LogicLoader™ User's Manual

User's Manual for Logic's SOM Development Kits (LogicLoader Version 2.3)

Logic Product Development
Published: January 2003
Last Revised: July 2007

REVISION HISTORY

REV	EDITOR	REVISION DESCRIPTION	LoLo Ver.	APPROVAL	DATE
A	Bill O'Donnell	Release	--	BOD	01/21/03
B	Bruce Rovner	Release	--	BR	05/05/03
C	Bruce Rovner	Release	--	BR	07/24/03
D	Hans Rempel	Edit and Section 6 Update; LoLo Ver. 1.2.0	--	HAR	09/15/03
E	James Wicks	Document Format and Edit	--	JAW	09/30/03
F	Bruce Rovner	Preliminary Release LogicLoader Version 1.4	--	BR	03/31/04
G	Bruce Rovner	Document Edit/ Test Commands Update LogicLoader Version 1.5/ MOT Pilot Release	--	JAW	06/08/04
H	Mike Aanenson	Added SOM to Acronyms; Added: mem-copy	--	JMC	08/30/04
I	James Wicks	Attached SOM Cover page	--	JAW	10/14/04
J	Robin Bhattacharyya, Bruce Rovner, Michael Erickson, Hans Rempel	Updated for LogicLoader 2.0 release; Command list is now in separate document	--	ME	06/03/05
K	Bruce Rovner	Section 5 "Program Loading": Added recommendation to user not to erase LogicLoader's RAM space until boot string has been recovered; Added description of exec command boot string structure	--	HAR	08/22/05
L	Jed Anderson, Eric Nelson	Sections 6.3.3 and 8.1.2: Fixed incorrect section numbers within the text referring to 'Section 0' to 'Section 7' and 'Section 5' respectively. Created Section 4 to support NAND secondary flash and booting from NAND. These features have been added to LoLo 2.2.0.	2.2.0	MT	07/12/06
M	Hans Rempel, Peter Barada, Jed Anderson	- Updated for LogicLoader 2.3.0 release. - Section 5.1: Added section describing TFTP usage. - Added Note to flat memory addressing subsection. - Section 5.3.2: Updated 'exec' command for use with an ARM Linux kernel. - Section 6: Scripting information – added '\n' requirement to end parser. - Section 6.3.1: Updated information about scripts used with the 'echo' command being stored in the serial EEPROM. - Added Section 6.4 "Settings that Affect Scripts" - Section 6.6.1.2: Added further explanatory information - Section 6.6.1.4: Added explanation of what happens when a new variable is referenced. - Section 6.6.1.6: Added info on 'if' and 'while' statements - Section 6.6.1.8: Added more operators supported by the shell, added section no Immediate expression evaluation - Added Section 6.6.1.9 "Escaping the variable character" - Section 9.1: Added Linux example - Section 9.2: Added information for NAND - General formatting and grammatical changes	2.3.0	MT	01/16/07
N	Jed Anderson	- Section 4.5.5: Corrected description of 'erase' command to indicate the second argument is "number of blocks" not "end block"	2.3.0	JCA	04/13/07
O	Jed Anderson	- Section 4.5.3: Corrected example in 'burn' command to include a B to specify the appropriate block location for the burn - General formatting and grammatical changes throughout	2.3.0	RGL	07/25/07

Table of Contents

Table of Figures and Tables

PRODUCT BRIEF:
Logic embedded product solutions

LOGICLOADER™ Bootloader/Monitor

LogicLoader™ is a bootloader/monitor program developed by Logic Product Development 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, SH, 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.

Product Features

Operating System (OS) Bootstrap

+ Load multiple OSes (Microsoft Windows Embedded CE, Linux, etc.)
+ Load an OS from 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

LOGICLOADER™ :: HIGHLIGHTS:

+ Ships standard on all Logic System on Modules
+ Conditional scripting
+ YAFFS (flash file system)

LOGIC WEBSITE :: DESIGN RESOURCES:

+ Logic Technical Support : http://www.logicpd.com/support/
+ Technical Discussion Group : http://www.logicpd.com/support/tdg/
+ Frequently Asked Questions (FAQ) : http://www.logicpd.com/support/faq/
+ For more information contact Logic Sales : product.sales@logicpd.com

LOGIC
embedded product solutions
411 N. Washington Ave. Suite 400
Minneapolis, MN 55401
T : 612.672.9495 F : 612.672.9489
I : www.logicpd.com

1.2 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
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

1.3 Technical Specifications

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

- SOM IO Controller Specification
- SOM Hardware Specification
- Applicable Processor Manual

1.4 LogicLoader Command Description Manual

For a complete description of LogicLoader's 'losh' commands, please see the *LogicLoader Command Description Manual* available from Logic's downloads page http://www.logicpd.com/auth/. The *LogicLoader Command Description Manual* explains how to use each LogicLoader command.

1.5 LogicLoader Addendums

Logic has written a SOM-specific addendum for each SOM that runs LogicLoader. LogicLoader Addendums are located under the "User Manuals" heading on Logic's Registered Products downloads page.

2 LogicLoader (LoLo)

2.1 LogicLoader Overview

The LogicLoader (LoLo) is a bootloader/firmware-monitor program developed by Logic Product Development. 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 operating systems rely on an underlying bootloader to initialize a device from its reset condition. In general, operating systems are designed with the assumption that the system will be in a specific pre-defined state before the operating system 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 Product Development's SOM platforms and prepares them for use by an operating system.

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 giving the system the ability to load system software from flash memory, a CompactFlash storage card, a Local Area Network, or even from a device attached to the system's serial port. LogicLoader also has the ability to upgrade an existing operating system 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. Included with LoLo is the LogicLoader shell, also known as "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. For example, formatted data in arbitrary memory locations can be read from, and written to, by using the 'x' and 'w' commands. Other commands run specific tests designed to verify Logic's SOM hardware platforms. All commands return a value to the command line that can be used to conditionally evaluate the command result. Refer to the *LogicLoader Command Description Manual* for a complete description of available commands.

