Broadcom Corporation

Common Firmware Environment (CFE) Functional Specification

ABSTRACT

This document describes a common firmware environment to be shared by all Broadcom MIPS64 processors and SOC designs. The goal is to provide a standard set of APIs and data structures for use by bootstrap, diagnostics, and initialization code for operating systems.

Author:	Mitch Lichtenberg (mpl@broadcom.com)
Document Version:	1.3
Edit Number:	248
Last Revised:	07/12/02 3:09 PM
Classification:	Broadcom Proprietary and Confidential

This page is supposed to be blank.
(of course, we've gone and put stuff on it, so it isn't really blank now is it?)

Table Of Contents

1. INTRODUCTION 1

1.1 PROJECT DESCRIPTION 1
1.2 LICENSE INFORMATION 2
1.3 RELATED DOCUMENTS 2
1.4 TRADEMARKS 2
1.5 REVISION HISTORY 3
1.6 NOTATION CONVENTIONS 3

2. OVERVIEW 4

2.1 SUPPORTED PLATFORMS 4
2.2 INITIALIZATION 4
2.3 EXECUTION MODEL 5
2.4 DEVICE DRIVERS 5
2.5 NETWORK SUPPORT 5
2.6 SYSTEM BOOTSTRAP 6
2.7 MEMORY MANAGEMENT 6
2.8 ENVIRONMENT 6
2.9 FIRMWARE API 6
2.10 VGA AND KEYBOARD SUPPORT 7
2.11 USB SUPPORT 7

3. USER INTERFACE 8

3.1 COMMAND SYNTAX 8
3.2 SPECIAL ENVIRONMENT VARIABLES 8
3.3 COMMAND LINE EDITOR 9
3.4 COMMAND DESCRIPTIONS 9
3.4.1 ARP 10
3.4.1.1 Usage 10
3.4.1.2 Description 10
3.4.1.3 Options 10
3.4.1.4 Example 10
3.4.2 BOOT, LOAD 11
3.4.2.1 Usage 11
3.4.2.2 Description 11
3.4.2.3 Options 11
3.4.2.4 Example 12
3.4.3 COPYDISK 13
3.4.3.1 Usage 13
3.4.3.2 Description 13
3.4.3.3 Options 13
3.4.3.4 Example 13
3.4.4 CPU1 (test command) 14
3.4.4.1 Usage 14
3.4.4.2 Description 14
3.4.4.3 Options 14
3.4.4.4 Example 14
3.4.5 D (dump) 15
3.4.5.1 Usage 15

3.4.5.2 Description 15
3.4.5.3 Options 15
3.4.5.4 Example 15
3.4.6 DEFEATURE 16
3.4.6.1 Usage 16
3.4.6.2 Description 16
3.4.6.3 Options 16
3.4.6.4 Example 16
3.4.7 E (edit) 17
3.4.7.1 Usage 17
3.4.7.2 Description 17
3.4.7.3 Options 17
3.4.7.4 Example 17
3.4.8 F (fill) 18
3.4.8.1 Usage 18
3.4.8.2 Description 18
3.4.8.3 Options 18
3.4.8.4 Example 18
3.4.9 FLASH 19
3.4.9.1 Usage 19
3.4.9.2 Description 19
3.4.9.3 Options 19
3.4.9.4 Example 19
3.4.10 GO 20
3.4.10.1 Usage 20
3.4.10.2 Description 20
3.4.10.3 Options 20
3.4.10.4 Example 20
3.4.11 HELP 21
3.4.11.1 Usage 21
3.4.11.2 Description 21
3.4.11.3 Options 21
3.4.11.4 Example 21
3.4.12 IFCONFIG 22
3.4.12.1 Usage 22
3.4.12.2 Description 22
3.4.12.3 Options 22
3.4.12.4 Example 23
3.4.13 LOOP 24
3.4.13.1 Usage 24
3.4.13.2 Description 24
3.4.13.3 Options 24
3.4.13.4 Example 24
3.4.14 MAP PCI 25
3.4.14.1 Usage 25
3.4.14.2 Description 25
3.4.14.3 Options 25
3.4.14.4 Example 25
3.4.15 MEMORYTEST 26
3.4.15.1 Usage 26
3.4.15.2 Description 26
3.4.15.3 Options 26
3.4.15.4 Example 26
3.4.16 MEMTEST 27
3.4.16.1 Usage 27
3.4.16.2 Description 27

3.4.16.3 Options 27
3.4.16.4 Example 27
3.4.17 PHY DUMP 28
3.4.17.1 Usage 28
3.4.17.2 Description 28
3.4.17.3 Options 28
3.4.17.4 Example 28
3.4.18 PHY SET 29
3.4.18.1 Usage 29
3.4.18.2 Description 29
3.4.18.3 Options 29
3.4.18.4 Example 29
3.4.19 PING 30
3.4.19.1 Usage 30
3.4.19.2 Description 30
3.4.19.3 Options 30
3.4.19.4 Example 30
3.4.20 PRINTENV 31
3.4.20.1 Usage 31
3.4.20.2 Description 31
3.4.20.3 Options 31
3.4.20.4 Example 31
3.4.21 RESET 32
3.4.21.1 Usage 32
3.4.21.2 Description 32
3.4.21.3 Options 32
3.4.21.4 Example 32
3.4.22 SAVE 33
3.4.22.1 Usage 33
3.4.22.2 Description 33
3.4.22.3 Options 33
3.4.22.4 Example 33
3.4.23 SET CONSOLE 34
3.4.23.1 Usage 34
3.4.23.2 Description 34
3.4.23.3 Options 34
3.4.23.4 Example 34
3.4.24 SET DATE 35
3.4.24.1 Usage 35
3.4.24.2 Description 35
3.4.24.3 Options 35
3.4.24.4 Example 35
3.4.25 SET TIME 36
3.4.25.1 Usage 36
3.4.25.2 Description 36
3.4.25.3 Options 36
3.4.25.4 Example 36
3.4.26 SETENV 37
3.4.26.1 Usage 37
3.4.26.2 Description 37
3.4.26.3 Options 37
3.4.26.4 Example 37
3.4.27 SHOW AGENTS 38
3.4.27.1 Usage 38
3.4.27.2 Description 38
3.4.27.3 Options 38

3.4.27.4 Example 38
3.4.28 SHOW BOOT 39
3.4.28.1 Usage 39
3.4.28.2 Description 39
3.4.28.3 Options 39
3.4.28.4 Example 39
3.4.29 SHOW DEFEATURE 40
3.4.29.1 Usage 40
3.4.29.2 Description 40
3.4.29.3 Options 40
3.4.29.4 Example 40
3.4.30 SHOW DEVICES 41
3.4.30.1 Usage 41
3.4.30.2 Description 41
3.4.30.3 Options 41
3.4.30.4 Example 41
3.4.31 SHOW FLASH 42
3.4.31.1 Usage 42
3.4.31.2 Description 42
3.4.31.3 Options 42
3.4.31.4 Example 42
3.4.32 SHOW HEAP 43
3.4.32.1 Usage 43
3.4.32.2 Description 43
3.4.32.3 Options 43
3.4.32.4 Example 43
3.4.33 SHOW MEMORY 44
3.4.33.1 Usage 44
3.4.33.2 Description 44
3.4.33.3 Options 44
3.4.33.4 Example 44
3.4.34 SHOW PCI 45
3.4.34.1 Usage 45
3.4.34.2 Description 45
3.4.34.3 Options 45
3.4.34.4 Example 45
3.4.35 SHOW SOC 46
3.4.35.1 Usage 46
3.4.35.2 Description 46
3.4.35.3 Options 46
3.4.35.4 Example 47
3.4.36 SHOW SPD 48
3.4.36.1 Usage 48
3.4.36.2 Description 48
3.4.36.3 Options 48
3.4.36.4 Example 48
3.4.37 SHOW TEMP 49
3.4.37.1 Usage 49
3.4.37.2 Description 49
3.4.37.3 Options 49
3.4.37.4 Example 49
3.4.38 SHOW TIME 50
3.4.38.1 Usage 50
3.4.38.2 Description 50
3.4.38.3 Options 50
3.4.38.4 Example 50

3.4.39 U (unassemble) 51
3.4.39.1 Usage 51
3.4.39.2 Description 51
3.4.39.3 Options 51
3.4.39.4 Example 51
3.4.40 UNSETENV 52
3.4.40.1 Usage 52
3.4.40.2 Description 52
3.4.40.3 Options 52
3.4.40.4 Example 52

4. SOFTWARE INTERNALS 53

4.1 MODULE OVERVIEW 53
4.2 LIBRARY MODULES 53
4.3 SYSTEM STARTUP 54
4.4 MULTIPROCESSOR STARTUP 55
4.5 "BI-ENDIAN" STARTUP 56
4.6 HEAP MANAGER 56
4.7 PHYSICAL MEMORY MANAGER 57
4.8 DEVICE MANAGER 58
4.9 CONSOLE INTERFACE 58
4.10 ENVIRONMENT MANAGER 59
4.11 TIMER MANAGER 61
4.12 NETWORK SUBSYSTEM 61
4.13 FILE SYSTEMS 62
4.14 PCI/LDT CONFIGURATION 63
4.15 USER INTERFACE 63
4.15.1 Adding a command 64
4.15.2 Calling the command function 65

5. THE BCM1250 REFERENCE DESIGNS 66

5.1 BOARD DESCRIPTION (SWARM) 66
5.1.1 Features 66
5.1.2 Jumpers and Settings 66
5.1.3 Firmware Devices 68
5.2 ADDRESSES OF ONBOARD PERIPHERALS 68
5.2.1 Generic Bus Assignments 68
5.2.2 GPIO Signals 68
5.3 BOARD DESCRIPTION (SENTOSA) 69
5.3.1 Features 69
5.3.2 Jumpers and Settings 69
5.4 BOARD DESCRIPTION (RHONE) 70
5.4.1 Features 70
5.4.2 Jumpers and Settings 71
5.4.3 Firmware Devices 72
5.4.4 Addresses of onboard peripherals 72
5.4.5 Generic Bus Assignments 72
5.4.6 GPIO Signals 72
5.5 LOADING CFE VIA A ROM EMULATOR 73
5.6 INSTALLING A NEW VERSION OF THE FIRMWARE INTO THE FLASH 74

6. PORTING CFE TO A NEW DESIGN 75

6.1 TOOLS REQUIRED FOR BUILDING CFE 75
6.2 DIRECTORY STRUCTURE 76
6.2.1 The build directory (build/) 76
6.2.2 The CFE source directory (cfe/) 76

6.2.3 Board, CPU, and Architecture directories ... 77
6.3 MAKEFILE FLOW ... 78
6.4 EXAMPLE MAKEFILE ... 79
6.5 SPECIAL SOURCE FILES ... 79
6.6 CONFIGURATION OPTIONS ... 80
6.6.1 Required Makefile macros ... 80
6.6.2 Options in the Makefile ... 80
6.6.3 Options in the bsp_config.h file ... 82
6.6.4 Startup Routines ... 83
6.6.5 Special caveats for *board_earlyinit* ... 84
6.6.6 Relocatable Code and Data ... 84
6.7 DRAM INITIALIZATION ON THE BCM1250 ... 85
6.7.1 DRAM Initialization Table ... 85
6.7.1.1 DRAM_GLOBALS(chintlv) ... 87
6.7.1.2 DRAM_GLOBALS(chintlv) ... 87
6.7.1.3 DRAM_CHAN_CFG(chan,tMEMCLK,dramtype,pagepolicy,blksize,csintlv,ecc,flg) ... 87
6.7.1.4 DRAM_CHAN_CLKCFG(addrskew,dqoskew,dqiskew,addrdrive,datadrive,clkdrive) ... 89
6.7.1.5 DRAM_CHAN_MANTIMING(tCK,rfsh,tval) ... 89
6.7.1.6 DRAM_CS_SPD(csel,flags,chan,dev) ... 89
6.7.1.7 DRAM_CS_GEOM(csel,rows,cols,banks) ... 90
6.7.1.8 DRAM_CS_TIMING(tCK,rfsh,caslatency,attributes,tRAS,tRP,tRRD,tRCD,tRFC,tRC) ... 90
6.7.2 Sample draminit tables ... 91
6.7.2.1 SWARM board ... 91
6.7.2.2 SENTOSA board ... 92
6.7.2.3 Large Memory (external decode mode) ... 93
6.8 LED MESSAGES ... 94

7. DEVICE DRIVERS ... 96

7.1 DEVICE DRIVER STRUCTURE ... 96
7.1.1 Device Descriptor ... 96
7.1.2 Device Classes ... 96
7.1.3 Function Dispatch ... 97
7.1.4 The Probe routine ... 97
7.2 ADDING A NEW DEVICE DRIVER ... 98
7.3 DEVICE DRIVER PROBE ARGUMENTS FOR SUPPLIED DEVICES ... 99
7.4 DEVICE DRIVER FUNCTIONS ... 99
7.4.1 The *dev_open* routine ... 100
7.4.2 The *dev_read* routine ... 100
7.4.3 The *dev_inpstat* routine ... 100
7.4.4 The *dev_write* routine ... 101
7.4.5 The *dev_ioctl* routine ... 101
7.4.6 The *dev_close* routine ... 102
7.4.7 The *dev_poll* routine ... 102
7.4.8 The *dev_reset* routine ... 102
7.5 STANDARD DEVICE IOCTLS AND READ/WRITE BEHAVIOR ... 102
7.5.1 Ethernet Devices ... 103
7.5.1.1 Read/Write behavior ... 103
7.5.1.2 Standard IOCTLs ... 103
7.5.2 Flash Memory Devices ... 103
7.5.2.1 Read/Write behavior ... 103
7.5.2.2 Standard IOCTLs ... 104
7.5.3 EEPROM Devices ... 104
7.5.3.1 Read/Write behavior ... 104
7.5.3.2 Standard IOCTLs ... 104
7.5.4 Serial Devices ... 105
7.5.4.1 Read/Write behavior ... 105

7.5.4.2 Standard IOCTLs 105
7.5.5 Disk Devices 105
7.5.5.1 Read/Write behavior 105
7.5.5.2 Standard IOCTLs 105

8. FIRMWARE API AND BOOT ENVIRONMENT 107

8.1 ENTRY POINT 107
8.2 BOOT ENVIRONMENT 108
8.2.1 Virtual Address Space 108
8.2.2 Environment Variables 108
8.2.3 Registers passed to boot loaders 109
8.2.4 Registers passed to secondary processors 109
8.2.5 Memory Map 110
8.3 DISK BOOTSTRAP 111
8.3.1 Generating a Boot Block 112
8.4 API FUNCTIONS 112
8.5 VENDOR EXTENSIONS 113
8.5.1 CFE_CMD_FW_GETINFO 114
8.5.2 CFE_CMD_FW_RESTART 115
8.5.3 CFE_CMD_FW_CPUCTL 116
8.5.4 CFE_CMD_FW_GETTIME 118
8.5.5 CFE_CMD_FW_MEMENUM 119
8.5.6 CFE_CMD_FW_FLUSHCACHE 120
8.5.7 CFE_CMD_DEV_GETHANDLE 121
8.5.8 CFE_CMD_DEV_ENUM 122
8.5.9 CFE_CMD_DEV_OPEN 123
8.5.10 CFE_CMD_DEV_INPSTAT 124
8.5.11 CFE_CMD_DEV_READ 125
8.5.12 CFE_CMD_DEV_WRITE 126
8.5.13 CFE_CMD_DEV_IOCTL 127
8.5.14 CFE_CMD_DEV_CLOSE 128
8.5.15 CFE_CMD_DEV_GETINFO 129
8.5.16 CFE_CMD_ENV_ENUM 130
8.5.17 CFE_CMD_ENV_GET 131
8.5.18 CFE_CMD_ENV_SET 132
8.5.19 CFE_CMD_ENV_DEL 133

1. Introduction

Note to customers: Please provide whatever feedback you can on CFE and this document. Our goals are to make this software useful to most customers, particularly those working on new designs that do not already have firmware to port. If you have comments, send them to me at mpl@broadcom.com**. Read the file 'TODO' in the root directory of the source tree for a list of the things we're planning in the future, and 'README' for a description of recent changes to CFE.**

1.1 Project Description

The Broadcom *Common Firmware Environment (CFE)* is a collection of software modules for initialization and bootstrap of designs incorporating Broadcom MIPS64™ processors. CFE can be used early in the development of designs using Broadcom processors to do bringup, and later be used to bootstrap the OS in a production environment.

CFE was designed with the following goals:

- It should be *simple*. Boot code isn't supposed to be very fancy. It should be easy to bring up and dependable. The "keep it simple" principle was applied liberally in the design of CFE.
- It should be easily portable to new designs incorporating Broadcom MIPS64 CPUs.
- It should support a variety of bootstrap devices, boot file systems, and console interfaces.
- It should be easy to add new device support
- It should be modular, and easy to remove unnecessary features
- It should serve as a collection of examples of simple device drivers for the integrated peripherals on Broadcom processors.

Therefore, there are certain "non-goals" in CFE:

- It is *not* designed to be portable to non-MIPS platforms. However, with some effort it can be used on MIPS designs that do not use Broadcom processors.
- It is *not* designed to be compatible with IEEE 1275 ("Open Firmware") or other established firmware standards. Similarly, it is also *not* designed to become a firmware standard.
- It is *not* meant to be a hardware-abstraction layer or BIOS usable by the operating system for normal device access.

1.2 License Information

1.3 Related Documents

Readers of this specification may find the following documents useful:

- *Broadcom BCM1250 User's Manual*, Broadcom Corporation
- *Broadcom SB-1 MIPS Core User's Manual*, Broadcom Corporation
- *See MIPS Run*, Dominic Sweetman, Morgan Kaufmann Publishers, Inc., ISBN 1-55860-410-3
- *MIPS64™ specification*, MIPS Incorporated.

1.4 Trademarks

The following trademarks are used in this document.

- *Mercurian* is a trademark of Broadcom Corporation
- *SiByte* is a trademark of Broadcom Corporation
- *MIPS* and *MIPS64* are trademarks of MIPS Corporation

1.5 Revision History

This section contains a list of revisions to this document.

Who	When	What
Lichtenberg	2/14/2001	Created this file.
Lichtenberg	2/19/2001	Added Justin's comments
Lichtenberg	3/15/2001	Filled out some of the empty sections, first release of CFE source to customers
Lichtenberg	5/8/2001	Updated to include changes for version 0.0.3
Lichtenberg	6/18/2001	Major updates since firmware became operational on real hardware
Lichtenberg	9/28/2001	Updated to reflect new directory structure (version 1.0.25)
Lichtenberg	11/29/2001	Updated to reflect version 1.0.26
Lichtenberg	2/4/2002	Updated to reflect version 1.0.27
Lichtenberg	5/8/2002	Updated to reflect version 1.0.32
Lichtenberg	7/11/2002	More updates to reflect reality

1.6 Notation Conventions

This section lists special notation that is used in this document.

Notation	Description
`Fixed-Width Text`	Information displayed by the computer, names of files, or directories.
`Italic Fixed Text`	Information you type into the computer (commands, answers to prompts)
Bold comments	Special notes or caveats
Symbols in *italics*	File names, structure and field names, argument names.

2. Overview

This chapter gives an overview of CFE's major features and internal subsystems. The internals will be covered in greater detail in subsequent chapters.

2.1 Supported Platforms

CFE is designed to be easily portable to designs incorporating current and future Broadcom MIPS64 compatible broadband processors. Supported platforms include:

- Broadcom's *SiByte* processor family (BCM1250, BCM1125, etc.)
- 32-bit and 64-bit memory models
- Big and little-endian operation

There are many configurable parameters at build time that may be used to customize CFE to suit your needs.

2.2 Initialization

On startup, CFE performs the following low-level initialization:

- Reset and ROM trap handler vectors
- CPU and FPU initialization
- L1 and L2 Cache initialization
- Multiprocessor initialization
- Memory controller initialization
- PCI and LDT bus configuration
- Environment variables
- Console device initialization
- Bootstrap device initialization

Certain information, such as the physical memory layout and other critical information, are stored by CFE and are made available to boot loaders and operating systems via CFE's external API.

Once initialization has completed, CFE is ready to load programs from the bootstrap device.

2.3 Execution Model

While running, CFE:

- Polls all I/O devices (it never touches the interrupt controller).
- Runs with interrupts completely disabled.
- On multiprocessor configurations, a spin lock is used to guard the entry point, allowing only one thread of execution into the firmware at a time.

2.4 Device Drivers

CFE also incorporates several built-in device drivers for console access and bootstrapping, including:

- BCM1250 UARTs
- NS16550-compatible UARTs
- BCM1250 Ethernet
- Digital/Intel DC21143 (“Tulip") PCI Ethernet
- National Semiconductor DP83815 PCI Ethernet
- Xicor X1240 SMBus real-time clock and EEPROM
- ST Micro 41T81C SMBus real-time clock
- Microchip 24LC128 serial EEPROMs
- Atmel AT24C02 serial EEPROMs
- Flash memory devices (AMD and Intel command sets)
- IDE disks, ATAPI disks, and ATAPI CD-ROMs via Generic Bus
- PCMCIA ATA Flash Devices
- Grammar Engine (http://www.gei.com) PromICE virtual serial port

New device drivers are easy to add to CFE, to permit initialization and bootstrap from external peripherals.

2.5 Network Support

CFE includes support for network bootstrap from the BCM1250’s Ethernet controllers. The network interface implements the following specifications:

- Address Resolution Protocol (ARP)
- Internet Protocol (IP)
- Internet Control Message Protocol (ICMP)
- Dynamic Host Configuration Protocol (DHCP)
- User Datagram Protocol (UDP)

- Trivial File Transfer Protocol (TFTP)
- Transmission Control Protocol (TCP)

The network interface is deliberately simple, providing only that functionality needed to read configuration files and boot the system. In particular, NFS support is not provided.

Customers: Some Unix OSes, such as NetBSD, bootstrap via NFS. The intent would be that the boot loader (loaded by CFE) would do this. NetBSD's current network boot loader uses CFE's API and implements its own NFS bootstrap. See netbsd/src/sys/arch/sbmips/stand for an example.

2.6 System Bootstrap

CFE can load programs from bootstrap devices in a variety of ways:

- Via the network from a TFTP server
- Via an IDE disk connected to the BCM1250's generic bus
- Via an IDE CD-ROM drive connected to the BCM1250's generic bus
- Via a PCMCIA ATA flash card in the PCMCIA slot.
- Via S-Records sent to the serial port

Loaded images can be S-records, raw binary files, or files in MIPS ELF format.

For disk devices (IDE disks and flash memory cards), the disk may either be unformatted (no file system) or formatted with a DOS FAT-style files system.

2.7 Memory Management

CFE creates and maintains a map of available physical memory. The operating system can query this map to determine what regions of memory are available and which are reserved by the hardware or firmware.

2.8 Environment

CFE maintains a global set of environment variables. The environment storage can be configured to live on any non-volatile device (EEPROMs, flash, even a disk). Environment storage is used to store system parameters such as Ethernet hardware addresses, IP addresses, startup scripts, and other information.

2.9 Firmware API

CFE exports an API that can be used by operating systems to access the console, bootstrap device, and to read system information. The API also permits control of secondary processor cores in multiprocessor designs.

For example, NetBSD has two levels of bootstrap. The boot loader reads the NetBSD loader from the boot device, and the NetBSD loader loads and launches the kernel. The NetBSD boot loader needs a device-independent way of accessing the boot device (network, disk, etc.), so CFE provides a simple mechanism for loaders to obtain data from the boot device. Operating systems typically need to access the firmware for certain configuration information, such as the available physical memory, MAC addresses of onboard Ethernet controllers, etc.

The CFE firmware API can be called from both 32-bit and 64-bit applications.

2.10 VGA and keyboard support

The BCM1250 evaluation board is packaged as an ATX (PC-style) board, including some PCI slots and an on-board USB interface. CFE has some minimal support for bootstrap using a "PC console" (using a USB PC keyboard, and a compatible VGA video device). See chapter XXX for more information on the PC console.[1]

The following adapters are known to work with CFE's VGA support:

- ATI Rage 128
- Nvidia Vanta-2 based cards
- 3Dfx Voodoo3 2000 PCI

These cards also support 3.3V signaling.

