells
- q Do not print LogicLoader error messages
- u Exit on expansion of unset variables
- v Echo input lines as they are read
- w Print warnings for possible errors
- c Allow prompt for user confirmation
- x Echo all user commands before executing them
- m Mute all output of the script

7.5 Using Boot-time Scripting

It is possible to execute a script automatically at startup. This is useful for making the device jump into an operating system or other program when powered on without requiring manual command-line input. This functionality can be described as being equivalent to the system automatically calling the *source* command on one of the boot-capable devices.

7.5.1 Boot-time Script Guidelines

All of the commands available in LogicLoader are also available to boot-time scripts. As in normal scripts, the '#' character can be used to indicate a comment line. A line starting with '#' will be

ignored by LogicLoader but is often used to place comments within the script. For a script to be loaded at power up, the script must:

- Be located on the boot device. This would be in the */lboot* partition if booting from NAND flash, or the *root directory* if booting from SD/MMC.
- Be named *lboot.lol*.

7.5.2 Exiting a Boot Script

A common need is to abort the execution of a boot script in order to exit into the command line for additional debugging, development, or simply to change the boot script. The primary way to accomplish this is by holding the *q* key down in a terminal emulator program attached to the device's debug serial port.

The system pauses for one/half of a second to read from debug serial port to determine if an abort request is being made.

7.5.3 Understanding the *echo* Command

The *echo* command can be used to store a script in the serial EEPROM. The *echo* command only writes the number of bytes contained in the string. If the string to be written is shorter than the previous contents, the result of the echo will not be what is intended. Use the *cat* or *hd* command to verify the contents of the serial EEPROM before using the *echo* command.

7.6 Conditional Scripting and Variables

7.6.1 Variables

Losh supports the concept of shell variables. The syntax and usage of these variables are patterned after the BASH shell.

7.6.1.1 Variable Names

A variable name may be any sequence of letters, numbers, or underscore tokens.

7.6.1.2 Variable Assignment

A variable is created and assigned a value by using the '=' operator. For example,

```
losh> foo = 1
```

creates a new variable named *foo* and assigns it the value of 1. Once a variable has been created, it may be assigned a new value at any time by using the '=' operator again. The right-hand side of an assignment statement is not limited to a simple number; it can be a complex expression involving other variables.

7.6.1.3 Internal Representation

Variables are internally represented as strings. For example,

```
losh> foo = 1
```

internally points the variable *foo* at a sequence of characters equivalent to: 0x31 0x00. Because variables are treated as strings, commands may be aliased as variables. For example:


```
losh> e = echo
losh> msg = “Hello World”
losh> $e $msg
Hello World
```

Notice the quotes used to ignore white space. If the created variable will be assigned to more than one token, the tokens must be included in double-quotation marks.

7.6.1.4 De-referencing a Variable

To de-reference a variable, that is, to access a variable’s assigned value, use the ‘$’ operator. For example:

```
losh> foo = “Hello World”
losh> echo $foo
Hello World
```

The ‘$’ operator causes the shell to substitute the variable with the string value assigned to it. In some cases, a variable’s assigned value will be converted into a numeric value. This occurs when the shell is evaluating a conditional expression and is described in more detail below.

If a variable is referenced that does not have a previous value, its value is assumed to be zero and a warning message is printed.

NOTE: Enclosing a sequence of tokens within double-quotes binds them together into a single token. For example,

```
losh> e = “echo Hello World”
losh> $e
echo Hello World: command not found
```

will not work because the parser only evaluates the string once. Thus, instead of being split up into three distinct tokens, the double-quotes cause the tokens to be bound and treated as one.

7.6.1.5 Built-in Variables

The shell contains two built-in variables, namely ‘?’ and ‘@’.

The ‘?’ variable is assigned to the return value of the last command executed. By convention, all shell commands return either a zero to indicate that it completed successfully or a non-zero error code to indicate a failure. To view a command’s return value, use the *echo* command and the value of the ‘?’ variable. For example:

```
losh> mount fatfs /dev/ata0a /cf   # Mount a FAT file system.
losh> echo $?           # Display the value returned from the mount
                          command.
```


The '@' variable is an auxiliary variable that is set by some commands. For instance, the *echo* command sets this value to the number of characters that it wrote. Therefore:

```
losh> echo "Hello"
losh> echo $@
0x5
```

The number *5* is printed because the string *Hello* contains five characters.

Please reference the *LogicLoader v2.5 Command Description Manual* for specific command descriptions in order to learn which commands set the '@' variable and how to use these commands.