Developers may code their own test programs using the provided GNU development tool chain and use the 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 operating system images.

2.4 Manufacturing Advantages with LoLo

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. Here is an example scenario: 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 for more information on using LogicLoader to streamline manufacturing.

3 The LogicLoader Shell (losh)

3.1 Losh 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 Losh 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 (i.e., 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) by default also refers to the SOM's debug serial port.

The LogicLoader Shell 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 system's various peripherals and their associated device drivers.

3.2.1 Using Losh

The losh shell includes a basic command line editing feature and a command history feature. This provides users with a quick way to repeat commands. Using the up and down arrow keys, the user can scroll through the list of previously executed commands. When a desired command is displayed, press the return key to repeat the command. The right and left arrow keys allow a user to position the cursor as desired on the current line so that text can be modified, deleted, or inserted at the appropriate location without having to "backspace" the entire line to access the portion of the command or command set being entered.

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 the LogicLoader shell.

Commands may be run in the background by adding a '&' 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 below:

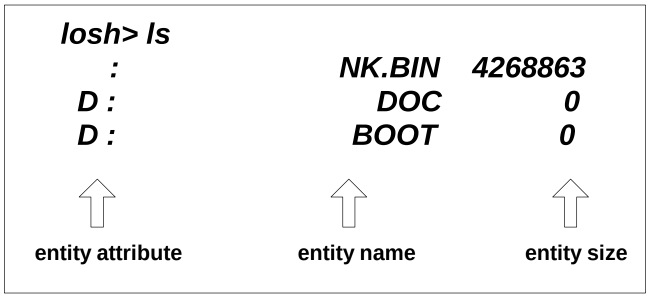
```
losh> ls

   :           NK.BIN   4268863

  D :             DOC         0

  D :            BOOT         0
```

In this example, the columns displayed when the 'ls' command is executed are (in order from left to right): entity attribute, entity name, and entity size. See Figure 3.1, below.



```
losh> ls
    :              NK.BIN   4268863
  D :                 DOC         0
  D :                BOOT         0
```

Figure 3.1: 'ls' Command Columns

The first column, entity attribute, can be blank, "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 reserved bits are set, and an "H" indicates a hidden attribute.

The second column, entity name, is simply the name of the entity as 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 NAND Devices and LogicLoader

LoLo supports NOR and NAND type flash devices. 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. LoLo 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. As of the time of writing this document, there is no common algorithm used to read or write to NAND flash devices; every manufacturer requires a unique algorithm.

4.1 Block 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 block and page.

NAND devices currently come in two flavors 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 type 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. LoLo dedicates a portion of the NAND spare area to YAFFS.

4.2 Bad Blocks

NAND devices can have bad blocks when shipped from the manufacturer, as well as bad blocks that develop over time. Most NAND blocks can be erased and rewritten on the order of 100,000 cycles before potentially going bad. 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 manufacturers state that the device integrity decays with every erase/program cycle. However, some third-party studies indicate that data integrity may decay with a large number of read cycles as well. LoLo 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.

4.3 NAND Programming

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

4.3.1 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 simply written contiguously on the device from low numbered blocks to higher numbered blocks, while skipping any bad blocks (as 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.

4.3.2 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.
- 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.

More information regarding how YAFFS operates in LoLo can be found in the "YAFFS" Section of this document.

4.4 NAND Boot Process

With conventional NOR flash, a typical boot process consists of an XIP boot loader that configures the system, loads the OS, and starts the OS. When using NAND flash, the only memory available after reset is the first page or sector (depending on the NAND controller and NAND device). A boot loader typically requires more code space than what is available in one page or sector. Because of this, the boot process requires an initial boot loader to load the main boot loader. In other words, booting from NAND requires one or more boot loader phases (see Figure 4.1).

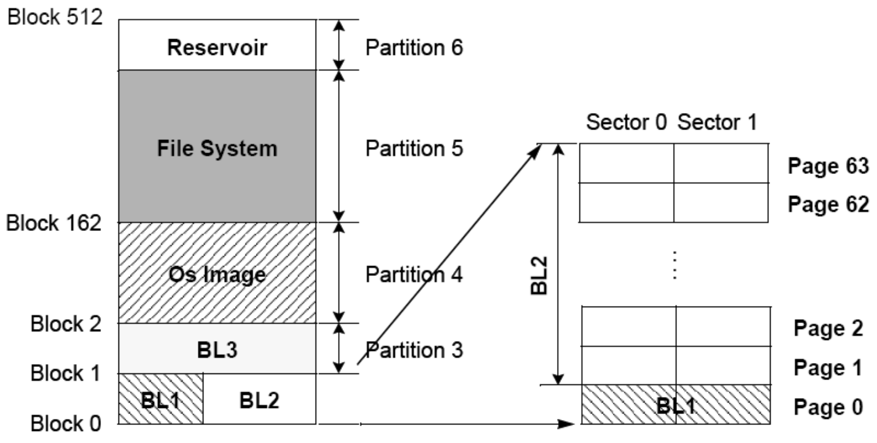


Figure 4.1: NAND Boot Process

4.4.1 NoLo

The job of NoLo (NAND Loader) is to load and execute LoLo. NoLo always resides at block zero, and uses no more than one block of the NAND device. Block zero is guaranteed by the NAND device manufacturer to be a good block for the life of the device.

4.4.2 BL1

Typically out of reset, the NAND controller will copy the first sector out of the first block and page of the NAND device to internal CPU SRAM. This is the “BL1” phase in Figure 4.1. The CPU program counter is then placed at the start of this SRAM memory space. This starts execution of NoLo. NoLo will begin by copying the remainder of the NAND flash block to internal CPU SRAM. When the entire NAND block is in SRAM, NoLo will continue to the “BL2” phase.