2.11 USB support

CFE includes a simple USB host stack with support for:

- An OHCI host controller
- Emulated root hub
- Standard USB hubs
- Keyboards and mice (boot protocol only)
- USB mass storage (SCSI command set)

Support is minimal, and this feature is generally unsupported.

[1] VGA support, while it has been shown to work, is not generally supported and is not tested very often at this tme.

3. User Interface

3.1 Command Syntax

CFE has a simple but "shell-like" command interpreter. Commands you enter are broken into words in a manner similar to the Unix shell. The double quote ("") characters may be used to group words together into a single word. Single quotes ('') do the same thing, except environment variable expansions will not be done.

Environment variables are expanded when CFE encounters a $ symbol when scanning the command. For example, in the command "foo $bar" the "$bar" portion will be replaced by the current value of the environment variable *bar*.

The backslash character is the shell escape character. To include a dollar sign in a string, for example, you must specify \$. To insert a backslash, use two backslashes (\\).

You can type more than one command on the command line by separating the commands with one of the *command separator* symbols.

Symbol	Description
`Semicolon (;)`	Run the following command regardless of the termination status of the current command.
`And (&&)`	Run the following command only if the current command returns a good termination status (0)
`Or (||)`	Run the following command only if the current command returns a bad termination status (not equal 0)

3.2 Special Environment Variables

The following special environment variables are used by CFE's command interpreter:

Variable	Description
`PROMPT`	Contains CFE's command prompt string. If unset, CFE will use the string "CFE>"
`STARTUP`	Contains one or more commands to be executed when the system completes initialization. If you type Ctrl-C during startup, you can prevent CFE from executing these commands.
`F1 through F12`	Contains commands to be run when you press function keys F1 through F12

3.3 Command line editor

CFE includes a rudimentary command line editor. The keys should be compatible with most ANSI-style terminals and terminal emulators. The command line editing keys are described below:

Key(s)	Description
`Ctrl-H, Backspace, Delete`	Delete the character to the left of the cursor
`Up-Arrow, Ctrl-P`	Recall the previous command
`Down-Arrow, Ctrl-N`	Recall the next command
`Left-arrow, Ctrl-B`	Move cursor left one character
`Right-arrow, Ctrl-F`	Move cursor right one character
`Ctrl-A`	Move cursor to the beginning of the line
`Ctrl-E`	Move cursor to the end of the line
`Ctrl-D`	Delete the character under the cursor
`Ctrl-U`	Erase the entire command
`Ctrl-R`	Redisplay the command
`Ctrl-K`	Delete all characters from the cursor to the end of the line and save in the kill buffer
`Ctrl-Y`	Insert characters from the kill buffer at the current cursor position
`F1 through F12`	Execute the command stored in the environment variable F1 through F12
`F12`	Repeat the last command (if F12 is not defined)

The command line editor will not operate properly if the command being edited exceeds the width of the terminal emulator's window.

3.4 Command Descriptions

The sections that follow describe the commands that are available in most versions of CFE. Some commands are only available by enabling certain compile-time configuration options.

3.4.1 ARP

3.4.1.1 Usage

```
arp [options] ip-address dest-address
```

3.4.1.2 Description

Display or modify the ARP table. The ARP table maps IP addresses to Ethernet (hardware) addresses on the network. Normally these addresses are obtained automatically by the ARP protocol. You can use this command to verify the contents of the ARP table or to force entries to appear in the table.

The *ip-address* parameter is an IP address to add to the table, in dotted-decimal notation. The *dest-address* is the hardware address, as 12 hex digits. If you add an entry manually, it will not time out.

3.4.1.3 Options

Option	Description
`-d`	Delete the specified entry, or all entries if *ip-address* is an asterisk.

3.4.1.4 Example

```
CFE> arp –d *
*** command status = 0
```

3.4.2 BOOT, LOAD

3.4.2.1 Usage

```
boot [options] file-name
load [options] file-name
```

3.4.2.2 Description

Bootstraps the system from the specified device and file name. By default, the *boot* command will read a raw binary from the specified boot device into the boot area at 0x2000_0000 and then jump to that address. Use of the *boot* command's options can cause CFE to use a different loader or override the file system defaults.

The *load* command functions exactly like the *boot* command except it does not start the loaded program.

The *file-name* parameter may be a device name or a network file name. CFE will take different actions based on the sort of device that *file-name* refers to:

Serial device	CFE will read S-records from the serial device and transfer control to the program when it receives a start-address record. This is equivalent to using the *S-record* loader and the *raw* filesystem.
Disk	CFE will look for a boot block on the disk device. If it finds a valid boot block, it will load the boot loader according to the instructions in the boot block and execute the boot loader. This is equivalent to using the *raw* loader and the *raw* filesystem.
host:filename	CFE will transfer the file from the network via TFTP and execute it. This is equivalent to using the *raw* loader and the *tftp* filesystem.

You can override both the file system and loader choice. For example, you can store S-records on the network or on an FAT-formatted PCMCIA card, or put an ELF binary into flash. Most reasonable combinations of loaders and file systems should work.

3.4.2.3 Options

Option	Description
`-elf`	Choose the ELF loader
`-srec`	Choose the S-Record loader
`-raw`	Choose the RAW loader
`-z`	Boot or load a compressed file (compressed via *gzip*). To use this option, CFE must be built with ZLIB support (CFG_ZLIB=1 in the Makefile)

`-loader=XXX`	Choose the loader by name (the –elf, -srec, and –raw options are shortcuts)
`-tftp`	Choose the TFTP file system
`-fatfs`	Choose the FAT file system
`-rawfs`	Choose the RAW file system
`-fs=XXX`	Choose the file system by name (the –tftp, -fatfs, and –rawfs options are shortcuts)
`-max=XXX`	Specifies the maximum number of bytes to load for the RAW file system. Defaults to 256KB
`-addr=XXX`	Specifies the address where the RAW filesystem will load the binary. Defaults to 0x2000_0000.
`-noclose`	Do not close the network before starting the program. (*boot* only)

3.4.2.4 Example

```
CFE> boot ide0:
CFE> boot –elf host:bootprogs/os_startup
CFE> boot –srec –fatfs pcmcia0:my_test.srec
CFE> load –raw –addr=80100000 host:data/my_data_file
```

3.4.3 COPYDISK

3.4.3.1 Usage

```
copydisk host:filename device-name [offset]
```

3.4.3.2 Description

Copies a remote file via TFTP to the specified offset on a local disk device. If you have built a RAM disk or other sector-by-sector disk image, you can use COPYDISK to transfer this image onto a real disk connected to your system. The *offset* parameter specifies a byte offset on the local disk that will receive the first byte of the image file.

3.4.3.3 Options

None

3.4.3.4 Example

```
CFE> copydisk mytftphost:path/to/disk.img ide0.0
*** command status = 0
CFE>
```

3.4.4 CPU1 (test command)

[Multiprocessor SiByte CPUs only]

3.4.4.1 Usage

```
cpu1 start|stop
```

3.4.4.2 Description

Provides a simple mechanism to verify the operation of CPU1 in multiprocessor versions of CFE. Normally this command would not be available in production environments.

By default, the *start* command causes a simple loop to run on core 1 that displays messages on the LED display, if present. The *stop* command forces the CPU back into CFE's secondary processor wait loop.

3.4.4.3 Options

Option	Description
`-addr=xxx`	Specify the starting address that CPU1 should jump to
`-a1=xxx`	Specify the value for register A1, passed to the startup routine
`-sp=xxx`	Specify the value for register SP, passed to the startup routine
`-gp=xxx`	Specify the value for register GP, passed to the startup routine

3.4.4.4 Example

```
CFE> cpu1 start
*** command status = 0
CFE> cpu1 stop
*** command status = 0
CFE>
```

3.4.5 D (dump)

3.4.5.1 Usage

```
d [options] address length
```

3.4.5.2 Description

Display a dump of memory, in hexadecimal.

The *address* option is the starting address to dump. *Length* specifies the number of bytes to display. If omitted, the dump command will use the end of the previous dump and its length as the starting address and length.

3.4.5.3 Options

Option	Description
-b	Dump memory as bytes
-h	Dump memory as halfwords (16 bits)
-w	Dump memory as words (32 bits)
-q	Dump memory as quadwords (64 bits)
-p	The *address* parameter is a physical address
-v	The *address* parameter is a virtual address

3.4.5.4 Example

```
CFE> d 80000000
FFFFFFFF80000000: 0000000000000000 0000000000000000  ................
FFFFFFFF80000010: 0000000000000000 0000000000000000  ................
FFFFFFFF80000020: 0000000000000000 0000000000000000  ................
FFFFFFFF80000030: 0000000000000000 0000000000000000  ................
*** command status = 0
CFE>
```

3.4.6 DEFEATURE

[SiByte CPUs only]

3.4.6.1 Usage

```
defeature new-value
```

3.4.6.2 Description

Sets the defeature mask for core 0's CPU. You should use great care when using this command, the behaviour of the defeature register changes from revision to revision of the CPU core.

3.4.6.3 Options

None

3.4.6.4 Example

```
CFE> unsetenv FOO
*** command status = 0
CFE>
```

3.4.7 E (edit)

3.4.7.1 Usage

```
e [options] address [data]
```

3.4.7.2 Description

Edit the contents of memory. *Address* specifies the starting address to modify. You can list data to be written into memory on the command line. If omitted, CFE will enter a "memory edit" mode to let you interactively edit memory values.

When in memory edit mode, you can enter '-' to back up, '=' to dump memory at the current location, or '.' to edit memory edit mode.

3.4.7.3 Options

Option	Description
-b	Edit memory as bytes
-h	Edit memory as halfwords (16 bits)
-w	Edit memory as words (32 bits)
-q	Edit memory as quadwords (64 bits)
-p	The *address* parameter is a physical address
-v	The *address* parameter is a virtual address

3.4.7.4 Example

```
CFE> e –q b0061000 1234567812345678
*** command status = 0
CFE> e –q 80000000
FFFFFFFF80000000: [0000000000000000]: 12345678
FFFFFFFF80000008: [0000000000000000]: .
*** command status = 0
CFE>
```

3.4.8 F (fill)

3.4.8.1 Usage

```
f [options] address length [pattern]
```

3.4.8.2 Description

Fills a region of memory with the specified pattern.

The *address* option is the starting address to fill. *Length* specifies the number of bytes, halfwords, words, or quads to fill. *Pattern* is the data to enter into the memory locations. If not specified, *pattern* defaults to zero.

3.4.8.3 Options

Option	Description
-b	Fill memory as bytes
-h	Fill memory as halfwords (16 bits)
-w	Fill memory as words (32 bits)
-q	Fill memory as quadwords (64 bits)
-p	The *address* parameter is a physical address
-v	The *address* parameter is a virtual address

3.4.8.4 Example

```
CFE> f 80000000 1000 ee
*** command status = 0
CFE> f –q –v A0000000 5000 0
*** command status = 0
```

3.4.9 FLASH

3.4.9.1 Usage

```
flash [options] source-file [destination-device]
```

3.4.9.2 Description

Updates the system's flash or EEPROM device with the specified file. Under most circumstances, the file to be written to flash must be the output of the *mkflashimage* program in the *cfe/hosttools* directory. The *mkflashimage* program writes a header on the front of the flash image that contains version information, the file's length and a CRC to prevent inadvertently writing a bad image into the system's boot ROM.

The *source-file* parameter may be a device name or a network file name. If *source-file* refers to a flash device, the *destination-device* is written with a copy of the data in the source device. If *source-file* refers to a serial device such as a UART, CFE will read S-records from the UART. The S-records must still be generated on the output of *mkflashimage* so that CFE can verify the CRC. Finally, if *source-file* refers to a network file name, in the form *host:path/filename.flash* CFE will use TFTP to obtain the file and write it into the destination device.

The *destination-device* parameter may be a flash device or an EEPROM. If not specified, *destination-device* defaults to *flash0*, which is typically the boot ROM.

3.4.9.3 Options

Option	Description
`-noheader`	Do not look for the header from *mkflashimage*. Use this carefully!
`-offset=value`	Copy the image file to an area starting at offset *value* in the flash device. You can specify hex values by using the "0x" notation. You should specify an offset that corresponds to a sector boundary to avoid erasing sectors that contain valid data.

3.4.9.4 Example

```
CFE> flash flash0 flash1
CFE> flash myhost:cfe_bins/latest_cfe.flash
CFE> flash uart0 flash0
```

3.4.10 GO

3.4.10.1 Usage

```
go [options] [address]
```

3.4.10.2 Description

Starts execution of a program that was loaded by the *load* command. If *address* is specified, CFE will transfer control to the specified address and ignore the start address of the loaded program.

3.4.10.3 Options

Option	Description
`-noclose`	Do not close network devices before executing the program. You can use this if you expect the program to return immediately to the firmware and do not want to reinitialize the network interface.

3.4.10.4 Example

```
CFE> go
(program begins execution)
```

3.4.11 HELP

3.4.11.1 Usage

```
help command-name
```

3.4.11.2 Description

Displays help about CFE commands. With no parameter, CFE will display a summary of all available commands.

3.4.11.3 Options

None

3.4.11.4 Example

```
CFE> help help

  SUMMARY

     Obtain help for CFE commands

  USAGE

     help [command]

     Without any parameters, the 'help' command will display a summary
     of available commands.  For more details on a command, type 'help'
     and the command name.

*** command status = 0
CFE>
```

3.4.12 IFCONFIG

3.4.12.1 Usage

```
ifconfig [options] device
```

3.4.12.2 Description

Configures the specified network interface. The *ifconfig* command activates the network interface, sets the IP addresses and other parameters, enabling other network-related commands such as *ping*, and *boot* (from network devices). Most of *ifconfig*'s functionality is accessed through options. The parameter *device* is the name of an Ethernet device.

Only one network device may be activated at any given time. If you enable a different network device while one is active, the active one will be deactivated first.

This command also sets the `NET_DEVICE`, `NET_DOMAIN`, `NET_IPADDR`, `NET_NETMASK`, `NET_GATEWAY`, and `NET_NAMESERVER` environment variables.

3.4.12.3 Options

Option	Description
`-auto`	Configure the interface automatically via DHCP
`-off`	Deactivates the interface.
`-addr=a.b.c.d`	Specifies the IP address of the interface.
`-mask=a.b.c.d`	Specifies the subnet mask of the interface
`-gw=a.b.c.d`	Specifies the default gateway of the interface.
`-dns=a.b.c.d`	Specifies the name server for the interface
`-domain=string`	Specifies the default domain name for DNS queries (for example, "broadcom.com")
`-speed=string`	Specifies the speed and duplex for the network interface, overriding automatic detection. Valid speed settings are auto, 10fdx, 10hdx, 100fdx, 100hdx, 1000fdx, and 1000hdx.
`-loopback=mode`	Specifies loopback mode options for the interface. Valid settings are off, internal, and external. External loopback will cause the Ethernet interface to enable loopback in the PHY. Internal loopback causes loopback within the controller itself.
`-hwaddr=xxx`	Overrides the system default hardware address for the interface.

3.4.12.4 Example

```
CFE> ifconfig eth0 -auto
eth0: Link speed: 100BaseT FDX
Device eth0:  hwaddr 40-00-00-00-01-00, ipaddr 10.21.2.10,
      mask 255.255.255.0, gateway 10.21.2.1,
      nameserver 10.21.192.10, domain broadcom.com
*** command status = 0
CFE> ifconfig eth0 -off
Device eth0 has been deactivated.
*** command status = 0
CFE>
```

3.4.13 LOOP

3.4.13.1 Usage

```
loop [options] "command"
```

3.4.13.2 Description

Causes CFE to execute the specified command repeatedly. The parameter *command* should be placed in quotes if it contains spaces or other punctuation.

3.4.13.3 Options

Option	Description
`-count=nnn`	Repeat *nnn* times.

3.4.13.4 Example

```
CFE> loop -count=5 "e 80000000 55"
*** command status = 0
```

3.4.14 MAP PCI

[SENTOSA and RHONE boards only]

3.4.14.1 Usage

```
map pci offset size paddr [-off] [-l2ca] [-matchbits]
```

3.4.14.2 Description

Maps a region of local physical memory to appear at the specified offset relative to the BAR0 PCI register. This is used on a CPU operating in device mode to cause local memory to be visible to the host processor. The memory at *paddr* for *size* bytes is made available at *offset* bytes relative to the BAR0 register.

3.4.14.3 Options

Option	Description
`-off`	Turn off the specified mapping
`-l2ca`	Make the region L2 cacheable
`-matchbits`	Use match bits policy for new region

3.4.14.4 Example

Make first 1MB of memory visible to host processor.

```
CFE> map pci 0 0x100000 0
*** command status = 0
CFE>
```

3.4.15 MEMORYTEST

[64-bit CFE version only]

3.4.15.1 Usage

```
memorytest [-cca=x] [-arena=x] [-stoponerror] [-loop]
```

3.4.15.2 Description

Tests all available memory. This command causes CFE to query the arena for available memory blocks (blocks not used by the firmware) and tests them using a pattern designed to help find addressing and data path problems. By default, the memory is tested using uncached accelerated writes to produce a predictable data pattern on the bus. By using the "-cca=5" switch, you can cause cacheable accesses to be used to test the memory and this will cause reads to be interleaved with writes as the cache is evicted.

3.4.15.3 Options

Option	Description
`-loop`	Loop forever or until a key is pressed
`-stoponerror`	Stop if error occurs while looping
`-cca=*`	Use specified MIPS cacheability attribute
`-arena=*`	Test only the specified block in the arena. In a "show memory" display, zero would be the first memory block, one the second, etc.

3.4.15.4 Example

```
CFE> memorytest
Available memory arenas:
phys = 0000000000000000, virt = B800000000000000, size = 000000000FE4F000

Testing memory.

Testing: phys = 0000000000000000, virt = B800000000000000, size =
000000000FE4F000
Writing: a/5/c/3
Reading: a/5/c/3
Writing: address|5555/inv/aaaa|address
Reading: address|5555/inv/aaaa|address

*** command status = 0
CFE>
```

3.4.16 MEMTEST

3.4.16.1 Usage

```
memtest [options] start-addr length
```

3.4.16.2 Description

Executes a very crude memory test, writing various patterns into memory and reading the results back for verification. Patterns are written 64 bits at a time on 64-bit boundaries.

The *start-addr* parameter is the beginning address in memory for the memory test. *Length* is the number of bytes to test.

3.4.16.3 Options

Option	Description
`-p`	The *start-addr* parameter is a physical address
`-v`	The *start-addr* parameter is a virtual address (default)
`-loop`	Loop continuously until a keypress or a failure

3.4.16.4 Example

```
CFE> memtest 80000000 10000
Pattern: 0000000000000000
Pattern: FFFFFFFFFFFFFFFF
Pattern: 5555555555555555
Pattern: AAAAAAAAAAAAAAAA
Pattern: 0000000000000000
Pattern: FF00FF00FF00FF00
Pattern: 00FF00FF00FF00FF
*** command status = 0
```

3.4.17 PHY DUMP

[SiByte CPUs only]

3.4.17.1 Usage

```
phy dump macid [register]
```

3.4.17.2 Description

Dumps the contents of the PHY registers on the specfied MAC (*macid*). If you also specify the *register* parameter, only the specified register will be displayed.

3.4.17.3 Options

Option	Description
`-phy=x`	Specifies the PHY address. This value is usually 1.

3.4.17.4 Example

```
CFE> phy dump 0
** PHY registers on MAC 0 PHY 1 **
Reg 0x00  =  0x1140   |  Reg 0x01  =  0x7969
Reg 0x02  =  0x0020   |  Reg 0x03  =  0x6071
Reg 0x04  =  0x01E1   |  Reg 0x05  =  0x40A1
Reg 0x06  =  0x0007   |  Reg 0x07  =  0x2001
Reg 0x08  =  0x0000   |  Reg 0x09  =  0x0300
Reg 0x0A  =  0x0000   |  Reg 0x0B  =  0x0000
Reg 0x0C  =  0x0000   |  Reg 0x0D  =  0x0000
Reg 0x0E  =  0x0000   |  Reg 0x0F  =  0x3000
Reg 0x10  =  0x0002   |  Reg 0x11  =  0x0301
Reg 0x12  =  0x0000   |  Reg 0x13  =  0x0000
Reg 0x14  =  0x0000   |  Reg 0x15  =  0x0000
Reg 0x16  =  0x0000   |  Reg 0x17  =  0x0000
Reg 0x18  =  0x0420   |  Reg 0x19  =  0xF314
Reg 0x1A  =  0x0406   |  Reg 0x1B  =  0xFFFF
Reg 0x1C  =  0x0000   |  Reg 0x1D  =  0x03AA
Reg 0x1E  =  0x0000   |  Reg 0x1F  =  0x0000
*** command status = 0
CFE> phy set 0 0 0x1140
```

3.4.18 PHY SET

[SiByte CPUs only]

3.4.18.1 Usage

```
phy set macid regnum value
```

3.4.18.2 Description

This command writes values to the PHYs connected to the BCM1250 Ethernet controller. The 16-bit value *value* is written to register number *regnum* (0..31) on the specifeid *macid* (0..2). If your PHY is not strapped at address 1, you can specify the PHY address with the –*phy* switch.

3.4.18.3 Options

Option	Description
`-phy=x`	Specifies the PHY address. This value is usually 1.

3.4.18.4 Example

```
CFE> phy set 0 0 0x1140
Wrote 0x1140 to phy 1 register 0x00 on mac 0
*** command status = 0
CFE>
```

3.4.19 PING

3.4.19.1 Usage

```
ping [options] host
```

3.4.19.2 Description

Sends ICMP echo messages to the specified host and waits for a reply. The network interface must be configured and operational (via *ifconfig*) for the ping command to work. Unlike many IP implementations, loopback is not treated specially. If the network interface is configured for loopback mode, *ping* will still transmit the packets through the interface. You can use ping as a quick test of the functionality of the interface.

3.4.19.3 Options

Option	Description
`-t`	Ping forever, or until the ENTER key is struck
`-x`	Exit immediately on the first error (use with –f or –t)
`-s=nnn`	Specify packet size in bytes
`-c=nnn`	Specify number of packets to echo
`-A`	Don't abort even if a key is pressed
`-E`	Require all packets sent to be returned for a successful return status

3.4.19.4 Example

```
CFE> ping myserver.broadcom.com
myserver.broadcom.com is alive
*** command status = 0
CFE>
```

3.4.20 PRINTENV

3.4.20.1 Usage

```
printenv
```

3.4.20.2 Description

Displays a table of the current environment variables and their values.

3.4.20.3 Options

None

3.4.20.4 Example

```
CFE> printenv
Variable Name        Value
-------------------- --------------------------------------------------
ETH0_HWADDR          40:00:00:00:01:00
net                  ifconfig eth0 -auto
ethdiag              boot -elf -fs=raw flash1:
F1                   f1 key macro
F3                   howdy doody!
CFE_VERSION          0.0.10
CFE_BOARDNAME        CSWARM
CFE_MEMORYSIZE       64
BOOT_CONSOLE         uart0
NET_DEVICE           eth0
NET_DOMAIN           broadcom.com
NET_IPADDR           10.21.2.10
NET_NETMASK          255.255.255.0
NET_GATEWAY          10.21.2.1
NET_NAMESERVER       10.21.192.10
*** command status = 0
```

3.4.21 RESET

[SiByte CPUs only]

3.4.21.1 Usage

```
reset [-yes] –softreset|-cpu|-unicpu1|-unicpu0|-sysreset
```

3.4.21.2 Description

Resets the system via the SCD. The switches indicate the type of reset desired (at least one must be specified). If you reset into uniprocessor mode (*-unicpu0* or *–unicpu1*), a BCM1250 processor will behave more like a BCM1125.

3.4.21.3 Options

Option	Description
`-yes`	Don't ask for confirmation
`-softreset`	Soft reset of the entire chip including all bus agents
`-cpu`	Reset just the CPU cores
`-unicpu0`	When resetting, restart in uniprocessor mode on CPU0
`-unicpu1`	When resetting, restart in uniprocessor mode on CPU1
`-sysreset`	Full system reset

3.4.21.4 Example

```
CFE> reset -sysreset -yes

CFE version 1.0.32 for SWARM (64bit,MP,LE)
Build Date: Thu Jul 11 11:38:59 PDT 2002 (mpl@hardy.sj.broadcom.com)
Copyright (C) 2000,2001,2002 Broadcom Corporation.

. . . etc.
```

3.4.22 SAVE

3.4.22.1 Usage

```
save [options] file-name start-addr length
```

3.4.22.2 Description

The *save* command uses the TFTP protocol to write a region of memory to a remote file. This can be useful if you are running test programs that log performance data or other results in memory and wish to postprocess them on the host.