7.6.1.6 Saving and Loading Variables

Shell variables can be saved to a file (see *save-var* command), and re-loaded (see the *load-var* command) on different shell sessions. This is useful to preserve a variable state between executions, creating a repeatable set of variables and values at startup or providing variable data values to an OS or user application.

Using a variable file to load variables is in essence the same as running a LogicLoader script that uses the *make-var* command to create one or more variables. The difference is that you can use LogicLoader to create a variable file that can be made to load at boot time, prior to any script starting.

7.6.1.6.1 Variable File Format

The file format is ASCII, where each variable is delineated by a new line, and each of the variable meta-data are separated by a comma. The parameters are as follows (in order):

- Variable name
- Type, where the acceptable types are:
  - 1 – Number
  - 2 – String
  - 3 – Reserved – do not use
  - 4 – Byte pointer
  - 5 – Word pointer
  - 6 – Long pointer
- Variable value
- Protection, where the protection attribute is a bit mask of the following:
  - 1 – Read only
  - 2 – Static
  - 4 – Global
  - 8 – Hidden

Comments within the file can be made using the '#' character at the start of a line. Comment lines will be ignored by LogicLoader when loading.

7.6.1.6.2 Example File

The following example file below will create three variables:

- *example_var*: This variable is of type number, with initial value set to 0x55aa55aa.

- *ro_string*: This variable is a string variable with initial value of *stuff*. This variable has a protection set to read only. Once this variable is loaded into the shell, its value cannot be changed.
- *gpt2_tcrr_reg*: This variable is a long-word pointer to address 0x49032028. Any references to this variable within the shell will return the 32-bit contents of the data at 0x49032028.

```
# lboot.var example
#
# This file is an example of how to create a LogicLoader .var file.
#
# name, type, value, protection
example_var, 1, 0x55aa55aa, 0
ro_string, 2,stuff, 1
gpt2_tcrr_reg, 6, 0x49032028, 0
```

7.6.1.6.3 Loading Variables at Boot Time

Shell variables can be automatically loaded at boot time. Placing the .var file in the boot time location (*/lboot* when booting from NAND, or the *root directory* when booting from SD/MMC), and naming it *lboot.var* will ensure the variable file is loaded into the shell at boot time.

7.6.2 Conditional Scripting

Losh supports an *if-else-endif* programming construct as well as a *while* construct. The syntax for an if-statement and an if-else statement is shown below:

```
if (expression)
      action
endif


if ( expression )
      action-1
else
      action-2
endif
```

Parentheses are not required around the expression, but they are encouraged to improve readability of the script. Similarly, tabs and new lines are not needed. The various elements of the construct may be separated by the ';' operator if so desired. For example:

```
losh> if expression echo “pass”; else echo “fail”; endif
pass
```

or

```
losh> if expression echo “pass”
      else echo “fail”;
      endif
```


The syntax for a *while* statement is shown below:

```
while ( expression )
      action
done
```

The expression is evaluated first. If the return is non-zero, then action is taken and control comes back to the expression evaluation. This is repeated until the expression evaluates to zero.

Note that *if* and *while* statements can be nested. The following example calculates the greatest common divisor of the numbers stored in the variables *a* and *b*, leaving the result in *a*:

```
losh> while ($a .ne $b) {
            if ($a .gt $b) {
                  a = $a - $b
            } else {
                  b = $b - $a
            }
      done
```

7.6.2.1 Expressions

An expression is defined as a number or a combination of a logical operator and a number or numbers. If a variable has been defined and is being de-referenced in an expression, its value is converted to a number. An expression evaluates to true if the result is non-zero and false if the result evaluates to zero. Therefore, the simplest expressions would be:

```
if ( 1 )    # evaluates to true.
if ( 0 )    # evaluates to false.
```

7.6.2.2 Using Shell Variables

```
losh> foo=1
losh> bar=0x0
```

```
if ( $foo ) # evaluates to true.
if ( $bar ) # evaluates to false.
```

The other operators supported by the shell are listed below in order of decreasing precedence.

'-' '!' '~'	unary minus, logical not, arithmetic not
'*' '/' '%'	multiplication, division, modulus
'+' '-'	addition, subtraction
'<<' '>>'	left shift, right shift
'.lt' '.le' '.gt' '.ge'	less than, less than or equal, greater than, greater than or equal
'.eq' '.ne'	equality, inequality
'<' '>'	less than, greater than
'==' '!='	equality, inequality
'$((' '))'	immediate evaluation open, immediate evaluation close
'^'	bitwise exclusive or
'|'	bitwise or
'&'	bitwise and
'&&'	logical and
'||'	logical or

Note that the operators '==', '!=', '>', '<' apply to either strings or integers, but the evaluation is done as string comparisons. The operators '`.eq`', '`.ne`', '`.lt`', '`.le`', '`.gt`', '`.ge`' apply to either strings or integers, but each side of the expression must evaluate to numbers.