4.4.3 BL2

From this phase there is enough XIP memory to setup chip selects, SDRAM, clocks, and load LoLo. To locate LoLo, NoLo will scan the YAFFS boot partition for a file called “lboot.elf”. If this file is not found, NoLo will assume the NAND device has been programmed with a skip bad block algorithm.

4.4.4 Loading from the YAFFS Boot Partition

The YAFFS boot partition will reside in the NAND flash device starting at block one. The boot partition is named “/lboot” in the file system. If a file named “lboot.elf” is found in the partition, that file is loaded into SDRAM and executed. The “lboot.elf” file does not necessarily have to be LoLo. Any file can be placed in the boot partition and named “lboot.elf”. For the sake of this document, however, we will assume LoLo is in the boot partition. Any files not named “lboot.elf”, are ignored.

4.4.5 Loading with the Skip Bad Block Algorithm

If no valid files are found in a YAFFS boot partition, NoLo will use the skip bad block algorithm (see Section 4.3.1) to begin loading a 512kB image starting at 1 MB plus 1 block offset of the NAND device. This location would be just after the YAFFS boot partition had a boot partition existed (see Figure 4.2). NoLo will then use the start address of the image as the entry point into the program. This is the case when the SOM has been programmed in production.

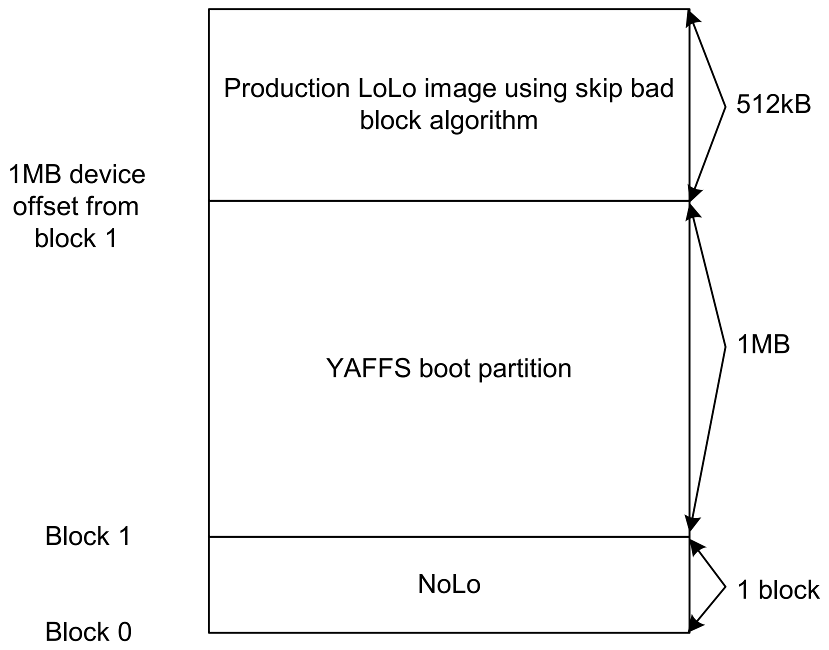

Production LoLo image using skip bad block algorithm
512kB
1MB device offset from block 1
YAFFS boot partition
1MB
Block 1
NoLo
1 block
Block 0

Figure 4.2: Location of LoLo Using the Skip Bad Block Algorithm

4.4.6 Error Handling

NoLo currently does not have a shell, so error handling is accomplished using the two status LEDs on the baseboard. When NoLo is loading LoLo from NAND, the status 1 and 2 LEDs will alternately flash. If NoLo has a problem booting the system, one of the LED sequences below will indicate the specific failure:

Status 1 LED state	**Status 2 LED state**	**Problem**
Flashing	Off	NoLo could not find a bootable file.
Off	Flashing	NoLo found a bootable file, but could not complete loading the file.
Flash in unison with status 2 LED	Flash in unison with status 1 LED	NoLo could not initialize the NAND device.
Flash alternating with status 2 LED	Flash alternating with status 1 LED	NoLo is loading the bootable file normally.

Table 4.1: Status LED Indicators

Note: LoLo also uses the status 1 and 2 LEDs. NoLo's use of the status LEDs will differ by speed or by pattern. For example, on the PXA270-10, NoLo will alternately flash status 1 and 2 LEDs; LoLo will also alternately flash status 1 and status 2 LEDs under normal operation. However, LoLo will flash the LEDs noticeably slower than how NoLo will flash the LEDs.

4.4.7 LoLo

A RAM version of LoLo resides in the YAFFS boot partition. LoLo is considered to be "BL3" in Figure 4.1. When LoLo starts, LoLo will check if it has been loaded from a YAFFS boot partition. If so, it will check to see if the device has been previously scanned for any manufacturer marked bad blocks.

4.4.8 Manufacturer Bad Block Scan

The NAND device manufacturer will identify blocks that are not reliable. The manufacturer will then mark these blocks as bad by writing certain values at certain locations within the block. If one of these bad blocks is ever erased, there is no way to recover the bad block information. So LoLo and YAFFS will track this bad block information and avoid using or erasing any blocks that are bad.

Unfortunately, not every manufacturer marks bad blocks the same way and YAFFS may also use a different method to indicate bad blocks. For this reason, at startup LoLo will scan for bad blocks as marked by the manufacturer and mark those blocks as bad using the YAFFS bad block marker. This only needs to occur once in the lifetime of the SOM. Once a NAND device has been scanned and marked with the YAFFS bad block marker, LoLo will never scan the NAND device again. To indicate that a NAND device has been scanned, LoLo will mark block zero as bad. Marking block zero as bad does not mean that block zero is actually bad. In fact, block zero is guaranteed to be good for the life of the NAND device by the manufacturer. This mark merely indicates the device has been scanned and re-marked with the YAFFS bad block markings. This block zero mark will be used the next time LoLo starts to indicate that LoLo does not need to scan the NAND device again.