The *filename* parameter is similar to the *load* command, in the format *host:filename*. See the examples below.

Like the *load* command, the network must be configured via the *ifconfig* command before this command is used.

3.4.22.3 Options

Option	Description

3.4.22.4 Example

```
CFE> save host:file_name 80000000 1000
4096 bytes written to host:filename
*** command status = 0
```

3.4.23 SET CONSOLE

3.4.23.1 Usage

```
set console device-name
```

3.4.23.2 Description

Switches CFE's console to the specified device. Care should be taken to ensure that the new console device is operational.

3.4.23.3 Options

None

3.4.23.4 Example

```
CFE> set console promice0
```

3.4.24 SET DATE

3.4.24.1 Usage

```
set date date
```

3.4.24.2 Description

Sets the current month, day, and year into the real-time-clock. The format of the date should be *mm/dd/yyyy*.

3.4.24.3 Options

none

3.4.24.4 Example

```
CFE> set date 10/04/2001
*** command status = 0
CFE>
```

3.4.25 SET TIME

3.4.25.1 Usage

```
set time time
```

3.4.25.2 Description

Sets the current hour, minute, and second into the real-time clock. The format of the time should be *hh:mm:ss*, where *hh* is a 24-hour time.

3.4.25.3 Options

none

3.4.25.4 Example

```
CFE> set time 18:12:01
*** command status = 0
CFE>
```

3.4.26 SETENV

3.4.26.1 Usage

```
setenv [options] varname value
```

3.4.26.2 Description

Sets the value of an environment variable. Optionally, the variable will also be stored in persistent storage (flash, EEPROM). Enclose the *value* in quotes if it contains spaces or other punctuation.

3.4.26.3 Options

Option	Description
`-ro`	Mark the variable read-only, preventing it from being changed in the future. The only way to delete a read-only variable is to erase the NVRAM, so use this carefully. It is intended for storing Ethernet hardware addresses, serial numbers, and other data that does not change.
`-p`	Set the environment variable permanently in the NVRAM device. Without this option, an environment variable will be retained in DRAM only and will not survive a system restart.

3.4.26.4 Example

```
CFE> setenv MYNAME “Hi there”
*** command status = 0
CFE>
```

3.4.27 SHOW AGENTS

[SiByte CPUs only]

3.4.27.1 Usage

```
show agents
```

3.4.27.2 Description

Display the list of agent names that are valid for the *show soc* command.

3.4.27.3 Options

None

3.4.27.4 Example

```
CFE> show agents
Available SOC agents: MC, L2, MACDMA, MACRMON, MAC, DUART, GENCS, GEN,
GPIO, SMBUS, TIMER, SCD, BUSERR, DM, IMR, SYNCSER, SERDMA
*** command status = 0
CFE>
```

3.4.28 SHOW BOOT

3.4.28.1 Usage

```
show boot device-name
```

3.4.28.2 Description

Display the boot block from the specified block device (disk, CD-ROM). You can use this command to verify that the boot block is correctly formatted and the checksums are correct.

3.4.28.3 Options

None

3.4.28.4 Example

```
CFE> show boot ide0:
```

3.4.29 SHOW DEFEATURE

[SiByte CPUs only]

3.4.29.1 Usage

```
show defeature
```

3.4.29.2 Description

Displays the current value of CPU0's defeature register.

3.4.29.3 Options

None

3.4.29.4 Example

```
CFE> show defeature
Defeature mask is currently 00080000
*** command status = 0
CFE>
```

3.4.30 SHOW DEVICES

3.4.30.1 Usage

```
show devices
```

3.4.30.2 Description

Displays a list of the device drivers that have been configured in CFE.

3.4.30.3 Options

None

3.4.30.4 Example

```
CFE> show devices
Device Name          Description
-------------------  ------------------------------------------------------
uart0                SB1250 DUART at 0x10060000 channel 0
promice0             PromICE AI2 Serial Port at 0x1FDFFC00
eeprom0              Xicor X1241 EEPROM on SMBus channel 1
uart1                SB1250 DUART at 0x10060000 channel 1
flash0               New CFI flash at 1FC00000 size 2048KB
flash1               New CFI flash at 1F800000 size 2048KB
eeprom1              Microchip 24LC128 EEPROM on SMBus channel 0 dev 0x50
eth0                 SB1250 Ethernet at 0x10064000 (00-02-4C-FE-09-32)
eth1                 SB1250 Ethernet at 0x10065000 (00-02-4C-FE-09-33)
ide0.0               IDE disk unit 0 at 100B3E00
pcmcia0              PCMCIA ATA disk unit 0 at 11000000
clock0               Xicor X1241 RTC on SMBus channel 1
*** command status = 0
CFE>
```

3.4.31 SHOW FLASH

3.4.31.1 Usage

```
show flash devicename [-sectors]
```

3.4.31.2 Description

Displays information about the specified flash device. If *–sectors* is specified, also displays the offsets of the flash sectors.

3.4.31.3 Options

Option	Description
-sectors	Display flash sector information

3.4.31.4 Example

```
CFE> show flash flash1
FLASH: Base 000000001F800000 size 00200000 type 03(Flash) flags
00000001
NVRAM: Not supported by this flash
*** command status = 0
CFE>
```

3.4.32 SHOW HEAP

3.4.32.1 Usage

```
show heap
```

3.4.32.2 Description

Displays statistics about CFE's internal heap.

3.4.32.3 Options

None

3.4.32.4 Example

```
CFE> show heap

Total bytes:       1048576
Free bytes:        896240
Free nodes:        2
Allocated bytes:   142816
Allocated nodes:   236
Largest free node: 895240
Heap status:       CONSISTENT

*** command status = 0
CFE>
```

3.4.33 SHOW MEMORY

3.4.33.1 Usage

```
show memory [options]
```

3.4.33.2 Description

Displays the contents of the *arena*, CFE's physical memory map. The arena describes the physical layout of memory, including which areas of memory are available for operating system use and which are in use by the firmware or devices.

3.4.33.3 Options

Option	Description
`-a`	Display all entries, not just available DRAM

3.4.33.4 Example

```
CFE> show memory -a
Range Start  Range End    Range Size     Description
------------ ------------ -------------- --------------------
000000000000-000003EAAFFF (000003EAB000) DRAM (available)
000003EAB000-000003FFFFFF (000000155000) DRAM (in use by firmware)
000004000000-00000FFFFFFF (00000C000000) Memory Controller (unused)
000010000000-00001002FFFF (000000030000) I/O Registers
000010030000-00001FBFFFFF (00000FBD0000) Reserved
00001FC00000-00001FDFFFFF (000000200000) ROM
00001FE00000-00003FFFFFFF (000020200000) Reserved
000040000000-00007FFFFFFF (000040000000) LDT/PCI
000080000000-00009FFFFFFF (000020000000) Memory Controller (unused)
0000A0000000-0000BFFFFFFF (000020000000) Reserved
0000C0000000-0000CFFFFFFF (000010000000) Memory Controller (unused)
0000D0000000-0000D7FFFFFF (000008000000) Reserved
0000D8000000-0000DFFFFFFF (000008000000) LDT/PCI
0000E0000000-0000F7FFFFFF (000018000000) Reserved
0000F8000000-0000FFFFFFFF (000008000000) LDT/PCI
000100000000-007FFFFFFFFF (007F00000000) Memory Controller (unused)
008000000000-00F7FFFFFFFF (007800000000) Reserved
00F800000000-00F9FFFFFFFF (000200000000) LDT/PCI
00FA00000000-00FCFFFFFFFF (000300000000) Reserved
00FD00000000-00FFFFFFFFFF (000300000000) LDT/PCI
*** command status = 0
CFE>
```

3.4.34 SHOW PCI

3.4.34.1 Usage

```
show pci [options] [bus/dev/func]
```

3.4.34.2 Description

Display information about the devices attached to the PCI and LDT buses. If specified, the *bus/dev/func* parameter indicates a particular device to display information about. If no parameter is specified, all attached devices are displayed.

The *show pci* command can also be used to rescan the bus. Use this with caution.

3.4.34.3 Options

Option	Description
`-v`	Display verbose information
`-init`	Reinitialize and rescan the PCI and LDT buses

3.4.34.4 Example

```
CFE> show pci
PCI bus 0 slot 0/0: vendor 0x166d product 0x0001 (host bridge, revision
0x01)
*** command status = 0
CFE>
```

3.4.35 SHOW SOC

[SiByte CPUs only]

3.4.35.1 Usage

```
show soc agent-name [instance [section]]
```

3.4.35.2 Description

Displays BCM1250 SOC registers in an easy-to-read format. This command is not expected to be included in production firmware, but it can be useful during bringup.

The *agent-name* parameter is an agent name. The list of valid agent names can be found by using the *show agents* command.

The *instance* parameter specifies a numeric instance (for example, there are three MAC agents, 0, 1, and 2). The *section* parameter further breaks down the set of registers to display. For example, the DMA controller has sets of registers for each channel and direction.

3.4.35.3 Options

Option	Description
`-v`	Display register fields where available
`-all`	Display all registers in all agents. You may get a trap on the functional simulator when using this option, since it does not implement all of the registers.

3.4.35.4 Example

```
CFE> show soc macdma 0 tx0
Register Name                  Address     Value
-----------------------------  ----------  ------------------
MACDMA 0 TX0 Config 0          0x10064C00  0000_0000_0008_0000
MACDMA 0 TX0 Config 1          0x10064C08  0000_0000_0000_0000
MACDMA 0 TX0 Descriptor Base   0x10064C10  0000_0000_03EB_C770
MACDMA 0 TX0 Descriptor Count  0x10064C18  0000_0000_0000_0000
MACDMA 0 TX0 Cur DSCR_A        0x10064C20  8002_0200_03EC_4FE0
MACDMA 0 TX0 Cur DSCR_B        0x10064C28  00A8_0000_0000_0003
MACDMA 0 TX0 Cur Dscr Addr     0x10064C30  0000_0000_03EB_C790
*** command status = 0
CFE> show soc scd
Register Name                  Address     Value
-----------------------------  ----------  ------------------
SCD System Revision            0x10020000  0000_0000_1250_01FF
SCD System Config              0x10020008  0000_0000_0048_0500
SCD Perf Cnt Config            0x100204C0  0000_0000_0000_0000
SCD Perf Counter 0             0x100204D0  0000_0000_0000_0000
SCD Perf Counter 1             0x100204D8  0000_0000_0000_0000
SCD Perf Counter 2             0x100204E0  0000_0000_0000_0000
SCD Perf Counter 3             0x100204E8  0000_0000_0000_0000
*** command status = 0
CFE>
```

3.4.36 SHOW SPD

[SiByte CPUs only]

3.4.36.1 Usage

```
show spd smbuschan smbusdev
```

3.4.36.2 Description

Display the contents of the specified *serial-presence-detect* ROM on the SMBus. This command can be useful in debugging memory problems.

3.4.36.3 Options

Option	Description
`-v`	Display entire SPD content in hex

3.4.36.4 Example

```
CFE> show spd 0 0x54
memtype  = 07 (7)          | [2 ] Memory type
rows     = 0D (13)         | [3 ] Number of row bits
cols     = 0A (10)         | [4 ] Number of column bits
sides    = 02 (2)          | [5 ] Number of sides
width    = 48 (72)         | [6 ] Module width
banks    = 04 (4)          | [17] Number of banks
tCK25    = 7.0             | [9 ] tCK value for CAS Latency
tCK20    = 7.5             | [23] tCK value for CAS Latency
tCK10    = 0.0             | [25] tCK value for CAS Latency
rfsh     = 0x82            | [12] Refresh rate (KHz)
caslat   = 0x0C            | [18] CAS Latencies supported
attrib   = 0x20            | [21] Module attributes
tRAS     = 2D (45)         | [30]
tRP      = 20.00           | [27]
tRRD     = 15.00           | [28]
tRCD     = 20.00           | [29]
tRFC     = 00 (0)          | [42]
tRC      = 00 (0)          | [41]
*** command status = 0
```

3.4.37 SHOW TEMP

3.4.37.1 Usage

```
show temp [options]
```

3.4.37.2 Description

Displays the CPU temperature, for boards equipped with a temperature sensor.

3.4.37.3 Options

Option	Description
`-continuous`	Write messages to the console as the temperature changes
`-stop`	Stop writing messages as the temperature changes

3.4.37.4 Example

```
CFE> show temp
Temperature:  CPU: 50C  Board: 21C  Status:00 [ ]
*** command status = 0
CFE>
```

3.4.38 SHOW TIME

3.4.38.1 Usage

```
show time
```

3.4.38.2 Description

Displays the current time from the real time-clock.

3.4.38.3 Options

none

3.4.38.4 Example

```
CFE> show time
Current date & time is:  10/04/2001 14:10:22
*** command status = 0
CFE>
```

3.4.39 U (unassemble)

3.4.39.1 Usage

```
u [options] address [length]
```

3.4.39.2 Description

Disassembles instructions. *Address* is the starting address and *length* is the number of instructions to disassemble. If not specified, *address* and *length* will be the ending address and length from the previous disassemble command.

3.4.39.3 Options

Option	Description
-p	The *address* parameter is a physical address
-v	The *address* parameter is a virtual address

3.4.39.4 Example

```
CFE> u bfc01000 5
FFFFFFFFBFC01000: 00000000    sll     zero,zero,#0
FFFFFFFFBFC01004: 241b1001    addiu   k1,zero,#4097
FFFFFFFFBFC01008: 135b0004    beq     k0,k1,0xffffffffbfc0101c
FFFFFFFFBFC0100C: 00000000    sll     zero,zero,#0
FFFFFFFFBFC01010: 241a0008    addiu   k0,zero,#8
*** command status = 0
CFE>
```

3.4.40 UNSETENV

3.4.40.1 Usage

```
unsetenv varname
```

3.4.40.2 Description

Deletes the specified environment variable. If the variable was stored in persistent storage (EEPROM, flash) it is removed from persistent storage as well..

3.4.40.3 Options

None

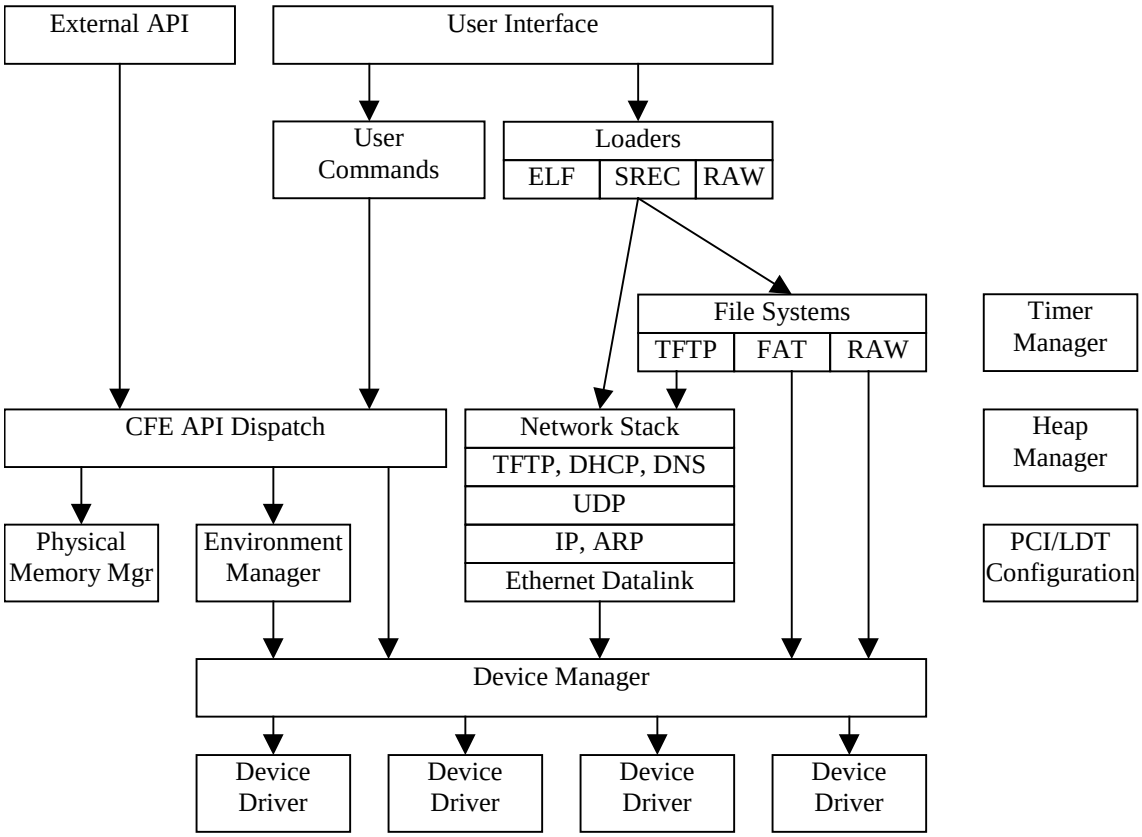
3.4.40.4 Example

```
CFE> unsetenv FOO
*** command status = 0
CFE>
```

4. Software Internals

4.1 Module Overview

The diagram below highlights some of the main modules within CFE:



4.2 Library Modules

CFE makes use of a number of "standard" C runtime library functions. The *lib/* directory contains the routines that make up the standard "C" functions.

The routines in the library are minimal (only those functions that are actually used by the firmware are present in the library).

4.3 System Startup

CFE's startup module is the assembly-language file *arch/mips/common/src/init_mips.S*. This module is responsible for getting the CPU ready to run the main "C" code of the firmware. *Init_mips* does the following:

- Perform any early initialization specific to the board CFE is running on *(board_init.S)*
- Initialize the CPU (CPU, CP0, FPU, and the TLB) *(arch/mips/common/src/init_mips.S)*
- Initialize the L1 cache *(arch/mips/cpu/sb1250/src/sb1250_l1cache.S)*
- Initialize the L2 cache *(arch/mips/cpu/sb1250/src/sb1250_l2cache.S)*
- Begin executing code in cached space *(arch/mips/common/src/init_mips.S)*
- Initialize the DRAM controller, including ECC if that is configured. The DRAM initialization can read the DRAM parameters from the SPD modules on the DIMMs if they are present. *(arch/mips/cpu/sb1250/src/sb1250_draminit.c)*
- Copy the initialized data from the ROM to the RAM. *(arch/mips/common/src/init_mips.S)*
- Zero the BSS area. *(arch/mips/common/src/init_mips.S)*
- If CFE is built to be relocatable, the data segment is relocated to the end of physical RAM, and all the internal references to data are fixed up to point to their new locations. *(arch/mips/common/src/init_mips.S)*
- Initialize the "C" stack and set up GP. *(arch/mips/common/src/init_mips.S)*
- Start the firmware. *(arch/mips/common/src/cfe_main.c)*

The *init_mips* module and related files in *arch/mips/cpu/sb1250* also includes:

- The basic trap & exception handlers. Since CFE does not make use of hardware interrupts, all exceptions are considered errors. Unhandled exceptions are currently handled by restarting the firmware.[2] The code dispatched by the exception handlers are in the CPU-specific area. For the bcm1250, exceptions are handled by *arch/mips/cpu/sb1250/src/exception.S*.
- The TLB exception handler (in the CPU area, at *arch/mips/cpu/sb1250/src/sb1250_cpu.S*) This TLB handler manages the *boot area*, a virtual address space assigned for boot loaders (see sectoion 8.2)
- If CFE is configured for multiple CPUs, it will call routines in a CPU-specific module to initialize the caches on secondary CPUs and keep those CPUs in a "holding pattern" until they are needed by the OS.
- The external API vector is also located here. When a boot loader calls CFE's API, the requests are routed through this vector to the "C" code.

[2] This could definitely be improved. Simple access violations inside of CFE should not warrant a restart.

4.4 Multiprocessor startup

The diagram below illustrates the startup process when using multiple processor cores:

CPU #0	**CPU #1..n**
Low-level board init	Held in reset
Basic CPU initialization (including CP0, FPU, etc.)	
L1 Cache init	
L2 Cache init	
Start secondary CPU	CPU ‘n’ starts
Wait for secondary CPUs to complete cache init	Basic CPU initialization (including CP0, FPU, etc.)
	L1 Cache init
	Notify CPU 0
CPU0 resumes execution	Wait for CPU 0 to initialize memory
Switch to KSEG0 to run cached	
Initialize memory controller	
Compute data segment relocation	
Copy data from ROM to RAM	
Zero BSS	
Fix up data segment relocations	
Notify other CPUs, pass GP (relocation value)	CPU ‘n’ resumes execution
Set up “C” stack	Switch to KSEG0 (cached)
CFE main program started	Enter idle loop (wait for command to jump to user code)

The code to handle the secondary processor cores startup sequence is in *arch/mips/cpu/sb1250/src/sb1250_altcpu.S*. Most of the calls into this module are made from *init_mips.S*.

4.5 "Bi-Endian" Startup

It is possible to build a variant of CFE that will operate regardless of the system endianness. That is, you can set the system for either big or little-endian and CFE will start from the same boot ROM.

When configured via CFG_BIENDIAN in the Makefile, the assembler places a special sequence of instructions at each of the exception vectors. These instructions are valid in either endianness, but decode differently. In particular, the opcode 0x10000014 decodes as "b .+0x54" when read as a big-endian instruction and "bne zero,zero,.+0x44" when read as a little-endian instruction. The latter case will be interpreted by the processor as a NOP.

Depending on system endianness, the code will either branch to the big-endian vector or fall through. The fall-through code contains instructions to jump to the corresponding little-endian vector 1MB into the ROM (at address 0xBFD00000). See the comments in *arch/mips/common/include/mipsmacros.h* for some more information.

To build bi-endian firmware, use the rules invoked by the "biend" target of the SWARM board's Makefile as follows:

```
gmake clean biend
```

This will produce two copies of CFE, one big, and one little. The big-endian version will have the special exception vectors, and the little-endian version will have its text segment set to start at 0xBFD00000. The *mkflashimage* program combines these two images together to make a single file you can flash into a ROM.

If your ROM is smaller than 2MB or you want to locate the little-endian firmware elsewhere, you will need to edit the following files:

File	Edit
arch/mips/common/src/init_mips.S	Change the value of `BIENDIAN_LE_BASE` to the actual base address that you will locate the little-endian ROM.
arch/mips/common/src/ cfe_rom_reloc_cached_biendian.lds	Change the base address of the text segment (and its absolute load address accordingly) to the base address you will locate the little-endian ROM.
hosttools/mkflashimage.c	Change the value of `CFE_BIENDIAN_LE_OFFSET` to the displacement into the ROM where the `BIENDIAN_LE_BASE` can be found.

4.6 Heap Manager

The heap manager is a very simple memory allocator for use by CFE to allocate and manage small memory objects. It is initialized early in CFE's startup process and is given a fixed amount of RAM to manage. It can allocate objects with a specified size and memory alignment, which can be very useful when dealing with devices that have alignment requirements.

Note: The heap manager is simple, but not particularly efficient. For example, it makes passes over all allocated objects to coalesce free regions. Its allocation policy is "first fit", which can lead to inefficiencies in some degenerate cases. In the unlikely event that you need high-performance memory allocation, consider allocating one block from the heap manager and manage pieces of it locally.

CFE uses the heap manager to create one heap (the default heap) at startup. You can also use it to create additional heaps by calling the functions in *lib/lib_malloc.c* directly.

The main calls to the heap manager are:

```
void *KMALLOC(int size,int align);
void KFREE(void *ptr);
```

KMALLOC allocates an object with the specified size (in bytes) and alignment. The alignment must be a power of 2, or zero if you have no specific alignment requirements. *KMALLOC* will return NULL if there is no memory left, but generally this condition is considered fatal for CFE.

KFREE will return an object to the heap.

The heap is implemented in the file *lib/lib_malloc.c*.

4.7 Physical Memory Manager

The *physical memory manager* keeps track of the physical address space and the attributes of ranges within that space. Internally, it is called the *arena* (named after an ancient DOS data structure).

It is *not* a memory allocator. The arena simply assigns attributes to address ranges, overwriting attributes previously assigned.