Immediate Expression Evaluation:

The immediate evaluation construct '$((' … '))' is used when a command needs an immediate value. In this case the expression contained in '$((' … ')) is immediately evaluated and returned as a number. For example, the *x* command cannot take an expression as its operand:

```
losh> x /x 0x80200000 + 0x10 4
error: x: wrong number of arguments
```

Using the immediate evaluation construct '$((' … '))' gives:

```
losh> x /x $(( 0x80200000 + 0x10 )) 4
0x80200010  04001000 eb000000 fe000001 40ea0003  ……….@
```

The following are all valid expressions that can be used as the right-hand side of an assignment, as an argument to a command (if enclosed in an immediate evaluation construct), or as the conditional expression in an *if* or *while* construct:

```
1 & 0            # evaluates to zero
1 | 0            # evaluates to one
0x01 ^ 0x02      # evaluates to 0x3
1 >= 2           # evaluates to zero
0 .ge 1          # evaluates to zero
1 + 3 * 5 ^ 7    # evaluates to 23 (reduces to 16 ^ 7)
```

As mentioned above, the shell exports two built-in variables. These are '?' and '@'. The variable '?' holds the return value of the last command executed. Therefore, constructs like the one below can prove to be very useful:

```
mount fatfs /dev/ata0a /cf
if ( $? )
      # Save current return values because 'echo' will overwrite them
      s_q = $?
      s_a = $@
      echo "Error, mount failed error codes: "
      echo $s_q
      echo $s_a
else
      echo "Mounted FAT file system at point '/cf'"
endif
```

NOTE: In the case of an error, the values of the '?' and '@' variables are saved. This is because the first call to the *echo* command will overwrite the value of those variables.

7.6.2.3 Escaping the Variable Character

If the *echo* command is used to store a variable reference in a script, the '\' operator must be used before that variable in order to defer evaluation of that variable until echoed. For example,

```
losh> echo "if ($a == 2) source bar;\n" /dev/serial_eeprom
```

needs to be written as

```
losh> echo "if (\$a == 2) source bar;\n" /dev/serial_eeprom
```

in order to prevent losh from evaluating the variable *a* in the string before the echo call is used. This method applies to any string which must include a literal '$' character.

7.6.2.4 Comments

In order to make it easy to self-document scripts, the shell recognizes and ignores comments. A comment begins with the character '#' and extends to the end of the current line.

7.6.2.5 Numbers

The shell recognizes the following number formats:

- decimal
  - contains the characters 0-9
  - does not start with a zero
- octal
  - contains the characters 0-7
  - starts with a zero
- hexadecimal
  - contains the characters 0-9, a-f, or A-F
  - starts with the sequence 0x or 0X

8 Video Interface

8.1 Video Interface Overview

LogicLoader includes the following video commands to configure the video controller:

- *video-clear*: clears the default video screen (sets the frame buffer to a monolithic color)
- *video-close*: turns off and un-initializes the default video device
- *video-fb*: sets or displays the video frame buffer address
- *video-init*: connects and initializes default video device settings, but does not enable the controller
- *video-off*: turns off an initialized display
- *video-on*: turns on an initialized display
- *video-open*: connects and initializes default video device settings and enables the display controller (equivalent of *video-init* and *video-on*)
- *video-add*: captures the current video controller register settings and assigns a name to that video mode; useful for creating custom display timings

8.2 Using the Video Interface after Initialization

Once the display has been initialized with either the *video-open* or the *video-init* commands, any of the drawing commands can be used. The *video-fb* command allows the user to change the frame buffer address.

After executing the *video-fb* command to change the frame buffer address, all drawing commands will use the new frame buffer address instead of the default. The *video-init* command can be used to connect and initialize the video controller without enabling the video display. Then use the *bitmap* command to draw to different areas in memory prior to using the *video-on* command to turn on the display. A typical command sequence might look like the following:

```
losh> video-init 7 16
video-init display: width: 640 height: 480 bpp: 16 disp: 7
losh> bitmap TEST1.BMP 0xc0400000
losh> bitmap TEST2.BMP 0xc0600000
losh> video-fb 0xc0400000
losh> video-on
.....other command sequences
losh> video-fb 0xc0600000

.....other command sequences
losh> video-off
```

8.3 Using a Custom Video Display

If using a custom video display, custom video timings can be assigned a name and be used to operate the display.

8.3.1 The *video-add* Command

Assuming the display timings have been determined, begin by writing those timings to the processor's video controller registers using the *w* command. Then, use the *video-add* command to assign a name to those settings. It may help to use **`info video < name>`** to view the display controller registers of a similar display. Then set those same registers with similar or new values depending on your video needs.

Once a name has been assigned to your video registers, other video commands like *video-open* and *video-close* can be used specifying your newly-created video name where needed.

Bear in mind that the *video-add* command will lose the name and register settings when the power is turned off. Be sure to record those timing settings and consider creating a LogicLoader script with the *video-add* command as a means to use your custom display between power cycles.

If you are unsure as to what the display register settings need to be for your display, please contact Logic PD for assistance.

NOTE: Some graphics controllers require the graphics controller to be disabled before setting the timing registers. In that case, use the *video-close* command, change your video settings, use the *video-add* command, and then use *video-open* with the new name to test out your settings.

9 CPU Pin Configuration

There may be times when it is necessary to prevent LogicLoader from using a pin on the CPU or to redirect LogicLoader to use a different CPU pin. LogicLoader associates an ID with every CPU pin it uses. Each pin ID can be configured to DNU (do not use) or configured to a different CPU pin mux register.

9.1 Pin IDs and Configuration

LogicLoader is designed to operate on many platforms. As such, LogicLoader has a unique ID for each pin function it needs. The command *info pin* is used to display the entire list of pin IDs within LogicLoader and the pin mux register currently associated with the pin ID. The CPU pin mux register address can be used to cross reference the exact hardware pin on the CPU using the CPU programming manual for your processor; this document can be obtained from the CPU manufacturer's website. For convenience, the *info pin* command will also display the GPIO number associated with that pin mux register. However, showing the GPIO number does not necessarily mean the pin ID is configured as a GPIO pin.

Further information about a specific pin can be obtained by specifying the pin ID with the *info pin [pin id]* command.

9.2 Disabling a Pin (DNU)

When there is a need to prevent LogicLoader from using a pin, the pin command can be used with the argument *dnu* (do not use) and a pin id. From that point on, any LogicLoader access to that pin will be ignored. Reinstating LogicLoader's use of that pin can be accomplished using the *clear* argument as in **`pin dnu <pin id> clear`**. In cases where it is necessary to inhibit LogicLoader from using a pin prior to when the shell is available, or even before LogicLoader can read in a script, a setup file can be created to inhibit the use of a pin. See the *LogicLoader Setup File* document for more information on creating a setup file.

9.3 Reconfiguring a Pin

There may be times when a user may want to redirect the use of a pin to some other pin. Consider, for example, a customer whose baseboard requires remapping the heartbeat LED used on the Logic PD development kit baseboard to a different pin.

1. Identify the pin ID of the LED on the Logic PD baseboard.
    a. From the schematic, identify the processor pin that is used to control the LED.
    b. From the processor pin identifier of the CPU, look up the pin mux register that is used to configure that pin in the CPU programming manual.
    c. Finally, the pin ID can be obtained by using the *info pin* command at the LogicLoader prompt. From the list, find the pin ID that is associated with the pin mux register identified in the CPU programming manual.
2. Identify the destination pin mux address. Each pin on the CPU has an associated pin mux address that identifies if the pin is to be used as a GPIO or dedicated to some other internal hardware interface. This information is available in the CPU programming manual.
3. Identify the destination GPIO number. Since the LED operation is controlled by turning a GPIO pin on or off, the GPIO number is needed.
4. Identify the mode of the pin mux. The pin mux mode is used to identify how the pin is to be used. Generally, it can be used as GPIO or as one of several other processor peripherals.

5. Identify the pull of the pin. Some processors support software configurable pull-up or pull-down states. If your processor does not support pull states, simply put any value here, as it will be ignored.
6. Identify the drive strength of the pin. Some processors support software-configurable drive strength. If your processor does not support drive strength configuration, simply put any value here or omit this argument, as it will be ignored.
7. Finally, supply this information to the LogicLoader shell command *pin redefine*, as in the following example:

```
losh> pin redefine 1 0x4800215e 133 4 disable nominal
```


10 The LogicLoader Setup *lboot.sup* File

The LogicLoader setup file is used in cases where LogicLoader needs to be configured/setup as soon as possible. Such uses may involve inhibiting LogicLoader from ever using a pin, where inhibiting such use at the losh prompt or in a script is simply too late in the boot process. Other uses may include writing to registers immediately on power up. Future LogicLoader versions may be used to configure shell UART baud rate, UART port number, SDRAM timing changes, etc.

10.1 Setup File Name

At boot time, one of the first things LogicLoader will do (prior to starting the system shell and most drivers) is look in the boot directory for a file named *lboot.sup*. If this file name is found, LogicLoader will process the keys contained within the file.

10.2 Setup File Keys

The setup file is loaded very early in the boot process. As such, the luxury of having a shell parser is not available. To simplify parsing the setup file, LogicLoader uses keys to identify what is being set. A list of keys can be found in Appendix A of this document.

10.3 Setup File Syntax

The setup file can be created and edited with any text editor. The file is in human readable form (non-binary). A "#" symbol can be used at the front of a line to indicate a comment. The file consists of a list of keys with associated parameters for each key. The number and definition of the parameters will be specific to each key. Below is an example of a simple LogicLoader setup file created using Windows Notepad:

```
# lboot.sup example
#
# This is an example LogicLoader setup file.
#
# The lboot.sup file is the first file loaded by LogicLoader at
# boot time.  See the example lboot.lol file for more info
# on boot file sequencing
#
#
# This setup file will:
#                 - Disable base board LED's
#                 - Write values to memory
#
# For each function, this file requires a key followed by
# a list of parameters.  Each key identifies the action to take.
# Each key has a specific number of parameters required.  Unknown
# keys are ignored.  Unknown parameters are ignored.
#
# See the LogicLoader user manual for more information regarding the
# lboot.sup file and a complete list of keys available.
#
# Keys used:
#         Key   Param 1   Param 2   Param 3  Description
#          1   Address    Value          Write byte to memory
#          2   Address    Value          Write word to memory
#          3   Address    Value          Write long to memory
#          4   pin ID                 Disable pin usage (dnu)

# Key, Param 1, Param 2
```



```
4, 1
4, 2
3, 0x81000000, 0xdeadbeaf
3, 0x81000004, 0x55aa5a5a
```


11 Partitions

11.1 Partitions Overview

Theoretically, partitions can be created on every block device. However, the current implementation of LogicLoader only allows users to create partitions on NOR and NAND devices. Partitions on other devices, such as ATA, CompactFlash, and SD cards, can be used, but new partitions cannot be created on those devices. There can be up to four partitions on a device; however, extended partition tables are not supported.

Inodes are created for every partition at boot time. For example,

```
losh> ls /dev
```

will return the output similar to that included below:

```
       S  :                          sdmmc0d        0
       S  :                          sdmmc0c        0
       S  :                          sdmmc0b        0
       S  :                          sdmmc0a        0
       S  :                           sdmmc0        0
       S  :                            ata0d        0
       S  :                            ata0c        0
       S  :                            ata0b        0
       S  :                            ata0a        0
       S  :                             ata0        0
       S  :                           nand0d        0
       S  :                           nand0c        0
       S  :                           nand0b        0
       S  :                           nand0a        0
       S  :                            nand0        0
       S  :                          flash0d        0
       S  :                          flash0c        0
       S  :                          flash0b        0
       S  :                          flash0a        0
       S  :                           flash0  2097152
       S  :                             null        0
       S  :                            uart0        0
```

Within this example, *nand0a* is only an empty inode at this time and cannot be accessed (unless previous partition tables have been created on the NAND device). Therefore,

```
losh> hd /dev/nand0a 512
```

will return the following error message:

```
Partition does not exist, type 0xff
error: hd: failed to open (/dev/nand0a)
```


The most important thing to know about partition handling is that there is a RAM-based partition table for every device (referred to as RAM-partition table in this document). At boot, LogicLoader tries to fill this partition with data from the device. If it does not find a partition table on the device, then it will be empty. This means that it will be filled with 0s or 1s, depending on the type of device (the partition is filled with 1s for NOR and NAND flash devices; the partition is filled with 0s for CompactFlash and SD cards).

Returning to the example above, the *nand0a* RAM-partition table is filled with 1s. The partition driver reads the corresponding entry from the RAM-partition table and finds that its type is 0xff or empty. This is why the error message states the partition does not exist.

So for every partition inode, there is a corresponding partition entry in the RAM-partition table.

11.2 Partition Creation in the RAM-Partition Table

Partitions can be created with the *part-add* command. For example,

```
losh> part-add /dev/nand0 b 1 2048
```

will create a partition entry in the *nand0* RAM-partition table. Note that the second partition of the device (*nand0b*) will be filled due to specifying *b* in the argument; this occurs because every device can have up to four partitions, which are labeled from *a* to *d*. The partition will be added beginning in block 1 and will have a length of 2048 blocks. For example, the following command prints the RAM-partition table for the device given as a parameter:

```
losh> info part /dev/nand0
```

Note that it is not possible to use a parameter such as */dev/nand0a* because this would instruct the command to access the RAM-partition table for the *nand0a* partition, but partitions within partitions are not supported. The following output table should be expected:

```
    ptype  pname    pstart      plength
a:   ff      a    0xffffffff  0xffffffff
b:   6c      b    0x00000001  0x00000800
c:   ff      c    0xffffffff  0xffffffff
d:   ff      d    0xffffffff  0xffffffff
```

Note that the *ptype* output for the *nand0b* is 0x6c; this verifies that the second entry is filled. However, this has no further significance for the user; it is only used to indicate what partitions have been created with the *part-add* command.

Now that the partition has been created, it can be accessed. Returning to the same example in the previous section, executing the following command will now succeed:

```
losh> hd /dev/nand0b 512
```


If successful, output similar to that included below should be seen.

```
0x80079310  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079320  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079330  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079340  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079350  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079360  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079370  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079380  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079390  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800793a0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800793b0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800793c0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800793d0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800793e0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800793f0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079400  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079410  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079420  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079430  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079440  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079450  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079460  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079470  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079480  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079490  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800794a0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800794b0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800794c0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800794d0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800794e0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x800794f0  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
0x80079500  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ..........
losh>
```

You can create up to four partitions on a device, although these partitions cannot overlap.

11.3 Partition Removal from RAM-Partition Table

Partitions can be removed with the *part-rem* command. For example,

```
losh> part-rem /dev/nand0 b
```

will remove the second partition entry from the RAM-partition table. Removing this partition means that the contents will be overwritten with 0s. For example,

```
losh> info part /dev/nand0
```


will output the following table:

```
    ptype  pname    pstart      plength
a:   ff      a    0xffffffff  0xffffffff
b:   0       b    0x00000000  0x00000000
c:   ff      c    0xffffffff  0xffffffff
d:   ff      d    0xffffffff  0xffffffff
```

This table shows that the second partition entry has been removed (it is all 0s). Attempting to access this device will return an error message that the partition does not exist.

12 File Systems

12.1 File System Types

Two file system types are supported in LogicLoader: FAT and YAFFS. YAFFS can only be mounted on NOR and NAND flash devices, while FAT file systems (FATFS) can only be mounted on ATA devices (e.g., CompactFlash cards) and SD cards.

12.2 Mount Command

The general syntax of the *mount* command is:

Mount <filesystem type> <device> <mount point> <flags>

File systems can be mounted on partitions or on a device. Wear-leveling file systems such as YAFFS can achieve greater performance mounting the entire device. FATFS can be mounted on a partition without loss of performance.

When a partition is added (using the *part-add* or *mount* command on a device), that region of memory is marked as protected. Protected areas in LogicLoader are areas of memory that are designated as in use. Whenever erasing a protected area, a warning will be presented on the shell to confirm the action. To display protected areas of memory, see the *LogicLoader v2.5 Command Description Manual* regarding the *info prot* command.

NOTE: LogicLoader will attempt to warn the user when performing actions may result in loss of data or unstable operation; however, LogicLoader will not restrict the user from performing such actions.

Specific examples of the *mount* command will be presented in the sections below.

12.3 Mounting FATFS

LogicLoader is capable of reading and writing FAT16 file systems, while it can only read FAT32 file systems. If the application requires writing to a FAT file system, the file system should be formatted as FAT16.

Use the following command to create a FATFS on the first partition of an ATA device:

```
losh> mount fatfs /dev/ata0a /cf
```

Use the following command to create a FATFS on the first partition of an SD card:

```
losh> mount fatfs /dev/sdmmc0a /sd
```

In the examples above, the FATFS will be read/write. If you would like to create a read-only file system, you have to add a *–ro* flag at the end of the command line. For example, the following command will create a read-only FATFS on the first partition of the ATA device:

```
losh> mount fatfs /dev/ata0a /cf –ro
```


12.4 Mounting YAFFS

12.4.1 Mounting YAFFS on NAND

The following command will create a YAFFS file system on the first partition of a NAND device:

```
losh> mount yaffs /dev/nand0a /yaffs1
```

NOTE: Before executing the *mount* command, the partition should be created first. The entire sequence would look like following:

```
losh> erase /dev/nand0 B0 B2048
losh> part-add /dev/nand0 a 1 2048
losh> mount yaffs /dev/nand0a /yaffs1
```

The *mount* command supports a special case where it can both create a partition and mount the partition within a single command. To do this, specify the device rather than the partition; the *mount* command will perform a part-add on the entire device and then mount it. If partitions already exist on the device, the *mount* command will create a new partition from the last partition to the end of the device. For example:

```
losh> erase /dev/nand0 B0 B2048
losh> mount yaffs /dev/nand0 /yaffs1
```

12.4.2 Mounting YAFFS on NOR

In order to mount YAFFS on a NOR flash device, an emulation layer must be created first. NOR flash devices do not have chunks, so a layer is required that emulates the NAND storage type to make the NOR device look like a NAND device. The emulation layer is created using the *mount* command, as seen below.

```
losh> mount emu /dev/flash0a /femu
```

Just like with NAND, mounting YAFFS on NOR requires a partition to be created first. So the entire sequence of commands to mount YAFFS on a NOR device would look like the following:

```
losh> part-add /dev/flash0 a 5 20
losh> erase /dev/flash0 B0 B20
losh> mount emu /dev/flash0a /femu1
losh> mount yaffs /femu1 /yaffs1
```

12.4.3 Unmounting YAFFS

LogicLoader supports the *unmount* command. However, the *unmount* command in LogicLoader serves no useful purpose, except in the case of unmounting YAFFS. Unmounting YAFFS creates a checkpoint which is written to the file system. A YAFFS checkpoint can greatly decrease the time it takes for future mounts.

YAFFS is a journaling file system. So, when mounting YAFFS, YAFFS must look through the file system history to identify which files are current. On large NAND devices, this may take a long

time. To overcome this delay, the file system state information can be written to the file system. On subsequent mounts, the file system state information is loaded via the checkpoint, rather than by looking through all of the file's histories. It should be noted that whenever the file system is updated in any way, the checkpoint is invalidated.

13 Yet Another Flash File System (YAFFS)

Please be aware that the YAFFS user interface was changed in LogicLoader version 2.4. The legacy user interface is still supported for backwards compatibility, but the new interface—as described in this section—should be used whenever possible.

13.1 YAFFS Overview

YAFFS was developed by a company named Aleph One Limited and incorporated by Logic PD into the LogicLoader software program.

Logic PD selected YAFFS to fill its file system requirements due to the flexible nature of the program, its licensing scheme, and the fact that it is available for Linux, Windows CE, and other operating systems. YAFFS also allows LogicLoader and an OS to view and modify the same partition. It also makes it easier for customers to work with embedded flash technology and perform in-field updates. For example, in Linux it is customary to have the Linux kernel reside in */boot/vmlinux*. So, using the commands below allows LogicLoader to mount, load, and boot the Linux kernel from the partition that is accessible from the Linux kernel:

```
losh> part-add /dev/nand0 a 9 500
losh> mount yaffs /dev/nand0a /nand-root
losh> load elf /nand-root/boot/vmlinux
losh> exec
```

NOTE: The partition entries for YAFFS partitions are not persistent—**they must be restored on each boot**. However, the partitions and data remain persistent.

13.2 Working with YAFFS in LogicLoader

13.2.1 Developing a Partition Scheme

In LogicLoader, YAFFS is mounted on partitions; there can be up to four partitions at a time on a NAND or NOR device.

Partitions are created with the *part-add* command, as in the examples below. (Partition handling is discussed in detail in Section 11 of this document.) Customers should design a partitioning scheme which suits their individual needs; however, for the purpose of providing examples within this document, the following partitioning scheme will be assumed for NOR flash:

- A partition named *boot* which contains a bitmap and OS image and spans the address space below:
  - * start: block 5
  - * length: 6 blocks (384kB) [remember that block sizes vary depending upon device; use the *info mem* command to display proper block size]
- A partition named *data* which contains customer-specific data.
  - * start: block 11
  - * length: 16 blocks (1 MB)

These two partitions are created with the following commands:

```
losh> part-add /dev/flash0 a 5 6
losh> part-add /dev/flash0 b 11 16
```

For the purpose of providing examples within this document, the following partitioning scheme will be assumed for NAND flash:

- A partition named *boot* which contains a bitmap and operating system image and spans the block range below:
  - * start: block 10
  - * length: 256 blocks (8 MB, assuming 16kB block size)
- A partition named *data* which contains customer specific data.
  - * start: block 266 (abuts boot partition)
  - * length: 128 blocks (4 MB, assuming 16kB block size)

These two partitions are created with the following commands:

```
losh> part-add /dev/nand0 a 10 256
losh> part-add /dev/nand0 b 266 128
```

13.2.2 Formatting YAFFS Partitions

All file systems need to be formatted before they can be mounted. Because YAFFS was designed from the ground up to work with embedded flash technologies, it understands an erased flash device to be both formatted and empty. To prepare your partition for mounting, use LogicLoader's *erase* command to erase the area of flash where the partition is to be located.

Using the example partition scheme in Section 13.2.1 above, the partitions could be prepared for initial use by erasing the regions of the flash device spanned by them.

For a NOR example

```
losh> erase /dev/flash0 B5 B6
losh> erase /dev/flash0 B11 B16
```

For a NAND example:

```
losh> erase /dev/nand0 B10 B256
losh> erase /dev/nand0 B266 B128
```

WARNING: Erasing flash blocks that will be used for YAFFS partitions will erase everything in those areas of flash. It is not required to format the partition every time the device is rebooted; the partition should only be formatted when an entirely new YAFFS partition is created or when the data on a stored partition needs to be completely erased. For NAND-based devices, the first few blocks of NAND (the actual number of blocks is dependent on the NAND device) are used to hold the */lboot* partition which is where LogicLoader resides. Modifying data in this partition can cause the board to fail to boot.

13.2.3 Mounting the Partition

To mount a partition, the *mount* command is used, as discussed in Section 12.4. This command takes the following arguments:

- <filesystem type> - the type of file system being mounted ('yaffs' here)
- <device> - the device on which the YAFFS partition is mounted
- <mountpoint> - the name of the YAFFS partition

For example, to mount YAFFS on NAND:

```
losh> mount yaffs /dev/nand0a /boot
losh> mount yaffs /dev/nand0b /data
```

NOTE: As previously discussed, you cannot mount YAFFS directly on NOR flash devices. First, you have to mount an emulation layer on top of the NOR flash device, then mount YAFFS on the emulation layer.

For example:

```
losh> mount emu /dev/flash0a /femu1
losh> mount emu /dev/flash0b /femu2

losh> mount yaffs /femu1 /boot
losh> mount yaffs /femu2 /data
```

13.2.4 Accessing YAFFS Partitions in an OS

A key advantage of the read/write YAFFS file system capability at the LogicLoader level is the ability to share data stored in the file system with an OS environment. If an OS environment (e.g., Linux, Windows CE, VxWorks) implements YAFFS as an OS-accessible file-system, any files available to LogicLoader are also available to the OS and vice-versa.

This contributes to significant benefits in the areas of system software upgrades (including OS upgrades), splash screen changes, script modifications, and other boot-time data that may need to be updated.

13.3 Summary

To use the YAFFS file system within LogicLoader, follow these steps:

1. Format the partitions by erasing the associated flash blocks.
2. Decide on a partitioning scheme and create partitions.
3. Mount the partitions using the *mount* command (first mount an emulation layer for NOR flash devices).

Steps 2 and 3 must be repeated every time the system is booted. If the YAFFS partitions are frequently accessed, consider implementing Steps 2 and 3 via a boot script. Step 1 only needs to be performed when creating a brand new partition or when the contents of an existing partition need to be completely erased.

NOTE: A partition is persistent. Re-adding a partition at boot time restores access to previously-saved data. Flash blocks must be erased to permanently remove a partition; otherwise, it can be recovered across boots.

Keep in mind the following when working with YAFFS and LogicLoader:

- Ensure partitions do not overlap each other or LogicLoader.
- Ensure that a partition is erased before it is mounted for the first time.

NOTE: The legacy YAFFS mounting scheme is still supported for backwards compatibility, but its use is discouraged.

Appendix A: Setup File Keys

Key	Function	Parameter 1	Parameter 2	Parameter 3	Parameter 4	Parameter 5
1	Write Byte	Address	Value	-	-	-
2	Write Word	Address	Value	-	-	-
3	Write Long	Address	Value	-	-	-
4	Disable Pin	Pin ID	-	-	-	-
5	-	-	-	-	-	-

Appendix B: LwIP License Agreement

LogicLoader uses the open source LwIP stack for networking support. The LwIP license requires the inclusion of the following license to satisfy Condition 2 below:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This file is part of the lwIP TCP/IP stack.

Author: Adam Dunkels <adam@sics.se>