4.4.9 YAFFS Boot Partition

After LoLo has verified the bad block scanning has been performed, it will automatically mount the boot partition of the NAND device. The boot partition always begins at block one and will occupy no less than 1 MB of space on the NAND device. Since LoLo is approximately 256kB, the 1 MB partition will allow room for LoLo to grow, as well as offer space for other LoLo system files, such as a config block.

If LoLo finds that the boot partition does not contain an “lboot.elf” file, it will assume that it has been booted from a production image. LoLo will then copy the first 512kB after the boot partition (see Figure 4.2) to a file named “lboot.elf” in the boot partition. This new “lboot.elf” file will be used to boot from in the future.

Note: The “lboot.elf” file may not actually require the full 512kB. LoLo copies this many bytes to accommodate any future growth of LoLo. When a user updates LoLo with the ‘update’ command, the actual file size of LoLo will be reflected in the directory.

Keep in mind that all the files in the boot partition can be manipulated just like any other YAFFS partition. Files can be copied, removed, and renamed. This gives the user unique abilities such as switching between two versions of LoLo in the boot partition by simply renaming one of the LoLo files to “lboot.elf”. Or a user could switch config blocks by copying any one of a list of config files from a user partition to the boot partition.

4.5 Losh and NAND

The LogicLoader shell supports NAND by extending functionality to some of the existing Losh commands. This section outlines areas that are unique to NAND. More detailed information for each command can be found in the *LogicLoader Command Description Manual*.

4.5.1 NAND Addressing

The NAND device uses a block, page, and sector to identify memory locations. In contrast, the NOR flash device uses a flat memory-mapped address. Losh will support both schemes when addressing the NAND device. For a flat memory address, losh takes the address as an offset into the device and calculates the block, page, and sector from that address. Losh will then use this calculated block, page, and sector internally when communicating to the NAND device. This scheme gives the user the ability to use the same losh commands with the same arguments on any SOM with any NAND device. Without this ability, the losh command line arguments would need to change according to the specific NAND device installed, because each NAND device has different block, page, and sector sizes.

4.5.1.1 Using Flat Memory Map Addressing

To use the flat memory map reference, use a 32-bit address just as if the NAND device were RAM or NOR flash. For example:

```
losh> add-yaffs nandy /dev/nand0 0x120000 0x3E80000
```

When considering the PXA270-10 NAND Card Engine, there are 0x20000 bytes per block, so the example above creates a YAFFS partition from block 9 (start = 9 x 0x20000 = 0x120000) to block 509 (length = 500 x 0x20000 = 0x3E80000).

Note: On some boards, such as the i.MX31-10 SOM-LV, flat memory addressing to access NAND will fail because the NAND device is accessible only through its NAND controller. In these situations Block References are required.

4.5.1.2 Using Block Reference

To directly reference a block on the NAND device, use a “B” in front of the block number. For example:

```
losh> add-yaffs nandy /dev/nand0 B9 B500
```

Similar to the example in 4.5.1.1, this creates a YAFFS partition starting at block 9 that is 500 blocks in size. The “B” in front of the 9 and the 500 indicate the 9 and 500 are block values.

4.5.2 add-yaffs

The ‘add-yaffs’ command is extended to support NAND by specifying the device type (/dev/nandx), a start block argument, and a total number of blocks argument. For example:

```
add-yaffs foo nand B9 B503
```

This example will create a YAFFS partition named “foo” in the first NAND flash device starting at block 9 and spanning 503 blocks.

4.5.3 burn

The ‘burn’ command has been extended for use with NAND devices. The ‘burn’ command will burn the NAND device using the skip bad block algorithm (see Section 4.3.1). In this case the ‘burn’ command will require a device name and start block number. The data to be burned is data that has previously been loaded with the ‘load’ command. For example:

```
burn /dev/nand0 B9
```

4.5.4 config

The ‘config’ command behaves nearly the same as for a SOM with NOR flash. The command line arguments are the same. The difference is, rather than creating a memory-mapped config block in the NOR flash device, an “lboot.cfg” file is created in the YAFFS boot partition.

4.5.5 erase

The 'erase' command has been expanded to include erasing NAND blocks. Care must be taken when erasing NAND blocks so as not to erase a YAFFS partition or NoLo in block zero. Any attempt to do so will require confirmation for erasing to continue. When using the 'erase' command with a NAND device, the arguments required are: start block, number of blocks to erase, and device name. Blocks with bad block markers will not be erased. For example:

```
erase B9 B503 /dev/nand0
```

There is an optional argument "force". The force argument will force the 'erase' command to erase all blocks in the specified address range regardless 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 or by YAFFS, and the block is erased with the "force" argument, there is no way to ever recover the bad block information. For example:

```
erase B9 B20 /dev/nand0 force
```

4.5.6 info

The 'info' command has expanded to include information regarding the NAND device and YAFFS boot partition; specifically using the 'info mem' and 'info YAFFS' arguments. The 'info mem' command now includes geometry data for all NAND flash devices. The geometry information includes:

- Base address of NAND
- Number of blocks
- Number of pages
- Number of sectors
- Sector size
- The overall NAND device byte size
- Bad block list

The 'info yaffs' command has been expanded to include YAFFS partition table information.

4.5.7 update

There command line arguments for the 'update' command remain the same. However, the 'update' command has been modified to support updating NoLo and updating LoLo in the YAFFS partition. When either of these update files are sent to the SOM via the 'update' command, LoLo will identify the update as NoLo or LoLo, and then program the NAND part as needed.