During startup, the physical memory manager will first create an arena to describe the BCM1250's entire 40-bit physical address map (this is done in the CPU-specific module), and then assign attributes to the ranges that correspond to physical memory, boot ROM, device registers, PCI and LDT spaces, etc.

At some point later, in CFE's generic code, CFE assigns attributes for the memory space that it consumes (its ROM and RAM requirements).

Later, when the OS loads, it can use CFE's external API to query the arena for ranges of a specific type, such as "available DRAM", to determine what regions of physical memory are available for OS use.

The arena is implemented in *lib/lib_arena.c* and is used in *main/cfe_mem.c*

4.8 Device Manager

The *device manager* maintains a simple list of device drivers that can be used by CFE as consoles, bootstrap devices, NVRAM storage, etc. For simplicity, CFE does not implement a "device tree" or other complex data structure to describe the devices it manages. It is expected that most implementations of CFE will include device support only for those devices needed for system bootstrap. The operating system's probe routines will be responsible for discovering all the devices that are present.

Devices are named with simple names such as "uart0" and "eth0", with a short alphabetic prefix followed by a unit number. Some devices are more complex, such as SCSI chains, and they may use multiple numeric suffixes to identify the device. For example, "scsi0.1.6" might mean "SCSI bus 0, unit 1, LUN 6."

The device manager is implemented in *main/cfe_attach.c*

For more information about how to write a device driver, see chapter 7.

4.9 Console Interface

The console interface provides a simple mechanism for displaying messages and reading user input from a CFE device. CFE console devices must be of the type "serial."

The following calls are useful for accessing the console:

```
xprintf(char *str,…)
console_readline(char *str,int len)
console_log(char *str,…)
```

The *xprintf* routine is a simple implementation of *printf*. It does not include all of the templates usually associated with printf, and it includes a few that are not, hence the different name. (there is a macro in the library to allow programs which call the real *printf* to work, but you should be careful with the templates. The table below lists the templates that are implemented in *xprintf*.

Template	Description
%s	String values.
%c	A single character
%d	A signed number
%u	An unsigned number

%x, %X	A hexadecimal number. Capital X will display capital letters for the hex digits (A-F)
%p, %P	A hexadecimal number, sized to the width of a pointer
%b	A byte (same as %02X)
%w	A 16-bit word (same as %04X)
%a	An Ethernet address, in the form `xx-xx-xx-xx-xx-xx`. To use this template, pass the address of a 6-byte buffer.
%I	An IP address, in the form nnn.nnn.nnn.nnn – to use this template, pass the IP address as the address of a 4-byte buffer.
%p, %P	A pointer, appropriately sized for 32-bit and 64-bit versions of CFE
%Z	Dump a buffer. To use this template, supply two arguments, a length in bytes and a buffer address. For example, *xprintf("%Z",10,buffer)* will display a hex dump of 10 bytes starting at *buffer*.

The templates %s, %d, %u, and %x take the following modifiers:

Modifier	**Description**
nn	Field width, for example %30s. Numbers will be right justified in the field, strings will be left-justified in the field.
0nn	For numeric templates, fill the leading spaces with '0' characters.
l	For numeric templates, indicates that the argument is of a "long" type.
ll	For numeric templates, indicates that the argument is of a "long long" (64-bit) type.

The *xprintf* routine does not support floating point.

The *console_log* routine is useful for "background" processes like polling loops. It works with the *console_readline* routine to avoid disturbing the current state of input on the console, so you can display messages (for example, Ethernet link status changes) without disrupting a partially-entered command line.

The *console_readline_noedit* routine reads a line of text from the console (similar to *gets*). It provides rudimentary line editing (basically, backspace) and will return when a complete line of text is entered. You can call *console_readline* to get line input with editing.

The console interface is implemented in *main/cfe_console.c*.

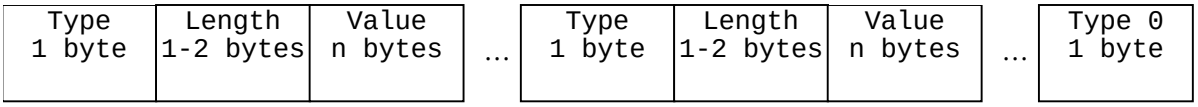
The *xprintf* routines are implemented in *lib/lib_printf.c*.

4.10 Environment Manager

The *environment manager* is responsible for tracking simple textual environment variables and other small values and storing them in persistent storage such as an EEPROM or flash device.

The environment manager is normally associated with a flash or NVRAM device driver during startup.

On the NVRAM device, the environment manager represents the environment strings as a set of TLV (type/length/value) encoded structures. The TLVs are represented on the NVRAM device as follows:

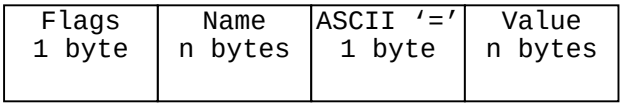


The length field represents the length of the data that follows the length field, not including the type and length fields. The low-order bit of the *type* field indicates whether the *length* field is one or two bytes. If the low-order bit is one the length field is one byte. If the low-order bit is zero, the length field is two bytes, with the high-order byte stored first.

The following types are defined:

TLV Type	**Symbol**	**Description**
0x00	`ENV_TLV_TYPE_END`	End. This is the last byte in a list of TLVs.
0x01	`ENV_TLV_TYPE_ENV`	Environment. This TLV is an environment variable's value.
0x80..0xEF	`User defined`	TLVs with the high-order bit set are reserved for use by customers. You should still set the low-order bit to indicate whether the TLV uses a one or two-byte length field.
0xF0..0xFF	`Reserved`	Reserved for future use.

Each environment variable's value has the following format:



The *flags* field indicates special flags to describe the environment variable. The flags may be OR'd together, but not all combinations make sense.

Flag	**Symbol**	**Description**
0x01	`ENV_FLG_BUILTIN`	"built-in" environment variable. These variables are *not* stored to the NVRAM (they are used at run time).
0x02	`ENV_FLG_READONLY`	"read only." These variables may not be altered or deleted after they are set the first time. This is useful for variables that contain serial numbers and Ethernet MAC addresses that are allocated once

		and never changed.

You can use either an EEPROM or a flash memory device to store the environment. In either case, the underlying device driver must implement the `IOCTL_NVRAM_GETINFO` call so that the environment manager can determine what region of the flash or EEPROM device it is permitted to use for environment storage (typically, one sector is enough).

Environment functions are implemented in *main/env_subr.c*. The routines to write the environment to the nonvolatile device are in *main/nvram_subr.c*.

4.11 Timer Manager

The *timer manager* provides a simple mechanism for device drivers and the network subsystem to keep track of the passage of time. Because CFE does not use interrupts for anything, even the timer is polled. The MIPS CP0 COUNT register is polled periodically to accumulate ticks into the global tick count.

The macro *POLL()* should be called periodically to keep the tick count up-to-date. At 800MHz, the counter will overflow every 5 seconds or so.

The global time is maintained in the variable *cfe_ticks* and represents the number of ticks since startup. This tick count is maintained in units of *CFE_CLOCKSPERTICK*, which defaults to 10 ticks per second.

The timer manager maintains a list of routines that need to be called periodically. You can use the *cfe_bg_add* routine to add a routine to the background processing list.

The timer manager includes some macros to make it easy to wait for device timeouts. An example device timeout is shown below:

```
int64_t timer;

TIMER_SET(timer,5*CFE_HZ);    /* wait for 5 seconds */
while (!TIMER_EXPIRED(timer)) {
      POLL();
      /* do stuff to device */
      }
```

The timer routines are implemented in *main/cfe_timer.c* and *main/cfe_background.c*

4.12 Network Subsystem

CFE includes support IPV4 networking for network bootstrap. The functions are relatively limited, but it should be easy to add support for more protocols and features should that become necessary.

The network subsystem consists of all of the files in the *net/* directory. It implements the following standards:

- Ethernet V2 (DIX) style datalink interface and protocol dispatch
- Address Resolution Protocol (ARP) (RFC826)
- Internet Protocol (IP) (RFC791)
- Internet Control Message Protocol (ICMP) (RFC792)
- User Datagram Protocol (UDP) (RFC768)
- Transmission Control Protocol (TCP)
- Trivial File Transfer Protocol (TFTP) (RFC1350)
- Domain Name System Protocol (query only) (DNS) (RFC1035)
- Dynamic Host Configuration Protocol (DHCP) (RFC2131)

The network subsystem is designed to be simple and portable, and operates on top of CFE's device interface for access to the Ethernet driver. Therefore, it is possible to re-use these components to write a second-level boot loader.[3]

The top-level interface to the network subsystem is in the file *net/net_api.c*. It includes high-level functions for activating the IP stack and transferring data.

The TCP stack is normally not configured into CFE, since it is generally not used. To enable the TCP stack, set CFG_TCP to 1 in your *bsp_config.h* file.

4.13 File Systems

CFE internally supports the notion of a "file system," but the concept is greatly scaled back for simplicity. Essentially, a CFE file system is a collection of routines for reading named files from some boot device. The boot device need not be a disk. The file systems allow common code to be used for parsing ELF headers and other formatted files.

CFE currently includes support for the following file systems:

Name	Description
`tftp`	The *tftp* filesystem provides access to files on a remote host's TFTP server. You can open, read, and seek (forward only) in a tftp file. The TFTP filesystem uses the default network device (the device must be configured before you attempt to open a file).
`fat` `rfat`	The *fat* filesystem provides access to files formatted with an MS-DOS style FAT16 or FAT12 filesystem. These filesystems are typically found on floppy disks, compactflash cards, and other small block devices. FAT32 support is currently *not* provided. There are two variants of the *fat* filesystem. The standard one, *fat*, is used with media that supports a hard-disk-style partition

[3] Of course, separating the network code into a second-level boot loader has not been tested yet, but it *is* possible.

	boot block. The *rfat* filesystem assumes there is no such boot block. *Fat* filesystems are found on hard drives, ZIP drives, and PCMCIA Flash cards. *Rfat* filesystems are on standard floppy disks.
`raw`	The *raw* filesystem provides access to a raw block device. If you stored an ELF file on a disk, tape, or flash memory, you can read the file from that device without having any standard on-disk structure. File names on raw devices represent byte offsets into the device.[4]

File systems are stored in a global file system table and may be accessed with the following call:

```
const fileio_dispatch_t *cfe_findfilesys(const char *name)
```

You can obtain a dispatch table by name.

4.14 PCI/LDT Configuration

The PCI and LDT configuration module is called during CFE's startup. It is responsible for initializing the PCI and LDT host (and subordinate) bridges, locating devices, and setting up the BARs. Operating systems need not reconfigure the PCI or LDT bus; it is expected that they can use the configuration that CFE has provided.

The main call (in *main/cfe_main.c*) is:

```
void pci_configure(flags);
```

This routine configures and initializes the PCI and LDT buses and displays some diagnostic information about devices present on the bus. Once initialized, device drivers within CFE can query for the presence of PCI and LDT devices and operating system software can scan configuration space to locate the devices that were configured.

CFE provides some basic functions to allow device drivers to query the bus and obtain the contents of registers in configuration space. These functions and macros are declared in *pci/pcivar.h*.

4.15 User Interface

The user interface routines were designed to make it easy to add new commands without changing a centralized command table. Internally, CFE maintains a tree of words, with pointers to command functions at the leaves of the tree. Commands must be unique (you cannot have just a "show" command if you also have "show devices") but otherwise you can make the command syntax as complex as you want.

[4] Well, eventually.

4.15.1 Adding a command

To add a command, call the *cmd_addcmd* routine from your *board_finalinit* routine as follows:

```
int cmd_addcmd(char *command,
               int (*func)(ui_cmdline_t *,int argc,char *argv[]),
               void *ref,
               char *help,
               char *usage,
               char *switches);
```

For example:

```
cmd_addcmd("arp",
       ui_cmd_arp,
       NULL,
       "Display or modify the ARP Table",
       "arp [-d] [ip-address] [dest-address]\n\n"
       "Without any parameters, the arp command will display the contents of the\n"
       "arp table.  With two parameters, arp can be used to add permanent arp\n"
       "entries to the table (permanent arp entries do not time out)",
       "-d;Delete the specified ARP entry.  If specified, ip-address\n"
       "may be * to delete all entries.");
```

Note: pay special attention to the presence or absence of commas to separate the string parameters in the above function call. Some of the strings are very long and are spread over several lines of source text.

The *command* parameter is the name of the command (use spaces to separate words). CFE will break this into tokens to find the appropriate place in the tree for the command descriptor.

The *func* parameter is a pointer to the function to call when the command is executed. The *ref* parameter will be placed in the *ui_cmdline_t* structure as an extra pointer. Most CFE functions do not use the *ref* parameter.

The *help* parameter is a one-line description of the command. It should not contain any newlines.

The *usage* parameter is the command's verbose description text. You can embed newline (\n) characters in this string and CFE will format the text appropriately when the *help* command is used to display the text.

The *switches* parameter serves to functions: it lists the valid switches for the command and also the help text for each of the switches. The general format of the *switches* parameter is:

```
-swname;description|-swname=*;description…
```

Each switch description is separated by a pipe (|) character. A semicolon separates switch names from their descriptions. If a switch accepts parameter data, you should list the switch in the form "-swname=*" so that CFE will know to expect a value for the switch.

4.15.2 Calling the command function

CFE will call the command function when your command is entered using the following prototype:

```
int (*func)(ui_cmdline_t *cmd,int argc,char *argv[]);
```

The *argc* and *argv* parameters are similar to what you would expect in "C" programs, except *argv[0]* is the first argument, not *argv[1]*.. The *argc* parameter indicates the number of arguments that followed your command when it was entered, with zero meaning no additional arguments were supplied.

The *cmd* parameter is a pointer to the context data structure that describes your command as it was parsed by the command interpreter. You can pass this value back to the command routines to obtain information about switches, values, and parameter values, using the routines in the following list:

- `int cmd_sw_value(ui_cmdline_t *cmd,char *swname,char **swvalue)`
  Looks up the switch *swname*. If present, fills in *swvalue* with a pointer to the switch's value and returns TRUE. Otherwise, returns FALSE.

- `int cmd_sw_isset(ui_cmdline_t *cmd,char *swname)`
  Returns TRUE if *swname* is present, FALSE otherwise.

- `char *cmd_getarg(ui_cmdline_t *cmd,int argnum)`
  Obtains the value of argument *argnum*. This is the same as *argv[argnum]* except *cmd_getarg* will return `NULL` if *argnum* is out of range.

- `char *cmd_sw_name(ui_cmdline_t *cmd,int swidx)`
  Gets the name of the nth (*swidx*) switch supplied for this command. This can be used if you want to allow the same switch to be specified multiple times.

- `int cmd_sw_posn(ui_cmdline_t *cmd,char *swname)`
  Obtains the position that *swname* appears relative to the arguments. If *swname* appears before the first argument, this function returns zero. Returns –1 if the switch was not specified at all.

When your function returns, it should return zero for a successful return status, or else one of the error codes in *cfe/include/cfe_error.h*.

5. The BCM1250 Reference Designs

There are two primary BCM1250 reference designs: *SWARM* (BCM912500A) and *SENTOSA* (BCM12500E). *SWARM* is an ATX-style board, and *SENTOSA* is a PCI card.

This chapter has an overview of the major features of these two designs.

5.1 Board Description (SWARM)

5.1.1 Features

The BCM912500A Reference Design (*SWARM*) is a demonstration board for the BCM1250 SOC. Some of the features of this board include:

- The BCM1250 Processor
- Four DDR SDRAM DIMM slots (two on each memory controller)
- Two Gigabit Ethernet ports with Broadcom BCM5411 PHYs
- PCI bus with two 32-bit 33/66Mhz slots
- LDT connector for user expansion
- LDT-to-PCI bridge (API Networks “Sturgeon”)
- PCI bus attached to LDT bridge with two 64-bit 66Mhz slots
- Opti USB controller connected to PCI bus
- 2Mbytes Flash for bootstrap
- Direct connector for Grammar Engine PromICE ROM emulator
- Audio codec connected to a synchronous serial port
- Four-character LED display connected to BCM1250 Generic Bus
- IDE disk interface connected to BCM1250 Generic Bus
- PCMCIA slot
- Philips video codec connected to 8-bit FIFO interface
- Maxim 1617A temperature sensor (SMBUS)
- Microchip 24LC128C serial EEPROM
- Xicor X1241 time-of-day clock and serial EERPOM, or ST Micro M41T81 clock
- Two UART ports
- EJTAG connector

5.1.2 Jumpers and Settings

Refer to the BCM912500A documentation for a complete list of jumper settings. The table below lists the important jumper settings and switch values for getting started with CFE:

Jumper	Default	Description
SW1	0	Configuration switch (see below)
SW2	0	CPU Core Frequency
J1	Installed	Selects big-endian operation.
J45	Removed	SYSRESET. Controls reset pin on SWARM peripherals
J41	Removed	COLD_RESET. Controls the cold reset input to the BCM1250.
J42	Removed	WARM_RESET. Controls the warm reset input the BCM1250.
SW3,4,5	0	Frequency dividers for the LDT bridge
J19	All Removed	When installed, USB controller will be available. When removed, USB controller is disabled. Since the USB controller is a 33Mhz PCI device, you need to remove the jumpers to test 66Mhz PCI operation with a card in the PCI slot.
J31	Installed	These jumpers are used to connect the ROM chip selects (CS0 and CS1) to the flash ROM or ROM emulator connectors, respectively. If the jumpers are installed such that they are perpendicular the ROM Emulator connector, the emulator will be on CS0 and the flash will be on CS1. If you turn the jumpers 90 degrees such that they are parallel to the ROM emulator connector, the flash appear on CS0 and the ROM Emulator will appear on CS1. On most newer-revision SWARM boards, there is only one jumper for J31. Install the jumper to use the ROM emulator on CS0, and remove the jumper for the flash on CS0.

The configuration switch can be used to set certain run-time parameters for CFE. Set the switch according to the table below:

Setting	Description
0	Console on UART 0 at 115200 baud. PCI/LDT will not be configured.
1	Console on PromICE virtual serial port. PCI/LDT will not be configured.
2	Console on UART 0 at 115200 baud. PCI/LDT will be configured
3	Console on PromICE virtual serial port. PCI/LDT will be configured.
4	not used
5	not used
6	Console on UART 0 at 115200 baud, LDT configured in *slave mode* for dual-hosted chains.
7	"Safe mode." Console on UART 0 at 115200 baud. CFE will not read the NVRAM contents but will allow you to write them. You can use this to recover from a corrupt NVRAM.
8-F	not used

5.1.3 Firmware Devices

The table below lists the CFE devices that are available in the SWARM version of CFE:

Device	Description
uart0	UART Channel A (bottom UART connector)
uart1	UART Channel B (top UART connector)
promice0	PromICE virtual serial port
eeprom0	Serial ROM on the Xicor X1241 clock. This eeprom is used to store the environment variables
eeprom1	Serial ROM on the Microchip 24LC128. You can boot from this serial ROM by putting a jumper on J2
eth0	Ethernet controller connected to MAC0 (J12)
eth1	Ethernet controller connected to MAC1 (J13)
ide0	IDE disk controller connector (J35)
pcmcia0	PCMCIA interface (J36)
flash0	Flash connected to CS0 (either the boot flash or the ROM emulator, depending on the setting of J31)
flash1	Flash connected to CS1 (either the boot flash or the ROM emulator, depending on the setting of J31)
clock0	The real-time clock section of the Xicor X1241 clock.

5.2 Addresses of onboard peripherals

This section contains a summary of the SWARM address map. These constants are defined in the file *swarm.h*.

5.2.1 Generic Bus Assignments

Address	Size	Chip Select	Description
0x1FC00000	2MB	CS0	Boot ROM
0x1F800000	2MB	CS1	Alternate Boot ROM
		CS2	Unused
0x100A0000	64KB	CS3	LED Display
0x100B0000	64KB	CS4	IDE Disk
		CS5	Unused
0x11000000	64MB	CS6	PCMCIA interface
		CS7	Unused

5.2.2 GPIO Signals

Signal	Direction	Description
GPIO0	Output	Debug LED

GPIO1	Output	Sturgeon NMI
GPIO2	Input	PHY Interrupt
GPIO3	Input	Nonmaskable Interrupt (SW11)
GPIO4	Input	IDE Disk Interrupt
GPIO5	Input	Temperature sensor alert
GPIO6	N/A	Used by PCMCIA
GPIO7	N/A	Used by PCMCIA
GPIO8	N/A	Used by PCMCIA
GPIO9	N/A	Used by PCMCIA
GPIO10	N/A	Used by PCMCIA
GPIO11	N/A	Used by PCMCIA
GPIO12	N/A	Used by PCMCIA
GPIO13	N/A	Used by PCMCIA
GPIO14	N/A	Used by PCMCIA
GPIO15	N/A	Used by PCMCIA

5.3 Board Description (SENTOSA)

5.3.1 Features

The BCM912500E Reference Design (*SENTOSA*) is a demonstration board for the BCM1250 SOC. Some of the features of this board include:

- The BCM1250 Processor
- 256MB of DDR memory soldered to the board (128MB on each memory channel)
- Two Gigabit Ethernet ports with Broadcom BCM5411 PHYs
- PCI device mode (card is configured as a PCI device, you can install it in a PC or other host).
- 2Mbytes Flash for bootstrap
- Direct connector for Grammar Engine PromICE ROM emulator
- Maxim 1617A temperature sensor (SMBUS)
- Two Microchip 24LC128C serial EEPROMs
- Xicor X1241 time-of-day clock and serial EERPOM, or ST Micro M41T81 clock
- One UART port
- EJTAG connector
- Samtec connector for LDT expansion

5.3.2 Jumpers and Settings

Refer to the BCM912500E documentation for a complete list of jumper settings. The table below lists the important jumper settings and switch values for getting started with CFE:

Jumper	Default	Description
SW1	0	Configuration switch (see below)
SW4-1	Off	ON to boot from PromICE, OFF to boot from flash
SW4-2	On	ON for big-endian, OFF for little-endian
SW2		COLD RESET. Press to reset the BCM1250
SW3	NMI	Wired to GPIO4 – if software configures this, you can cause use it to cause an NMI

The configuration switch can be used to set certain run-time parameters for CFE. Set the switch according to the table below:

Setting	Description
0	Console on UART 0 at 115200 baud. PCI/LDT will not be configured.
1	Console on PromICE virtual serial port. PCI/LDT will not be configured.
2	Console on UART 0 at 115200 baud. PCI/LDT will be configured
3	Console on PromICE virtual serial port. PCI/LDT will be configured.
4	not used
5	not used
6	Console on UART 0 at 115200 baud, LDT configured in *slave mode* for dual-hosted chains.
7	"Safe mode." Console on UART 0 at 115200 baud. CFE will not read the NVRAM contents but will allow you to write them. You can use this to recover from a corrupt NVRAM.
8-F	not used

5.4 Board Description (RHONE)

5.4.1 Features

The BCM91125E Reference Design (*RHONE*) is a demonstration board for the BCM1125/H SOC. Some of the features of this board include:

- The BCM1125/H Processor
- 128MB of DDR memory soldered to the board
- Two Gigabit Ethernet ports with Broadcom BCM5421 PHYs
- PCI device mode (card is configured as a PCI device, you can install it in a PC or other host).
- 16Mbytes Flash for bootstrap
- Direct connector for Grammar Engine PromICE ROM emulator
- Maxim 6654 temperature sensor (SMBUS)
- Two Microchip 24LC128C serial EEPROMs

- ST Micro M41T81 real-time clock
- One UART port
- EJTAG connector
- Samtec connector for LDT expansion
- Four character LED display connected to BCM1125/H Generic Bus

5.4.2 Jumpers and Settings

Refer to the BCM91125E documentation for a complete list of jumper settings. The table below lists the important jumper settings and switch values for getting started with CFE:

Jumper	Default	Description
SW1	0	Configuration switch (see below)
SW4-1	Off	ON to boot from PromICE, OFF to boot from flash
SW4-2	On	ON for big-endian, OFF for little-endian
SW2		COLD RESET. Press to reset the BCM1250
SW3	NMI	Wired to GPIO4 – if software configures this, you can cause use it to cause an NMI

The configuration switch can be used to set certain run-time parameters for CFE. Set the switch according to the table below:

Setting	Description
0	Console on UART 0 at 115200 baud. PCI/LDT will not be configured.
1	Console on PromICE virtual serial port. PCI/LDT will not be configured.
2	Console on UART 0 at 115200 baud. PCI/LDT will be configured
3	Console on PromICE virtual serial port. PCI/LDT will be configured.
4	not used
5	not used
6	Console on UART 0 at 115200 baud, LDT configured in *slave mode* for dual-hosted chains.
7	"Safe mode." Console on UART 0 at 115200 baud. CFE will not read the NVRAM contents but will allow you to write them. You can use this to recover from a corrupt NVRAM.
8-F	not used

5.4.3 Firmware Devices

The table below lists the CFE devices that are available in the SENTOSA version of CFE:

Device	Description
uart0	UART Channel A (bottom UART connector)
promice0	PromICE virtual serial port
eeprom0	Serial ROM on the Xicor X1241 clock. This eeprom is used to store the environment variables
eth0	Ethernet controller connected to MAC0
eth1	Ethernet controller connected to MAC1
flash0	Flash connected to CS0 (either the boot flash or the ROM emulator, depending on the setting of SW4-1)
flash1	Flash connected to CS1 (either the boot flash or the ROM emulator, depending on the setting of SW4-1)
clock0	The real-time clock section of the Xicor X1241 clock.