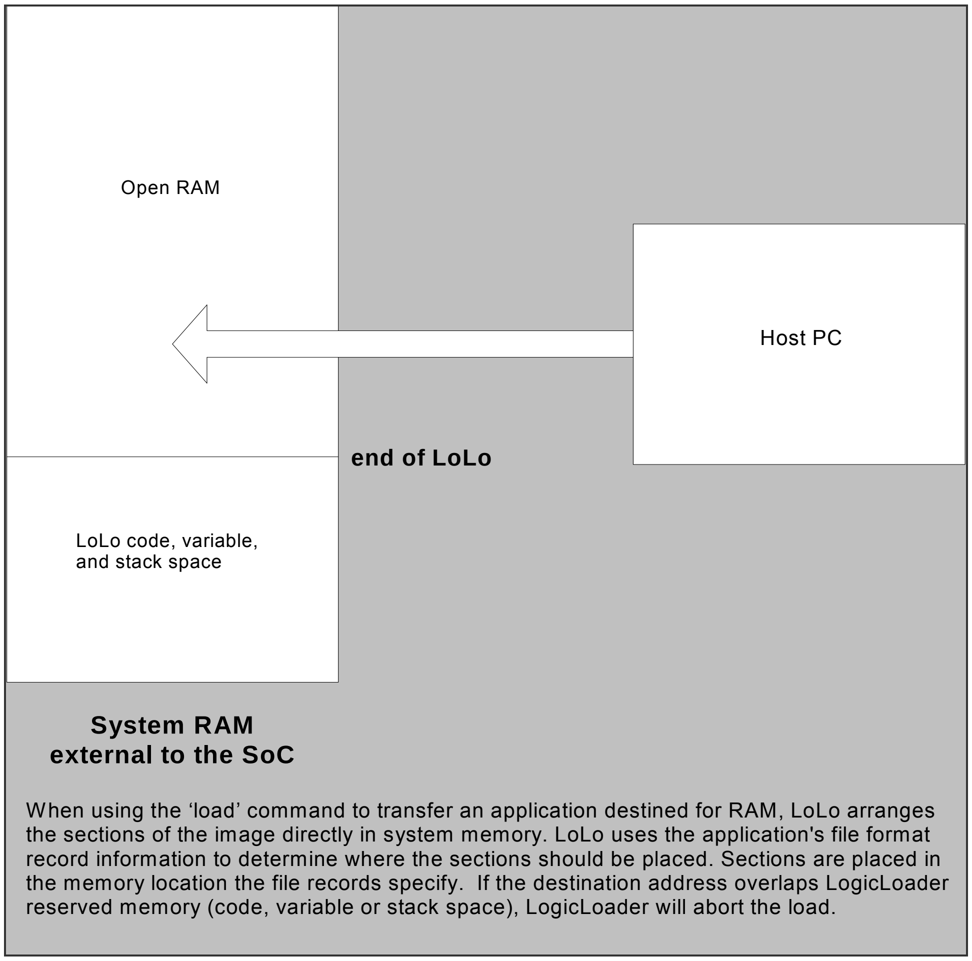
5 Program Loading

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.

5.1 Understanding the 'load' Command

The purpose of the 'load' command is to transfer an executable 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, 'load' 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.

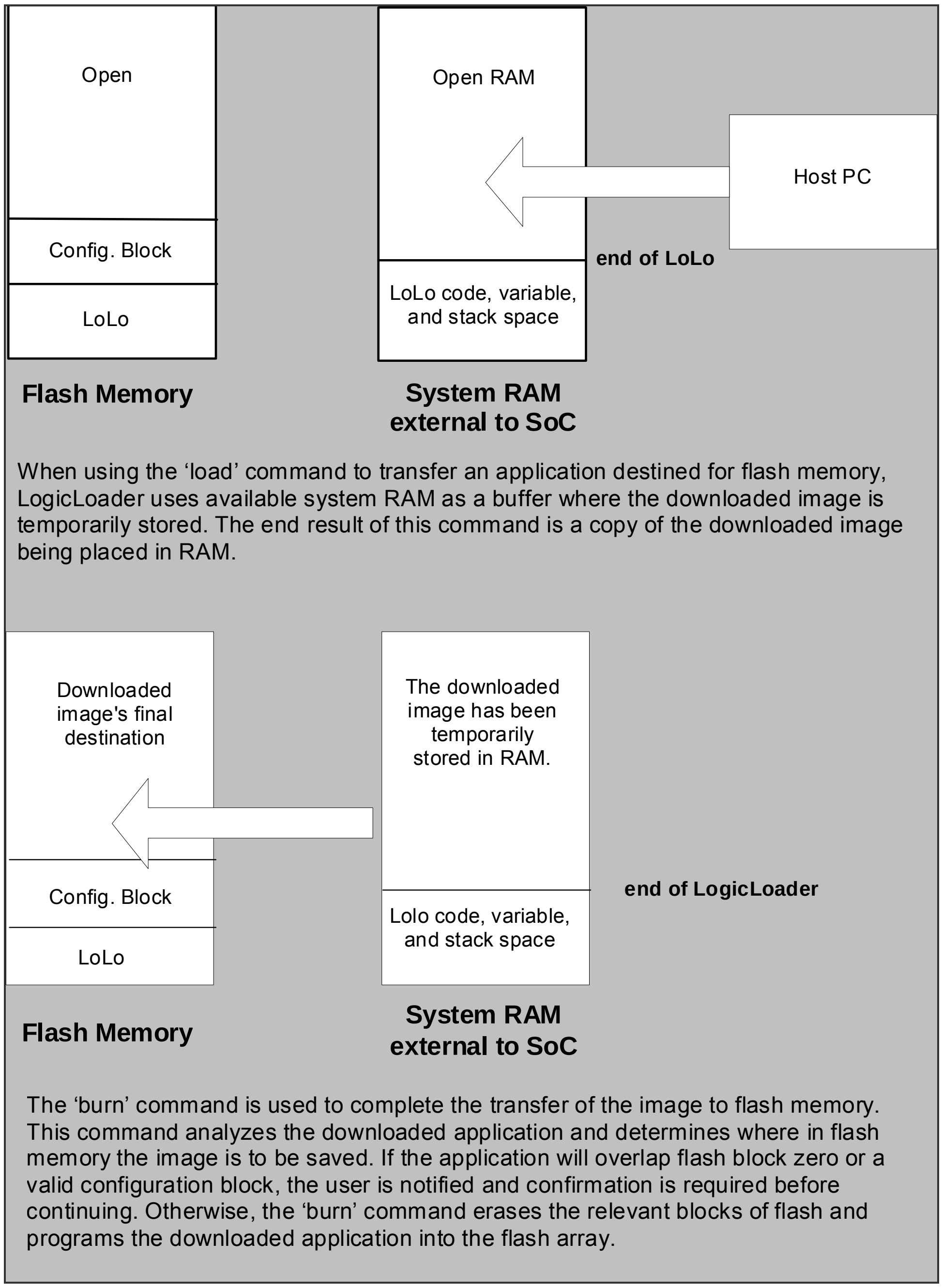
If an image is destined for system RAM or on-chip SRAM, the 'load' command stores the image directly to its run-time location. Refer to Figure 5.1 for a graphic representation of this process.


Open RAM
Host PC
end of LoLo
LoLo code, variable, and stack space
System RAM external to the SoC

When using the 'load' command to transfer an application destined for RAM, LoLo arranges the sections of the image directly in system memory. LoLo uses the application's file format record information to determine where the sections should be placed. Sections are placed in the memory location the file records specify. If the destination address overlaps LogicLoader reserved memory (code, variable or stack space), LogicLoader will abort the load.

Figure 5.1: 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 5.2 for a graphic representation of this process.


Open
Config. Block
LoLo
Flash Memory
Open RAM
Host PC
end of LoLo
LoLo code, variable, and stack space
System RAM external to SoC
Downloaded image's final destination
The downloaded image has been temporarily stored in RAM.
end of LogicLoader
Config. Block
LoLo
Lolo code, variable, and stack space
Flash Memory
System RAM external to SoC

When using the 'load' command to transfer an application destined for flash memory, LogicLoader uses available system RAM as a buffer where the downloaded image is temporarily stored. The end result of this command is a copy of the downloaded image being placed in RAM.

The 'burn' command is used to complete the transfer of the image to flash memory. This command analyzes the downloaded application and determines where in flash memory the image is to be saved. If the application will overlap flash block zero or a valid configuration block, the user is notified and confirmation is required before continuing. Otherwise, the 'burn' command erases the relevant blocks of flash and programs the downloaded application into the flash array.

Figure 5.2: Downloading to Flash

5.1.1 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:data-file
```

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
```

5.2 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 5.2 for more information.

5.3 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 operating system to setup the runtime environment. The 'exec' command is more useful for launching an application such as an operating system that will take over total control of the hardware and the environment. The differences between the 'jump' and 'exec' command are that 'exec' can pass a command line argument to the program being executed and that 'exec' disables interrupts, the cache, and the MMU (if present).

5.3.1 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, rather the program inherits LogicLoader's current environment. It is the software engineer's responsibility to ensure that the hardware is setup in the desired manner.

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

```
int my_jump_function(void);
```

5.3.2 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:

```
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 get 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.

5.3.3 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 anytime but 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:

1. `losh> load elf`
   This transfers the image to the device.
2. `losh> burn`
   This programs the image into flash at the destination address stored by the 'load' command.
3. `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:

1. `losh> load elf`
   This transfers the image to the device.
2. `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 available as well.

5.4 Understanding the 'update' Command

Logic 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 this way: only one command implements the entire update process without any user interaction or confirmation.

6 Scripting

6.1 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.

6.1.1 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).

6.2 Launching Scripts

The process of launching a script manually or post-boot time employs 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 reference the *LogicLoader 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, with their ability to be automatically run at startup differentiating them from normal scripts. One 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 simply '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: a command sequence initializes the Ethernet interface, downloads a Windows CE OS image, and then launches the OS image with a specific command line.

6.3 Persistent Script Storage

In order for a script to persist across power cycles, the script must be stored to 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 on each system. The primary storage mechanisms supported by LogicLoader are the serial EEPROM, the resident flash array (dev/config or YAFFS), and the CompactFlash 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's Manual Addendum* document for specific persistent storage interface support.

6.3.1 Persisting Scripts with the Echo Command

The 'echo' command can be used to store a script in the serial EEPROM or /dev/config. 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 or /dev/config are not stored as actual files, it is important that any previous information in the serial EEPROM or /dev/config is not interpreted as part of the script. Check the contents of the serial EEPROM or /dev/config with 'cat', or 'hd' to verify that the contents are as desired. If not, the 'erase' command should be used to erase any previous information before the 'echo' command is executed.

6.3.2 Serial EEPROM Scripts

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 fairly short and may re-direct to a secondary script on an interface capable of a 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:

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

6.3.3 Configuration Block Scripts

The system's configuration block is another persistent storage area that supports the storage and execution of scripts. The configuration block is located in system flash memory and is the secondary boot-time script storage location on systems with serial EEPROM.

Store a script to the configuration block interface (/dev/config) by using the 'echo' command or the 'config S' command. Here is an example of using the 'echo' command to store a script to the configuration block:

```
echo "LOLOmount fatfs /cf; source /cf/B.BAT; exit;\n" /dev/config
```

Note: The configuration block must be initialized before using it for scripting commands. For more information on how to create and use the configuration block, please see Section 8.

6.4 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 following flags 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 don't execute; ignored by interactive shells;
- q Don't print LoLo error messages;
- u Exit on expansion of unset variables;
- v Echo input lines as they are read;
- w Print warnings for possible errors;
- x Echo all user commands before executing them.

6.5 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.

6.5.1 Boot-time Script Guidelines

All of the commands available in LogicLoader are also available to boot-time scripts. As in normal scripts, a semi-colon must be used to separate commands and the exit command must be used to terminate a boot-time script.

In order for a script to be boot-capable, the script must be stored in a boot-capable location and must contain the necessary "magic" string prefix. A boot-time script may reside either in the on-board serial EEPROM or in the flash-based configuration block. The order of boot-time execution is first the EEPROM, then the configuration block. In order to differentiate between auto-booting scripts and non auto-booting, LogicLoader checks the first four bytes of the boot-capable devices to see if they contain a "magic" string indicating that the following script should run automatically.

6.5.2 Boot Script Magic Strings

The "magic" string for LogicLoader is "LOLO" for silent execution or "VOLO" for verbose script execution. If the string "LOLO" prefixes the boot script, the script's commands and terminal output will be completely silent. By using "LOLO" as the prefix, it is possible to fully boot the system without ever sending any information out the debug serial port. If "VOLO" is used as the boot script prefix, the boot script commands, return codes, and other "normal" information is displayed via the serial port as if the script was running post-boot time.

6.5.3 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 does pause for one/half of a second to read from debug serial port to determine if an abort request is being made. Some of Logic's SOM products implement an external mode line that allows LogicLoader to ignore the assertion of the 'q' key – thereby skipping the one/half second wait time and decreasing the overall boot time of the system when a boot script is desired. For more information on the hardware line that provides this functionality, check the *LogicLoader User's Manual Addendum* for your hardware

6.5.4 Understanding the Echo Command

The 'echo' command can be used to store a script in the serial EEPROM or /dev/config. 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 or /dev/config before using the 'echo' command.

6.5.5 Boot-time Script Example

The following example creates a simple LoLo boot script that first mounts the CompactFlash card and then runs a second script "B.BAT" on the CompactFlash card that automates software.

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

or

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

6.6 Conditional Scripting and Variables

6.6.1 Variables

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

6.6.1.1 Variable Names

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

6.6.1.2 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.

6.6.1.3 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.

6.6.1.4 De-referencing a Variable

To dereference 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. This 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.

6.6.1.5 Built-in Variables

The shell contains two built-in variables, '?' and '@'.

The '?' variable is assigned to the return value of the last command executed. By convention, all shell commands return zero to indicate that it completed successfully and 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 /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 Command Description Manual* for specific command descriptions in order to learn which commands set the '@' variable, and if so, the usage of these commands.

6.6.1.6 Conditional Scripting

LogicLoader's shell 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
```

6.6.1.7 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.
```

6.6.1.8 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 can not 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 /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.

6.6.1.9 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:

```
echo “if ($a == 2) source bar;\n” /dev/config
```

needs to be written as

```
echo “if (\$a == 2) source bar;\n” /dev/config
```

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.

6.6.1.10 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.

6.6.1.11 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'

7 Video Interface

7.1 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)

7.2 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
```

If using the configuration block, up to eight uniquely named custom screen settings can be saved. The stored settings include the frame buffer address so that the frame buffer will be initialized to a user specified address upon executing the 'video-open' or 'video-init' command. The current frame buffer address can be ascertained by issuing the 'video-fb' command.

8 Configuration Block

8.1 Configuration Block Overview

Logic has added an optional configuration block that is located in the first 64K of the second 256K block of flash, immediately following the location of LogicLoader. The purpose of the configuration block is to allow our customers the ability to store larger scripts, change the baud rate of the debug serial port, store default settings for the Ethernet, and save custom LCD controller settings. The script area accommodates 16Kbytes of script storage space, and the peripheral settings allow for up to eight different settings each to be saved for the video, Ethernet, and serial. Custom settings can be downloaded and saved, or pre-programmed at the factory.

The configuration block is designed to be easily accessible from a customer’s application. The structure and field definition header files are available from Logic. Each section has its own checksum, and there is also a checksum and version id for the entire configuration block. It is possible to store additional customer-defined settings to the configuration block for later use by customer software. In order to do this, a custom configuration block must be created. Please contact Logic for more information.

The configuration block is optional for most SOMs. Normal operation, with the default settings, is available to customers who do not wish to use the configuration block.

8.1.1 Initializing

The configuration block must be initialized on systems that have never implemented a configuration block. If, for some reason, the system’s configuration block needs to be cleared, this initialization step will also provide for that.

Enter 'config CREATE' at the losh prompt to initialize (or re-initialize) the configuration block. The configuration block will be located in the 64K of flash immediately following the LogicLoader flash storage location.

8.1.2 Scripting

The default mode of the configuration block is scripting. Once the configuration block has been initialized, users can ‘cat’ and ‘source’ /dev/config in order to view or execute their script. The ‘echo’ command may be used to create the script, or larger scripts can be downloaded and saved using the ‘config’ command with the ‘S’ option. The scripts in the configuration block can be boot scripts or ordinary scripts. For more information on scripting, please refer to Section 6.

8.1.3 Video

A customer may save up to eight custom LCD screen settings, including the frame buffer location, by using the ‘config’ command with the ‘V’ option. In order to store a custom setting, set up the controller as desired and then enter ‘config V screen_name X Y’ to store it (where X and Y are dimensions of the screen—e.g., 640 and 480). Once this configuration step is implemented, the system can use the command sequence ‘video-open screen_name depth” to use the new configuration.

8.1.4 Serial with the Configuration Block

A customer can change the baud rate of the debug port with the ‘B’ option. The setting will be stored under the debug serial port’s UART index, and will automatically be used after the next reset. Only valid settings will be allowed.

8.1.5 Ethernet

A customer may save a default Ethernet setting that consists of a MAC address, and IP address, a subnet mask, and a gateway.