5.4.4 Addresses of onboard peripherals

This section contains a summary of the SENTOSA address map. These constants are defined in the file *sentosa.h*.

5.4.5 Generic Bus Assignments

Address	Size	Chip Select	Description
0x1FC00000	2MB	CS0	Boot ROM
0x1F800000	2MB	CS1	Alternate Boot ROM
		CS2	Unused
		CS3	Unused
		CS4	Unused
		CS5	Unused
		CS6	Unused
		CS7	Unused

5.4.6 GPIO Signals

Signal	Direction	Description
GPIO0	Output	Debug LED
GPIO1	N/A	Not used
GPIO2	N/A	Not used
GPIO3	N/A	Not used

GPIO4	N/A	Not used
GPIO5	N/A	Not used
GPIO6	N/A	Not used
GPIO7	N/A	Not used
GPIO8	N/A	Not used
GPIO9	N/A	Not used
GPIO10	N/A	Not used
GPIO11	N/A	Not used
GPIO12	N/A	Not used
GPIO13	N/A	Not used
GPIO14	N/A	Not used
GPIO15	N/A	Not used

5.5 Loading CFE via a ROM Emulator

When developing your own version of CFE, or adapting CFE's routines for your own firmware, you can use the BCM912500A's ROM Emulator connector to connect a Grammar Engine *PromICE*. See http://www.gei.com for more details on Grammar Engine products.

The PromICE model used most commonly with the BCM912500A is the P1160-90, a 16Mbit (2Mbyte) emulator with 90ns access time.

If you have another ROM emulator, it should be simple to adapt the connector on the BCM912500A for other emulation products. The ROM emulator connector (J30) is 5V-tolerant. You can get the pinout from the schematic.

To connect a PromICE to the BCM912500A, make the following connections:

- Attach the emulation cable to J30 such that it exits towards the Ethernet connectors. Be sure pin 1 is correctly aligned. *Note: Most of GEI's emulation connectors have pin 1 marked on the wrong end of the cable. Be sure pin 1 on the PromICE unit itself is on the same side of the connector as pin 1 on the BCM912500A.*
- Place jumpers on the EXT and 32 pins on the PromICE unit. Remove the jumper from the ROM pins if present.
- Set the jumpers on J31 so that they are perpendicular to the ROM Emulator connector.
- Connect the PromICE to your host computer.
- If you want to use the virtual console port, or want to use the emulated ROM as RAM, connect the write jumper from J67 pin 1 to the MWR pin on the PromICE.

Below is a sample *loadice.ini* file for the PromICE. This assumes you have connected your PromICE to the parallel port on your PC:

```
output=com1
pponly=LPT1
```

```
rom 2M
word 8
burst 0
file cfe.srec
ailoc 1FFC00,19200
aidirt
```

Press the COLD RESET button (SW9) on the BCM912500A after reloading the PromICE to start the new firmware.

5.6 Installing a new version of the firmware into the flash

If you have the ROM Emulator attached, you can easily copy the contents of the ROM Emulator to the onboard flash by using the following CFE command:

```
CFE> flash flash0 flash1
```

This command causes CFE to copy the contents of flash0 (the ROM Emulator) to flash1 (the onboard flash chip). After you have copied the data, you can rotate J31 90 degrees and the BCM912500A will boot from the onboard flash.

If you want to upgrade CFE without a ROM emulator and have generated a *cfe.flash* file, you can place this file on your TFTP server and do:

```
CFE> ifconfig eth0 …    /* configure Ethernet interface */
CFE> flash hostname:path/to/cfe.flash flash0
```

Once the flash update is complete, you can restart your board to run the newly-installed CFE.

6. Porting CFE to a new design

6.1 Tools required for building CFE

The Broadcom tool chain is used for building CFE from sources. Among the compilers supplied, the *mips64-sb1sim* toolchain is appropriate for building firmware, boot loaders, and other standalone applications.

The *mips64-sb1sim* tools are somewhat of a misnomer, since resulting binaries work just fine on actual hardware. You will probably run into build issues if you use the Linux or NetBSD toolchains to compile CFE.

For a "quick start", you can compile the CFE source for the SWARM or SENTOSA (BCM1250 Reference Design) by following the steps shown below:

```
setenv PATH /path/to/mips64-sb1sim-tools/bin:$PATH
cd cfe/swarm
gmake
```

The build procedure should produce the following files:

File	Description
`cfe`	CFE binary (ELF executable)
`cfe.bin`	CFE binary (ROM image file)
`cfe.srec`	CFE binary (S-records for a ROM emulator or programmer)
`cfe.flash`	Flash update file (can be put on a TFTP server and downloaded to a target to update its flash)
`cfe.flash.srec`	S-Records for the flash update file (cfe.flash) – this is used if you want to update the flash of a target that does not support TFTP. In this case, you update the flash via a serial port.
`cfe.dis`	Disassembly output
`cfe.map`	Linker map file

6.2 Directory structure

The top of the CFE source tree contains the following directories:

Path	Contents
`cfe/`	Main CFE sources (see next section)
`docs/`	Documentation, including this file
`build/`	Build areas

6.2.1 The build directory (build/)

The build area, under "build/" contains the makefiles for various targets. In particular, the directory "build/broadcom" contains the build areas for supported BCM1250 reference designs.

Path	Contents
`broadcom/swarm/`	Broadcom evaluation board build area
`broadcom/sentosa/`	Broadcom evaluation board build area
`broadcom/rhone/`	Broadcom evaluation board build area
`broadcom/sim/`	A build area for a variant of CFE suitable for use under the functional simulator.
`broadcom/vcs/`	A minimal build area for making a variant of CFE suitable for execution under the RTL simulator
`algor/p5064`	A port of CFE to the Algorithmics P5064
`algor/p6064`	A port of CFE to the Algorithmics P6064

6.2.2 The CFE source directory (cfe/)

Below this level, in the "*cfe/*" directory are the following:

Path	Contents
`cfe/arch/mips`	Contains all MIPS architecture processor-related directories. While CFE is not intended for use on other CPU architectures, the directory structure permits it.
`cfe/arch/mips/cpu/sb1250`	Contains BCM1250-related sources and includes. Initialization code, drivers, and include files specific to the BCM1250 are placed here.
`cfe/arch/mips/board/`*`boardname`*	Contains sources used by a particular board (e.g., *swarm*). Sources common to a particular board or variants of similar boards are placed here. The "board

	support package" files that contian initialization routines to personalize CFE with device drivers and custom commands are also placed here.
`cfe/arch/mips/common`	Contains sources and includes common to all MIPS designs. The *init_mips.S* file is common across all MIPS designs.
`cfe/applets/`	Sample programs that call CFE's external API
`cfe/dev/`	Device drivers for the BCM1250's peripherals and other devices
`cfe/include/`	Shared include files
`cfe/lib/`	Standard library functions (strcpy, malloc, etc.)
`cfe/main/`	Main program and startup routines
`cfe/net/`	Network stack (ARP, IP, ICMP, UDP, TFTP, etc.)
`cfe/pccons/`	Special routines for initializing the "PC console"
`cfe/pci/`	PCI and LDT enumeration and configuration code
`cfe/ui/`	User interface functions (implementations of user commands)
`cfe/usb/`	USB host stack for the PC console's keyboard
`cfe/x86emu/`	An X86 emulator (from the Xfree86 source) that is used to initialize a VGA adapter. This code has a BSD-style license.
`cfe/verif/`	Some special routines used for running verification test programs under control of the firmware. Normally these routines are not used.
`cfe/vendor/`	Directory containing vendor extensions to CFE

6.2.3 Board, CPU, and Architecture directories

While CFE is primarily intended for MIPS audiences, particularly users of the BCM1250 and other members of the Broadcom Broadband Processor product line, some changes have been made to the directory tree to reduce the pain of moving CFE to other MIPS processors and other processor architectures.

Unfortunately, this has complicated the directory tree. This section attempts to explain the intent of the different directories to aid you in locating files.

- Architecture (MIPS) specific files are in the *cfe/arch/mips/common* directory. This directory contains code and include files that are common to all MIPS platforms, such as assembly-language macros, the disassembler, and the startup routine.
- CPU (BCM1250) specific files are in the *cfe/arch/mips/cpu/sb1250* directory. This directory contains code and include files that pertain only to the SB1250, such as multiprocessor startup code, cache initialization, memory initialization, and device drivers for on-chip peripherals.

- Board (SWARM) specific files are in the *cfe/arch/mips/board/swarm* directory. This directory contains code and include files that are used on the SWARM board and boards similar to it, such as SWARM-specific device drivers and include files, and the startup routines that personalize CFE to the devices present on the SWARM.

When you create your own BCM1250-based design, you can populate your *cfe/arch/mips/board/xxxx* directory, where *xxxx* is your board name, and then generate a *build/companyname/xxxx* directory to contain the makefile and the build area.

6.3 Makefile flow

CFE's makefile is broken into many small pieces to separate functionality into small modules. The "sub-makefiles" are responsible for communicating the names of object files and C compiler flags to the main makefile.

The makefile that *make* initially reads is in your target's build directory. This makefile reads the platform-independent makefile in *main/cfe.mk* which in turn reads other makefiles. Refer to the table below for the flow of makefiles:

Step	**Makefile**
Make reads the Makefile in your build directory. This makefile sets the TOP macro and some global configuration options and includes the platform-independent Makefile	`Makefile`
The platform-independent Makefile is read	`${TOP}/main/cfe.mk`
The version number is set	`${TOP}/main/cfe_version.mk`
Defaults are set for configuration values not set by the makefile in the build directory	`${TOP}/main/cfe.mk`
Directory names are calculated for the architecture, CPU, and board-specific directories. These directories are also converted into the INCLUDE path for header files and the VPATH for source files	`${TOP}/main/cfe.mk`
The tools are configured from the architecture-specific directory. This makefile names the version of GCC and the specific compiler flags needed for building on the MIPS architecture	`${TOP}/arch/mips/common/src/Makefile`
The main target (all) is defined	`${TOP}/main/cfe.mk`
The architecture Makefile is read. This makefile appends object names to *ALLOBJS*	`${TOP}/arch/mips/common/src/Makefile`
The CPU-specific Makefile is read. This makefile appends object names to *ALLOBJS*	`${TOP}/arch/mips/cpu/sb1250/src/Makefile`
The board-specific Makefile is read. This makefile appends object names to *ALLOBJS*	`${TOP}/arch/mips/board/swarm/src/Makefile`

Some default rules are declared for certain locally-built tools.	`${TOP}/main/cfe.mk`
Finally, the ALL target (in caps) in the original Makefile is referenced, which causes Make to build the files 'cfe' and 'cfe.flash'	`Makefile`
The linker-script Makefile is read to read the rules for building the target.	`${TOP}/main/cfe_link.mk`

6.4 Example Makefile

An example Makefile (in the target directory) appears below:

```
TOP = ../cfe
ARCH = mips
BOARD = swarm
CPU = sb1250

CFG_MLONG64 ?= 0
CFG_LITTLE ?= 0
CFG_RELOC ?= 1
CFG_UNCACHED ?= 0
CFG_VAPI ?= 0
CFG_BOOTRAM ?= 0
CFG_BOARDNAME = "SWARM"
CFG_PCI = 1

include ${TOP}/main/cfe.mk

BSPOBJS = swarm_init.o swarm_devs.o

ALL : cfe cfe.flash
        echo done

include ${TOP}/main/cfe_link.mk
```

6.5 Special source files

Each port of CFE will need at least a few special source files for customization and board initialization. These files generally live in the board support directory for your target board. For example, the special source files for the CSWARM checkout board live in the *arch/mips/board/swarm/* directory.

If you want, you can also place these files in the build directory instead, should you wish to have variants of a particular board that are very similar but differ in device configuration or *bsp_config.h* options. If you do this, you can append special files to the BSPOBJS macro in the

Makefile that lives in the build directory. Since the current directory is first on the search path, files in your build directory should override ones in the board-specific directory.

File	Purpose
`Makefile`	Main makefile for your target. This file has certain build-time options that cannot be placed in a source file.
`bsp_config.h`	Most of CFE's compile-time options are selected in this file.
`board_init.S`	Your board's low-level initialization code. This is called very early in the startup sequence to give your code a chance to set up critical hardware (for example, diagnostic LEDs)
`board_devs.c`	Your board's device startup file. Most other initialization happens in this file. CFE will make calls to routines in this file at various stages of initialization.

You can add as many additional source files as you need for your port and place those files in the board support directory.

6.6 Configuration options

6.6.1 Required Makefile macros

Certain macros in the *Makefile* are required for building CFE. These settings are summarized in the table below:

Option	Description
`TOP`	Set to the directory name of the "cfe" directory (that is, the directory that contains "lib", "arch", and other top-level source directories). For the default distribution, this should be set to "../cfe"
`ARCH`	Should be set to "mips"
`BOARD`	Should be set to the name of the directory under *arch/mips/board* where your board-specific files live. For example, the SWARM board sets this to *swarm*.
`CPU`	Should be set to "sb1250", the name of the directory under *arch/mips/cpu* containing CPU-specific files.

Note that the *BOARD* macro is different from *CFG_BOARDNAME*. This is useful for subtle variants of a board that has common board-specific files.

6.6.2 Options in the Makefile

Options in *Makefile* are things that affect the compiler and build procedure and cannot be placed in a source file.

Option	Description
`CFG_MLONG64`	Set to '1' to build a 64-bit version of CFE, or '0' to build a 32-bit

	version.
CFG_LITTLE	Set to '1' to build a little-endian version of CFE, or '0' to build a big-endian version.
CFG_RELOC	Set to '1' to build a relocatable version of CFE. The relocatable version will automatically move its code and data segments as high in physical memory as is possible, depending on the amount of installed DRAM. Since this can make debugging difficult, it is a good idea to enable this later after your board is operational.
CFG_UNCACHED	Set to '1' to run CFE entirely uncached (from KSEG1). Generally this is not advisable, but it may be useful in some debug environments.
CFG_BOOTRAM	Set to'1' to run CFE from the boot ROM area, using the boot ROM for RAM as well. This assumes you have connected a ROM emulator that supports SRAM emulation or have actual SRAM in the bootstrap location. Since most ROM emulators have some sort of RAM emulation (provided you have attached the write line to your target board) this can be useful for debugging your SDRAM initialization routines.
CFG_VAPI	Set to '1' to configure the "verification API" – this is generally not needed
CFG_BOARDNAME	Sets the name of the board's BSP.
CFG_PCI	Set to '1' to configure support for PCI devices.
CFG_VGACONSOLE	Set to '1' to configure support for VGA consoles
CFG_ZLIB	Configure the decompression library, *zlib*. This enables booting of compressed images via the *–z* switch to the *boot* and *load* commands.
CFG_RAMAPP	Set to '1' to build a version of CFE that runs like an application – it does not initialize the CPU or memory controller and can be loaded like any other application. This version is not suitable for putting in ROM, but can be useful if you want to use CFE's code like a library.
CFG_USB	Set to '1' to configure support for USB.
CFG_DOWNLOAD	Set to '1' to configure download support for PCI devices. On the SENTOSA, this configures CFE to wait for an image to be sent to it via the PCI connector.

6.6.3 Options in the bsp_config.h file

Most configuration options can be placed in *bsp_config.h*. This file is included in many "C" source files and assembly modules to trigger conditional compilation of features and options.

Option	Description
CFG_INIT_L1	Set to '1' to enable initialization of the L1 cache. It is best to leave this set to '1'.
CFG_INIT_L2	Set to '1' to enable initialization of the L2 cache. It is best to leave this set to '1'
CFG_INIT_DRAM	Set this to '1' to initialize the DRAM controller. Unless you have made other arrangements to initialize the DRAM from your board_init module, this should be set to '1'.
CFG_NETWORK	Set this to '1' to include support for the network stack, including network bootstrap.
CFG_TCP	Set this to '1' to include support for a simple TCP stack. Normally, only UDP is supported.
CFG_UI	Set this to '1' to include the user interface. If you set this to '0', you need to launch the boot program yourself at the end of CFE's initialization.
CFG_UNIPROCESSOR_CPU0	For BCM1250 CPUs, start the processor in CPU0 mode, effectively disabling CPU1 and making the part look more like a BCM1125. You must also set CFG_MULTI_CPUS to zero for this to work.
CFG_MULTI_CPUS	Set this to '1' to initialize secondary processor core(s). If you leave this set to '0', the secondary core(s) will be held in reset.
CFG_HEAP_SIZE	Set this to the size of the heap in kilobytes. This memory will be marked 'in use' by the firmware and will not be made available to the operating system, so don't overspecify it.
CFG_SERIAL_BAUD_RATE	Sets the console speed (for serial ports). 115200 is the normal default.
CFG_FATFS	Set this to '1' to include support for FAT file systems.
CFG_DRAM_INTERLEAVE	Set this to '1' to interleave chip selects 0..3 (128 byte interleave). At present there is no way to do 64-byte interleave with the DRAM initialization module
CFG_DRAM_ECC	Set this to '1' to enable ECC. The memory controller will be initialized, all of physical memory will be zeroed, and ECC will be enabled.
CFG_DRAM_SMBUS_CHANNEL	Set this to the SMBus channel (0 or 1) where the serial presence detect ROMs are wired for the memory channels. This is used only if you are making use of the DRAM controller module's default initialization table.
CFG_DRAM_SMBUS_BASE	Set this to the first SMBus address used for serial presence detect modules. This is usually coded in the pullups and pulldowns on the DIMM slots. It is assumed that the SPDs are

	addressed in sequential order (see the comments in sb1250_draminit.c for more info). This is used only if you are making use of the DRAM controller module's default initialization table.
CFG_DRAM_BLOCK_SIZE	Set this to 32, 64, or 128 to specify the amount of column interleaving. This is used only if you are making use of the DRAM controller module's default initialization table. The default is 32.
CFG_DRAM_CSINTERLEAVE	Set this to 0, 1, or 2 to specify the number of bits of chip-select interleaving. This is used only if you are making use of the DRAM controller module's default initialization table.
CFG_VENDOR_EXTENSIONS	Set this to 1 to enable dispatching of vendor extended IOCBs.
CFG_MINIMAL_SIZE	Set this to 1 to disable various debug features (examine/deposit commands, disassembler, etc.), and turn off other features that consume lots of memory (command line recall, etc.). This can substantially reduce the minimum memory footprint of CFE.

6.6.4 Startup Routines

CFE makes several calls to routines in the board support package during initialization. The most important calls are described in the following table:

Routine	Description
board_earlyinit	This routine is called extremely early in CFE's startup. At that time, there will be no DRAM, no cache, no TLB, and the CPU/FPU registers will not be initialized yet. It will be called at a KSEG1 address, so special care must be taken if this routine calls any other routines. CFE is customarily linked to run at a KSEG0 address, so a vanilla "*jal*" instruction will cause you to execute in KSEG0 space before the cache is initialized. (see the discussion in the next section) This routine normally sets up the "generic bus" on the BCM1250 to speed up access to the generic bus and to enable the chip selects for the LEDs or other diagnostic peripherals, if any.
board_draminfo	Return the address of a DRAM information table, if needed. This table is used to control the operation of *sb1250_draminit.c* and lists the rows, columns, banks, and other information about each DRAM module. To use the built-in table, which should be sufficient for applications that use DIMM slots, return zero in the *v0* register. Otherwise, return the address of the table. CFE will be running cached by this point, so it is safe to return a cacheable (KSEG0) address. However, the initialized data

	segment will not have been copied yet, so store the table in the text segment and use a position-independent (not the *la* instruction) means to get the address of the table. See *init_mips.S* and the *LOADREL* macro for an example.
`board_setleds`	This routine is called from many parts of CFE. If you have a small LED display on your board, this routine will be called to put characters on the LED. The A0 register will contain four packed ASCII characters. This routine may be called from both a KSEG0 or KSEG1 address, and is only permitted to use A0 and temporary registers T0..T3. If you don't have an LED, just return to the caller.
`board_console_init`	This routine is called from "C". Its main job is to add the console device (using *cfe_add_device*), and then set the current console to the device name that was added using *cfe_set_console*. After this init routine completes, CFE will be able to display messages on the console.
`board_device_init`	This routine is called from "C". Its job is to add all other devices besides the console (using *cfe_add_device*). It will be called after the PCI bus has been scanned, so device drivers that rely on the PCI bus may be added here.
`board_device_reset`	This routine is called when CFE is re-entered after an operating system exits and attempts a "warm start." It will be called just before the device drivers' *dev_reset* routines are called, to give you a chance to turn off any dangerous I/O activity, like background DMA that should not be running when the firmware is idle.
`board_final_init`	This is the final routine that will be called before CFE displays the command prompt. If you are not using the user interface, this is a good place to call the bootstrap routines to load the operating system. If you are using the user interface, you can add customized commands to the command table at this point.

6.6.5 Special caveats for *board_earlyinit*

All the startup routines except *board_earlyinit* may be implemented in "C". The *board_earlyinit* routine is called before the stack, memory controller, or caches are enabled. It is also called at a KSEG1 address, but CFE is linked at a KSEG0 address.

Each time you want to call a subroutine inside your *board_earlyinit* routine, use the CALLKSEG1 macro (you can find a definition of it in *init_mips.S*). This macro loads the address of the target routine and manually sets the KSEG1 bits in the address to ensure that it will be called in KSEG1, then jumps to the computed value.

6.6.6 Relocatable Code and Data

CFE can be built to relocate its code and data segments automatically to the highest point in physical memory below the 256MB segment of 32-bit addressable space. This is very useful in systems that have removable SDRAM DIMMs, since the total amount of memory may change. Operating systems typically want to use the low physical addresses.

When you activate CFG_RELOC in *Makefile*, the compiler emits code and data references to be "position independent.": All data references are offsets from the GP register, and code references are relative to the current PC.

There a number of limitations to this scheme, the first of which is that you must be careful when writing assembly language routines. Any "*la*" (load address) instruction involving a label will be converted into an addition operation using the GP register. If you need to get the address of something in the text segment before GP is initialized, use the *LOADREL* macro in *init_mips.S* or something similar.

During linking, the compiler will initially locate the data segment at the address specified in the linker script (it will still be stored in the ROM, of course). The linker also supplies a condensed form of the relocation table and stores it in the text segment along with the main program. The *init_mips* module uses these relocations to fix up references to data from initialized data structures after it moves the data segment to its final location. Finally, the GP register itself is relocated and CFE can run normally.

Special care must be taken when using relocatable code and data. The linker will put all structures that contain pointers into the initialized data segment, even if they are declared *const*. This needs to be done in order to apply fixups, since the data and code will both be moved. The initialized data segment must fit completely in the reach of the GP register, which limits its size to about 64 Kbytes. If you declare pointers within the text (code) segment, these pointers will *not* be fixed up, so you must apply the relocation offset manually. An example of this is the "init table" in *arch/mips/common/src/init_mips.S* which contains pointers to init-time routines.

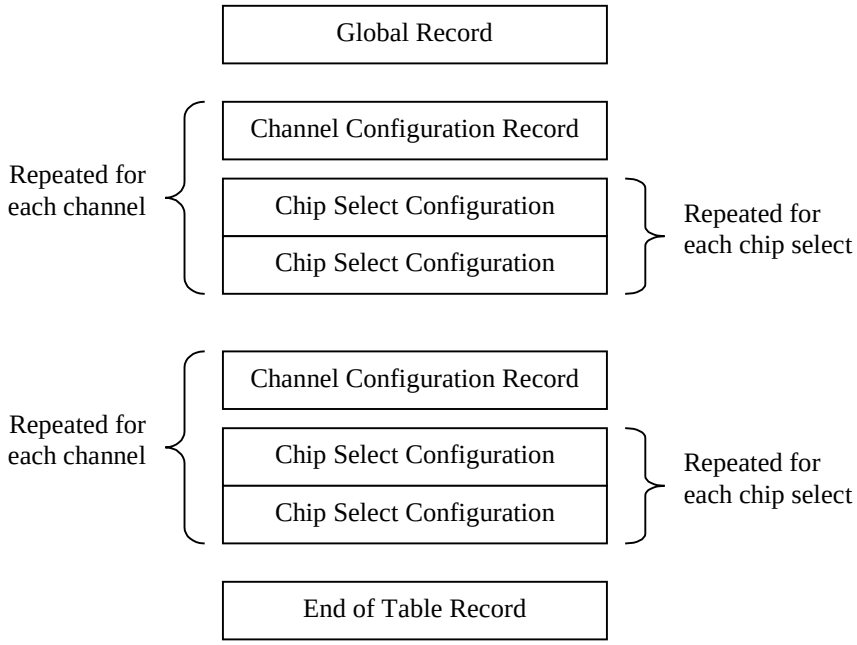
6.7 DRAM Initialization on the BCM1250

6.7.1 DRAM Initialization Table

The code in *sb1250_draminit.c* takes most of the hard work out of initializing the DRAM controller. Under normal circumstances, the built-in defaults can be used to obtain information about the SDRAM DIMMs from the on-module *serial-presence-detect* (SPD) ROMs. For more flexibility, it can be customized by writing a *DRAM initialization table* and supplying the address of this table to the startup routines when the *board_draminfo* routine is called by the startup code.

Note: The DRAM Initialization Table has changed significantly from previous versions of CFE. In particular, it now supports a more general method of specifying DRAM options, moving most of them into the table.

The table is composed of 12-byte records that are declared in *sb1250_draminit.h*. Several records are required to make a complete table, and the order of the records are important for proper initialization. The diagram below illustrates the ordering of records



The following record types are defined:

Record	**Purpose**
DRAM_GLOBALS	Declares global parameters that affect the entire memory system. This should be the first record in the table.
DRAM_CHAN_CFG	Selects a memory channel as the "current" memory channel (for subsequent DRAM_CS_xxx records) and configures parameters related to the memory channel
DRAM_CHAN_CLKCFG	Overrides the default BCM1250 clock configuration values to set the drive strengths and skew values for the memory channel.
DRAM_CHAN_MANTIMING	Overrides all automatic memory timing calculations for this channel. The macro provides the value to be programmed into the timing register.
DRAM_CS_SPD	Selects a chip select for the current memory channel and provides SMBus address information where *draminit* can find the geometry and timing information for the memory attached to this chip select. This is typically used if your memory channel is populated with DIMMs.
DRAM_CS_GEOM	Selects a chip select for the current memory channel and

	provides the geometry information for the memory on that channel. This is used when you have soldered-down memory. Typically, the DRAM_CS_GEOM record is followed by a DRAM_CS_TIMING record to specify the timing parameters that would have been obtained from the SPD on a DIMM.
DRAM_CS_TIMING	Specifies the timing information for the current chip select and the current memory channel. This macro provides the information that would have been obtained from an SPD, and is used in the case where SPDs are not available.
DRAM_EOT	This must be the last record in the table.

The *sb1250_draminit* routine walks through this table to build an internal data structure that represents the memory system, and in particular the relationships between memory controllers, chip selects, and timing data. There is a "current" memory channel and chip select that is maintained by the routine as it processes the records in the table.

The sections below detail the parameters for each record:

6.7.1.1 DRAM_GLOBALS(chintlv)

Specifies global parameters for the memory system. This should be the first record in the table.

Arg	**Value**
chintlv	Nonzero to enable channel interleaving. If this is nonzero, and the geometries all the chip selects on the two memory channels are identical, channel interleaving will be enabled.

6.7.1.2 DRAM_GLOBALS(chintlv)

Specifies global parameters for the memory system. This should be the first record in the table.

Arg	**Value**
chintlv	Nonzero to enable channel interleaving. If this is nonzero, and the geometries all the chip selects on the two memory channels are identical, channel interleaving will be enabled.

6.7.1.3 DRAM_CHAN_CFG(chan,tMEMCLK,dramtype,pagepolicy,blksize,csintlv,ecc,flg)

Selects a memory channel and configures basic parameters. This record should be the first in a group of records for a specific memory controller channel.

Arg	Value
chan	Selects the memory channel number (0 or 1)
tMEMCLK	Specifies the *minimum* value of tMEMCLK, the period of the memory clock. This will set an upper bound on the memory system frequency. This value should be specified in using the *DRT10(units,tenths)* macro – for example, to limit the memory system to 125Mhz, set tMEMCLK to *DRT10(8,0)* for an 8 nanosecond minimum clock.
dramtype	Selects the type of memory attached to this memory channel. Possible values are JEDEC, SGRAM, and FCRAM. JEDEC memory is the value for normal DDR SDRAMs.
pagepolicy	Specifies the page policy to be used for open memory pages. Refer to the BCM1250 manual for the details of each policy. The policy values you can provide here are CLOSED, CASCHECK, HINT, and OPEN.
blksize	Specifies the block size (column interleaving) for this channel – the number of column bits that will appear at the bottom of the address before the bank address bits are introduced. Supported values are 32, 64, and 128 for 0, 1, and 2 bits of column interleaving respectively.
csintlv	Specifies the number of bits of chip select interleaving. Possible values or 0, 1, or 2. Zero disables chip select interleaving. Note that when you use only one bit of chip select interleaving, the memory controller will be configured for mixed-CS mode.
ecc	Set to a nonzero value to enable ECC on this memory controller. If ECC is enabled, *draminit* will zero all of system memory before it returns to CFE.
flg	A bit mask of miscellaneous flags. Supported flags are MCFLG_BIGMEM to enable "large-memory" support (the external decode feature in the 1250's memory controller) and MCFLG_FORCEREG to force *draminit* to include the extra cycle for an external register, even if the SPD on the DIMM reports as an unbuffered device. For large memory systems, both of these are usually specified together, since large memory systems will often need an external register to drive the extra memory devices.

There is also another version of this macro:

DRAM_CHAN_CFG2(chan,tMEMCLK,tROUNDTRIP,dramtype,pagepolicy,blksize,csintlv,ecc,flg)

Arg	Value
tROUNDTRIP	Specifies the *round trip* time of the memory system, in tenths of nanoseconds (use the DRT10 macro as you would for tMEMCLK, shown above). tROUNDTRIP is the round-trip time from the pins on the processor to the memory and back. The default value for this is 2ns.

You should specify either a *DRAM_CHAN_CFG* or *DRAM_CHAN_CFG2* but not both.

6.7.1.4 DRAM_CHAN_CLKCFG(addrskew,dqoskew,dqiskew,addrdrive,datadrive,clkdrive)

Specifies the clock configuration value for the currently selected memory controller. If this record is absent, a default value will be used. The clock configuration parameters you can specify here are all of the drive and skew controls for the memory channel. These values are copied directly into the BCM1250's registers, so refer to the manual for more detailed information on the values you should choose here.

Arg	**Value**
`addrskew`	Address skew value (see the BCM1250 manual)
`dqoskew`	DQO skew value (see the BCM1250 manual)
`dqiskew`	DQI skew value (see the BCM1250 manual)
`addrdrive`	Address drive value and class (see the BCM1250 manual)
`datadrive`	Data drive value and class (see the BCM1250 manual)
`clkdrive`	Clock drive value and class (see the BCM1250 manual)

6.7.1.5 DRAM_CHAN_MANTIMING(tCK,rfsh,tval)

Overrides the timing value for the current memory channel. If you want *draminit* to skip all calculations of timing parameters and load a specific value into the 1250's memory timing registers, use this record.

Arg	**Value**
`tCK`	Configures the memory clock. Enter a BCD value using the DRT10(units,tenths) macro.
`rfsh`	Configures the refresh rate. Use a JEDEC refresh value (JEDEC_RFSH_xxx constants).
`tval`	The 64-bit value to be written to the MC_TIMING1 register in the memory controller. See the BCM1250 user's manual for details on the register's format.

6.7.1.6 DRAM_CS_SPD(csel,flags,chan,dev)

Selects a chip select on the current memory channel and configures the SMBus information for the SPD that contains timing and geometry information. If the DIMM reports as a double-sided device (two chip selects) it is assumed that the 'odd' chip select is also present. Therefore, it is only necessary to use DRAM_CS_SPD records for CS0 and CS2 in systems with two DIMM slots per channel, since a double sided DIMM installed in slot 0 will consume both CS0 and CS1.

Arg	Value
csel	Chip select number (0,1,2,3)
flags	not currently used. Specify 0
chan	SMBus channel number where SPD can be found (0,1)
dev	SMBus device where SPD can be found. SMBus device numbers are 7 bits long – typical values used by the SPDs on DIMMs range from 0x50 through 0x57.

6.7.1.7 DRAM_CS_GEOM(csel,rows,cols,banks)

Selects a chip select on the current memory channel and configures the geometry of the devices connected to the chip select. This record is usually followed by a DRAM_CS_TIMING record to specify timing information.

Arg	Value
csel	Chip select number (0,1,2,3)
rows	Number of row bits
cols	Number of column bits
banks	Number of bank bits (note: not number of banks!)

6.7.1.8 DRAM_CS_TIMING(tCK,rfsh,caslatency,attributes,tRAS,tRP,tRRD,tRCD,tRFC,tRC)

Selects a chip select on the current memory channel and configures the timing for the devices on this channel. These values are essentially the values from the datasheet, converted into a format similar to those usually provided in an SPD ROM on a DIMM. Refer to *sb1250_draminit.h* for the values of constants that can be used for the encoded values.

Arg	Units	Value
tCK	ns (DRT10)	The tCK (memory clock speed)
rfsh	JEDEC_RFSH_xxx	Refresh rate (encoded)
caslatency	JEDEC_CASLAT_xxx	CAS latency value (bitmask)
attributes	JEDEC_ATTRIB_xxx	Attribute bits (usually zero)
tRAS	ns	tRAS – time from ACTIVE to PRECHARGE command
tRP	ns (DRT4)	tRP – PRECHARGE command period
tRRD	ns (DRT4)	tRRD – ACTIVE bank “A” to ACTIVE bank “B” command time
tRCD	ns (DRT4)	tRCD – ACTIVE to READ or WRITE delay
tRFC	ns	tRFC – auto refresh command period. If left zero, this can be calculated automatically from the other parameters.

tRC	ns	tRC. – ACTIVE to ACTIVE/AUTOREFRESH comamnd period. If left zero, this can be calculated automatically from the other parameters.

The ‘units’ field specifies the format that the timing parameters should be entered with. DRT10 is a BCD format, with the upper 4 bits used for the units and the lower 4 bits used for the tenths. DRT4 is a fixed-point format, with the upper 6 bits for the units and the lower 2 bits used for the “quarters.” Examples:

```
DRT10(8,0) = 8.0
DRT10(7,5) = 7.5
DRT4(19,75) = 19.75
DRT4(20,0) = 20.0
```

Be sure to use the correct units for the parameters in the above table, or the memory initialization will not work properly.

6.7.2 Sample draminit tables

This section contains some sample tables for the *sb1250_draminit* routine.

6.7.2.1 SWARM board

The SWARM evaluation board has DIMM slots, so we can get the DIMM geometry and timing information from the SPD ROMs

```
    /*
     * Global data: Interleave mode from bsp_config.h
     */

    DRAM_GLOBALS(CFG_DRAM_INTERLEAVE),        /* do port interleaving if possible */

    /*
     * Memory channel 0: Configure via SMBUS, Automatic Timing
     * Assumes SMBus device numbers are arranged such
     * that the first two addresses are CS0,1 and CS2,3 on MC0
     * and the second two addresses are CS0,1 and CS2,3 on MC1
     */

    DRAM_CHAN_CFG(MC_CHAN0, CFG_DRAM_MIN_tMEMCLK, JEDEC,
                CASCHECK, CFG_DRAM_BLOCK_SIZE, NOCSINTLV, CFG_DRAM_ECC, 0),

    DRAM_CS_SPD(MC_CS0, 0, DEFCHAN, DEVADDR+0),
    DRAM_CS_SPD(MC_CS2, 0, DEFCHAN, DEVADDR+1),

    /*
     * Memory channel 1: Configure via SMBUS
     */

    DRAM_CHAN_CFG(MC_CHAN1, CFG_DRAM_MIN_tMEMCLK, JEDEC,
                CASCHECK, CFG_DRAM_BLOCK_SIZE, NOCSINTLV, CFG_DRAM_ECC, 0),

    DRAM_CS_SPD(MC_CS0, 0, DEFCHAN, DEVADDR+2),
    DRAM_CS_SPD(MC_CS2, 0, DEFCHAN, DEVADDR+3),

   /*
    * End of Table
```

```
*/

DRAM_EOT
```

6.7.2.2 SENTOSA board

The SENTOSA evaluation board has soldered-down memory, so the memory geometries and timing parameters must be specified manually.

```
/*
 * DRAM globals: interleave OK
 */

DRAM_GLOBALS(CFG_DRAM_INTERLEAVE)


/*
 * 128MB on MC 0 (JEDEC SDRAM)
 * Samsung K4H561638B - 16Mx16 chips
 *
 * Minimum tMEMCLK: 8.0ns (125Mhz max freq)
 *
 * CS0 Geometry: 13 rows, 9 columns, 2 bankbits
 *
 * 64khz refresh, CAS Latency 2.5
 * Timing (ns):   tCK=7.50 tRAS=45 tRP=20.00 tRRD=15.0 tRCD=20.0 tRFC=auto tRC=auto
 *
 * Clock Config: Default
 */

 DRAM_CHAN_CFG(MC_CHAN0, DRT10(8,0), JEDEC, CASCHECK, BLKSIZE32,
               CFG_DRAM_CSINTERLEAVE, CFG_DRAM_ECC, 0)
DRAM_CS_GEOM(MC_CS0, 13, 9, 2)
DRAM_CS_TIMING(DRT10(7,5), JEDEC_RFSH_64khz, JEDEC_CASLAT_25,
               0,  45, DRT4(20,0), DRT4(15,0),  DRT4(20,0),  0, 0)


/*
 * 128MB on MC 1 (JEDEC SDRAM)
 * Samsung K4H561638B - 16Mx16 chips
 *
 * Minimum tMEMCLK: 8.0ns (125Mhz max freq)
 *
 * CS0 Geometry: 13 rows, 9 columns, 2 bankbits
 *
 * 64khz refresh, CAS Latency 2.5
 * Timing (ns):   tCK=7.50 tRAS=45 tRP=20.00 tRRD=15.0 tRCD=20.0 tRFC=auto tRC=auto
 *
 * Clock Config: Default
 */

 DRAM_CHAN_CFG(MC_CHAN1, DRT10(8,0), JEDEC, CASCHECK, BLKSIZE32,
                CFG_DRAM_CSINTERLEAVE, CFG_DRAM_ECC, 0)
DRAM_CS_GEOM(MC_CS0, 13, 9, 2)
DRAM_CS_TIMING(DRT10(7,5), JEDEC_RFSH_64khz, JEDEC_CASLAT_25,
                0,  45, DRT4(20,0), DRT4(15,0),  DRT4(20,0),  0, 0)

DRAM_EOT
```

6.7.2.3 Large Memory (external decode mode)

This is an example table for a board with eight DIMM slots, using the “external decode” feature of the BCM1250’s memory controller.

```
/*
 * Global data: Interleave mode from bsp_config.h
 */

DRAM_GLOBALS(CFG_DRAM_INTERLEAVE)               /* do port interleaving if possible */

/*
 * Memory channel 0: Configure via SMBUS, Automatic Timing, Big Memory mode, Force Register
 *
 * There’s an external register on the board, so dimms appear to be "registered" even though
 * the SPD says they’re not.
 *
 * SPD SMBus Channel 0 Device 0x50       -> MC0 slot 0  \___ MC0 CS0 via external decode
 * SPD SMBus Channel 0 Device 0x51       -> MC0 slot 1  /
 * SPD SMBus Channel 0 Device 0x52       -> MC0 slot 2  \___ MC0 CS1 via external decode
 * SPD SMBus Channel 0 Device 0x53       -> MC0 slot 3  /
 *
 * DRAM must always be added in pairs!
 */

DRAM_CHAN_CFG(MC_CHAN0, CFG_DRAM_MIN_tMEMCLK, JEDEC, CASCHECK,
            CFG_DRAM_BLOCK_SIZE, NOCSINTLV, CFG_DRAM_ECC, (MCFLG_BIGMEM | MCFLG_FORCEREG))

DRAM_CS_SPD(MC_CS0, 0, 0, 0x50)
DRAM_CS_SPD(MC_CS1, 0, 0, 0x52)

/*
 * Memory channel 1: Configure via SMBUS, Automatic Timing, Big Memory mode, Force Register
 *
 * There’s an external register on the board, so dimms appear to be "registered" even though
 * the SPD says they’re not.
 *
 * SPD SMBus Channel 0 Device 0x54       -> MC1 slot 0  \___ MC1 CS0 via external decode
 * SPD SMBus Channel 0 Device 0x55       -> MC1 slot 1  /
 * SPD SMBus Channel 0 Device 0x56       -> MC1 slot 2  \___ MC1 CS1 via external decode
 * SPD SMBus Channel 0 Device 0x57       -> MC1 slot 3  /
 *
 * DRAM must always be added in pairs!
 */

DRAM_CHAN_CFG(MC_CHAN1, CFG_DRAM_MIN_tMEMCLK, JEDEC, CASCHECK,
            CFG_DRAM_BLOCK_SIZE, NOCSINTLV, CFG_DRAM_ECC, (MCFLG_BIGMEM | MCFLG_FORCEREG))

DRAM_CS_SPD(MC_CS0, 0, 0, 0x54)
DRAM_CS_SPD(MC_CS1, 0, 0, 0x56)

/*
 * End of Table
 */

DRAM_EOT
```

6.8 LED messages

One of the earliest I/O operations in CFE's startup is to initialize access to an external LED display, if you have one. Display units similar to HP/Agilent's DL-2416 are easy to attach to the BCM1250's generic bus and can provide valuable information during bringup. If you have an LED configured, the following messages are displayed during startup:

Message	Meaning
HELO	Very first message displayed by the firmware. This is message is displayed just after your *board_earlyinit* routine returns.
L1CI	Displayed just before L1 cache initialization.
L2CI	Displayed just before L2 cache initialization.
DRAM	Displayed just before DRAM controller initialization
RAMX	**Fatal error.** Displayed if there is no RAM in the system.
ZBSS	Displayed just before CFE zeroes the BSS region.
DATA	Displayed just before copying the initialized data segment from flash to DRAM
RELO	Displayed just before performing fixups on the data segment (relocatable version only)
L12F	Displayed just before flushing the caches after code relocation.
MAIN	Displayed just before jumping to "C" code
cpui	Displayed just before starting the secondary CPU
cpu1	Displayed by CPU1 just before it enters its idle loop
KMEM	Displayed just before CFE initializes its heap
CONS	Displayed just before attaching the console device
CIOK	Displayed just before displaying copyright notice
AREN	Displayed just before initializing the arena (physical memory map)
PCIH	Displayed just before initializing the PCI host bridge.
PCIS	Displayed just before scanning PCI configuration space for devices
PCIR	Displayed just before reconfiguring the PCI devices based on information gathered from the scan
PCIW	Displayed just before assigning PCI resources (I/O, memory windows)
PCID	Displayed just before activating PCI devices with assigned resources
vgai	Displayed while running the X86 VGA BIOS (if configured)
DEVI	Displayed just before initializing the rest of the devices
ENVI	Displayed just before reading the environment from the NVRAM device
CFE	Displayed at completion of all initialization. CFE is now at the command prompt.
EXC!	**Error.** Displayed if an exception occurs.
RUN!	Displayed just before CFE launches a program.

If you do not have an LED display in your system, and you can monitor the generic bus with a logic analyzer, you can write a routine in your *board_init.S* file to write the LED value (in the A0 register) to some address that you can monitor externally.

7. Device Drivers

7.1 Device driver structure

Device drivers in CFE are extremely simple. They are not meant to provide a high-performance data path, and they generally do not expose all of a device's features or functions. They provide the minimal support necessary to read and write data from a device in such a way that an operating system or bootstrap program can be loaded, and status can be communicated to the user. In particular, CFE's drivers are entirely *polled*, since CFE has no interrupt dispatcher and never enables interrupts.

Device drivers are simple "C" routines that are linked into the firmware at build time. They all share a few common data structures:

7.1.1 Device Descriptor

Device drivers are managed by CFE's *device manager*, which simply keeps a list of device drivers and their names for applications to use. A device driver is declared by creating a *cfe_driver_t* structure:

```
typedef struct cfe_driver_s {
    char *drv_description;
    char *drv_bootname;
    int drv_class;
    const cfe_devdisp_t *drv_dispatch;
    void (*drv_probe)(. . .);
} cfe_driver_t;
```

The fields in this structure describe the device and declare its entry points. The fields are:

Field	Description
`drv_description`	Full name of the device, for display purposes
`drv_bootname`	Device name prefix (without a unit number suffix, such as "uart")
`drv_class`	Type of device (see below)
`drv_dispatch`	Pointer to dispatch table to device's access methods
`drv_probe`	Probe routine, which eventually calls *cfe_add_device* to instantiate the device.

7.1.2 Device Classes

The following device classes are defined. The device class should be filled into the *drv_class* field:

Class	Description
CFE_DEV_NETWORK	Network device, such as an Ethernet controller
CFE_DEV_DISK	Disk (or other random-access block-structured device)
CFE_DEV_FLASH	Flash memory device, for boot ROMs
CFE_DEV_SERIAL	Serial port devices or other character-oriented devices that are suitable for use as consoles
CFE_DEV_CPU	CPUs (not used)
CFE_DEV_NVRAM	NVRAMs such as EEPROMs that can be used to store the environment
CFE_DEV_CLOCK	Real-time (time of year) clock devices
CFE_DEV_OTHER	Any other devices that do not fit into the above categories

7.1.3 Function Dispatch

Each device driver supplies a function dispatch table in the form of a *cfe_devdisp_t* structure. This structure is defined as:

```
struct cfe_devdisp_s {
    int (*dev_open)(cfe_devctx_t *ctx);
    int (*dev_read)(cfe_devctx_t *ctx,iocb_buffer_t *buffer);
    int (*dev_inpstat)(cfe_devctx_t *ctx,iocb_inpstat_t *inpstat);
    int (*dev_write)(cfe_devctx_t *ctx,iocb_buffer_t *buffer);
    int (*dev_ioctl)(cfe_devctx_t *ctx,iocb_buffer_t *buffer);
    int (*dev_close)(cfe_devctx_t *ctx);
    void (*dev_poll)(cfe_devctx_t *ctx,int64_t ticks);
    void (*dev_reset)(void *softc);
};
```

The functions in this dispatch table correspond to the CFE functions exported by the external API. All of the functions are required except *dev_poll* and *dev_reset* which may be null if you do not need them. The *dev_reset* routine's parameter list differs from the others since it will be called on a warm restart when the device is not yet open.

7.1.4 The Probe routine

The probe routine is called during startup to add a device to the system. It is passed three generic parameters: *probe_a, probe_b,* and *probe_ptr*. The purpose of these parameters is completely unspecified; the probe routine can use it for any purpose that it wants. Generally these parameters are used to communicate the device's bus address or other information that is specific to the target. In the target's *board_device_init* routine, there will be one or more calls to *cfe_add_device*, as in the following example:

```
cfe_add_device(&flashdrv,BOOTROM_PHYS,BOOTROM_SIZE*K64,NULL);
cfe_add_device(&flashdrv,ALT_BOOTROM_PHYS,ALT_BOOTROM_SIZE*K64,NULL);
```

The second, third, and fourth arguments to *cfe_add_device* will be passed to the probe routine as *probe_a, probe_b,* and *probe_ptr*. In this example, *probe_a* is being used to pass the physical address of a flash ROM area, and *probe_b* is being used to pass the size of the flash ROM. *Probe_ptr* is not used.

The *probe* routine can verify that the device is present in the system and call *cfe_attach* to install the device in CFE's device list. The *cfe_attach* routine is declared as:

```
void cfe_attach(cfe_driver_t *devname,void *softc,
                char *bootinfo,char *description);
```

The *devname* parameter is the *cfe_driver_t* structure passed to the probe routine. The *softc* parameter is your "soft context" data structure, which is generally allocated from the heap via *KMALLOC* and contains any information you wish to store about the device, such as its current state, device register addresses, etc. The *bootinfo* parameter is the suffix to add to the device's full name. Generally it is a "dotted decimal" sequence of numbers like "1.2" which will be appended to the device name string. If you are installing "mydevice0" and you pass a *bootinfo* string of "1.2", the device's full name will be "mydevice0.1.2" Finally, the *description* parameter is a verbose description of the device (it can be displayed by the user interface). CFE will make a copy of this string for you, so it is not necessary to declare it in your *softc* structure.

7.2 Adding a new device driver

It is easiest to start with one of the existing device drivers, such as the serial driver (*dev_sb1250_uart.c*).

Decide what your probe routine's arguments will be, and in your probe routine allocate space for a private data structure to hold this data and anything else you need to keep track of the current state of the device.

When writing the I/O routines, try to avoid doing any device-specific setup in the probe routine. (you can allocate memory and calculate I/O register offsets and values, but avoid actually writing the values to the I/O registers). Move these accesses to the open routine to minimize the possibility of bad hardware causing the firmware to hang, and to increase the likelihood that a warm restart will work when an OS exits.

Call the *cfe_attach* routine to create the device. If you want, you can call *cfe_attach* multiple times to create additional instances of the device.

In your Makefile, add the object file name of your new device driver to the BSPOBJS list.

In your *board_device_init* routine, add a call to *cfe_add_device*, passing any necessary addressing information in the *probe_a, probe_b,* and *probe_ptr* arguments.

7.3 Device driver probe arguments for supplied devices

The device drivers included in the base CFE distribution pass the following data in their probe arguments:

Device	probe_a	probe_b	probe_ptr
`null`	unused	unused	unused
`flash`	Physical address of the base of the flash memory	Total size of the flash memory	if non-NULL, points at a *flash_info_t*, which describes advanced flash options.
`ide`	Physical address of the base of the IDE drive's registers	Unit number (0 or 1) and flags	unused
`jtag`	Physical address of JTAG communication area	unused	unused
`pcconsole`	unused	unused	unused
`promice`	Physical address of the AI2 interface on the PromICE	Word size (1,2 or 4 bytes). '1' is normal.	unused
`eth`	Ethernet controller's physical base address	unused	String pointer to hardware address in the form "xx:xx:xx:xx:xx:xx"
`24lc128eeprom`	SMBus channel number (0 or 1)	SMBus device address	unused
`x1240eeprom`	SMBus channel number (0 or 1)	unused (SMBus device address is fixed)	unused
`uart`	DUART's base phys address	Channel number within the DUART (0 or 1)	unused
`ns16550`	Physical address of the base of the NS16550's registers.	unused	unused
`ns16550_pci`	unused	unused	unused

7.4 Device driver functions

The sections below describe the routines that your device driver implements to perform I/O for the firmware. Pointers to these routines are placed in the *cfe_devdisp_s* structure.

Most of the device driver functions are passed a pointer to the *device context* in the *cfe_devctx_t* structure. This structure maintains information needed for the device while it is open. You can retrieve the pointer to the *softc* structure that you created in the probe routine by referencing the *ctx->dev_softc* member of the *cfe_devctx_t* structure.

With the exception of *dev_poll* and *dev_reset*, all device functions should return zero if successful, or an error code from *include/cfe_error.h*.

7.4.1 The *dev_open* routine

The *dev_open* routine is called when an application uses the CFE_DEV_OPEN firmware API call. In general, your routine should initialize the device and prepare it for I/O operations. For example, a serial device driver will enable the port, set the modem control signals to indicate data transfer is OK, configure the baud rate, etc.

7.4.2 The *dev_read* routine

The *dev_read* routine is used to transfer data from the device. It is called from the device manager when an application uses the CFE_DEV_READ firmware API call. The destination for the transferred data is described in the *iocb_buffer_t* argument. The fields are used as follows:

field	Description
`buf_ptr`	Points to the user buffer to receive the data from the device
`buf_length`	Contains the size of the user's buffer, in bytes
`buf_offset`	For block devices such as disks, contains the byte offset within the device where the transfer will begin.

The *dev_read* routine is non-blocking. If there is insufficient data from the device to transfer all requested bytes, the data that is available is read and the function returns. Higher-level routines will repeat the *dev_read* call until all the requested data is read. On return, *dev_read* will fill in the *iocb_buffer_t* fields as follows:

field	Description
`buf_retlen`	Contains the actual number of bytes of data that were read (zero means no data is available to read)

7.4.3 The *dev_inpstat* routine

The *dev_inpstat* routine is used to test the status of an input device. It is called from the device manager when an application uses the CFE_DEV_INPSTAT firmware API call. In the case of a

console, *dev_inpstat* will return an indication that there are characters available to be read by the *dev_read* routine. The status is returned in the *iocb_inpstat_t* structure as follows:

field	Description
`inp_status`	Contains zero if no bytes are available to be read, or one if one or more bytes are available to be read.

7.4.4 The *dev_write* routine

The *dev_wite* routine is used to transfer data to the device. It is called from the device manager when an application uses the CFE_DEV_WRITE firmware API call. The source for the transferred data is described in the *iocb_buffer_t* argument. The fields are used as follows:

field	Description
`buf_ptr`	Points to the user buffer containing data to be sent to the device
`buf_length`	Contains the number of bytes of data to send to the device
`buf_offset`	For block devices such as disks, contains the byte offset within the device where the transfer will begin.

The *dev_write* routine is non-blocking. If there is insufficient buffer space in the device to transfer all requested bytes, the data that can be transferred will be transferred and the function returns. Higher-level routines will repeat the *dev_write* call until all the requested data is written. On return, *dev_write* will fill in the *iocb_buffer_t* fields as follows:

field	Description
`buf_retlen`	Contains the actual number of bytes of data that were written (zero will indicate that the device is blocked – in the case of a serial port, for example, zero would be returned if the FIFO was full)

7.4.5 The *dev_ioctl* routine

The *dev_ioctl* routine is used to provide device-specific API calls. The exact functions of the *dev_ioctl* routine are up to you, but certain classes of devices have some standardized IOCTL functions. See section 7.5.5.2 for details on the standard IOCTL functions.

The *dev_ioctl* function's parameters are passed in the *iocb_buffer_t* structure. The fields are as follows:

field	Description
`buf_ioctlcmd`	Contains a code number to specify which IOCTL function is being requested. Standard IOCTL codes are defined in

	include/cfe_ioctl.h.
`buf_ptr`	Points to a user buffer. If not used, this field should be NULL.
`buf_length`	Contains the length of the user's buffer.
`buf_retlen`	May be used by the IOCTL routine to return the number of bytes transferred.
`buf_offset`	May be used by the IOCTL routine to pass additional parameters.

If your device driver does not support a particular IOCTL routine, you should return an error code.

Be careful to avoid problems with 32/64-bit issues when writing an IOCTL handler. For example, if you are passing a structure in the *buf_ptr* field and this structure does not have the same layout on 32-bit and 64-bit versions of CFE, applications will pass incorrect data if the firmware is not compiled with the same memory model as the application.

7.4.6 The *dev_close* routine

The *dev_close* routine is called from the device manager when an application uses the CFE_DEV_READ firmware API call. Your device driver should stop any pending I/O, disable the device, and release any resources associated with the device. Note that the *softc* structure is not freed – it was allocated in the *probe* routine and remains in memory at all times.

7.4.7 The *dev_poll* routine

Some devices require periodic polling to avoid missing data or perform other background tasks. If you declare a pointer to a *dev_poll* routine, it will be called when your driver is open each time CFE goes through its polling loop. When your driver is not open, the *dev_poll* routine is not called. The *dev_poll* routine is optional; if you do not intend to use it, you can leave its entry point as NULL.

7.4.8 The *dev_reset* routine

The *dev_reset* routine is called when the firmware does a "warm start." It gives your driver a chance to undo any potentially dangerous device settings that an application or operating system may have done while it was running. Because the *dev_poll* routine is not called when the device is open, the dispatcher will not have created the *cfe_devctx_t* structure. Therefore, *dev_reset* is passed your *softc* pointer (to the data allocated in your *probe* routine). The *dev_reset* routine is not called on a cold start.

7.5 Standard device IOCTLs and read/write behavior

Certain classes of device drivers respond to common sets of IOCTL functions (for example, Ethernet devices have IOCTLs to obtain the Ethernet hardware address). The following sections describe the read/write behavior and the standard IOCTL functions for each class of device.

Constants and data structures related to the IOCTL commands can be found in *include/cfe_ioctl.h*.

7.5.1 Ethernet Devices

7.5.1.1 Read/Write behavior

Ethernet devices are record-oriented sequential devices. The *read* and *write* calls receive and transmit packets. If there are no packets available when a *read* call is issued, the driver returns zero indicating that no packets are available. When a packet is received, the buffer for the *read* call must be large enough to hold the packet (any portion that does not fit in the buffer will be dropped).

7.5.1.2 Standard IOCTLs

IOCTL Code	Parameter Data	Description
IOCTL_ETHER_GETHWADDR	6 bytes	Return the hardware address assigned to the interface.
IOCTL_ETHER_SETHWADDR	6 bytes	Set the hardware address for this interface. The buffer points to 6 bytes containing the address
IOCTL_ETHER_GETSPEED	int	Return the configured link speed (ETHER_SPEED_xxx)
IOCTL_ETHER_SETSPEED	int	Manually set the link speed (ETHER_SPEED_xxx)
IOCTL_ETHER_GETLINK	int	Return the current link status and speed (ETHER_SPEED_xxx)
IOCTL_ETHER_GETLOOPBACK	int	Return the current loopback mode setting (ETHER_LOOPBACK_xxx)
IOCTL_ETHER_SETLOOPBACK	int	Set the interface's loopback status (ETHER_LOOPBACK_xxx)

7.5.2 Flash Memory Devices

7.5.2.1 Read/Write behavior

Flash memory devices are random-access storage devices. The *read* and *write* calls make use of the *buf_offset* field of the buffer structure to specify the starting offset for data transfers.

7.5.2.2 Standard IOCTLs

IOCTL Code	Parameter Data	Description
IOCTL_NVRAM_GETINFO	nvram_info	Return data describing which portion of the flash may be used as an NVRAM device
IOCTL_NVRAM_ERASE	none	Erase the NVRAM area of the flash
IOCTL_FLASH_ERASE_SECTOR	offset	Erase the sector at the offset specified in the buffer descriptor's *buf_offset* field.
IOCTL_FLASH_ERASE_ALL	none	Erase the entire flash device
IOCTL_FLASH_WRITE_ALL	buffer,length	Relocate a programming routine to RAM, then erase and write the entire device using the buffer pointer and length supplied in the descriptor.
IOCTL_FLASH_GETINFO	flash_info	Return information about the flash in the *flash_info_t* structure. In particular, the *flash_type* field returns whether the device is actually a flash device. This IOCTL attempts to distinguish among flash, SRAM, and ROMs.
IOCTL_FLASH_GETSECTORS	flash_sector	Return information about a flash sector. The sector's index is filled into *flash_sector_idx* in the *flash_sector_t* structure. The sector's size and offset are returned, or else the *flash_sector_status* field returns an error code.
IOCTL_FLASH_UNLOCK	none	"unlocks" the NVRAM area of the flash, making it possible to write data beyond the reported end of the flash device. The environment manager uses this to write the NVRAM sector (usually the last sector in the flash).

7.5.3 EEPROM Devices

7.5.3.1 Read/Write behavior

Flash memory devices are random-access devices. The *read* and *write* calls make use of the *buf_offset* field of the buffer structure to specify the starting offset for data transfers.

7.5.3.2 Standard IOCTLs

IOCTL Code	Parameter	Description

	Data	
IOCTL_NVRAM_GETINFO	nvram_info	Return data describing which portion of the flash may be used as an NVRAM device
IOCTL_NVRAM_ERASE	none	Erase the NVRAM area of the flash

7.5.4 Serial Devices

7.5.4.1 Read/Write behavior

Serial ports are sequential, byte-stream devices. The *read* and *write* calls receive and transmit data. If no data is available to be received, the *read* call returns zero.

7.5.4.2 Standard IOCTLs

IOCTL Code	Parameter Data	Description
IOCTL_SERIAL_SETSPEED	int	Set the serial port's speed (in bits/second)
IOCTL_SERIAL_GETSPEED	int	Return the serial port's speed
IOCTL_SERIAL_SETFLOW	int	Set the flow control mode (SERIAL_FLOW_xxx)
IOCTL_SERIAL_GETFLOW	int	Return the current flow control mode (SERIAL_FLOW_xxx)

7.5.5 Disk Devices

7.5.5.1 Read/Write behavior

Disks are random-access storage devices. The *read* and *write* calls make use of the *buf_offset* field of the buffer structure to specify the starting offset for data transfers. The device driver is responsible for handling the case where the offset is not aligned on a sector boundary.

7.5.5.2 Standard IOCTLs

IOCTL Code	Parameter Data	Description
IOCTL_BLOCK_GETBLOCKSIZE	int	Return the size of each block (sector) on the device
IOCTL_BLOCK_GETTOTALBLOCKS	long long	Return the total number of blocks on the

		device
IOCTL_BLOCK_GETDEVTYPE	blockdev_info	Return information describing the block device in a *blockdev_info_t* structure.

8. Firmware API and Boot Environment

The firmware API is used by operating systems and loaders to communicate with the bootstrap device, obtain environment variables, and request information about the system. CFE uses a simple control-block style API, where the address of a parameter block is passed to the firmware for all operations.

8.1 Entry point

The entry point is located in the ROM, with an entry vector of 0xBFC00500 (uncached) or 0x9FC00500 (cached). The entry point has the following structure:

Address (cached)	Address (uncached)	Contents
0x9FC00500	0xBFC00500	Branch to entry point
0x9FC00504	0xBFC00504	NOP for branch delay slot
0x9FC00508	0xBFC00508	Entry point seal (0x43464531)
0x9FC0050C	0xBFC0050C	Entry point seal (0x43464531)

Software should check for the presence of the seal before calling the entry point. Most software can call the cached entry point, since CFE will have initialized the cache. If you have built CFE to run entirely uncached, then programs loaded via CFE should not use the cache, and those programs should call the uncached entry point.

When calling CFE, the following parameters are passed in registers:

Register	Value
A0	Firmware Handle (see below)
A1	Pointer to IO control block (*cfe_xiocb_t* structure pointer)

Results (status) are returned in the V0 register. The firmware will save the S0..S7 registers, but may use the temporary registers for itself.

The *firmware handle* value (passed in A0) is the value that was passed to the application when it was launched by CFE. This value is the pointer to CFE's data segment, which cannot be automatically determined in an implementation-independent way. CFE normally relocates its data segment to the top of physical memory to avoid conflicts with OS boot loaders and other software. It is the application's responsibility to remember the handle passed by the firmware and pass it back to CFE when making API calls.

The *I/O Control Block (IOCB)* pointer is the address of a data structure that describes the firmware request. The IOCB structure is divided into two parts, a fixed portion and a parameter list. The contents of the parameter list depend on the function number. The sections that follow describe each IOCB call.

8.2 Boot Environment

8.2.1 Virtual Address Space

CFE creates a special environment for boot loaders to help them avoid writing on the memory they will be loading the operating system into. It is assumed that the OS will load into KSEG0 and begin operation there, and once running it will create page tables and establish a TLB handler.

CFE includes a simple TLB handler and a page table for the following address:

Virtual Address	Size	Physical Address
0x20000000	256KB	Calculated; near the top of physical memory but within the first 256MB

The page table is linear mapping to the physical address. By locating the physical memory near the top of the available address range, an OS kernel can safely be loaded (presumably the kernel is smaller than physical memory and does not know how large the memory is, so it will be loaded at the *bottom* of the address space).

When loading "raw" (i.e., not ELF format) binary files, CFE will always load them to 0x20000000 and begin execution at that location. If you load ELF files, CFE will follow the load instructions in the program header. It will reject any ELF binary that tries to load data that conflicts with CFE or areas of memory that do not exist or lie outside the boot environment (in other words, you *can* load ELF files which are loaded to run at 0x20000000).

As long as the CP0 "BEV" (boot exception vector) bit is set, CFE will be able to service TLB exceptions for the boot area. Once control is transferred to the kernel, do not depend on the presence of the boot environment.

8.2.2 Environment Variables

CFE sets up a number of environment variables for use by the boot loader and the operating system. They are summarized in the table below:

Variable	Value
BOOT_DEVICE	The full name of the device containing the boot file
BOOT_FILE	The name of the file read from the boot device, if any
BOOT_FLAGS	Any command-line switches or options specified along with the file

	name on the boot command

For network bootstrap, the following additional environment variables are available:

Variable	Value
NET_DEVICE	The name of the network device (for example, “eth0”). This should be the same as the value of BOOT_DEVICE.
NET_IPADDR	The IP address for this network device (dotted decimal notation)
NET_NETMASK	The netmask for this network device (dotted decimal notation)
NET_GATEWAY	The gateway address, if any (dotted decimal notation)
NET_NAMESERVER	The name server address (dotted decimal notation)
NET_DOMAIN	The domain name, if available (e.g., “broadcom.com”)
NET_HOSTNAME	The host’s name, if available (not including the domain name)

If a parameter is not present (either not set explicitly or obtained via the DHCP server), the environment variable will not exist.

8.2.3 Registers passed to boot loaders

When CFE invokes the boot loader, it passes the following values in registers:

Register	Value
A0	Firmware handle. This value must be passed back to CFE each time the loaded program wants to use CFE’s API. The firmware handle is a pointer to CFE’s data segment.
A1	Zero
A2	Firmware’s entry vector (usually 0x9FC0500, but could be different if CFE has been relocated into RAM)
A3	Entry point seal. You can use this value to make sure you were invoked from CFE. The value is 0x43464531, the same as the one in the ROM entry point vector.

When a program is invoked, it will be using CFE’s stack and the BEV (boot exception vector) bit will still be set. The trap handler will cause the firmware to restart.

8.2.4 Registers passed to secondary processors

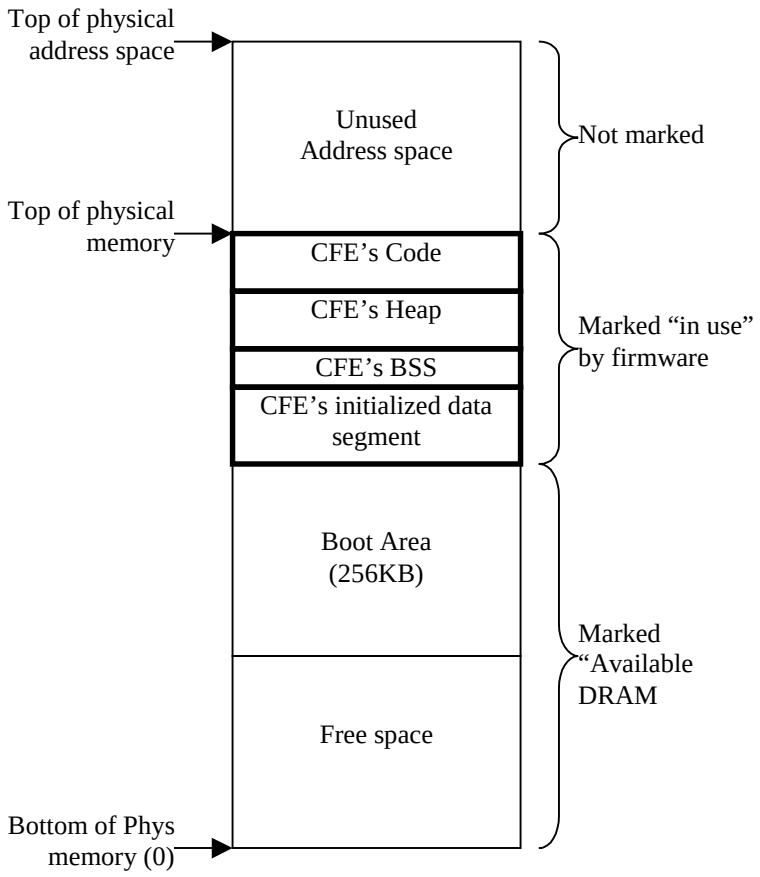
When CFE starts a secondary processor core, the following values will be passed in the registers:

Register	Value
A0	Firmware handle. This value must be passed back to CFE each time the loaded program wants to use CFE’s API. The firmware handle is a pointer to CFE’s data segment.
A1	Value from the *a1_val* parameter list member from the CPU startup API command

A2	Firmware's entry vector (usually 0x9FC0500, but could be different if CFE has been loaded into RAM)
A3	Entry point seal. You can use this value to make sure you were invoked from CFE. The value is 0x43464531, the same as the one in the ROM entry point vector.
SP	Value from the *sp_val* parameter list member of the CPU startup API command
GP	Value from the *gp_val* parameter list member of the CPU startup API command

8.2.5 Memory Map

Depending on how CFE is built, the memory map can vary greatly. The most common and perhaps the most useful CFE configuration is to include a relocatable data segment. When built in this manner, CFE will try to locate its data structures as high in physical memory as is possible. For systems with more than 256MB of memory, the data structures will be as high in the first 256MB block as possible, to ensure that regular (not extended) addressing can be used at all times.



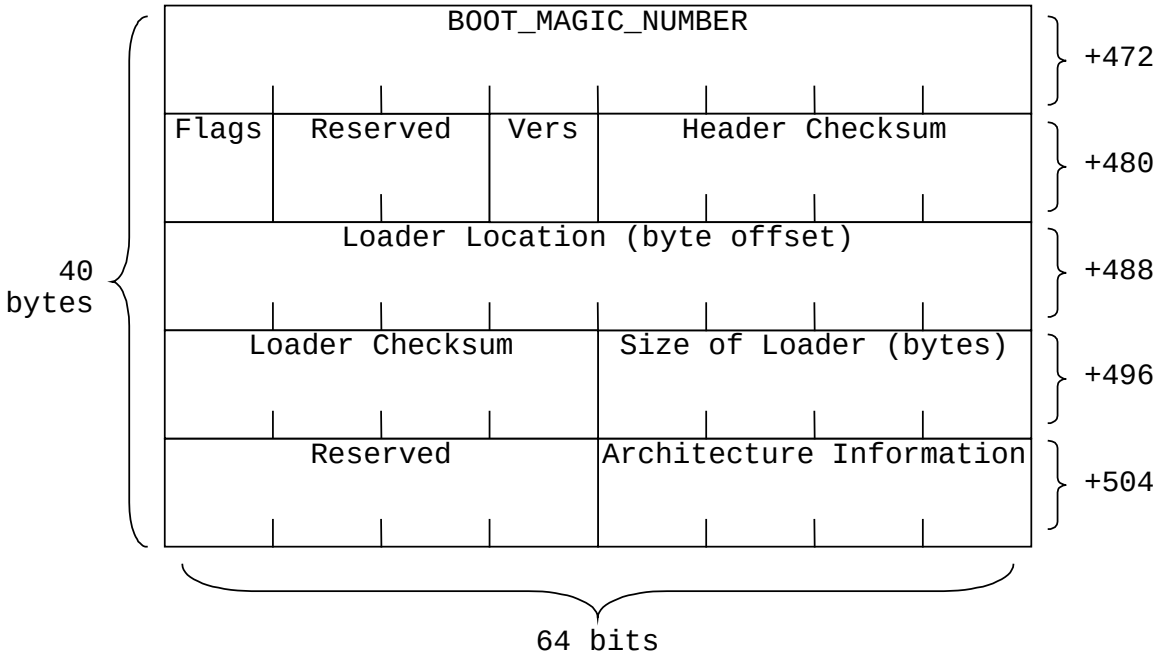
If CFE is built to use a specific data segment address the boot area will be placed at the top of physical memory, unless it conflicts with the firmware's data segment.

8.3 Disk Bootstrap

When loading a bootstrap program from the network, you can specify the file's name on the remote system.

When booting from the disk, you do not necessarily have a file system to read the bootstrap from, so the bootstrap generally does not have a name or a predefined size.

CFE looks for a special record called the *boot block* on a block-structured (disk, tape, CD-ROM) device. The record has the following layout:



The boot record is located at offset 472 in the a given 512-byte sector and is 40 bytes (five 64-bit words) long.

The boot record is normally located in the first sector of the boot device. To accommodate multiple architectures sharing the boot media (for example, on CD-ROMs), CFE will search the first 16 sectors of the boot device to find a valid boot block.

The fields in the boot block are stored in native byte order, and should be accessed as 64-bit quantities to make parsing the fields easier. The fields are summarized below:

Field	Value
`BOOT_MAGIC_NUMBER`	The constant `0x43465631424f4f5`
`Flags`	Special flags for using this boot block (none currently defined, so this field should be zero)
`Reserved`	Ignored; set to zero

`Vers`	Boot block version. The current version is 1.
`Header Checksum`	Header checksum – the checksum of the 40 bytes of this boot block (see below)
`Loader Location`	The byte offset into the media where the loader begins
`Loader Checksum`	The checksum for the loader code (see below)
`Size of Loader`	The size of the loader, in bytes
`Architecture Information`	Any architecture-specific information. None are currently defined, so this field should be set to zero.

The checksum is a simple checksum, generated by totaling all of the 32-bit words in the header or loader program. For loader programs that are not even multiples of 32 bits in length, the fill bytes should be considered as zero. The header checksum is calculated after the loader checksum has been placed in the data structure, and while calculating the header checksum, the header checksum field itself is assumed to be zero.

Pay particular attention to the host and target endianness, especially in a cross-development environment. The checksum calculation must be made reading the words of the header and loader in target-endian format, just as they will be when running on the hardware.

8.3.1 Generating a Boot Block

There are two programs in the `hosttools/` directory for generating boot blocks. The first is *mkbootimage*, which takes a binary file (a raw binary executable linked to run in the boot environment) and preprends a 512-byte record containing a CFE boot block to it. This image may then be placed on a real or simulated hard disk for booting. The *mkbootimage* program is run as follows:

```
mkbootimage [-v] [-EB] [-EL] inputfile outputfile
```

Where *–EB* and *–EL* specify the target endianness. The *–v* switch causes *mkbootimage* to print out messages describing its progress. The *inputfile* parameter is the raw binary image, and *outputfile* is the file to write that will contain the boot block prepended to the binary image.

The second program is *installboot*, which is only useful if you are working with the simulated IDE disk in the BCM1250's functional simulator. The *installboot* program takes the boot block (output from *mkbootimage*, above) and writes it to the beginning of a simulated disk file. The first 480 bytes of the first sector will be preserved by *installboot* so as not to conflict with BSD disklabels. The *installboot* program is run as follows:

```
installboot inputfile outputfile
```

where *inputfile* is the output from *mkbootimage* and *outputfile* is your simulated disk file.

8.4 API functions

All of CFE's API function parameters are passed in an *I/O Control Block (IOCB)*. The "C" data structure name for this structure is *cfe_xiocb_t*, where "xiocb" stands for an *external* IOCB. Internally, CFE uses the same structure for communicating with itself, but the XIOCB differs in that all of the fields are 64 bits wide (even on 32-bit versions of CFE) and the XIOCB structures will remain the same even if the internal IOCB functions are modified. A translation module inside CFE converts the XIOCB into an IOCB, and should an incompatible change to the IOCB structure be needed, the translation module can be enhanced to make up for the differences that were introduced.

The XIOCB structure has two parts: a fixed *header* and a *parameter list*. The contents of the parameter list depend on the function code and the value of the *xiocb_psize* member, which contains the length of the parameter list. The XIOCB is defined as the following "C" structure:

```
typedef struct cfe_xiocb_s {
    cfe_xuint_t xiocb_fcode;        /* IOCB function code */
    cfe_xint_t  xiocb_status;       /* return status */
    cfe_xint_t  xiocb_handle;       /* file/device handle */
    cfe_xuint_t xiocb_flags;        /* flags for this IOCB */
    cfe_xuint_t xiocb_psize;        /* size of parameter list */
    union {
      xiocb_buffer_t  xiocb_buffer; /* buffer parameters */
      xiocb_inpstat_t xiocb_inpstat;/* input status parameters */
      xiocb_envbuf_t  xiocb_envbuf; /* environment function parameters */
      xiocb_cpuctl_t  xiocb_cpuctl; /* CPU control parameters */
      xiocb_time_t    xiocb_time;   /* timer parameters */
      xiocb_meminfo_t xiocb_meminfo;/* memory arena info parameters */
      xiocb_exitstat_t xiocb_exitstat; /* exit status */
    } plist;
} cfe_xiocb_t;
```

The types *cfe_xuint_t* and *cfe_xint_t* are 64-bit unsigned and signed integers, respectively. When placing a buffer address in a parameter list member from a 32-bit application, be sure to sign-extend the address in case you will be calling the 64-bit version of CFE.[5]

8.5 Vendor Extensions

Vendors who port CFE to their designs can extend the IOCB interface. If the *bsp_config.h* option `CFG_VENDOR_EXTENSIONS` is defined, all commands with function codes above the constant `CFE_FW_CMD_VENDOR_USE` are directed to a dispatch routine in the *vendor/* directory in the source tree. This dispatch routine works in a manner very similar to the standard dispatch routine, except Broadcom will attempt to minimize changes to the files in the *vendor/* directory from release to release.

[5] Would it be a good idea to have a whole section on 64/32 bit issues?

8.5.1 CFE_CMD_FW_GETINFO

This function returns important global information about CFE.

Request structure fields:

xiocb field	Description
xiocb_fcode	0 (CFE_CMD_FW_GETINFO)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0
xiocb_psize	sizeof(xiocb_fwinfo_t)

Return structure fields:

xiocb field	Description
plist:fwi_version	Major, minor, and ECO version of CFE. For example, the value 0x010203 would mean version 1.2.3.
plist:fwi_totalmem	Total installed memory in all memory regions.
plist:fwi_flags	Flags about this version of CFE: CFE_FWI_64BIT – set if using the 64-bit version CFE_FWI_32BIT – set if using the 32-bit version CFE_FWI_RELOC – set if data segment is relocatable CFE_FWI_UNCACHED – set if running in KSEG1 CFE_FWI_MULTICPU – more than one CPU supported CFE_FWI_FUNCSIM – set if running in functional simulator CFE_FWI_RTLSIM – set if running in the RTL simulator
plist:fwi_boardid	A number specified in the CFG_BOARD_ID parameter in the *bsp_config.h* file. You can use this to detect subtle differences in firmware or options.
plist:fwi_bootarea_va	The virtual address of the boot area, 0x20000000.
plist:fwi_bootarea_pa	The physical address of the boot area
plist:fwi_bootarea_size	The size of the boot area, 256KB (0x40000)
plist:fwi_reserved1 plist:fwi_reserved2 plist:fwi_reserved3	Reserved, will return zero

Error codes:

Code	Description
(none)	

8.5.2 CFE_CMD_FW_RESTART

Restarts the firmware. This function is used when an operating system exits or an application loaded by CFE wishes to return to the CFE command prompt.

A warm start may not work in all circumstances, particularly if devices are not shut down cleanly and the hardware is left in an unstable state. CFE will restore the boot exception vectors, reinitialize the CP0 registers, and invalidate all of the TLB entries when restarted via a warm start.

Request structure fields:

xiocb field	Description
xiocb_fcode	1 (CFE_CMD_FW_RESTART)
xiocb_status	0
xiocb_handle	0
xiocb_flags	CFE_FLG_COLDSTART for a cold start CFE_FLG_WARMSTART to skip initializing devices
xiocb_psize	sizeof(cfe_exitstat_t)
plist:status	Exit status (0 = successful)

Return structure fields:

xiocb field	Description
none	function does not return

Error codes:

Code	Description
none	function does not return

8.5.3 CFE_CMD_FW_CPUCTL

Control secondary processor cores. This routine is used to direct control of secondary processor cores to a user routine. During startup, the firmware initializes the processor cores and leaves secondary processors in a "holding pattern" waiting for work to do. When this firmware call is issued, control on the secondary processor can be passed to a user routine. The CPU commands understood by this firmware call are:

Command (cpu_command field)	Description
CFE_CPU_CMD_START	Start the specified CPU at the *start_addr* address.
CFE_CPU_CMD_STOP	Stop the specified CPU, causing it to return to the idle loop. This call must be made from CPU0.
CFE_CPU_CMD_IDLE	Return to the idle loop. This is called from the secondary CPU, and may not be called from CPU0. The current CPU enters CFE's idle loop and stays there. The function does not return. **Note:** Not implemented.

Request structure fields:

xiocb field	Description
xiocb_fcode	3 (CFE_CMD_FW_CPUCTL)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0
xiocb_psize	sizeof(xiocb_cpuctl_t)
plist:cpu_number	CPU number (starting with 1) for CFE_CPU_CMD_START and CFE_CPU_CMD_STOP
plist:cpu_command	Command to issue to the specified CPU
plist:start_addr	Start address (if starting a CPU)
plist:sp_val	Initial value of the SP register
plist:gp_val	Initial value of the GP register
plist:a1_val	Initial value of the A1 register

Return structure fields:

xiocb field	Description
none	none

Error codes:

Code	Description
CFE_ERR_INV_COMMAND	Firmware does not support secondary CPUs
CFE_ERR_INV_PARAM	CPU number or CPU command are invalid

8.5.4 CFE_CMD_FW_GETTIME

Obtain system time and call internal polling functions. This routine can be used to obtain CFE's idea of the system time (see the *timer manager* for details). It also causes the internal device polling routines to be run, so boot loaders should periodically call this service to ensure that network timeouts occur and packets are processed.

Request structure fields:

xiocb field	Description
xiocb_fcode	4 (CFE_CMD_FW_GETTIME)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0
xiocb_psize	sizeof(xiocb_time_t)

Return structure fields:

xiocb field	Description
plist:ticks	Current system time in ticks (units of CFE_HZ)

Error codes:

Code	Description
none	none

8.5.5 CFE_CMD_FW_MEMENUM

Enumerate the contents of the arena for available DRAM. This routine is used by operating systems to determine the areas of physical memory that are available for use. It is an enumerator, so it must be called repeatedly until it returns an error to obtain all of the memory block information. The first time it is called, set *mi_idx* to zero, and then increment it for each call until an error is returned.

Request structure fields:

xiocb field	Description
xiocb_fcode	5 (CFE_CMD_FW_MEMENUM)
xiocb_status	0
xiocb_handle	0
xiocb_flags	CFG_FLG_FULL_ARENA set to return all memory blocks, otherwise only available DRAM blocks are returned.
xiocb_psize	sizeof(xiocb_meminfo_t)
plist:mi_idx	index of entry to query, starting with zero.

Return structure fields:

xiocb field	Description
plist:mi_type	Type of memory block: CFE_MI_AVAILABLE = block is available for use CFE_MI_RESERVED = reserved memory block other values are reserved by applications.
plist:mi_addr	Physical address of the start of the block
plist:mi_size	Size of the memory block, in bytes

Error codes:

Code	Description
CFE_ERR_NOMORE	No more memory blocks

8.5.6 CFE_CMD_FW_FLUSHCACHE

Perform cache operations. After loading software from a boot device, it is important to flush the Dcache and invalidate the Icache to ensure that the correct instructions will be executed. Cache operations are applied to the entire cache.

Request structure fields:

xiocb field	Description
xiocb_fcode	6 (CFE_CMD_FW_FLUSHCACHE)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0 to flush D and invalidate I, else a combination of: CFE_CACHE_FLUSH_D : flush D cache (write back to memory) CFE_CACHE_INVAL_I : invalidate I cache CFE_CACHE_INVAL_D : invalidate D cache CFE_CACHE_INVAL_L2 : invalidate L2 cache
xiocb_psize	0
plist	none

Return structure fields:

xiocb field	Description
none	none

Error codes:

Code	Description
none	none

8.5.7 CFE_CMD_DEV_GETHANDLE

Obtain a standard file handle. This function is normally used to get the handle for the console device. Since CFE internally uses the console for its own purposes, it does not close the console when it transfers control to another program. You can obtain the handle that CFE uses for the console with the CFE_CMD_DEV_GETHANDLE function and then use it for writing console messages.

Request structure fields:

xiocb field	Description
xiocb_fcode	9 (CFE_CMD_DEV_GETHANDLE)
xiocb_status	0
xiocb_handle	0
xiocb_flags	Indicates which handle to get: CFE_STDHANDLE_CONSOLE : obtain console handle
xiocb_psize	0
plist	none

Return structure fields:

xiocb field	Description
xiocb_handle	File handle, if successful

Error codes:

Code	Description
CFE_ERR_DEVNOTFOUND	Console handle is not open, or there is no console
CFE_ERR_INV_PARAM	The xiocb_flags parameter is invalid

8.5.8 CFE_CMD_DEV_ENUM

Enumerate the devices that are present in the system. This function allows a boot loader to scan the list of device drivers known to CFE. Note that this does not necessarily mean all of the devices present in the system will be enumerated, just the ones that CFE has been built to recognize. The function returns the device names (boot names and device names).

Request structure fields:

xiocb field	Description
xiocb_fcode	10 (CFE_CMD_DEV_ENUM)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0
xiocb_psize	sizeof(xiocb_envbuf_t)
plist:enum_idx	Enumeration index. Start with 0, and increment each time you call this function until it returns an error.
plist:name_ptr	Points to a buffer to receive the device's boot name.
plist:name_length	Length of buffer receiving device's boot name
plist:val_ptr	Points to a buffer to receive the device's full name
plist:val_length	Length of buffer receiving device's full name

Return structure fields:

xiocb field	Description
plist:name_length	Length of device's boot name
plist:val_length	Length of device's full name

Error codes:

Code	Description
CFE_ERR_DEVNOTFOUND	No device at the specified index

8.5.9 CFE_CMD_DEV_OPEN

Open a device. CFE will search the device table for the specified device and open it. You may specify either the boot name or the full name for the device. Once the device is open, CFE will return a handle that you can use for subsequent READ, WRITE, IOCTL, and CLOSE calls.

Request structure fields:

xiocb field	Description
xiocb_fcode	11 (CFE_CMD_DEV_OPEN)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0
xiocb_psize	sizeof(xiocb_buffer_t)
plist:buf_ptr	Points to name of device to open (null terminated)
plist:buf_length	Size of the name of the device, including null byte

Return structure fields:

xiocb field	Description
xiocb_handle	File handle if the operation was successful

Error codes:

Code	Description
CFE_ERR_DEVNOTFOUND	No device with the specified name was found to open
CFE_ERR_DEVOPEN	The device is already open.
CFE_ERR_NOMEM	Insufficient memory to open device
others	Device-specific error codes may be returned

8.5.10 CFE_CMD_DEV_INPSTAT

Request input status for a device. This command tests to see if data is ready to be read for an open device and returns a flag. For serial port drivers, this indicates that characters are waiting in the receive buffer. For network drivers, this indicates that packets are waiting.

Request structure fields:

xiocb field	Description
xiocb_fcode	12 (CFE_CMD_DEV_INPSTAT)
xiocb_status	0
xiocb_handle	An open file handle, from CFE_CMD_DEV_OPEN
xiocb_flags	0
xiocb_psize	sizeof(xiocb_inpstat_t)
plist	

Return structure fields:

xiocb field	Description
plist:inp_status	Zero if no data is available to read, one if data is available to read

Error codes:

Code	Description
CFE_ERR_INV_PARAM	The file handle is invalid.

8.5.11 CFE_CMD_DEV_READ

Read data from a device. The amount of data read depends on the type of device. For example, a serial port device will only return the amount of data available in its FIFO, while a network device will return an entire packet (discarding the portion of the packet beyond the length of the user's request buffer). Block devices such as disks and CD-ROMs must specify an offset in the *xiocb_buffer_t* parameter list. This is a byte offset into the device (should be aligned to a natural sector boundary compatible with the device, but it is not required).

Request structure fields:

xiocb field	Description
xiocb_fcode	13 (CFE_CMD_DEV_READ)
xiocb_status	0
xiocb_handle	An open file handle from CFE_CMD_DEV_OPEN
xiocb_flags	0
xiocb_psize	sizeof(xiocb_buffer_t)
plist:buf_offset	Offset (in bytes) into the device. This is used only for block devices such as CD-ROMs and disks.
plist:buf_ptr	Pointer to user buffer to receive the data
plist:buf_length	Length of buffer receiving the data

Return structure fields:

xiocb field	Description
plist:buf_retlen	Actual number of bytes returned from the device

Error codes:

Code	Description
CFE_ERR_INV_PARAM	File handle is invalid
others	Device may return device-specific error codes

8.5.12 CFE_CMD_DEV_WRITE

Write data to the specified device. The way the data is written to the device depends on the device's type. For example, network devices will write an entire packet using this call (the buffer points at the MAC header, usually the destination address). UART devices write as many characters as will fill up the FIFO. Disk devices write blocks of data and must specify the *buf_offset* field in the *xiocb_buffer_t* parameter list.

CFE device write operations do not block, so if more characters are written to a device (most likely a serial device) than will fit in the FIFO, the call will write as many as it can and return (passing back the actual number of characters written). The user must advance the write pointer and repeat the call until all of the characters are flushed to the device. This scheme gives CFE and the calling program the opportunity to poll for other devices.

Request structure fields:

xiocb field	Description
`xiocb_fcode`	14 (CFE_CMD_DEV_WRITE)
`xiocb_status`	0
`xiocb_handle`	An open file handle, from CFE_CMD_DEV_OPEN
`xiocb_flags`	0
`xiocb_psize`	sizeof(xiocb_buffer_t)
`plist:buf_offset`	For block devices such as disks, the offset (in bytes) into the device where the write is to start. If this offset is not sector-aligned, the device driver will read/modify/write the partial sector (at the beginning and end, if necessary).
`plist:buf_ptr`	Pointer to the user buffer containing the data to write
`plist:buf_length`	Length of the user buffer, in bytes

Return structure fields:

xiocb field	Description
`plist:buf_retlen`	Number of bytes actually written to the device

Error codes:

Code	Description
`CFE_ERR_INV_PARAM`	File handle is invalid
`others`	Device may return device-specific error codes

8.5.13 CFE_CMD_DEV_IOCTL

Perform device-specific I/O operations. This is an "escape" call for accessing device functions particular to a device or device type. Refer to section XXX for details on the operations of individual device IOCTL calls.

Request structure fields:

xiocb field	Description
xiocb_fcode	15 (CFE_CMD_DEV_IOCTL)
xiocb_status	0
xiocb_handle	An open file handle, from CFE_CMD_DEV_OPEN
xiocb_flags	0
xiocb_psize	sizeof(xiocb_buffer_t)
plist:buf_offset	Buffer offset, if needed by IOCTL function
plist:buf_ptr	pointer to user buffer, if needed by IOCTL function
plist:buf_length	Length of user buffer, if needed by IOCTL function
plist:buf_ioctlcmd	IOCTL command code. This code distinguishes among different IOCTLs supported by a device.

Return structure fields:

xiocb field	Description
plist	Parameter list members may be modified by the IOCTL function

Error codes:

Code	Description
CFE_ERR_INV_PARAM	File handle is invalid
others	Device may return device-specific error codes

8.5.14 CFE_CMD_DEV_CLOSE

Close a device handle. You should close the device when you are finished using it to prevent resources from being consumed and to make the device available for other callers.

Request structure fields:

xiocb field	Description
xiocb_fcode	16 (CFE_CMD_DEV_CLOSE)
xiocb_status	0
xiocb_handle	A file handle, from CFE_CMD_DEV_OPEN
xiocb_flags	0
xiocb_psize	0
plist	none

Return structure fields:

xiocb field	Description
none	none

Error codes:

Code	Description
CFE_ERR_INV_PARAM	File handle is invalid

8.5.15 CFE_CMD_DEV_GETINFO

Obtain information about a device given the device's name. You can use this function to test if a device exists and find out what type of device (serial, disk, etc.) before opening the device.

Request structure fields:

xiocb field	Description
xiocb_fcode	17 (CFE_CMD_DEV_GETINFO)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0
xiocb_psize	sizeof(xiocb_buffer_t)
plist:buf_ptr	Pointer to user buffer containing device name (null-terminated)
plist:buf_length	Size of user buffer containing device name (including null byte)

Return structure fields:

xiocb field	Description
plist:buf_devflags	Device type flags. You can use the mask CFE_DEV_MASK with this field to determine the device type, from one of the CFE_DEV_xxx constants.

Error codes:

Code	Description
CFE_ERR_DEVNOTFOUND	There is no device with the specified name.

8.5.16 CFE_CMD_ENV_ENUM

Enumerate environment variables. This function is used to walk through the environment, obtaining all of the environment variable names and values. You call this function repeatedly starting with *enum_idx* equal to zero and increment it until it returns an error.

Request structure fields:

xiocb field	Description
xiocb_fcode	20 (CFE_CMD_ENV_ENUM)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0
xiocb_psize	sizeof(iocb_envbuf_t)
plist:enum_idx	Index of environment variable to retrieve, starting with zero
plist:name_ptr	Pointer to user buffer to receive variable name (will be null terminated)
plist:name_length	Size of buffer to receive variable name
plist:val_ptr	Pointer to user buffer to receive variable value (will be null terminated)
plist:val_length	Size of buffer to receive variable value

Return structure fields:

xiocb field	Description
none	none

Error codes:

Code	Description
CFE_ERR_ENVNOTFOUND	*enum_idx* is too large, no more environment variables.

8.5.17 CFE_CMD_ENV_GET

Get the value of an environment variable.

Request structure fields:

xiocb field	Description
xiocb_fcode	22 (CFE_CMD_ENV_GET)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0
xiocb_psize	sizeof(iocb_envbuf_t)
plist:name_ptr	Pointer to user buffer containing variable name to retrieve (null-terminated)
plist:name_length	Size of buffer containing variable name (including null byte)
plist:val_ptr	Pointer to user buffer to receive variable value
plist:val_length	Size of buffer to receive variable value

Return structure fields:

xiocb field	Description
none	none

Error codes:

Code	Description
CFE_ERR_ENVNOTFOUND	Specified environment variable was not found.

8.5.18 CFE_CMD_ENV_SET

Set the value of an environment variable.

Request structure fields:

xiocb field	Description
xiocb_fcode	23 (CFE_CMD_ENV_SET)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0 for normal variables (will not be saved to the NVRAM), else CFE_FLG_ENV_PERMANENT to cause the environment variable to be written to NVRAM.
xiocb_psize	sizeof(iocb_envbuf_t)
plist:name_ptr	Pointer to user buffer containing variable name, null terminated
plist:name_length	Size of buffer containing variable name, including null byte
plist:val_ptr	Pointer to user buffer containing variable's new value, null terminated
plist:val_length	Size of buffer containing variable's new value, including null byte

Return structure fields:

xiocb field	Description
none	none

Error codes:

Code	Description
CFE_ERR_NOMEM	Insufficient memory
CFE_ERR_ENV_READONLY	The specified environment variable is read-only
CFE_ERR_IOERR	I/O error writing to the NVRAM device

8.5.19 CFE_CMD_ENV_DEL

Delete the value of an environment variable. The variable will also be deleted from the nonvolatile device, if present.

Request structure fields:

xiocb field	Description
xiocb_fcode	24 (CFE_CMD_ENV_DEL)
xiocb_status	0
xiocb_handle	0
xiocb_flags	0
xiocb_psize	sizeof(iocb_envbuf_t)
plist:name_ptr	Pointer to user buffer containing variable name, null terminated
plist:name_length	Size of buffer containing variable name, including null byte

Return structure fields:

xiocb field	Description
none	none

Error codes:

Code	Description
CFE_ERR_NOMEM	Insufficient memory
CFE_ERR_ENV_READONLY	Environment variable is read-only
CFE_ERR_IOERR	I/O error writing to the NVRAM device

LAST PAGE