To accomplish this, set up the controller using the 'ifmac' and the 'ifconfig' commands, and then use the 'config' command 'E' option with an index (e.g., 'config E 0') to save the settings. The index used must correspond to the hardware name index used in the ifconfig command (e.g., sm0 – where 0 is the valid index).

Use the default setting by typing 'ifconfig sm0 /dev/config' (where sm0 indicates index 0) to re-load the values stored in the configuration block. In general, the MAC address is stored in the config block for viewing purposes only, and the actual MAC address used is accessed directly by the Ethernet chip out of its dedicated serial EEPROM. On SOMs without a serial_eeprom, the config block is required if the customer wishes to use the Ethernet feature from within LogicLoader.

The configuration block allows storage for up to eight Ethernet interface configurations, but the config E command is only able to store information for SOM hardware that is supported by LogicLoader.

9 YAFFS (Yet Another Flash File System)

9.1 YAFFS Overview

The acronym YAFFS stands for the phrase "Yet Another Flash Filing System.” YAFFS was developed by a company named Aleph One Limited and incorporated by Logic Product Development into the LogicLoader (LoLo) software program.

Logic 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 RTOS 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. As an 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> add-yaffs nand-root nand B9 B500
losh> mount yaffs /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.

9.2 Working with YAFFS in LogicLoader

9.2.1 Developing a Partition Scheme

The LogicLoader may mount up to four YAFFS partitions at a time. Customers should design a partitioning scheme which suits their individual needs. The following limits are imposed on partitions:

- Each partition must have a unique name.
- Each partition must exist on local flash accessible from LogicLoader’s '/dev/flashx' device file (where ‘x’ is an instance index). For example: /dev/flash0 or /dev/flash1.
- Each partition must span at least 4 physical flash blocks.
- A partition must *not* overlap the flash blocks that contain LogicLoader or its configuration block. For information about the location of these items, check the *LogicLoader User's Manual* addendum for your hardware.
- A partition (in NOR flash) has to be aligned on and exactly span the largest sector boundary found within the address range specified.

For the remainder of this document, the following partitioning scheme for demonstration of NOR flash will be used:

- A partition named “boot” which contains a bitmap and operating system image and spans the address space below:
    - * start: 0x000C0000
    - * length: 0x00800000 (8 MBytes)
- A partition named “data” which contains customer specific data.
    - * start: 0x00900000
    - * length: 0x00400000 (4 MBytes)

For boards with NAND devices, the following partition scheme for demonstration purposes will be used:

- A partition named "boot" which contains a bitmap and operating system image and spans the block range below:
  - * start: B10
  - * length: B256 (8 MBytes, assuming 32KByte block size)
- A partition named "data" which contains customer specific data.
  - * start: B266 (abuts boot partition)
  - * length: B128 (4 MBytes, assuming 32KByte block size)

9.2.2 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, simply use LogicLoader's 'erase' command to erase the area of flash where the partition is to be located.

Using the example partition scheme in the "Developing a Partition Scheme" section, above, the partitions could be prepared for initial use by erasing the regions of the flash device spanned by them.

For a NOR example (LH7A404-11 system address used):

```
losh> erase 0x000C0000 0x00800000
losh > erase 0x00900000 0x00400000
```

For a NAND example:

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

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.

9.2.3 Adding YAFFS Type Partitions

LogicLoader maintains a partition table in RAM. Before a YAFFS partition can be mounted, it must be added to the partition table. To do this, the 'add-yaffs' command is used. The 'add-yaffs' command takes the following arguments:

■ <name>	a unique string which identifies the partition
■ <type>	type of flash device the partition resides on
■ <start>	the physical starting address of the partition
■ <length>	the length (in bytes) of the partition

Continuing with the example partitions above, LogicLoader can be instructed to add the partitions by executing the commands as shown below (LH7A404-11 system address used):

```
        losh> add-yaffs boot nor 0x000C0000 0x00800000
        losh> add-yaffs data nor 0x00900000 0x00400000
```

For NAND partitions, the numbers change to block addresses:

```
        losh> add-yaffs boot nand B10 B256
        losh> add-yaffs data nand B266 B128
```

Note: The above steps must be performed every time LogicLoader boots. Because LogicLoader keeps the partition table in RAM, the existence and locations of YAFFS partitions (for both NAND and NOR) does not persist across resets or power cycles.

9.2.4 Mounting the Partition

To mount a partition in the partition table, the 'mount' command is used. That command takes the following arguments:

■ <fstype>	the type of file system being mounted ('yaffs' here)
■ [drive addr]	not used when mounting a YAFFS partition
■ <point>	the name of the YAFFS partition

For example:

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

Of note is that the 'drive addr' argument is not used when mounting a YAFFS partition. Also of note is that the 'point' argument needs to correspond to the name of the partition (as defined by the add-yaffs command) preceded by a forward slash. LogicLoader needs to mount all YAFFS partitions at the root-directory level. Thus, a partition added using:

```
        'add-yaffs boot ...'
```

will be mounted using:

```
        'mount ... /boot'.
```

Note the absence of the '/' character during the 'add-yaffs' command and its presence during the 'mount' command.

9.2.5 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.

9.3 Summary

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

1. Decide on a partitioning scheme.
2. Format the partitions by erasing the associated flash blocks.
3. Add the partitions to LogicLoader using the 'add-yaffs' command.
4. Mount the partitions using the 'mount' command.

Steps 3 and 4 must be repeated every time the system is booted. If the YAFFS partitions are frequently accessed, consider implementing steps 3 and 4 via a boot script. Step 2 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 that a partition name does *not* begin with a '/'. LogicLoader's virtual file system uses the forward slash to indicate the root directory.
- Ensure partitions do not overlap each other, LogicLoader, or the configuration block.
- Ensure that a partition is erased before it is mounted for the first time.

Appendix: 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: