Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor: Using CompactFlash

Application Note

December 2004

Order Number: 302456-003

Contents

Figures

Tables

Revision History

Date	Revision	Description
November 2004	003	Updated product branding. Change bars were retained from the previous release of this document (002).
August 2004	002	Added Section 1.2 and Section 6.0, plus replaced Appendix A, "Source Code".
June 2004	001	Initial release.

1.0 Introduction

This application note describes the hardware interface to a CompactFlash (CF) card connected in 'True IDE' mode to the Expansion Bus of Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor, and presents a Linux* device driver with basic functions to access the CF card. The device driver initializes the Expansion Bus timing and control registers to set up the interface that allows reading/writing the CF card. A simple Linux application program is provided to view directories, change directories, and view files in a CF card that has a FAT16 file system. This application note also briefly reviews the CF architecture, FAT16 file system, Expansion Bus architecture, and the platform used to test the device driver.

The basic functions in the device driver are necessary to connect the CF card to system-level software. The device driver can also be used to debug the platform, and although it is written for Linux, the details can pertain to most operating systems.

The following sections cover: Expansion Bus and CF architecture, Expansion Bus initialization, reading/writing the Expansion Bus, accessing the CF registers, reading/writing the sectors in the CF card, FAT16 file system on CF card, device driver architecture, and functions in the application code used to test the driver.

1.1 Related Documentation

Title	Document Number
Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Developer's Manual	252480
Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Datasheet	252479
Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Specification Update	252702
Intel® IXP400 Software Programmer's Guide	252539
CompactFlash Specification	(www.compactflash.org)

1.2 References

1. "CF+ and CompactFlash Specification Revision 2.0", CompactFlash Association, May 2003.
2. "Microsoft Extensible Firmware Initiative, FAT32 File System Specification, FAT: General Overview of On-Disk Format", Microsoft Corp., version 1.03, December 6, 2000.
3. "FAT16 Structure Information", Jack Dobiash, June 17, 1999, http://home.teleport.com/~brainy/fat16.htm.

1.3 Acronyms

ATA	AT Attachment
CF	CompactFlash
CFA	CompactFlash Association

CFI	CompactFlash Interface
DMA	Direct Memory Access
GPIO	General-Purpose Input/Output
IDE	Integrated Device Electronics
FAT	File Allocation Table
LBA	Logical Block Addressing
LSP	Linux Support Package
MVL	MontaVista* Linux
PIO	Programmed Input/Output
PHY	Physical Layer Device
PLL	Phase Lock Loop
SoC	System-on-Chip
XCVR	Transceiver

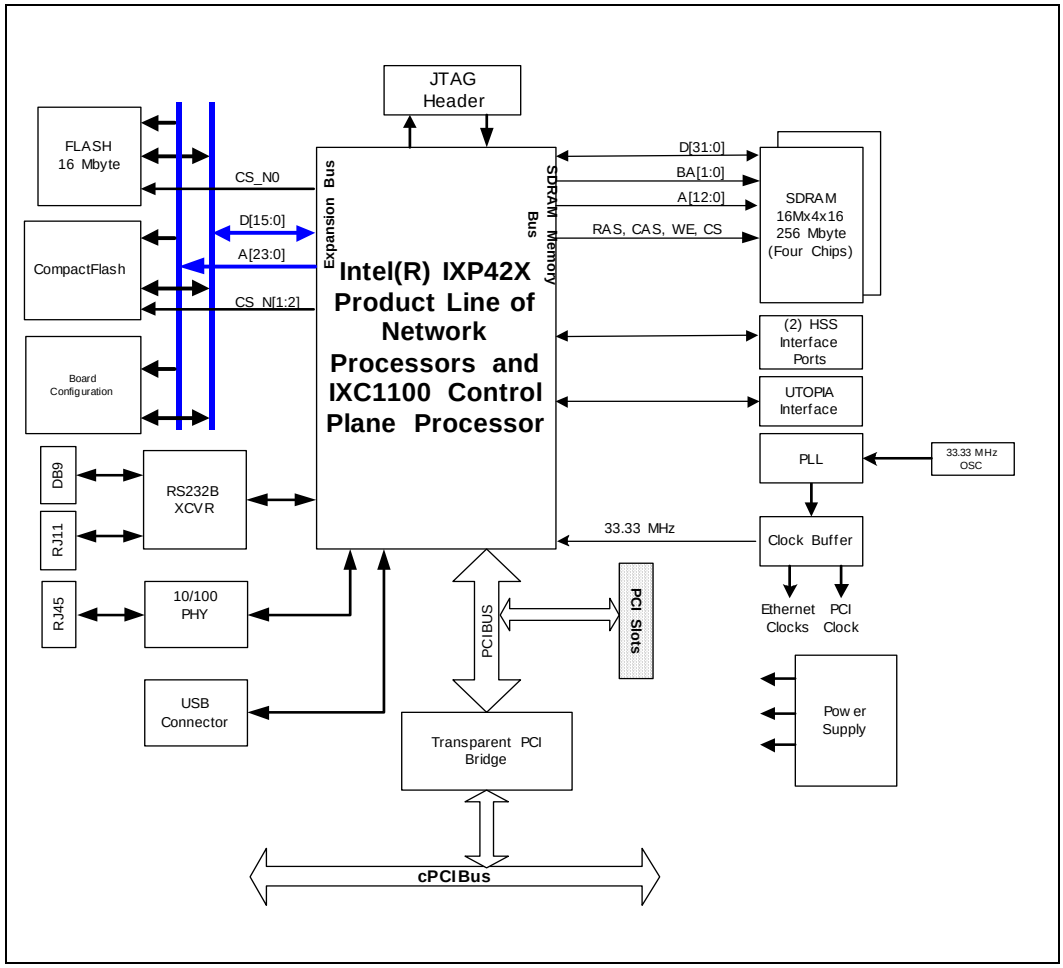
2.0 Hardware Overview

This section provides an overview of how to connect a CF card to the Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Expansion Bus.

2.1 The Processor

The Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor are highly integrated System-on-a-Chip (SoC) designs that provide greater flexibility and reduce system-development costs. These devices include features such as the UARTs, watchdog timers (WDT), general-purpose timers, three Network Processor Engines (NPEs) for two Ethernet and one UTOPIA interface, PC133 SDRAM, GPIO, PCI 2.2, and Expansion Bus controllers that can be interfaced and implemented in many applications such as embedded networking and communications. Figure 1 illustrates an example of the system interfaces of the IXP42X product line processors.

Figure 1. Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor System Block Diagram



2.2 Expansion Bus Overview

The Expansion Bus of the IXP42X product line processors supports a variety of types and speeds of I/O accesses for devices such as flash, SRAM, CF, Intel and Motorola*–style microprocessor interfaces, and the Texas Instruments* (TI) DSP standard Host-Port Interfaces (HPI). In most cases, these devices are supported seamlessly, without any additional glue logic.

The Expansion Bus provides a 24-bit address bus and an 8- or 16-bit-wide data interface for each of its eight independent chip-selects, and maps transfers between the internal bus and the external devices. Multiplexed and non-multiplexed address/data buses are both supported. Devices with a wider than 16-bit data bus interface are not supported; however, TI DSPs with internal bus widths of 32 bits can be integrated using the multiplexed HPI-16 interface. Please refer to the *Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Developer's Manual* for more details regarding the Expansion Bus Controller. The Expansion Bus interface signals are described in Table 1.

2.2.1 Expansion Bus Interface Signals

Table 1. Expansion Bus Interface Signals

Name	Type	Description
EX_CLK	I	Input clock signal – Not used in this application note.
EX_ALE	O	Address-latch enable – Not used in this application note.
EX_ADDR[23:0]	I/O	Expansion Bus address lines. Only EX_ADDR[10:0] are used in this application note.
EX_WR_N	O	Write strobe signal.
EX_RD_N	O	Read strobe signal.
EX_CS_N[7:0]	O	External chip selects for Expansion Bus. Only EX_CS1_N and EX_CS2_N are used in this application note.
EX_DATA[15:0]	I/O	Expansion Bus, bidirectional data.
EX_IOWAIT_N	I	Data ready/acknowledge from Expansion Bus devices – Not used in this application note.
EX_RDY[3:0]	I	Ready signals – Not used in this application note.

2.3 Expansion Bus Control and Configuration Registers

The Expansion Bus is controlled and configured by ten registers: eight timing and control registers, and two configuration registers.

Table 2. Expansion Bus Register Overview

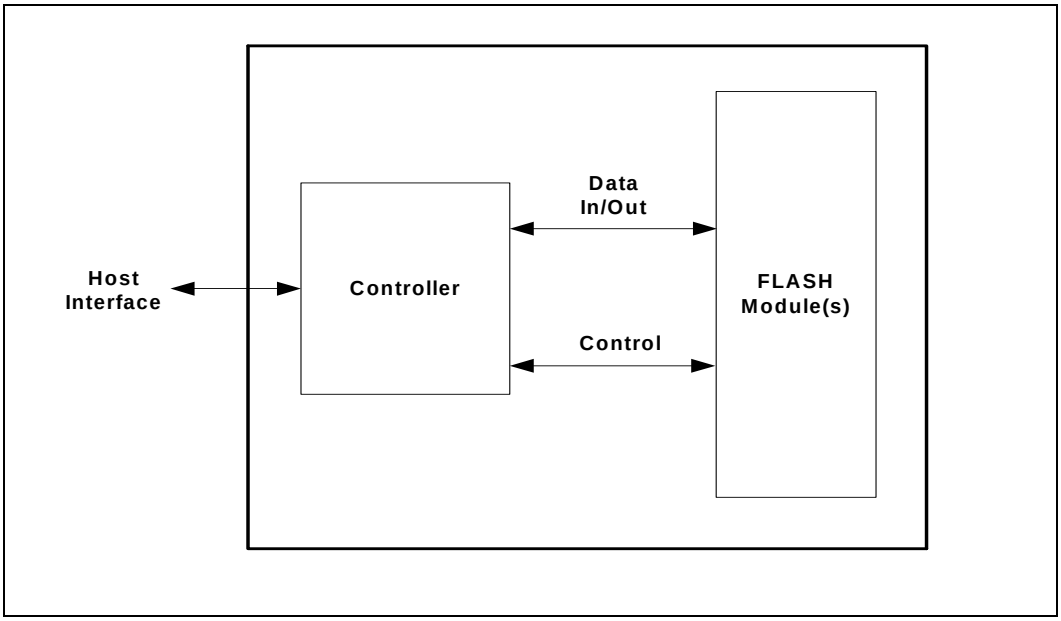
Address	R/W	Name	Description
0xC4000000	R/W	EXP_TIMING_CS0	Timing and Control Register for Chip Select 0
0xC4000004	R/W	EXP_TIMING_CS1	Timing and Control Register for Chip Select 1
0xC4000008	R/W	EXP_TIMING_CS2	Timing and Control Register for Chip Select 2
0xC400000C	R/W	EXP_TIMING_CS3	Timing and Control Register for Chip Select 3
0xC4000010	R/W	EXP_TIMING_CS4	Timing and Control Register for Chip Select 4
0xC4000014	R/W	EXP_TIMING_CS5	Timing and Control Register for Chip Select 5
0xC4000018	R/W	EXP_TIMING_CS6	Timing and Control Register for Chip Select 6
0xC400001C	R/W	EXP_TIMING_CS7	Timing and Control Register for Chip Select 7
0xC4000020	R/W	EXP_CNFG0	General Purpose Configuration Register 0
0xC4000024	R/W	EXP_CNFG1	General Purpose Configuration Register 1
0xC4000028	–	–	Reserved

2.4 CompactFlash

CompactFlash (CF) is a standard specification maintained by the CompactFlash Association (CFA) (www.compactflash.org). The CF Specification describes the connectivity and communications with I/O, storage modules, and compact memory devices. It is widely used in many applications such as portable and desktop computers, digital cameras, handheld data collection scanners, PCS phones, Pocket PCs, PDAs, handy terminals, personal communicators, audio recorders, MP3

players, monitoring devices, and set-top boxes. A block diagram of a CF storage card is shown in Figure 2. The controller interfaces with a host system allowing data to transfer to and from the flash memory module.

Figure 2. CF Storage Card Block Diagram



The CF card is a small-form-factor, PCMCIA-compatible, storage and I/O card based on the PCMCIA PC Card ATA specification, and includes a True IDE mode, which is compatible with the ATA/ATAPI-4 standard. CF cards function in three basic interface modes:

- **True IDE Mode**
- **PC Card I/O Mode**
- **PC Card Memory Mode**

The following generic descriptions of these modes should help designers choose one of them for connection to the Expansion Bus of the IXP42X product line processors.

TRUE IDE – A CF storage card also runs in True IDE mode that is electrically compatible with an IDE disk drive. A CF storage card is configured in True IDE mode only when the OE# pin (also called ATA SEL#) is grounded by the host during the power-off to power-on cycle. In this mode, the task file registers are also mapped into I/O address space, and the control signals IORD# and IOWR# are used to access I/O locations.

PC Card I/O – The control signals **IORD#** and **IOWR#** are also used to access I/O locations in the PC Card I/O mode, and the task file registers are mapped into I/O address space.

PC Card Memory – The control signals **OE#** and **WE#** are also used to access memory locations, and the task file registers are mapped into the memory space. In this mode, **REG#** pin is used during Memory Cycles to distinguish between Common Memory and Register (Attribute) Memory accesses. When **REG#** pin is High **(H**), it is used to denote a Common Memory access. When **REG#** is Low **(L**), it is used to denote an Attribute Memory access.

2.4.1 Interface Signals

Table 3 lists the CF interface signals. Note that the signals listed are for all three common modes of the CF card: PC Card I/O, True IDE and PC Card Memory. Some uncommon control signals, which may not be used in some modes, are also listed.

Table 3. Interface Signal Description

Signal Name	Type	Description
RESET#	I	Active Low – Reset CF. When the pin is high, this resets the CompactFlash Card.
RESET	I	Active High – Reset CF
CS0# (CE1)	I	Chip Select 0 (Card Select 0)
CS1# (CE2)	I	Chip Select 1 (Card Select 1)
A10-A0	I	Address Bits [10:0]
D15-D0	I/O	Data Bits [15:0]
INTRQ	O	Interrupt Request to the Host
REG#	I	Register Select
OE#/ATA SEL#	I	Output Enable/IDE Mode Enable
CSEL#	I	Cable Select
IOWR#	I	I/O Write Strobe
IORD#	I	I/O Read Strobe
VS1#, VS2#	O	Voltage Sense
WE#	I	Write Enable
INPACK#	O	Input. Acknowledge
IOIS16#/IOCS16#	O	16-Bit Transfer
PDIAG#	I/O	Pass Diagnostic
CD1#, CD2#	O	Card Detect
DASP#	I/O	Drive Active/Slave Present
Wait/IORDY	O	Wait/Ready
BVD1	I/O	Bus Voltage
BVD2	I/O	Bus Voltage
SPKR	I/O	Speaker
STSCHG#	I/O	Status Changed
RDY/BSY#	O	Ready/Busy
IREG#	O	Interrupt Request
WP	O	Write Protect
VCC		3.3 V
Ground (GND)		Ground

3.0 Hardware Interface Considerations

In every embedded application, implementations and requirements are different from one platform to another. Choosing which mode of the CF card to interface to the Expansion Bus depends upon product requirements. This section describes how to interface all three basic modes of the CF card to the Expansion Bus, but note that only the True IDE mode is supported by the device driver described in this application note.

CF cards supports both 3.3V and 5.0V operation and can interchange between 3.3V and 5.0V systems. Compatible with CF card 3.3V signals, the Expansion Bus operates at 3.3V I/O; therefore, the interface between the two requires no voltage-shift-level conversion. However, when interfacing a 5.0V CF card, voltage-shift-level converters are required. The Expansion Bus I/O buffers are designed to support up to eight loads, but the devices on the bus may not be able to quickly drive the large load. To account for this, timing on the Expansion Bus may be adjusted using the Expansion Bus timing and control registers for each control signal and chip select. If an edge rises slowly due to low drive strength, the IXP42X product line processors should wait an extra cycle before the value is read.

3.1 True IDE Mode Hardware Interface

This section describes the physical and logic interface of the CF card in True IDE mode to the Expansion Bus. The device driver presented in this application note assumes the CF card is connected in this mode. A CF card is configured in a True IDE mode of operation only when the **OE# (ATASEL#)** input signal is grounded by the host during the power-up sequence. In this mode, the CF card is accessible as if it were an IDE drive operating in PIO mode (non-DMA), and neither Memory nor Attribute registers are accessible. Figure 3 shows the interface of the Expansion Bus to the CF card in True IDE mode. The details of this interface signals are covered in the following paragraphs.

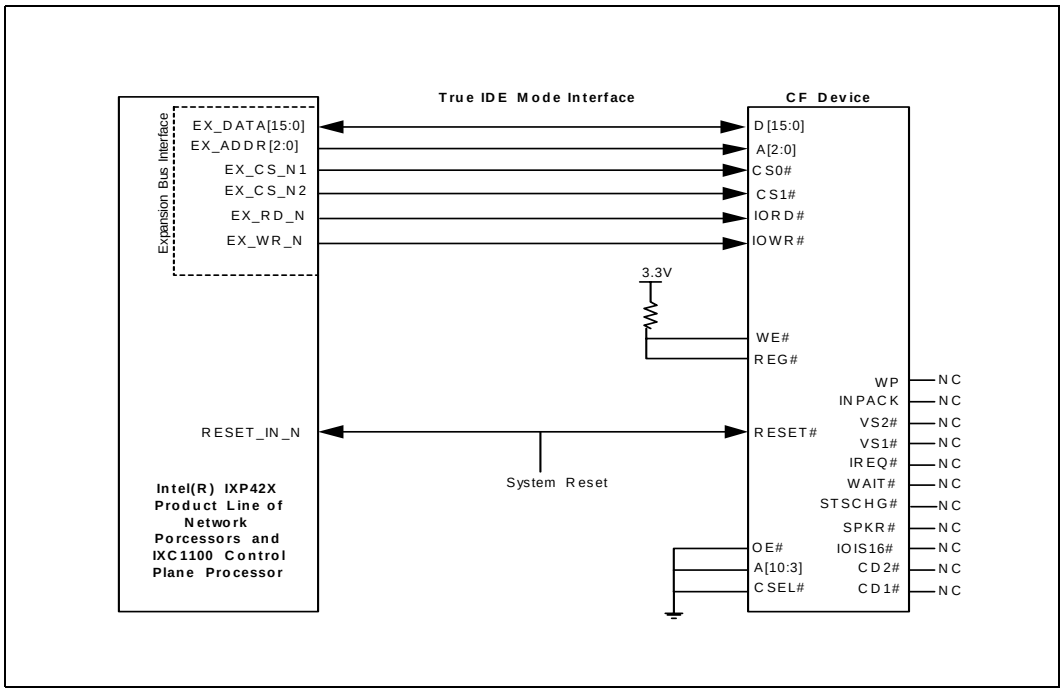
In True IDE mode, the CF card can simply be interfaced to the IXP42X product line processors through the Expansion Bus.

Note: The interface shown in Figure 3 does not support the IDE DMA mode. The Expansion Bus does not have DMA capabilities.

The CF chip select **CS0#** and **CS1#** are enabled by two chip select signals (**EX_CS_N1** and **EX_CS_N2**) from the Expansion Bus, and **IORD#** and **IOWR#** are controlled by **EX_RD_N** and **EX_WR_N** from the Expansion Bus. To meet the timing required by the CF Specification, the chip select is deasserted at least 20 ns after the IORD# or IOWR# is deasserted. Note that in True IDE mode, the CF **CS0#** is used for the Task File registers, and the chip select **CS1#** is used for the Device Control registers.

In True IDE mode, the CF **IOIS16#** is asserted low when the CF card is expecting a 16-bit data transfer. All Task File operations take place in byte mode using D7-D0, while all data transfers are using 16-bit word data. It is not necessary to control this signal; as shown in Figure 3, **IOIS16#** is not used. The A2-A0 address lines are used to select one of eight registers in the Task File. The usage of the required interface signals (in True IDE mode) are described and shown below.

Figure 3. CompactFlash – True IDE Mode Interface



- **A2-A0 –** The address lines from the CF card are directly connected to EX_ADDR[2:0] on the Expansion Bus.
- **A10-A3 –** The address lines from the CF card are not used and connected to Ground (GND).
- **D15-D0 –** The data lines from the CF card are directly connected EX_DATA[15:0] on the Expansion Bus.
- **CS0 –** Chip Select 0 line from the CF card is connected EX_CS_N1 on the Expansion Bus.
- **CS1 –** Chip Select 1 line from the CF card is connected EX_CS_N2 on the Expansion Bus.
- **IORD# –** The IO Read Strobe line from the CF card is connected to EX_RD_N on the Expansion Bus.
- **IOWR# –** The IO Write Strobe line from the CF card is connected to EX_WR_N on the Expansion Bus.
- **RESET# –** The Reset line from the CF card is directly connected to the power-on reset circuitry to reset the CF device every power-up sequence.
- **(ATASEL#)** OE# **–** The Output Enable line from the CF card is connected to Ground. To ensure the CF device operates in True IDE mode, this pin has to be grounded.
- **CSEL# –** The Card Select line from the CF card is not used and connected to Ground.
- **REG# –** The Register Select line from the CF card is not used in True IDE mode and connected to Ground.
- **WE# –** The Write Enable line from the CF card is not used in True IDE mode and connected to Ground.

CSEL# - The Card Select line from the CF card is not used and connected to Ground. This signal is used to select Master or Slave drive. In this hardware interface, **CSEL#** is connected to ground to indicate there is only one drive — the master drive — can be connected.

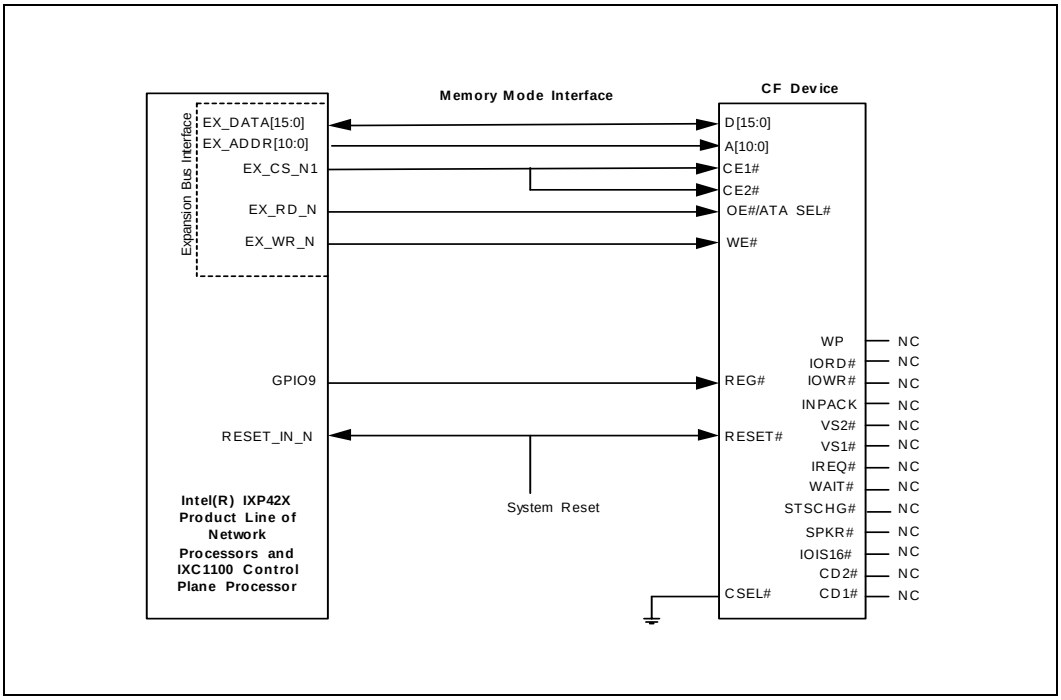
As discussed, the interface between the CF card and the IXP42X product line processors through the Expansion Bus in True IDE mode requires no external glue logic. Unused signals and control signals are also shown and designated as "NC" (No Connection). Only power and ground signals are not shown. The **RESET#** signal is required to directly connect to the power-on reset circuitry to reset the CF device every power-up sequence.

3.2 Memory Mode Hardware Interface

This section covers the physical and logic interface of the CF cards in Memory mode to the Expansion Bus. A CF card is configured in a Common Memory mode of operation only when the **OE# (ATASEL#)** input signal is high during the power-up sequence.

According to the CF Specification, the interface of the CF card to the host and access in Memory mode is similar to True IDE mode. The ATA registers are accessible through an external memory address generated by the host. However, True IDE mode is a 16-bit scheme, and Memory mode supports either 8- or 16-bit interface. True IDE mode and Memory mode use different Read/Write control signals to transfer data. In Memory mode, the **REG#** signal is used during Memory Cycles to distinguish between Common Memory and Register (Attribute) Memory accesses. When **REG#** is High **(H)**, it is used to denote a Common Memory access. When **REG#** is Low **(L)**, it is used to denote an Attribute Memory access. Common Memory mode is the default mode for the CF, and in this mode, the control signals **OE#** and **WE#** are also used to access memory locations, and the task file registers are mapped into direct addressing space. Figure 4 shows the main signal-to-signal connections between the Expansion Bus.

Figure 4. CompactFlash – Memory Mode Interface



As shown in Figure 4, the implementation requires no external glue logic for 16-bit interface. All the required interface signals are as follows:

- **A10-A0 –** The address lines from the CF card are directly connected to EX_ADDR[10:0] on the Expansion Bus.
- **D15-D0 –** The data lines from the CF card are directly connected EX_DATA[15:0] on the Expansion Bus.
- **CE1#(CS0#) and CE2#(CS1#) –** Both CE1# and CE2# lines from the CF card are connected EX_CS_N1 on the Expansion Bus.
- **IORD# –** The I/O Read Strobe line from the CF card is not used.
- **IOWR# –** The I/O Write Strobe line from the CF card is not used.
- **RESET# –** The Reset line from the CF card is directly connected to the power-on reset circuitry to reset the CF card at every power-up sequence.
- **OE# –** The Output Enable line from the CF card is connected EX_RD_N on the Expansion Bus.
- **CSEL# –** The Card Select line from the CF card is not used and connected to Ground.
- **REG# –** The Register Select line from the CF card is connected to GPIO pin 9 of the **IXP42X product line processors**.
- **WE# –** The Write Enable line from the CF card is connected to EX_RD_N on the Expansion Bus.

The NC (No Connection) signals can be ignored in this application, and in this I/O mode, both the **IOS16#** and **CSEL#** signals are not used and left NC.

The hardware control of the bus width is selected through the control of the **CE1#** and **CE2#** pins as summarized in Table 4.

Table 4. CE1# and CE2# Control Logic

CE1#	CE2#	Access Mode
0	0	D0-D15
1	0	D8-D15
0	1	D0-D7
1	1	Standby

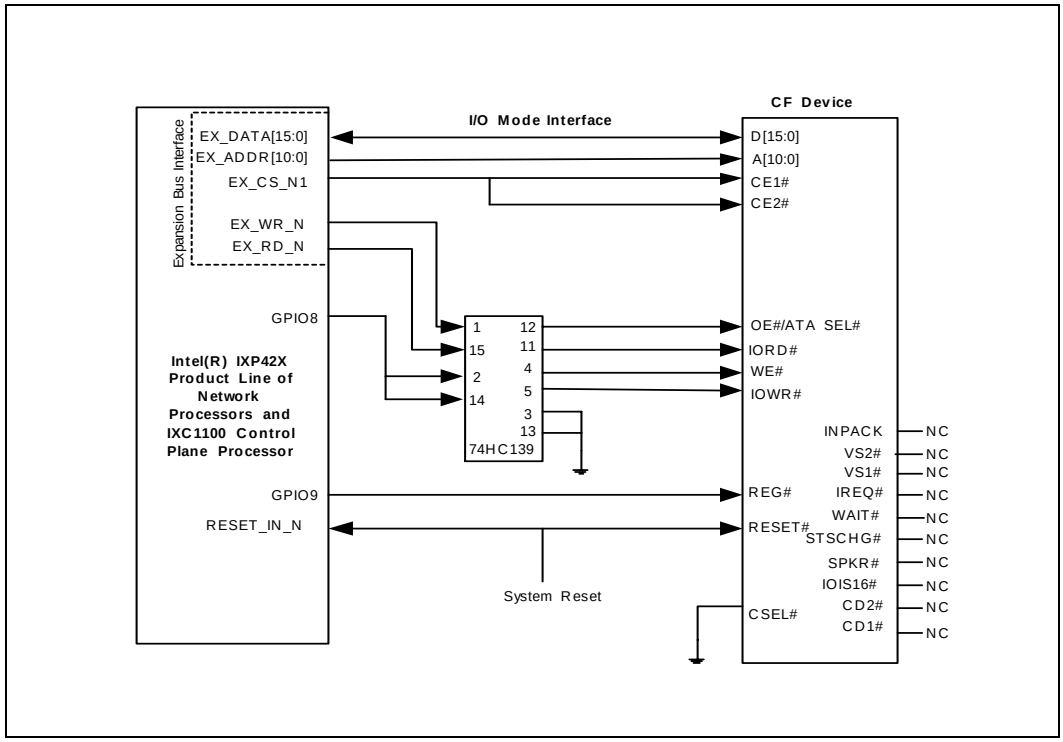
3.3 I/O Mode Hardware Interface

The previous sections have discussed the interfaces and the required signals between the CF device and the Expansion Bus in True IDE and Memory modes. This section presents the physical and logic interface of the CF devices to the Expansion Bus in I/O mode. In this mode, the control signals **IORD#** and **IOWR#** are used to access I/O locations in the PC Card I/O mode, and the task file registers are mapped into I/O address space. Note that in Memory mode, the CF card requires the **OE# (ATASEL#)** and **WE#** signals to access attribute memory when the **REG#** signal is low and access to common memory when the **REG#** signal is high. The **OE#** and **WE#** signals are needed in I/O mode to access to attribute memory, and the **IORD#** and **IOWR#** signals are used to access to common memory. Since the Expansion Bus has only two control signals — **EX_WR_N** and **EX_RD_N** — to implement and meet the logic interface and requirements between the CF card and the Expansion Bus, a decoder is needed. This decoder demuxes the **EX_WR_N** and

EX_RD_N signals to **OE#** and **WE#** or **IORD#** and **IOWR#.** As illustrated in Figure 5, GPIO pin 8 of the IXP42X product line processors controls the selections of the **OE#** and **WE#** or **IORD#** and **IOWR#** signals. If GPIO pin 8 is high, the **IORD#** and **IOWR#** are selected, and the **OE#** and **WE#** are selected when GPIO pin 8 is low.

As shown in Figure 5, the interface requires a 74HC139 Decoder or equivalent device. Unused control and status signals in this mode as well as the other two modes are designated NC "No Connection". The power signals are not shown. The following are the descriptions of the signals used for the interface:

Figure 5. CompactFlash – I/O Mode Interface



- **A10-A0 –** The address lines from the CF card are directly connected to EX_ADDR[10:0] on the Expansion Bus.
- **D15-D0 –** The data lines from the CF card are directly connected EX_DATA[15:0] on the Expansion Bus.
- **CE1#(CS0#) and CE2#(CS1#) –** Both CE1# and CE2# lines from the CF card are connected EX_CS_N1 on the Expansion Bus.
- **IORD# –** The IO Read Strobe line from the CF card is connected to Pin 11 of the 74HC139 decoder.
- **IOWR# –** The IO Write Strobe from line the CF is connected to Pin 5 of the 74HC139 decoder.
- **RESET# –** The Reset line from the CF card is directly connected to the power on reset circuitry in reset the CF card every power up sequence.
- **OE# –** The Output Enable line from the CF card is connected to Pin 12 of the 74HC139 decoder.

- **CSEL# –** The Card Select line from the CF card is not used and connected to Ground.
- **REG# –** The Register Select line from the CF card is connected to GPIO pin 9 of the IXP42X product line processors.
- **WE# –** The Write Enable line from the CF card is connected to Pin 4 of the 74HC139 decoder.

Eight- or 16-bit mode access refers to whether the data lines **D0-D15** are used to present one complete word transfer. 8/16-bit access controlled by **CE1#** and **CE2#** of the CF card is shown in Table 5.

Table 5. CE1# and CE2# Control Logic

CE#1	CE2#	Access Mode
0	0	D0-D15
1	0	D8-D15
0	1	D0-D7
1	1	Standby

4.0 Expansion Bus Operation

This section describes how to configure the CF and how to read/write from/to registers and sectors in a CF card.

4.1 Expansion Bus Configuration

In this application note, the device driver assumes the CF card is connected in True IDE mode to the Expansion Bus as shown in Figure 3 in Section 3.1.

In this application note it is assumed the CF chips selects are connected to Expansion Bus chip select 1 and 2. The bit-level layout for these two particular registers are shown below. For current information, please see the *Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Specification Update* and *Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Developer's Manual*):

Table 6. Timing and Control Registers for Chip Select 1

Register Name:	**EXP_TIMING_CS1**																												
Hex Offset Address:	0XC4000004															**Reset Hex Value:**	0x00000000										
Register Description:	Timing and Control Registers																												
Access: Read/Write																													
31	**30**	**29**	**28**	**27**	**26**	**25**			**22**	**21**	**20**	**19**			**16**	**15**	**14**	**13**			**10**	**9**			**6**	**5**	**4**	**3**	**2**	**1**	**0**
CSx_EN	(Rsvd)	T1		T2		T3			T4			T5			CYCLE_ TYPE	CNFG[3:0]				(Rsvd)			BYTE_RD16	HRDY_POL	MUX_EN	SPLT_EN	(Rsvd)	WR_EN	BYTE_EN

Table 7. Timing and Control Registers for Chip Select 2

Register Name:	**EXP_TIMING_CS2**																														
Hex Offset Address:	0XC4000008															**Reset Hex Value:**	0x00000000												
Register Description:	Timing and Control Registers																														
Access: Read/Write.																															
31	**30**	**29**	**28**	**27**	**26**	**25**			**22**	**21**	**20**	**19**			**16**	**15**	**14**	**13**			**10**	**9**			**6**	**5**	**4**	**3**	**2**	**1**	**0**
CSx_EN	(Rsvd)	T1		T2		T3				T4		T5				CYCLE_ TYPE		CNFG[3:0]				(Rsvd)			BYTE_RD16	HRDY_POL	MUX_EN	SPLT_EN	(Rsvd)	WR_EN	BYTE_EN

Table 8. Bit Level Definition for the Timing and Control Registers

Bits	Name	Description
31	CSx_EN	0 = Chip Select x disabled 1 = Chip Select x enabled
30		(Reserved)
29:28	T1 – Address timing	00 = Generate normal address phase timing 01 - 11 = Extend address phase by 1 - 3 clocks
27:26	T2 – Setup / Chip Select Timing	00 = Generate normal setup phase timing 01 - 11 = Extend setup phase by 1 - 3 clocks
25:22	T3 – Strobe Timing	0000 = Generate normal strobe phase timing 0001-1111 = Extend strobe phase by 1 - 15 clocks
21:20	T4 – Hold Timing	00 = Generate normal hold phase timing 01 - 11 = Extend hold phase by 1 - 3 clocks
19:16	T5 – Recovery Timing	0000 = Generate normal recovery phase timing 0001-1111 = Extend recovery phase by 1 - 15 clocks
15:14	CYC_TYPE	00 = Configures the Expansion Bus for Intel cycles. 01 = Configures the Expansion Bus for Motorola* cycles. 10 = Configures the Expansion Bus for HPI cycles. (HPI reserved for chip selects [7:4] only) 11 = Reserved
13:10	CNFG[3:0]	Device Configuration Size. Calculated using the formula: SIZE OF ADDR SPACE = $2^{(9+CNFG[3:0])}$ For Example: 0000 = Address space of 2^9 = 512 Bytes … 1000 = Address space of 2^{17} = 128 Kbytes … 1111 = Address space of 2^{24} = 16 Mbytes
9:7		(Reserved)
6	BYTE_RD16	Byte read access to Half Word device 0 = Byte access disabled. 1 = Byte access enabled.
5	HRDY_POL	HPI HRDY polarity (reserved for exp_cs_n[7:4] only) 0 = Polarity low true. 1 = Polarity high true.
4	MUX_EN	0 = Separate address and data buses. 1 = Multiplexed address / data on data bus.
3	SPLT_EN	0 = AHB split transfers disabled. 1 = AHB split transfers enabled.
2		(Reserved)
1	WR_EN	0 = Writes to CS region are disabled. 1 = Writes to CS region are enabled.
0	BYTE_EN	0 = Expansion Bus uses 16-bit-wide data bus. 1 = Expansion Bus uses only 8-bit data bus.

These two timing and control registers, EXP_TIMING_CS1 and EXP_TIMING_CS2, are configured as follows:

- Bit 31 will be set to '1' to enable the Expansion Bus.
- Bits 13 to 10 are set to '0' because the CF control registers only occupy a very small amount of memory space (512 bytes is ample)
- Bit 6 will be set to '1' to allow byte read access in the bus.
- Bit 1 will be set to '1' to allow write operation in the bus.
- Bit 0 will be set to '1' or '0', depending on 8-bit-wide or 16-bit-wide data bus is used.
- All other bits are set to 0

Note: When the data register in the CF card is accessed, 16-bit-wide data bus will be used, while 8-bit-wide data bus will be used for other register in the CF card. Refer to Section 5.3.

Configuration of these two registers is done in function CompactFlashExpBusInit() in Appendix A.2, on page 42. They are initialized with value `0xbfff0043`:

```
cs = (unsigned int *)IXP425_EXP_CS1;
*cs = 0xbfff0043;// 8-bit data bus as default

cs = (unsigned int *)IXP425_EXP_CS2;
*cs = 0xbfff0043;// 8-bit data bus as default
```

where `IXP425_EXP_CS1` and `IXP425_EXP_CS2` are defined in ixp425.h.

4.2 Switching Data Bus Width

The CF card has both 8-bit and 16-bit registers, which are explained in Section 5.3. The data bus width of the Expansion Bus must be switched to match the width of the CF register before the device driver tries to access the register.

The Expansion Bus can be switched into 16-bit-wide data bus with the following instructions (as in function setExpBusCS1To16BitDataBus() in Appendix A.2, on page 42):

```
cs = (unsigned int *)IXP425_EXP_CS1;
value = *cs;
*cs = value&(~1);  // set bit 0 to 0
```

Or switched into 8-bit-wide data bus with the following instructions (as in function setExpBusCS1To8BitDataBus() in Appendix A.2, on page 42):

```
cs = (unsigned int *)IXP425_EXP_CS1;
value = *cs;
*cs = value|1;  // set bit 0 to 1
```

4.3 Reading/Writing Expansion Bus

Reading/writing the Expansion Bus is done by first calling the ioremap() function defined in #include <asm/io.h> to prompt the memory management to update its page attributes tables and map the memory space for Chip Select 1 of the Expansion Bus to pointer ixp_exp_bus_cs1, and memory space for Chip Select 2 to ixp_exp_bus_cs2, as is done in function CompactFlashExpBusInit() in Appendix A.2, on page 42:

```
ixp_exp_bus_cs1=(unsigned

long)ioremap(IXP425_EXP_BUS_CS1_BASE_PHYS,512);

ixp_exp_bus_cs2 = (unsigned

long)ioremap(IXP425_EXP_BUS_CS2_BASE_PHYS, 512);
```

where `IXP425_EXP_BUS_CS1_BASE_PHYS` and `IXP425_EXP_BUS_CS2_BASE_PHYS` are defined in ixp425.h.

Then a call to one of the following functions, defined in #include <asm/io.h>

```
__raw_writew(data,__mem_pci(addr))
__raw_writeb(data,__mem_pci(addr))
__raw_readw(__mem_pci(addr))
__raw_readb(__mem_pci(addr))
```

where addr=ixp_exp_bus_cs1+offset or ixp_exp_bus_cs2+offset, will read/write a word or a byte from/to the Expansion Bus. The application note provides the following four functions in Appendix A.2, on page 42 to perform these operations easily:

- CompactFlashExpBusWriteW()
- CompactFlashExpBusWriteB(),
- CompactFlashExpBusReadW(),
- CompactFlashExpBusReadB();

5.0 CompactFlash Operations

Note: The functions described in this section are shown in Appendix A.2, on page 42, "CompactFlashIDE.c".

5.1 Access to the CompactFlash Registers

Control and access to the CF card in True IDE mode is done through a set of registers, the so-called CF-ATA registers or 'task file':

Table 9. True IDE Mode I/O Decoding

-CS1	-CS0	A2	A1	A0	-IORD=0	-IOWR=0	Note
1	0	0	0	0	RD Data	WR Data	8 or 16 bit
1	0	0	0	1	Error Register	Features	8 bit
1	0	0	1	0	Sector Count	Sector Count	8 bit
1	0	0	1	1	Sector No	Sector No	8 bit
1	0	1	0	0	Cylinder Low	Cylinder Low	8 bit
1	0	1	0	1	Cylinder High	Cylinder High	8 bit
1	0	1	1	0	Select Card/Head	Select Card/Head	8 bit
1	0	1	1	1	Status	Command	8 bit
0	1	1	1	0	Alt Status	Device Control	8 bit

These registers are addressed by three address lines and two chip select lines: A0, A1, A2, CS0, and CS1. To get access to these registers, a 8-bit value is used as offset. The two most significant bits of the offsets are used to distinguish Chip Select 1 or 2, and the four least significant bits are address offsets of the registers. Based on Table 9, these registers are hence denoted as follows:

```
#define CF_DATA         0x20
#define CF_ERROR        0x21
#define CF_SECT_CNT     0x22
#define CF_SECT_NUM     0x23
#define CF_CYL_L        0x24
#define CF_CYL_H        0x25
#define CF_DRV_HEAD     0x26
#define CF_STATUS       0x27

#define CF_FEATURES     0x21
#define CF_COMMAND      0x27

#define CF_ALTSTATUS    0x16
#define CF_DEV_CTR      0x16
```

These are used to read/write the CF registers, as in the following example for setting the CF card into Logical Block Address (LBA) mode:

```
WriteRegB(CF_DRV_HEAD, 0xE0);
```

where WriteRegB() is one of the following four functions in Appendix A.2, on page 42 that make use of the functions at the end of Section 4.3:

- WriteRegW()
- WriteRegB()
- ReadRegW()
- ReadRegB()

5.2 Wait for CompactFlash To Get Ready

Before any command is issued to the CF card, the card needs to be checked for readiness. The CF_STATUS register provides this status information. When the CF card is ready, the ready bit (bit 6 of the CF_STATUS register) must be 1, and the Busy bit (bit 7 of the CF_STATUS register) must be zero.

The application note provides a function Waiting_RDY_TO() in Appendix A.2, on page 42 calling ReadRegB(CF_STATUS) to check if the CF card is ready.

5.3 Switching Expansion Bus Data Width

In True IDE mode all CF registers are 8 bits wide and reside on byte-aligned addresses except for the CF_DATA register, which is 16 bits wide. The CF_DATA register is used by the host to read/write the CF data buffer.

In order to read/write from the CF_DATA register, the Expansion Bus must be configured to produce 16-bit-wide data access. In order to read/write the other CF internal register, the Expansion Bus must be configured to produce 8-bit-wide data access. Hence, depending on which CF registers are being accessed, it is necessary to switch the Expansion Bus data width.

Before the Expansion Bus data width is switched the code must ensure the last Expansion Bus transaction has completed. This is done by reading a CF internal register then using the returned value to force the host to stall until the data is returned.

The two functions in Appendix A.2, on page 42, setExpBusCS1To16BitDataBus()) and setExpBusCS1To8BitDataBus(), are used to switch the Expansion Bus data width.

5.4 Little and Big Endian Conversion

The data in the CF card is stored in little-endian format, while the host CPU is set into big-endian mode. It is therefore required to convert the data by calling the following function when reading from or writing to the CF card, as described in Section 5.5 and Section 5.6.

```
unsigned short byteSwap(unsigned short data)
{
    unsigned short tmp;
    tmp=(data<<8)|(data>>8);
    return tmp;
}
```

Note: The byte order is maintained when reading from or writing to the CF card.

5.5 Read from a Sector

Once a read command is issued by the host, the CF card fills the internal data buffer inside the CF with one sector worth of data. The host then repeatedly reads the CF_DATA register to retrieve that sector worth of data from the CF internal data buffer.

When reading from CF, logical block addressing (LBA) is used. The next sequence of steps show how this is set up.

- The LBA is written to the following registers:
  - Sector number: `CF_SECT_NUM`=LBA7~0
  - Cylinder Low: `CF_CYL_L` = LBA15~8
  - Cylinder High: `CF_CYL_H` = LBA23~16
  - Head: `CF_DRV_HEAD(LSB3~0)` = LBA27~24
- The sector count register CF_SECT_CNT is loaded with a value to indicate how many sectors to read.
- A read sectors command 0x20 is written to the command register `CF_COMMAND` to start the reading process:

  ```
  WriteRegB(CF_COMMAND, 0x20);
  ```

- The CF card will put a sector of data in the internal buffer, and then set the DRQ bit and clear the BSY bit in the CF_STATUS register.
- The host then can read the data from the internal buffer by repeatedly reading the CF_DATA register, as follows:

  ```
  ReadRegW(CF_DATA)
  ```

This application note provides the function `ReadSectorW()`in Appendix A.2, on page 42 to read a sector. This function also makes use of the operations described in Section 5.3 and Section 5.4.

5.6 Write to a Sector

In write operation, the host (after issuing a write sector command to the CF card) repeatedly writes to the CF_DATA register. Once the CF card's internal buffer is filled, the buffer's content is then written to a sector.

Steps to write to a sector are similar to those in Section 5.5 except that (after setting up all the other registers) a write sector command 0x30 is written to the CF_COMMAND register:

```
WriteRegB(CF_COMMAND, 0x30);
```

After this command is issued, the CF card will indicate it is ready by setting the DRQ bit and clearing the BSY bit in the CF_STATUS register. The host then can repeatedly write data to the internal buffer using:

```
WriteRegW(CF_DATA, val);
```

When finishing writing the data, the Expansion Bus is switched back to 8-bit width after issuing a read instruction (`ReadRegW(CF_STATUS)`) to make sure the Expansion Bus write operation is completed.

This application note provides function `WriteSectorW()` in Appendix A.2, on page 42 to write to a sector.

5.7 Read the Identify Information

When a value 0xEC is written to the CF_COMMAND register of the CF card:

```
WriteRegB(CF_COMMAND, 0xEC);
```

the internal buffer of the CF card is filled with 512 bytes of information, including the signature of the CF card, the default number of heads, cylinders, sectors per track, capability, as well as other parameters. To show the identify information, this application note provides function ReadIdenfyInformationW() in Appendix A.2, on page 42.

6.0 FAT16 File System on the CF Card

Note: The functions described in this section are shown in Section A.6, "CompactFlashFat16.c" on page 64.

This section discusses the File Allocation Table (FAT) file system and how it is used on the CF card. The demo code in this application note does not implement all the functions required to support the file system. The code only provides the basic functions to process the information in the CF card for the FAT16 file system, and to view directories, change directories, or view files in the CF card.

6.1 Master Boot Record

Depending on how the CF card is formatted, the first sector in a CF card contains either a Master Boot Record (MBR), or a BIOS Parameter Block (BPB). Before the CF card can be used, it must be formatted. This application note does not provide information about formatting the CF card,

which can be done on a PC. The MBR contains code to boot the computer and partition tables defining different sections of the drive. The MBR also provides information about where the BPB is located. The BPB contains parameters for the FAT file system.

After the CF card is initialized, the first thing to do is to call function checkMBR() in the demo code to check if an MBR exists in the CF Card. This function makes use of the fact that the first sector can only be an MBR or a BPB, and if it is a BPB, the first byte can only be 0xeb or 0xe9 which are jump instructions.

If a MBR does not exist, the first sector must be the boot sector that contains the BPB. Hence in this case the sector number for BPB is 0, BPB_LBA=0;

If a MBR exists, the 32-bit word starting at byte 0x1c6 in the first sector on the CF card is the sector number for the BPB of the first partition, namely, BPB_LBA=*(0x1c6);

The byte in 0x1c2 indicates the FAT file system type. There are three file system types: FAT12, FAT16, and FAT32. This application note only covers FAT16.

Table 10 and Table 11 show the MBR structure.

Table 10. MBR Structure

Offset	Description	Size
000h	Executable Code (Boots Computer)	446 Bytes
1BEh	1st Partition Entry (See Table 11)	16 Bytes
1CEh	2nd Partition Entry	16 Bytes
1DEh	3rd Partition Entry	16 Bytes
1EEh	4th Partition Entry	16 Bytes
1FEh	Executable Marker (55h AAh)	2 Bytes

Table 11. Partition Entry (Part of MBR)

Offset	Description	Size
00h	Current State of Partition (00h=Inactive, 80h=Active)	1 Byte
01h	Beginning of Partition - Head	1 Byte
02h	Beginning of Partition - Cylinder/Sector (See Below)	1 Word
04h	Type of Partition (See List Below)	1 Byte
05h	End of Partition - Head	1 Byte
06h	End of Partition - Cylinder/Sector	1 Word
08h	Number of Sectors Between the MBR and the First Sector in the Partition	1 Double Word
0Ch	Number of Sectors in the Partition	1 Double Word

6.2 BIOS Parameter Block

The BPB is at the beginning of the boot sector (see reference for the detailed BPB structure). The location of the boot sector is discussed in Section 6.1. The BPB contains parameters for the file system. The function ProcessBPB() in the demo code uses the information in BPB to calculate parameters, such as location of the root directory, etc.

6.3 Root Directory Location

After the parameters for the file system is obtained from the BPB, the first thing to do is to locate the root directory. The root directory information is contained in a set of sectors, and each sector in the set consists of 32-byte entries that employ the FAT directory structure as is described in Section 6.4. The total number of sectors in the set is `RootDirSectors` sectors, and the sector number of the first sector in the set is (BPB_LBA+FirstRootDirSecNum).

The parameters `RootDirSectors` and FirstRootDirSecNum are calculated using the information in the BPB as follows. The function GetBit(n) used in the following gets a 16-bit integer starting at byte n from the beginning of the boot sector.

RootDirSectors=((BPB_RootEntCnt*32)+ (BPB_BytsPerSec-1))/BPB_BytsPerSec;

where

BPB_RootEntCnt=Get16Bits(17), is the count of 32-byte directory entries in the root directory,

BPB_BytsPerSec=Get16Bits(11), is the count of bytes per sector,

FirstRootDirSecNum= BPB_RsvdSecCnt + (BPB_NumFATs*FATSz)

where

BPB_RsvdSecCnt=Get16Bits(14), is the number of reserved sectors in the reserved region,

BPB_NumFATs=Get8Bits(16), is the count of the FAT data structures (which is always 2).

FATSz= BPB_FATSz16=Get16Bits(22), is the count of sectors occupied by one FAT.

6.4 FAT Directory Structure

A FAT directory entry is a 32-byte structure, which represents either a file or a subdirectory.

If the file name or the subdirectory name has only 11 characters (8 characters for name, and 3 characters for name extension), only one 32-byte structure is needed, and this is a short directory entry.

If the file name or the subdirectory name has more than 11 characters, it needs multiple 32-byte structures. This is a long directory entry.

If byte 0 of the 32-byte structure is 0, this directory entry is free and there are no more directory entries after this one.

If byte 11 of the structure is 0x0f, this entry is part of a directory entry with long name.

If byte 11 of the structure is 0x10, this entry is for a subdirectory.

For a short directory entry (entry with a short name), the first 8 bytes of the structure are for the file or subdirectory name, and the following 3 bytes are for the file or subdirectory name extension. If the name requires less than 11 bytes, it is padded with 0x020. Function getShortFileName() in the demo code gets the short file name from the entry.

For a long directory entry (entry with a long name), the entry will start with a sequence of 32-byte structures with byte 11 equal to 0x0f, and end with a short directory entry. Each character of the file name will be represented by 2 bytes. Function getLongFileName() gets the long file name from the entry.

Bytes 20, 21, 26 and 27 of the 32-byte structure in a short directory entry comprise the first cluster number Dir_FstClus, which is used to determine either where the file content is located for a file name entry, or where the subdirectory list is located for a subdirectory entry.

For a subdirectory, the first two entries are the dot entry and dot-dot entry. The dot entry is for the current directory, and dot-dot entry is for the parent directory of the subdirectory. If the parent directory is the root directory, Dir_FstClus will be 0 for the dot-dot entry.

6.5 List the Root Directory

In this application note demo code, the function changeToRootDirectory() is called to list the root directory entries after checkMBR() and ProcessBPB() are called to process the MBR and BPB.

The set of sectors that contain the 32-byte FAT directory structure entries for the root directory is specified in Section 6.3. Function changeToRootDirectory() reads each sector in the set, and calls function listFileEntryInOneSector() to list all the file or directory entries in each sector. As described in Section 6.4, each sector is a sequence of 32-byte FAT directory structures.

6.6 List a Subdirectory

If a 32-byte FAT directory structure entry in the current directory is a subdirectory entry (namely, byte 11 is 0x10), the subdirectory can be browsed by passing the first cluster number Dir_FstClus (comprised of bytes 20, 21, 26 and 27 of the entry) to the function getClusFstSet() to get the first sector in the cluster that contains the directory entries for this subdirectory. The number of sectors in each cluster is BPB_SecPerClus, which is calculated in ProcessBPB(). After all the sectors in the current cluster are listed, call getNextFATentry() to get the next cluster. This is continued until isEndOfclusterChain() returns true.

The entries in each sector are listed by listFileEntryInOneSector(). And the function changeToDirectory() lists all the entries for a subdirectory entry.

As described in Section 6.4, the dot-dot entry in a subdirectory points to the parent directory. If Dir_FstClus of the dot-dot entry is 0, the parent is the root directory, and hence needs to be processed as discussed in Section 6.5. If Dir_FstClus is not 0, the Dir_FstClus in the dot-dot entry is processed as a regular directory entry.

6.7 Get Access to File Content

If the current directory entry is a file entry, namely byte 11 of the structure is not 0x10, then bytes 28, 29, 30, 31 of the structure comprise the file size DIR_FileSize. The first sector in the cluster that contains the file content can be obtained by passing the first cluster number Dir_FstClus (comprised of bytes 20, 21, 26 and 27) to the function getClusFstSet(). The number of sectors in each cluster is BPB_SecPerClus, which is calculated in ProcessBPB(). After the file content in all the sectors in the current cluster is read, call getNextFATentry() to get the next cluster. This process continues until all DIR_FileSize bytes are read. Function readFile() in this application note demo code performs this function.

7.0 CompactFlash Linux* Device Driver

The functions described previously for the Expansion Bus configuration and reading/writing the CF card are wrapped in a device driver, which is presented in this section. The CF device driver in this application note is a character driver. It provides a set of file operations for applications to get access to the CF card by going through the normal file system in Linux.

Note: The functions described in this section are shown in Appendix A.1, on page 36, "CompactFlashModuleSymbols.c".

The following file operations are defined in this device driver:

```
struct file_operations
CompactFlashModuleOperations = {
     NULL,
     NULL,                              /* lseek */
     CompactFlashModule_read,           /* read */
     CompactFlashModule_write,          /* write
*/
     NULL,                              /* readdir
*/
     NULL,                              /* poll */
     CompactFlashModule_ioctl,          /* ioctl
*/
     NULL,                              /* mmap */
     CompactFlashModule_open,           /* open */
     NULL,                              /* flush */
     CompactFlashModule_close,          /* release
*/
     NULL,                              /* sync */
     NULL,                              /* async */
     NULL,                              /* lock */
     NULL,                              /* ready */
     NULL,                              /* written
*/
```

```
    NULL,                                   /* sendPage
*/
     NULL                                     /*
get_umpatted_area */
   };
```

The driver also uses the following two functions when the device driver is loaded or unloaded:

```
            static int __init
CompactFlashModule_init_module(void);
            static void __init
CompactFlashModule_cleanup_module(void);
```

When the device driver is loaded into the system, CompactFlashModule_init_module() is called and the device is registered with register_chrdev(). When it is unloaded, the driver is unregistered with unregister_chrdev().

When applications open or close the device, the following functions are called, respectively:

```
            int CompactFlashModule_open (struct inode *inNum,
struct file *fp)
            int CompactFlashModule_close(struct inode *inNum,
struct file *fp)
```

The main operations are in functions CompactFlashModule_read(), CompactFlashModule_write(), and CompactFlashModule_ioctl(), which are described in the following sections.

7.1 Read the Device

The read function CompactFlashModule_read() in the driver reads a sector in the CF card by calling ReadSectorW() (described in Section 5.5) and using function copy_to_user() to pass the data to the application. The function copy_to_user() is required because applications cannot directly get access to the memory areas managed by the kernel. This function can also read the identify sector in the CF card.

7.2 Write the Device

The write function CompactFlashModule_write() in the driver writes a sector to the CF card by calling WriteSectorW() (described in Section 5.6) and using function copy_from_user() to pass the data from the application. The function copy_from_user() is required because applications cannot directly get access to the memory areas managed by the kernel.

7.3 Control the Device

The control function CompactFlashModule_ioctl() is used by the applications to read/write the registers in the CF card, check if the card is ready, display the timing and control registers in the Expansion Bus, start initializing the Expansion Bus, and set a flag `file-private_data` used by the device read/write functions. It also provide functions to view directories, change directories, view files, find the MBR, and process the BPB.

8.0 Application Code

Note: The functions described in this section are shown in Appendix A.8, on page 92, "CompactFlashApp.c".

This simple application code is used to test the driver. It starts with opening the device:

```
CFdriver = open("/dev/CompactFlashModule",O_RDWR);
```

Then it calls the device driver's control function:

```
rc = ioctl(CFdriver,CF_INIT_IDE, &passedArg);
```

to initialize the Expansion Bus, check if the CF card is ready, and read the identify information from the CF card.

The application then calls testFileSystemMenu() to display the following menu and wait for user input:

```
--------------------------------------------------------------------------------
- IxCompactFlashCodelet File System Demo    -
--------------------------------------------------------------------------------

Commands: cd/dir [/][.][..][dir name]; file or dir name; test!, exit!
```

The command "cd" is used to change the directory, and command "dir" lists the content of a directory. If a file name is entered, the content of the file is displayed (each byte is displayed as a character). Command "exit!" terminates the application.

If the command "test!" is entered, the application will switch to a test menu (TestMenu() is called) so that the user can perform the following low-level testing on the CF card: checking if the CF is ready, reading CF registers, writing to CF registers, showing Exp Bus Regs, reading the identify sector, reading from one sector, writing to one sector, showing content in a sector, finding the MBR, and processing the BPB.

```
------------------------------------------------------------------------
- IxCompactFlashCodelet Demo Menu      -
------------------------------------------------------------------------

Read/Write to CF Registers:
      1: check if the flash card is ready
      2: read all the CF registers
      3: read one CF register
      4: write to one CF register
      5: show Exp Bus Regs

View CF Identify Information:
      6: read the identify sector

Read/Write to sectors:
      7: read from one sector
      8: write to one sector

Display data:
      9: show one byte in a sector
      10: show next 10 bytes
      11: show one word in a sector
      12: show next 10 words

Display MBR & BPB:
      13: find MBR and BPB
      14: Process BPB data

Display Other Information:
      15: display a number to the LED on RF board
      16: toggle gpio pin 6 on RF board
100: Exit
```

9.0 Platform Used for Testing

The platform used to test the demo code in this application note is the Avila* GW2342 single-board computer made by Gateworks Corporation* (http://www.gateworks.com/avila_sbc.htm). This board supports the IXP42X product line processors at speeds up to 533 MHz. It provides a CF socket attached to the Expansion Bus, as described in Section 3.1.

The board also supports up to four Type III Mini PCI slots, two 10/100 Base-TX Ethernet channels, and two RS232 ports for management and debug. Additional features include up to 128 Mbytes SDRAM, five bits digital I/O, optional USB device port, real time clock, watchdog timer and a voltage/temperature monitor. Program storage consists of up to 32 Mbytes of on-board flash memory in addition to the CF socket. Software support includes Linux, VxWorks*, and Windows* CE .NET operating systems.

The Avila board is compatible with the IXDP425 / IXCDP1100 platform; therefore, the Linux Support Package (LSP) from MontaVista* Linux (MVL) 3.0 for the IXDP425 / IXCDP1100 platform is used for the board. No change is required to the software except when issuing the command to execute the kernel (as shown in Section 10.0), the user must specify the board's memory amount (64 Mbytes).

Note: This system is NOT suitable for HOT plugging. So even though the CF card is removable, the system must be powered off before changing the CF card.

10.0 Demo and 'Screen Shot'

Along with IXP400 software v1.3, the demo code was compiled using MontaVista Linux 3.0 with Red Hat* 7.3.

Note: The code in the application note is not IXP400 software release-dependent. Only the build setup of the IXP400 software release is used to build the device driver and the application.

Refer to the *Intel® IXP400 Software Release 1.5 Software Release Notes* for details about building modules for the IXP400 software release.

In the codelet subdirectory in IXP400 software, create a subdirectory "cfEng" and put the files in Appendix A, "Source Code" into this subdirectory with the following file structure:

```
ixp425_xscale_sw
   \------src
      \------codelets
            \------cfEng
               \------CompactFlash.h
               \------CompactFlashIDE.c
               \------CompactFlashIDE.h
               \------CompactFlashFat16.c
               \------CompactFlashFat16.h
               \------CompactFlashModuleSymbols.c
               \------component.mk
               \------cfApp
                     \------CompactFlashApp.c
                     \------MakeFile
```

To include the CF driver into the building process, the Makefile in \ixp425_xscale_sw is modified such that cfEng is added as follows:

BI_ENDIAN_CODELETS_COMPONENTS := hssAcc ethAcc usb timers dspEng cfEng

The following command will build all the modules:

```
make modules
```

To build the CF test application, go to subdirectory "cfApp" and execute a "make" command.

Refer to the user guide for the Avila GW2342 single-board computer for details about setting up the board.

The kernel can be downloaded to the Avila single-board computer using the following command:

```
load -r -v -b 0x001600000 zImage
```

Due to the size of the SDRAM on the board (64 Mbytes), the kernel is executed with the following command:

```
exec -c "console=ttyS0,115200 root=/dev/nfs ip=bootp mem=64M@0x00000000"
```

The IXP400 software library and the driver module for the CompactFlash are then loaded:

```
insmod ixp400_codelets_cfEng.o
mknod /dev/CompactFlashModule c 253 0
```

The CF test application is then started:

```
./CompactFlashApp
```

10.1 CompactFlash Demo Screen Shot

The following is a 'screen shot' of the demo code:

```
CompactFlash_module :: initialize IDE ...
CompactFlash_module :: initialize Exp Bus ...
CompactFlash_module :: Exp Bus cs1 =c5880000
CompactFlash_module :: Exp Bus cs2 =c5882000
CompactFlash_module :: CF is ready ...
CompactFlash_module :: show CF regs ...
CompactFlash_module :: CF_STATUS=50
CompactFlash_module :: CF_ALTSTATUS=50
CompactFlash_module :: CF_
CompactFlash_module :: CF_CYL_H=0
CompactFlash_module :: CF_CYL_L=0
CompactFlash_module :: CF_SECT_NUM=1
CompactFlash_module :: CF_SECT_CNT=1
CompactFlash_module :: CF_ERROR=1

CompactFlash_module :: read the identify block ...
CompactFlash_module :: CF signature =0X848a
CompactFlash_module :: Number of Cylinders =490 (0x1ea)
CompactFlash_module :: Number of Heads =8 (0x8)
CompactFlash_module :: Number of Sectors per tarck=32 (0x20)
CompactFlash_module :: Number of sectors per card =125440
```

```
(0x1ea00)
CompactFlash_module :: Capabilities =0x200
CompactFlash_module :: LBA supported

CompactFlash_module :: checking if there is a master boot record
...
processing MBR
CompactFlash_module :: it is FAT16 (larger than 32MB)
CompactFlash_module :: BPB_LBA=32

CompactFlash_module :: processing BPB ...
CompactFlash_module :: checking FAT type ...
CompactFlash_module :: CountofClusters=14834 ...
CompactFlash_module :: RootDirSectors=32 ...
CompactFlash_module :: BPB_RsvdSecCnt=1 ...
CompactFlash_module :: BPB_RootEntCnt
CompactFlash_module :: BPB_BytsPerSec=512 ...
CompactFlash_module :: BPB_FATSz16=122 ...
CompactFlash_module :: BPB_FATSz32=-450297728 ...
CompactFlash_module :: BPB_NumFATs=2 ...
CompactFlash_module :: BPB_TotSec16=0 ...
CompactFlash_module :: BPB_TotSec32=59616 ...
CompactFlash_module :: BPB_SecPerClus=4 ...
CompactFlash_module :: FATSz=122 ...
CompactFlash_module :: TotSec=59616 ...
CompactFlash_module :: it is FAT16

CompactFlash_module :: FirstDataSec=277 ...
CompactFlash_module :: FirstRootDirSecNum=24
CompactFlash_module :: FATstart=33 ...

CompactFlash_module :: List Root Directory
06/29/2004  03:28:16 PM   <DIR>      DIR1
06/29/2004  03:28:24 PM   <DIR>      DIR2
05/04/2004 02:00:12 PM       3304 README.TXT
05/07/2004 02:44:14 AM     13446132 QFJE.MP3
06/30/2004  04:46:20  PM    <DIR>          New Folder
efriewriewrwirwiuiw
05/04/2004 02:00:12 PM     3304 Copy of readme.txt
```

```
-------------------------------------------
- IxCompactFlashCodelet File System Demo    -
-------------------------------------------

Commands: cd/dir [/][.][..][dir name]; file or dir name; test!,
exit!


dir
*************************************************
CompactFlash_module :: List Root Directory
06/29/2004  03:28:16  PM    <DIR>          DIR1
06/29/2004  03:28:24  PM    <DIR>          DIR2
05/04/2004  02:00:12  PM              3304 README.TXT
05/07/2004  02:44:14  AM          13446132 QFJE.MP3
06/30/2004  04:46:20  PM    <DIR>          New Folder
05/04/2004  02:00:12  PM              3304 Copy of readme.txt

-------------------------------------------
- IxCompactFlashCodelet File System Demo    -
-------------------------------------------

Commands: cd/dir [/][.][..][dir name]; file or dir name; test!,
exit!



dir DIR1
*************************************************
06/29/2004  03:28:16  PM    <DIR>          DIR1
06/29/2004  03:28:16  PM    <DIR>          .
06/29/2004  03:28:16  PM    <DIR>          ..
05/04/2004  02:00:12  PM              3304 Copy of readme.txt
05/04/2004  02:00:12  PM              3304 readme.txt



test!
*************************************************
```

```
----------------------------------------
- IxCompactFlashCodelet Demo Menu     -
----------------------------------------

Read/Write to CF Registers:
     1: check if the flash card is ready
       2: read all the CF registers
        3: read one CF register
       4: write to one CF register
        5: show Exp Bus Regs

View CF Identify Information:
        6: read the identify sector

Read/Write to sectors:
        7: read from one sector
        8: write to one sector

Display data:
        9: show one byte in a sector
        10: show next 10 bytes
        11: show one word in a sector
        12: show next 10 words

Display MBR & BPB:
        13: find MBR and BPB
        14: Process BPB data

100: Exit
```

Appendix A Source Code

The source code shown in this appendix consists of the following files:

- CF device driver: **CompactFlashModuleSymbols.c**
- CF register and sector access: **CompactFlashIDE.c**
- Include file used by the device driver and the application: **CompactFlash.h**
- Include file used by the device driver: **CompactFlashIDE.h**
- FAT16 processing: **CompactFlashFat16.C**
- Include file used by the device driver: **CompactFlashFat16.h**
- Make file for the device driver: **component.mk**
- Application: **CompactFlashApp.c**
- Make file for the application: **Makefile.mk**

```
/**
 *
 * @author Intel Corporation
 * @date 5 May 2004
 *
 * @brief This file declares exported symbols for linux kernel module builds
 *
 * -- Intel Copyright Notice --
 *
 *
 * @par
 * Copyright 2004 Intel Corporation All Rights Reserved.
 *
 * @par
 * The source code contained or described herein and all documents
 * related to the source code ("Material") are owned by Intel Corporation
 * or its suppliers or licensors.  Title to the Material remains with
 * Intel Corporation or its suppliers and licensors.  The Material
 * contains trade secrets and proprietary and confidential information of
 * Intel or its suppliers and licensors.  The Material is protected by
 * worldwide copyright and trade secret laws and treaty provisions. Except for the
 * licensing of the source code hereunder, no part of the Material may be used,
 * copied, reproduced, modified, published, uploaded, posted, transmitted,
 * distributed, or disclosed in any way without Intel's prior express written
 * permission.
 *
 * @par
```

```
 * Except for the licensing of the source code as provided hereunder, no license under
 * any patent, copyright, trade secret or other intellectual property right is granted
 * to or conferred upon you by disclosure or delivery of the Materials, either
* expressly, by implication, inducement, estoppel or otherwise and any license under
 * such intellectual property rights must be express and approved by Intel in writing.
 *
 * @par
 * For further details, please see the file README.TXT distributed with
 * this software.
 * -- End Intel Copyright Notice --
 */
```

A.1 CompactFlashModuleSymbols.c

```
#define EXPORT_SYMTAB  1
/*
  * Put the system defined include files required.
  */

#include <stdio.h>
#include <taskLib.h>
#include <string.h>

#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <asm/uaccess.h>
#include <asm/arch/irqs.h>
#include <asm/io.h>

#include <linux/types.h>
#include <asm/hardware.h>

#include "IxAssert.h"
#include "ixp425.h"

#include "CompactFlash.h"
#include "CompactFlashIDE.h"
```

```
#define MSG(string, args...) printk(KERN_DEBUG "CompactFlashModule:" string, ##args)

#define MODULE_NAME "CompactFlashModule"
#define MODULE_VERSION "0.0.3"
MODULE_DESCRIPTION("CompactFlashModule for IXP425");
ssize_t CompactFlashModule_read(struct file *, char *, size_t , loff_t *);
ssize_t CompactFlashModule_write(struct file *, const char *, size_t, loff_t *);
int CompactFlashModule_open (struct inode *, struct file *);
int CompactFlashModule_close (struct inode *, struct file *);
int CompactFlashModule_ioctl(struct inode *,struct file *,unsigned int,unsigned long);
int CompactFlashModule_readdir(struct file *, void *, filldir_t);

extern void checkMBR();
extern void ProcessBPB(void);
extern void listCurrentDirectory();
extern void changeToDir(char *str, unsigned long changeFlag);
extern void readFileByName(char *str);
extern void changeToUpperDir(unsigned long changeFlag);
extern void changeToRootDirectory(unsigned long changeFlag);


struct file_operations CompactFlashModuleOperations = {
      NULL,
      NULL,                             /* lseek function ptr */
      CompactFlashModule_read,          /* read function ptr */
      CompactFlashModule_write,         /* write function ptr */
      NULL,                             /* readdir function ptr */
      NULL,                             /* poll function ptr */
      CompactFlashModule_ioctl,         /* ioctl function ptr */
      NULL,                             /* mmap function ptr */
      CompactFlashModule_open,           /* open function ptr */
      NULL,                             /* flush function ptr */
      CompactFlashModule_close,          /* release function ptr */
      NULL,                             /* sync function ptr */
      NULL,                             /* async function ptr */
      NULL,                             /* lock function ptr */
      NULL,                             /* readv function ptr */
      NULL,                             /* writev function ptr */
      NULL,                             /* sendPage function ptr */
      NULL                              /* get_umpatted_area */
    };

int CompactFlashChrDevVer=0;
```

```
static int __init CompactFlashModule_init_module(void)
{
    CompactFlashChrDevVer=register_chrdev(253,"CompactFlashModule",
                                          &CompactFlashModuleOperations);

    printk("CompactFlash_init_module :: "
        "LOADED SUCCESSFULLY CompactFlash MODULE\n");
    return IX_SUCCESS;
}

static void __init CompactFlashModule_cleanup_module(void)
{
    printk("Value=%d\n",unregister_chrdev(253, "CompactFlashModule"));
    printk("CompactFlash_cleanup_module :: UNLOADED CompactFlash MODULE\n");
}


ssize_t CompactFlashModule_write(struct file *fp,const char *buf,size_t num,loff_t *off)
{

    unsigned char x[512];

    copy_from_user(x, buf, 512);

    switch((int)fp->private_data)
    {
        case Byte_Access:
            WriteSectorB(x, (unsigned long) num);
            break;
        case Word_Access:
            WriteSectorW((unsigned short *)x, (unsigned long) num);
            break;
    }
    return(num);
}

ssize_t CompactFlashModule_read(struct file *file, char *buffer, size_t size, loff_t *ppos)
{
    size_t length = 0;
    unsigned char x[512];

    //x = (unsigned char *) kmalloc(512,GFP_KERNEL);
```

```
    switch((int)file->private_data)
    {
        case Byte_Access:
            ReadSectorB(x, (unsigned long) size);
            break;
        case Word_Access:
            ReadSectorW((unsigned short *)x, (unsigned long) size);
            break;

        case Read_Identify_Sector:
            ReadIdenfyInformationW((unsigned short *)x);
            ReadIdenfyInformation();
            break;
    }
    if(copy_to_user(buffer, (char *)x, 512))
        return -EFAULT;

    return(512);
}
int CompactFlashModule_ioctl(struct inode *inode, struct file *file,
                             unsigned int functionId, unsigned long arg)
{
    int CntCd;
    unsigned long x;
    char str[100];
    switch(functionId)
    {
        case IX_CF_CODELET_READ_REG:
            x=*IXP425_EXP_REG(*((unsigned long *)arg));
            if(copy_to_user((char *)arg, (char *)&x, 4))
            return -EFAULT;
        break;

    case IX_CF_CODELET_INIT_IDE:

        IDE_Init(0);
        break;

     case Byte_Access:
        (int)file->private_data = Byte_Access;
        break;
     case Word_Access:
```

```
    (int)file->private_data = Word_Access;
    break;

case Read_Identify_Sector:
    (int)file->private_data = Read_Identify_Sector;
    break;

case Check_Card:
    *(unsigned long *)arg=Check_RDY();
    break;

case Read_All_Regs:
    ReadAllCF_regs();
    break;
case Read_One_Reg:
    ReadOneCF_reg(arg);
    break;
case Write_One_Reg:
    WriteOneCF_reg((arg>>8)&0x0ff, arg&0x0ff);
    break;

case Show_Exp_Regs:
    x=*IXP425_EXP_REG(*((unsigned long *)arg));
    if(copy_to_user((char *)arg, (char *)&x, 4))
        return -EFAULT;
    break;



case FindMBRandBPB:
    checkMBR();
    break;

case ProcessBPBdata:
    ProcessBPB();
    break;

case ChangeToDir:
    if(copy_from_user(str, (char *)arg, 100))
        return -EFAULT;

    if((str[0]==0) || ((str[0]=='.')&&(str[1]==0)) )
        listCurrentDirectory();
    else if( (str[0]=='/')&&(str[1]==0) )
```

```
            changeToRootDirectory(1);
        else if( (str[0]=='.')&&(str[1]=='.')&&(str[2]==0))
            changeToUpperDir(1);
        else
            changeToDir(str, 1);
        break;

     case ShowDir:
        if(copy_from_user(str, (char *)arg, 100))
            return -EFAULT;

        if((str[0]==0) || ((str[0]=='.')&&(str[1]==0) ))
            listCurrentDirectory();
        else if( (str[0]=='/')&&(str[1]==0))
            changeToRootDirectory(0);
        else if( (str[0]=='.')&&(str[1]=='.')&&(str[2]==0))
            changeToUpperDir(0);
        else
            changeToDir(str, 0);
        break;

     case ReadFile:
        if(copy_from_user(str, (char *)arg, 100))
            return -EFAULT;
        readFileByName(str);
        break;



      default:
        printf("Invalid IOCTL is passed to driver %x \n", functionId);
        break;
    }

    return IX_SUCCESS;
}

int CompactFlashModule_open (struct inode *inNum, struct file *fp)
{
    return IX_SUCCESS;
}

int CompactFlashModule_close (struct inode *inNum, struct file *fp)
{
```

```
    return IX_SUCCESS;
}


module_init(CompactFlashModule_init_module);
module_exit(CompactFlashModule_cleanup_module);
```

A.2 CompactFlashIDE.c

```
#include <stdio.h>
#include <string.h>

#include <linux/module.h>

#include <asm/uaccess.h>
#include <asm/arch/irqs.h>
#include <asm/io.h>

#include <linux/types.h>
#include <asm/hardware.h>
#include <linux/delay.h>
#include "ixp425.h"

#include "CompactFlash.h"

#include "CompactFlashIDE.h"
#include "CompactFlashFat16.h"



unsigned char   Heads;      // number of heads as read from CF
unsigned short  SecTrack;   // sectors petr track as read from CF
unsigned long   LastSect=1000000;
```

```
unsigned long ixp_exp_bus_cs1, ixp_exp_bus_cs2;


void CompactFlashExpBusInit(void)
{
    unsigned int *cs;
    unsigned int value;


    printk("CompactFlash_module :: "     "initialize Exp Bus ... \n");


    cs = (unsigned int *)IXP425_EXP_CS1;
    value = 0xbfff0043;// 8-bit data bus as default
    *cs = value;
    ixp_exp_bus_cs1 = (unsigned long)ioremap(IXP425_EXP_BUS_CS1_BASE_PHYS, 512);


    cs = (unsigned int *)IXP425_EXP_CS2;
    value = 0xbfff0043;// 8-bit data bus as default
    *cs = value;
    ixp_exp_bus_cs2 = (unsigned long)ioremap(IXP425_EXP_BUS_CS2_BASE_PHYS, 512);


    printk("CompactFlash_module :: "     "Exp Bus cs1 =%x\n", ixp_exp_bus_cs1);
    printk("CompactFlash_module :: "     "Exp Bus cs2 =%x\n", ixp_exp_bus_cs2);
}


void setExpBusCS1To16BitDataBus(void)
{
    unsigned int *cs;
    unsigned int value;


    cs = (unsigned int *)IXP425_EXP_CS1;
    value = *cs;
    *cs = value&(~1);  // set bit 0 to 0
}
```

```
void setExpBusCS1To8BitDataBus(void)

{

    unsigned int *cs;

    unsigned int value;


    cs = (unsigned int *)IXP425_EXP_CS1;

    value = *cs;

    *cs = value|1;  // set bit 0 to 1

}



unsigned short byteSwap(unsigned short data)

{

    //convert from little to big endian

    unsigned short tmp;

    tmp=(data<<8)|(data>>8);

    return tmp;

}



void CompactFlashExpBusWriteW(unsigned long ixp_exp_bus_cs, unsigned short reg,
unsigned short data)

{

    __raw_writew(data,__mem_pci(ixp_exp_bus_cs + reg));

}


void CompactFlashExpBusWriteB(unsigned long ixp_exp_bus_cs, unsigned short reg,
unsigned short data)

{

    __raw_writeb(data,__mem_pci(ixp_exp_bus_cs + reg));

}


unsigned short CompactFlashExpBusReadW(unsigned long ixp_exp_bus_cs, unsigned
short reg)
```

```
{
    return ( __raw_readw(__mem_pci(ixp_exp_bus_cs + reg)) );
}

unsigned char CompactFlashExpBusReadB(unsigned long ixp_exp_bus_cs, unsigned short
reg)
{
    return ( __raw_readb(__mem_pci(ixp_exp_bus_cs + reg)) );
}

void WriteRegW(unsigned short addr, unsigned short data)
{
    if (addr&0x20)
        CompactFlashExpBusWriteW(ixp_exp_bus_cs1, addr&0x0f, data);
    else
        CompactFlashExpBusWriteW(ixp_exp_bus_cs2, addr&0x0f, data);
}

void WriteRegB(unsigned short addr, unsigned char  data)
{
    if (addr&0x20)
        CompactFlashExpBusWriteB(ixp_exp_bus_cs1, addr&0x0f, data);
    else
        CompactFlashExpBusWriteB(ixp_exp_bus_cs2, addr&0x0f, data);
}

unsigned short ReadRegW(unsigned short addr)
{
    unsigned short val;
    if (addr&0x20)
        val=CompactFlashExpBusReadW(ixp_exp_bus_cs1, addr&0x0f);
    else
        val=CompactFlashExpBusReadW(ixp_exp_bus_cs2, addr&0x0f);
    return (val);
```

```
}


unsigned char  ReadRegB(unsigned short addr)
{
    unsigned char val;
    if (addr&0x20)
        val=CompactFlashExpBusReadB(ixp_exp_bus_cs1, addr&0x0f);
    else
        val=CompactFlashExpBusReadB(ixp_exp_bus_cs2, addr&0x0f);
    return (val);
}



// INT will be cleared
void Waiting_RDY(void)
{
    unsigned short  Status, noReady;
    noReady=1;
    while(noReady)
    {
        Status=ReadRegB(CF_STATUS);
        if((Status & 0x40) && ((Status & 0x80) == 0)) noReady=0;
        // ready bit must be 1 AND Busy bit must be Zero...
    }
}


unsigned char Check_RDY(void)
{
    unsigned short  Status, Ready;
    Ready=0;
    Status=ReadRegB(CF_STATUS);
    if((Status & 0x40) && ((Status & 0x80) == 0)) Ready=1;
    return Ready;
}
```

```
// wait for delay with time out
unsigned char Waiting_RDY_TO(void)
{
    unsigned short    Status, noReady;
    unsigned short    count=0;

    noReady=1;
    while(noReady)
    {
        udelay(500);
        Status=ReadRegB(CF_STATUS);
        if((Status & 0x40) && ((Status & 0x80) == 0))
            noReady=0;
        else
            count++;
        if(count >= 50000)
        {
            if((Status & 0x40) == 0)
                return(2);
            else if((Status & 0x80))
                return(1);
            else if(((Status & 0x80)) && ((Status & 0x40) == 0))
                return(3);
        }
        // ready bit must be 1 AND Busy bit must be Zero...
    }

    return(0);
}


// INT will not be cleared
```

```
void Waiting_RDY_Alt(void)
{
    unsigned short  Status, noReady;
    noReady=1;
    while(noReady)
    {
        Status=ReadRegB(CF_ALTSTATUS);
        if((Status & 0x40) && ((Status & 0x80) == 0)) noReady=0;
    }
}



// identify drive
void ReadIdenfyInformation(void)
{
    unsigned short DriveID[256];
    unsigned char *ptr;
    ptr=DriveID;

    WriteRegB(CF_DRV_HEAD, 0xA0);// identify drive
        Waiting_RDY();

        WriteRegB(CF_DEV_CTR, 0x02);      // disable Inten
            Waiting_RDY();

    ReadIdenfyInformationW(DriveID);

    printk("CompactFlash_module :: "     "CF signature =0X%x\n",
byteSwap(DriveID[0]));

    printk("CompactFlash_module :: "     "Number of Cylinders =%d (0x%x)\n",
byteSwap(DriveID[1]),byteSwap(DriveID[1]));

        Heads=byteSwap(DriveID[3]);
        printk("CompactFlash_module :: "     "Number of Heads =%d (0x%x)\n",
```

```
byteSwap(DriveID[3]),byteSwap(DriveID[3]));



        SecTrack=byteSwap(DriveID[6]);

        printk("CompactFlash_module :: "     "Number of Sectors per tarck =%d
(0x%x)\n", byteSwap(DriveID[6]),byteSwap(DriveID[6]));




        printk("CompactFlash_module :: "     "Number of sectors per card =%d
(0x%x)\n",
(byteSwap(DriveID[7])<<16)+byteSwap(DriveID[8]),(byteSwap(DriveID[7])<<16)+byteSwa
p(DriveID[8]));



        printk("CompactFlash_module :: "     "Capabilities =0x%x\n",
byteSwap(DriveID[49]));

        if(!((byteSwap(DriveID[49]) >> 8) & BIT(1)))

        printk("CompactFlash_module :: "     "LBA not supported\n");

        else

        printk("CompactFlash_module :: "     "LBA supported\n");

}



void ReadAllCF_regs()

{

    unsigned char tmp;

    tmp=ReadRegB(CF_STATUS);

    printk("CompactFlash_module :: "     "CF_STATUS=%x\n", tmp);



    tmp=ReadRegB(CF_ALTSTATUS);

    printk("CompactFlash_module :: "     "CF_ALTSTATUS=%x\n", tmp);



    tmp=ReadRegB(CF_DRV_HEAD);

    printk("CompactFlash_module :: "     "CF_DRV_HEAD=%x\n", tmp);



    tmp=ReadRegB(CF_CYL_H);

    printk("CompactFlash_module :: "     "CF_CYL_H=%x\n", tmp);



    tmp=ReadRegB(CF_CYL_L);
```

```
    printk("CompactFlash_module :: "     "CF_CYL_L=%x\n", tmp);

    tmp=ReadRegB(CF_SECT_NUM);
    printk("CompactFlash_module :: "     "CF_SECT_NUM=%x\n", tmp);

    tmp=ReadRegB(CF_SECT_CNT);
    printk("CompactFlash_module :: "     "CF_SECT_CNT=%x\n", tmp);

    tmp=ReadRegB(CF_ERROR);
    printk("CompactFlash_module :: "     "CF_ERROR=%x\n\n", tmp);
}

void ReadOneCF_reg(unsigned char reg)
{
    unsigned char tmp;

    switch(reg)
    {
    case CF_STATUS:
        tmp=ReadRegB(CF_STATUS);
        printk("CompactFlash_module :: "     "CF_STATUS=0x%x\n", tmp);
    break;

    case CF_ALTSTATUS:
        tmp=ReadRegB(CF_ALTSTATUS);
        printk("CompactFlash_module :: "     "CF_ALTSTATUS=0x%x\n", tmp);
    break;

    case CF_DRV_HEAD:
        tmp=ReadRegB(CF_DRV_HEAD);
        printk("CompactFlash_module :: "     "CF_DRV_HEAD=0x%x\n", tmp);
    break;

    case CF_CYL_H:
```

```
            tmp=ReadRegB(CF_CYL_H);
            printk("CompactFlash_module :: "     "CF_CYL_H=0x%x\n", tmp);
        break;

        case CF_CYL_L:
            tmp=ReadRegB(CF_CYL_L);
            printk("CompactFlash_module :: "     "CF_CYL_L=0x%x\n", tmp);
        break;

        case CF_SECT_NUM:
            tmp=ReadRegB(CF_SECT_NUM);
            printk("CompactFlash_module :: "     "CF_SECT_NUM=0x%x\n", tmp);
        break;

        case CF_SECT_CNT:
            tmp=ReadRegB(CF_SECT_CNT);
            printk("CompactFlash_module :: "     "CF_SECT_CNT=0x%x\n", tmp);
        break;

        case CF_ERROR:
            tmp=ReadRegB(CF_ERROR);
            printk("CompactFlash_module :: "     "CF_ERROR=0x%x\n\n", tmp);
        break;
        }
}


void WriteOneCF_reg(unsigned short reg, unsigned char  data)
{
    switch(reg)
    {
    case CF_COMMAND:
        WriteRegB(CF_COMMAND, data);
    break;
```

```
        case CF_DEV_CTR :
            WriteRegB(CF_DEV_CTR , data);
        break;

        case CF_DRV_HEAD:
            WriteRegB(CF_DRV_HEAD, data);
        break;

        case CF_CYL_H:
            WriteRegB(CF_CYL_H, data);
        break;

        case CF_CYL_L:
            WriteRegB(CF_CYL_L, data);
        break;

        case CF_SECT_NUM:
            WriteRegB(CF_SECT_NUM, data);
        break;

        case CF_SECT_CNT:
            WriteRegB(CF_SECT_CNT, data);
        break;

        case CF_FEATURES:
            WriteRegB(CF_FEATURES, data);
        break;
        }
    }
```

```
void IDE_Init(unsigned char drive)
{
    unsigned short tmp, x[1000];

    printk("CompactFlash_module :: "     "initialize IDE ... \n");
    CompactFlashExpBusInit();

    tmp=Waiting_RDY_TO();
    if(!tmp)
        printk("CompactFlash_module :: "     "CF is ready ... \n");
    else
    {
            tmp=ReadRegB(CF_STATUS);
        printk("CompactFlash_module :: "     "CF is not ready; CF_STATUS=%x\n",
tmp);
    }

    printk("CompactFlash_module :: "     "show CF regs ... \n");
    ReadAllCF_regs();

    ReadIdenfyInformation();

    // set to LBA
    WriteRegB(CF_DRV_HEAD, 0xE0);

    Fat16Init();
}




void ReadSectorW(unsigned short *buff, unsigned long LBALocation)
{
```

```
    unsigned short cnt, tmp, *ptr;
    LastSect=LBALocation;

    WriteRegB(CF_DRV_HEAD, ((LBALocation >> 24) & 0xFF) | 0xE0);   // BitsForDH
    WriteRegB(CF_CYL_H, (LBALocation >> 16) & 0xFF); // BitsForCF_CYL_H
    WriteRegB(CF_CYL_L, (LBALocation >> 8) & 0xFF); // BitsForCF_CYL_L
    WriteRegB(CF_SECT_NUM, LBALocation & 0xFF);// BitsForSect
    WriteRegB(CF_SECT_CNT, 1);
    WriteRegB(CF_COMMAND, 0x20);    // read sectors

    Waiting_RDY_Alt();

    setExpBusCS1To16BitDataBus();
    ptr=buff;
    for(cnt=0; cnt < 256; cnt++)
    {
    tmp=ReadRegW(CF_DATA);
    *ptr++=byteSwap(tmp);
    }
    setExpBusCS1To8BitDataBus();
}


void ReadSectorB(unsigned char *buff, unsigned long LBALocation)
{
    unsigned short cnt;
    unsigned char *ptr;
    LastSect=LBALocation;

    WriteRegB(CF_DRV_HEAD, ((LBALocation >> 24) & 0xFF) | 0xE0);   // BitsForDH
    WriteRegB(CF_CYL_H, (LBALocation >> 16) & 0xFF); // BitsForCF_CYL_H
    WriteRegB(CF_CYL_L, (LBALocation >> 8) & 0xFF); // BitsForCF_CYL_L
    WriteRegB(CF_SECT_NUM, LBALocation & 0xFF);// BitsForSect
    WriteRegB(CF_SECT_CNT, 1);
```

```
    WriteRegB(CF_COMMAND, 0x20);    // read sectors


    Waiting_RDY_Alt();

    WriteRegB(CF_FEATURES, 0x01);    // enable 8 bit transfer
    WriteRegB(CF_COMMAND, 0xEF);    // set features

    Waiting_RDY_Alt();

    ptr=buff;
    for(cnt=0; cnt < 512; cnt++)
    *ptr++=ReadRegB(CF_DATA);

    WriteRegB(CF_FEATURES, 0x81);    // disable 8bit transfer
    WriteRegB(CF_COMMAND, 0xEF);    // set features
}


void ReadIdenfyInformationW(unsigned short *buff)
{
    unsigned short cnt, tmp, *ptr;

    printk("CompactFlash_module :: "     "read the identify block ... \n");
    WriteRegB(CF_COMMAND, 0xEC);          // identify drive

    Waiting_RDY_Alt();

    ptr=buff;
    setExpBusCS1To16BitDataBus();
    for(cnt=0; cnt < 256; cnt++)
    {
    tmp=ReadRegW(CF_DATA);
    *ptr++=byteSwap(tmp);
```

```
    }

    ReadRegB(CF_STATUS);

    setExpBusCS1To8BitDataBus();
}

void ReadIdenfyInformationB(unsigned char *buff)
{
    unsigned short cnt;
    unsigned char *ptr;

    printk("CompactFlash_module :: " "read the identify block ... \n");
    WriteRegB(CF_COMMAND, 0xEC);  // identify drive

    Waiting_RDY_Alt();

    WriteRegB(CF_FEATURES, 0x01);     // enable 8 bit transfer
    WriteRegB(CF_COMMAND, 0xEF);    // set features

    Waiting_RDY_Alt();

    ptr=buff;
    for(cnt=0; cnt < 512; cnt++)
    *ptr++=ReadRegB(CF_DATA);

    WriteRegB(CF_FEATURES, 0x81);    // disable 8bit transfer
    WriteRegB(CF_COMMAND, 0xEF);    // set features
}

void ReadSectorMod(unsigned char *buff, unsigned long LBALocation)
```

```
{
    unsigned short cnt, *ptr;
    LastSect=0;

    WriteRegB(CF_DRV_HEAD, ((LBALocation >> 24) & 0xFF) | 0xA0);   // BitsForDH
    WriteRegB(CF_CYL_H, (LBALocation >> 16) & 0xFF); // BitsForCF_CYL_H
    WriteRegB(CF_CYL_L, (LBALocation >> 8) & 0xFF); // BitsForCF_CYL_L
    WriteRegB(CF_SECT_NUM, LBALocation & 0xFF);// BitsForSect
    WriteRegB(CF_SECT_CNT, 1);
    WriteRegB(CF_COMMAND, 0x20);    // read sectors

    Waiting_RDY_Alt();

    setExpBusCS1To16BitDataBus();
    ptr=(unsigned short *)buff;
    for(cnt=0; cnt < 256; cnt++)
    *ptr++=ReadRegW(CF_DATA);
    setExpBusCS1To8BitDataBus();
}


void Waiting_DRQ_Alt(void)
{
    unsigned short  Status, noReady;
    noReady=1;
    while(noReady)
    {
        Status=ReadRegB(CF_ALTSTATUS);
        if((Status & 0x08) && ((Status & 0x80) == 0)) noReady=0;
    }
}


void WriteSectorW(unsigned short *buff, unsigned long LBALocation)
```

```
{
    unsigned short cnt, *ptr, tmp;
    LastSect=LBALocation;

    printk("CompactFlash_module :: "     "writing to a sector ... \n");

    WriteRegB(CF_DRV_HEAD, ((LBALocation >> 24) & 0xFF) | 0xE0);   // BitsForDH
    WriteRegB(CF_CYL_H, (LBALocation >> 16) & 0xFF); // BitsForCF_CYL_H
    WriteRegB(CF_CYL_L, (LBALocation >> 8) & 0xFF); // BitsForCF_CYL_L
    WriteRegB(CF_SECT_NUM, LBALocation & 0xFF);// BitsForSect
    WriteRegB(CF_SECT_CNT, 1);
    WriteRegB(CF_COMMAND, 0x30);    // write sectors

    Waiting_DRQ_Alt();

    setExpBusCS1To16BitDataBus();
    ptr=buff;
    for(cnt=0; cnt < 256; cnt++)
    {
    tmp=byteSwap(*ptr++);
    WriteRegW(CF_DATA, tmp);
    }

    ReadRegB(CF_STATUS);
    setExpBusCS1To8BitDataBus();

    Waiting_RDY_Alt();

    printk("CompactFlash_module :: "     "done with writing to a sector\n");
}


void WriteSectorB(unsigned char *buff, unsigned long LBALocation)
{
```

```
    unsigned short cnt, *ptr;
    LastSect=LBALocation;

    printk("CompactFlash_module :: "     "writing to a sector ... \n");

    WriteRegB(CF_DRV_HEAD, ((LBALocation >> 24) & 0xFF) | 0xE0);   // BitsForDH
    WriteRegB(CF_CYL_H, (LBALocation >> 16) & 0xFF); // BitsForCF_CYL_H
    WriteRegB(CF_CYL_L, (LBALocation >> 8) & 0xFF); // BitsForCF_CYL_L
    WriteRegB(CF_SECT_NUM, LBALocation & 0xFF);// BitsForSect
    WriteRegB(CF_SECT_CNT, 1);
    WriteRegB(CF_COMMAND, 0x30);    // write sectors

    Waiting_DRQ_Alt();

    setExpBusCS1To16BitDataBus();
    ptr=(unsigned short *)buff;
    for(cnt=0; cnt < 256; cnt++)
    WriteRegW(CF_DATA, *ptr++);
    setExpBusCS1To8BitDataBus();

    Waiting_RDY_Alt();

    printk("CompactFlash_module :: "     "done with writing to a sector\n");
}
```

A.3 CompactFlash.h

```
#ifndef __COMPACTFLASH_H__
#define __COMPACTFLASH_H__


/* Ioctl values for Linux */
#define IX_CF_CODELET_READ_REG         800001
```

```
#define IX_CF_CODELET_INIT_IDE         800002
#define IX_CF_CODELET_TOGGLE_GPIO      800003
#define IX_CF_CODELET_DSIPLAY_HEX      800004



#define IXP425_EXP_CS0_OFFSET  0x00
#define IXP425_EXP_CS1_OFFSET   0x04
#define IXP425_EXP_CS2_OFFSET   0x08
#define IXP425_EXP_CS3_OFFSET   0x0C
#define IXP425_EXP_CS4_OFFSET   0x10
#define IXP425_EXP_CS5_OFFSET   0x14
#define IXP425_EXP_CS6_OFFSET   0x18
#define IXP425_EXP_CS7_OFFSET   0x1C
#define IXP425_EXP_CFG0_OFFSET 0x20
#define IXP425_EXP_CFG1_OFFSET 0x24
#define IXP425_EXP_CFG2_OFFSET 0x28
#define IXP425_EXP_CFG3_OFFSET 0x2C

#define Check_Card    1
#define Read_All_Regs 2
#define Read_One_Reg  3
#define Write_One_Reg 4
#define Show_Exp_Regs 5

#define Read_Identify_Sector   6

#define Read_From_One_Sector   7
#define Write_To_One_Sector    8



#define Show_One_Byte        9
#define Show_Next_10_Bytes  10
#define Show_One_Word       11
#define Show_Next_10_Words  12
```

```
#define FindMBRandBPB        13
#define ProcessBPBdata       14

#define DSIPLAY_HEX         15
#define TOGGLE_GPIO         16


#define ChangeToDir         18// cd dir_name, cd /, cd .,  cd ..
#define ShowDir             19// dir
#define ReadFile            20// file name

#define GoTestMenu          21// test     //go to test menu


#define Byte_Access         51
#define Word_Access         52


#define Exit_Now            100


#endif
/* __COMPACTFLASH_H__ */
```

A.4 CompactFlashIDE.h

```
#ifndef __COMPACTFLASHIDE_H
#define __COMPACTFLASHIDE_H

// _CS1 _CS0  0  A2 A1 A0  // table 35 in CF Spe 2.0
#define CF_DATA        0x20
#define CF_ERROR       0x21
```

```
#define CF_SECT_CNT    0x22

#define CF_SECT_NUM    0x23

#define CF_CYL_L       0x24

#define CF_CYL_H       0x25

#define CF_DRV_HEAD    0x26

#define CF_STATUS      0x27


#define CF_FEATURES    0x21

#define CF_COMMAND     0x27


#define CF_ALTSTATUS   0x16

#define CF_DEV_CTR     0x16


#define LED_RF      0x10// LED in RF board



void CompactFlashExpBusInit(void);

void setExpBusCS1To16BitDataBus(void);

void setExpBusCS1To8BitDataBus(void);


void CompactFlashExpBusWriteW(unsigned long ixp_exp_bus_cs, unsigned short reg,
unsigned short data);

void CompactFlashExpBusWriteB(unsigned long ixp_exp_bus_cs, unsigned short reg,
unsigned short data);

unsigned short CompactFlashExpBusReadW(unsigned long ixp_exp_bus_cs, unsigned
short reg);

unsigned char CompactFlashExpBusReadB(unsigned long ixp_exp_bus_cs, unsigned short
reg);


void WriteRegW(unsigned short addr, unsigned short data);

void WriteRegB(unsigned short addr, unsigned char  data);

unsigned short ReadRegW(unsigned short addr);

unsigned char  ReadRegB(unsigned short addr);


unsigned short getLEW(unsigned char *addr);

unsigned char Check_RDY(void);
```

```
void Waiting_RDY(void);

unsigned char Waiting_RDY_TO(void);

void Waiting_RDY_Alt(void);

void Waiting_DRQ_Alt(void);


void ReadAllCF_regs();

void ReadOneCF_reg(unsigned char reg);

void WriteOneCF_reg(unsigned short addr, unsigned char  data);


void ReadIdenfyInformation();

void ReadIdenfyInformationW(unsigned short *buff);

void ReadIdenfyInformationB(unsigned char *buff);


void IDE_Init(unsigned char drive);


void ReadSectorW(unsigned short *buff, unsigned long LBALocation);

void ReadSectorB(unsigned char *buff, unsigned long LBALocation);


void ReadSectorMod(unsigned char *buff, unsigned long LBALocation);

void WriteSectorW(unsigned short *buff, unsigned long LBALocation);

void WriteSectorB(unsigned char *buff, unsigned long LBALocation);


#endif
```

A.5 component.mk

```
ifeq ($(IX_TARGET_OS),linux)

codelets_cfEng_OBJ := CompactFlashIDE.o \

            CompactFlashFat16.o \

            CompactFlashModuleSymbols.o


codelets_cfEng_CFLAGS := -DOS_EMBLINUX

else
```

```
codelets_cfEng_OBJ := CompactFlashIDE.o \
            CompactFlashFat16.o \
            CompactFlashModuleSymbols.o

codelets_cfEng_CFLAGS := -DOS_VXWORK
endif

codelets_mabbDemo_CFLAGS :=

codelets_cfEng_test_DEPS := ethAcc ethDB hssAcc npeDl npeMh qmgr osServices ossl
```

A.6 CompactFlashFat16.c

```
#include <stdio.h>
#include <string.h>

#include <linux/module.h>

#include <asm/uaccess.h>
#include <asm/arch/irqs.h>
#include <asm/io.h>

#include <linux/types.h>
#include <asm/hardware.h>
#include <linux/delay.h>
#include "ixp425.h"

#include "CompactFlash.h"

#include "CompactFlashIDE.h"
#include "CompactFlashFat16.h"

extern unsigned char   Heads;  // number of heads as read from CF
```

```
extern unsigned short  SecTrack;   // sectors petr track as read from CF
extern unsigned long   LastSect;

#define _WORD               2
#define _BYTE               1

/*-----------------------------------*/
unsigned long     BPB_LBA;
unsigned char     Sector_Buff[512];

unsigned short    BPB_RootEntCnt;
unsigned short    BPB_BytsPerSec;
unsigned short    BPB_FATSz16;
unsigned long     BPB_FATSz32;
unsigned char     BPB_NumFATs;
unsigned char     BPB_SecPerClus;
unsigned short    BPB_RsvdSecCnt;
unsigned short    BPB_TotSec16;
unsigned short    BPB_TotSec32;


unsigned long   RootDirSectors;
unsigned long   FATtype;
unsigned long   FATSz;
unsigned long   TotSec;
unsigned long   TotSec;
unsigned long   DataSec;
unsigned long   CountofClusters;
unsigned long   FirstDataSecNum;
unsigned long   FirstRootDirSecNum;
unsigned long   FirstRootDirClus;
unsigned long   FirstDataClus;
unsigned long   FATstart;
```

```
unsigned long   currentDirfstclus;

char fileName[262];

unsigned short longNameFlag;



unsigned short Get8Bits(unsigned short address)

{

            return(unsigned short) (  Sector_Buff[address] );

}

unsigned short Get16Bits(unsigned short address)

{

            return(unsigned short) (  (Sector_Buff[address +1]<<8) |
Sector_Buff[address] );

}

unsigned long Get32Bits(unsigned short address)

{

        return(unsigned long) (  (Sector_Buff[address +3]<<24)|(Sector_Buff[address
+2]<<16)|(Sector_Buff[address +1]<<8) | Sector_Buff[address] );

}




void DisplayBufChar(int N)

{

    int i;


    for(i=0; i<N; i++)

    {

            printk("  "     "%c",Get8Bits(i));

    }

}



//Extract Bios partition block info


void ProcessBPB(void)
```

```
{
    printk("\nCompactFlash_module :: "     "processing BPB ... \n");


    printk("\nCompactFlash_module :: "     "checking FAT type ... \n");
    BPB_RootEntCnt=Get16Bits(17);
    BPB_BytsPerSec=Get16Bits(11);
    BPB_FATSz16=Get16Bits(22);
    BPB_FATSz32=Get32Bits(36);
    BPB_NumFATs=Get8Bits(16);
    BPB_RsvdSecCnt=Get16Bits(14);
    BPB_TotSec16=Get16Bits(19);
    BPB_TotSec32=Get16Bits(32);
    BPB_SecPerClus=Get8Bits(13);


    RootDirSectors=( (BPB_RootEntCnt*32)+ ( BPB_BytsPerSec-1) )/BPB_BytsPerSec;


    if(BPB_FATSz16 !=0)
    FATSz = BPB_FATSz16;
    else
    FATSz = BPB_FATSz32;


    if(BPB_TotSec16 !=0)
    TotSec = BPB_TotSec16;
    else
    TotSec = BPB_TotSec32;


    DataSec =TotSec - (BPB_RsvdSecCnt + (BPB_NumFATs*FATSz) + RootDirSectors);
    CountofClusters = DataSec / BPB_SecPerClus;


    printk("\nCompactFlash_module :: "     "CountofClusters=%d ... \n",
CountofClusters);
    printk("\nCompactFlash_module :: "     "RootDirSectors=%d ... \n",
RootDirSectors);
    printk("\nCompactFlash_module :: "     "BPB_RsvdSecCnt=%d ... \n",
BPB_RsvdSecCnt);
```

```
    printk("\nCompactFlash_module :: "     "BPB_RootEntCnt=%d ... \n",
BPB_RootEntCnt);

    printk("\nCompactFlash_module :: "     "BPB_BytsPerSec=%d ... \n",
BPB_BytsPerSec);

    printk("\nCompactFlash_module :: "     "BPB_FATSz16=%d ... \n", BPB_FATSz16);

    printk("\nCompactFlash_module :: "     "BPB_FATSz32=%d ... \n", BPB_FATSz32);

    printk("\nCompactFlash_module :: "     "BPB_NumFATs=%d ... \n", BPB_NumFATs);

    printk("\nCompactFlash_module :: "     "BPB_TotSec16=%d ... \n", BPB_TotSec16);

    printk("\nCompactFlash_module :: "     "BPB_TotSec32=%d ... \n", BPB_TotSec32);

    printk("\nCompactFlash_module :: "     "BPB_SecPerClus=%d ... \n",
BPB_SecPerClus);


    printk("\nCompactFlash_module :: "     "FATSz=%d ... \n", FATSz);

    printk("\nCompactFlash_module :: "     "TotSec=%d ... \n", TotSec);



    if(CountofClusters < 4085)

    {

    FATtype=12;

    printk("CompactFlash_module :: "     "it is FAT12\n");

    }

    else if(CountofClusters < 65525)

    {

    FATtype=16;

    printk("CompactFlash_module :: "     "it is FAT16\n");

    }

    else

    {

    FATtype=32;

        printk("CompactFlash_module :: "     "it is FAT32\n");

    }


    FirstDataSecNum = BPB_RsvdSecCnt + (BPB_NumFATs*FATSz) + RootDirSectors;

    FirstRootDirSecNum= BPB_RsvdSecCnt + (BPB_NumFATs*FATSz);

    FATstart =  BPB_LBA+BPB_RsvdSecCnt ;
```

```
    printk("\nCompactFlash_module :: "     "FirstDataSec=%d ... \n",
FirstDataSecNum);

    printk("\nCompactFlash_module :: "     "FirstRootDirSecNum=%d ... \n",
FirstRootDirSecNum);

    printk("\nCompactFlash_module :: "     "FATstart=%d ... \n", FATstart);


    if(FATtype==32)

    {

        FirstRootDirClus = Get32Bits(44);

    FirstDataClus=FATstart +(BPB_NumFATs*FATSz);

    }

}



unsigned long showFileAttr(unsigned long ent)

{

        unsigned long tmp, dirFlag;


    tmp=Get16Bits(32*ent+24); /* date */

    printk(""  "%02d/%02d/%4d  ",(tmp>>5)&0x0f,tmp&0x1f, ((tmp>>9)&0x7f)+1980);


    tmp=Get16Bits(32*ent+22); /* time */

    if(((tmp>>11)&0x1f)>12)

        printk(""  "%02d:%02d:%02d  PM",((tmp>>11)&0x1f)-12,(tmp>>5)&0x3f,
(tmp&0x1f)*2);

    else

        printk(""  "%02d:%02d:%02d  AM",(tmp>>11)&0x1f,(tmp>>5)&0x3f,
(tmp&0x1f)*2);



    dirFlag=isItDir(ent);

    if(dirFlag)

    {

        printk(""  "    <DIR>          ");// it is a  directory

    }
```

```
    else
    {
        tmp=Get32Bits(32*ent+28);
        printk(""  "%18d ",tmp);   // file size
    }

    return dirFlag;
}

void strcpyW(char *dst, char *src)
{
    short *dstPtr, *srcPtr;
    int i;

    dstPtr=(short *)dst;
    srcPtr=(short *)src;

    i=0;
    while(srcPtr[i]!=0)
        dstPtr[i]=srcPtr[i++];
    dstPtr[i]=0;// the null
}

int strcmpW(char *dst, char *src)
{
    short *dstPtr, *srcPtr;
    int i, j;

    dstPtr=(short *)dst;
    srcPtr=(short *)src;

    i=0;
    j=0;
    while(srcPtr[i]!=0)
```

```
    {
        if(dstPtr[i]!=srcPtr[i++])
        {
            j=1;
            break;
        }
    }
    return j;
}

void printfW(char *src)
{
    short *srcPtr;
    int i;

    srcPtr=(short *)src;
    i=0;
    while(srcPtr[i]!=0)
    {
        printk(""  "%c",src[2*i]);
        printk(""  "%c",src[2*i+1]);
        i++;
    }
}

void getLongFileName(unsigned long ent)
{
    char tmp[262];

    memcpy(tmp, Sector_Buff+32*ent+1,10);
    memcpy(tmp+10, Sector_Buff+32*ent+14,12);
    memcpy(tmp+22, Sector_Buff+32*ent+28,4);

    strcpyW(tmp+26, fileName);
```

```
    strcpyW(fileName, tmp);
}

void getShortFileName(unsigned long ent)
{
        unsigned long tmp, i, j, dirFlag;

    dirFlag=isItDir(ent);

    for(i=0, j=0; i<11; i++)
    {
        tmp=Get8Bits(32*ent+i);
        if(tmp!=0x20)
            fileName[j++]=tmp;
        if((i==7)&&(dirFlag==0))
            fileName[j++]='.';
    }
    fileName[j]=0;
}




unsigned long getFileFstClus(unsigned long ent)
{
    unsigned long tmp;
    tmp=Get16Bits(32*ent+20);
    tmp=(tmp<<16)|(Get16Bits(32*ent+26));
    return tmp;
}

unsigned long getClusFstSet(unsigned long clus)
{
    unsigned long tmp;
```

```
    tmp=(clus-2)*BPB_SecPerClus + FirstDataSecNum;
    return tmp;
}



unsigned long getNextFATentry(unsigned long clus)
{
    unsigned long FAToffset, thisFATsecNum, thisFATentOffset, nextClus;

    if(FATtype==16)
        FAToffset= clus*2;
    else if (FATtype==32)
        FAToffset= clus*4;

    thisFATsecNum=BPB_RsvdSecCnt+FAToffset/BPB_BytsPerSec;
    thisFATentOffset=FAToffset%BPB_BytsPerSec;

    ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA+thisFATsecNum);



    if(FATtype==16)
        nextClus=Get16Bits(thisFATentOffset);
    else if (FATtype==32)
        nextClus=Get32Bits(thisFATentOffset)&0x0fffffff;

    return nextClus;
}
unsigned long isEndOfclusterChain(unsigned long clus)
{
    if(FATtype==16)
    {
        if(clus>=0x0fff8)
            return (1);
    }
```

```
    else if (FATtype==32)
    {
        if(clus>=0x0ffffff8)
            return (1);
    }
    return (0);
}


unsigned long getFileSize(unsigned long ent)
{
    unsigned long tmp;
    tmp=Get32Bits(32*ent+28);
    return tmp;
}

unsigned long isItDir(unsigned long ent)
{
    unsigned long dirFlag;
    dirFlag=Get8Bits(32*ent+11)&0x10;
    return dirFlag;
}

void readFile(unsigned long ent)
{
    unsigned long fileSize, clus, firstSet, sec, bytCnt;

    if((int)ent<0)
    {
        printk(""  "can not find the file\n");
        return;
    }

    clus=getFileFstClus(ent);
```

```
    if(clus==0)
    {
        printk(""  "the file is empty\n");
        return;
    }

    fileSize=getFileSize(ent);
    bytCnt=0;
    do
    {
        firstSet=getClusFstSet(clus);
        for(sec=0; sec<BPB_SecPerClus; sec++)
        {
            ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA+firstSet+sec);
            bytCnt+=512;
            if(bytCnt<=fileSize)
                DisplayBufChar(512);
            else
                DisplayBufChar(512-(bytCnt-fileSize));

            if(bytCnt>=fileSize)
                break;
        }
        if(bytCnt>=fileSize)
            break;
        clus=getNextFATentry(clus);
    }while(!isEndOfclusterChain(clus));

}


void changeToDirectory(unsigned long ent, unsigned long changeFlag)
{
    unsigned long clus, firstSet, sec;
```

```
    short *fileNamePtr;


    fileNamePtr=(short *)fileName;
    longNameFlag=0;
    fileNamePtr[0]=0;


    if((int) ent<0)
    {
        printf("can not found the dir\n");
        return;
    }


    clus=getFileFstClus(ent);
    if(changeFlag)
        currentDirfstclus=clus;


    if(clus==0)
    {
        ChangeToRootDirectoryFAT16(changeFlag);
        return;
    }


    do
    {
        firstSet=getClusFstSet(clus);
        for(sec=0; sec<BPB_SecPerClus; sec++)
        {
            ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA+firstSet+sec);
            listFileEntryInOneSector(fileNamePtr);
        }
        clus=getNextFATentry(clus);
    }while(!isEndOfclusterChain(clus));
}
```

```
void listCurrentDirectory()
{
    unsigned long clus, firstSet, sec;
    short *fileNamePtr;

    fileNamePtr=(short *)fileName;
    longNameFlag=0;
    fileNamePtr[0]=0;

        clus=currentDirfstclus;
    if(clus==0)
    {
        // the parent is the root dir
        ChangeToRootDirectoryFAT16(0);
        return;
    }

    do
    {
        firstSet=getClusFstSet(clus);
        for(sec=0; sec<BPB_SecPerClus; sec++)
        {
            ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA+firstSet+sec);
            listFileEntryInOneSector(fileNamePtr);
        }
        clus=getNextFATentry(clus);
    }while(!isEndOfclusterChain(clus));
}

unsigned long searchFileEntry(short *fileNamePtr, unsigned long *index, unsigned
long fileIndex)
```

```
{
        unsigned long ent, tmp;
    for(ent=0; ent<16; ent++)
    {
        tmp=Get8Bits(32*ent+0);
        if(tmp==0x00)
            break;  // no more entry
        else if(tmp==0x05)
            continue;  // this entry is free for Japanese
        else if(tmp==0xe5)
            continue;  // this entry is free
        else
        {
            if((Get8Bits(32*ent+11)&0x0f)==0x0f) // one entry in a long name set
            {
                if(((*index)+1)==fileIndex)
                {   longNameFlag=1;
                    getLongFileName(ent);
                }
            }
            else
            {
                (*index)++;
                if((*index)==fileIndex)
                {
                    if (longNameFlag==1)
                    {
                        // this is a long name
                        showFileAttr(ent);
                        printfW(fileName);
                        printk(""  "\n");
                    }
                    else
                    {
```

```
                                // this is a short name
                                showFileAttr(ent);
                                getShortFileName(ent);
                                printk("" "%s\n", fileName);
                            }
                            break;
                        }
                    }
                }
            }
            return ent;
}


int searchFileEntryByName(short *fileNamePtr, char *str)
{
            unsigned long ent, tmp, flag;

            flag=0;
            for(ent=0; ent<16; ent++)
            {
                tmp=Get8Bits(32*ent+0);
                if(tmp==0x00)
                    break;  // no more entry
                else if(tmp==0x05)
                    continue;  // this entry is free for Japanese
                else if(tmp==0xe5)
                    continue;  // this entry is free
                else
                {
                    if((Get8Bits(32*ent+11)&0x0f)==0x0f) // one entry in a long name set
                    {
                        longNameFlag=1;
                        getLongFileName(ent);
```

```
                }
                else
                {
                        if (longNameFlag==1)
                        {
                            // this is a long name


                            if(strcmpW(str, fileName)==0)
                            {
                                flag=1;
                                showFileAttr(ent);
                                printfW(fileName);
                                printk(""  "\n");
                                break;
                            }
                            longNameFlag=0;
                            fileNamePtr[0]=0;
                        }
                        else
                        {
                            // this is a short name
                            getShortFileName(ent);
                            if(strcmp(str, fileName)==0)
                            {
                                flag=1;
                                showFileAttr(ent);
                                printk(""  "%s\n", fileName);
                                break;
                            }
                        }
                }
            }
        }
```

```
        if(flag==1)
            return ent;
        else
            return (-1);
}



unsigned long getRootFileEntFAT16(unsigned long fileIndex)  // return the entry in
the directory
{
    unsigned long sec, ent, index;
    short *fileNamePtr;


printk("\nCompactFlash_module ::" "file information for fileIndex= %d\n",
fileIndex);


    index=-1;
    fileNamePtr=(short *)fileName;
    longNameFlag=0;
    fileNamePtr[0]=0;
    for(sec=0; sec<RootDirSectors; sec++)
    {
        ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA+FirstRootDirSecNum+sec);
        ent=searchFileEntry(fileNamePtr, &index, fileIndex);
        if(index==fileIndex)
            break;
    }
    return (ent);
}


unsigned long getRootFileEntByFileNameFAT16(char *str)  // return the entry in the
directory
{
    unsigned long sec, ent;
```

```
    short *fileNamePtr;


    ent=-1;
    fileNamePtr=(short *)fileName;
    longNameFlag=0;
    fileNamePtr[0]=0;
    for(sec=0; sec<RootDirSectors; sec++)
    {
        ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA+FirstRootDirSecNum+sec);
        ent=searchFileEntryByName(fileNamePtr, str);
        if((int)ent>=0)
            break;
    }
    return (ent);
}



unsigned long getFileEntFAT16(unsigned long fileIndex)
{
    unsigned long clus, firstSet, sec, ent, index;
    short *fileNamePtr;


    printk("\nCompactFlash_module :: "     "file information for fileIndex= %d\n",
fileIndex);


    index=-1;
    ent=-1;


    fileNamePtr=(short *)fileName;
    longNameFlag=0;
    fileNamePtr[0]=0;


   clus=currentDirfstclus;
    if(clus==0)
```

```
    {
        ent=getRootFileEntFAT16(fileIndex);
        return (ent);
    }

    do
    {
        firstSet=getClusFstSet(clus);
        for(sec=0; sec<BPB_SecPerClus; sec++)
        {
            ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA+firstSet+sec);
            ent=searchFileEntry(fileNamePtr, &index, fileIndex);
            if(index==fileIndex)
                break;
        }
        if(index==fileIndex)
            break;
        clus=getNextFATentry(clus);
    }while(!isEndOfclusterChain(clus));


    return (ent);
}


unsigned long getFileEntByFileNameFAT16(char *str)
{
    unsigned long clus, firstSet, sec, ent;
    short *fileNamePtr;

    ent=-1;

    fileNamePtr=(short *)fileName;
    longNameFlag=0;
```

```
    fileNamePtr[0]=0;


        clus=currentDirfstclus;
    if(clus==0)
    {
        ent=getRootFileEntByFileNameFAT16(str);
        return (ent);
    }


    do
    {
        firstSet=getClusFstSet(clus);
        for(sec=0; sec<BPB_SecPerClus; sec++)
        {
            ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA+firstSet+sec);
            ent=searchFileEntryByName(fileNamePtr, str);
            if((int)ent>=0)
                break;
        }
        if((int) ent>=0)
            break;
        clus=getNextFATentry(clus);
    }while(!isEndOfclusterChain(clus));



    return (ent);
}



void listFileEntryInOneSector(short *fileNamePtr)
{
        unsigned long ent, tmp;


        for(ent=0; ent<16; ent++)
```

```
{
    tmp=Get8Bits(32*ent+0);
    if(tmp==0x00)
        break;  // no more entry
    else if(tmp==0x05)
        continue;  // this entry is free for Japanese
    else if(tmp==0xe5)
        continue;  // this entry is free
    else
    {
        if((Get8Bits(32*ent+11)&0x0f)==0x0f) // one entry in a long name set
        {
            longNameFlag=1;
            getLongFileName(ent);
        }
        else
        {
            if (longNameFlag==1)
            {
                // this is a long name
                showFileAttr(ent);
                printfW(fileName);
                printk(""  "\n");
                longNameFlag=0;
                fileNamePtr[0]=0;
            }
            else
            {
                // this is a short name
                showFileAttr(ent);
                getShortFileName(ent);
                printk(""  "%s\n", fileName);
                //printk(""  "sec=%d,ent=%d\n",sec,ent);
            }
```

```
                }
            }
        }

}


void ChangeToRootDirectoryFAT16(unsigned long changeFlag)
{
    unsigned long sec;
    short *fileNamePtr;

    printk("\nCompactFlash_module :: "     "List Root Directory\n");

    fileNamePtr=(short *)fileName;
    longNameFlag=0;
    fileNamePtr[0]=0;
    for(sec=0; sec<RootDirSectors; sec++)
    {
        ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA+FirstRootDirSecNum+sec);
        listFileEntryInOneSector(fileNamePtr);
    }

    if(changeFlag)
        currentDirfstclus=0;
}


void changeToRootDirectory(unsigned long changeFlag)
{
    if(FATtype==16)
        ChangeToRootDirectoryFAT16(changeFlag);
}
```

```
unsigned long getRootFileEnt(unsigned long fileIndex)

{

    if(FATtype==16)

        return (getRootFileEntFAT16(fileIndex));

}


unsigned long getFileEnt(unsigned long fileIndex)

{

    if(FATtype==16)

        return (getFileEntFAT16(fileIndex));

}


unsigned long getFileEntByName(char *str)

{

    if(FATtype==16)

        return (getFileEntByFileNameFAT16(str));

}


void checkMBR()

{

    unsigned long tmp;

    printk("\nCompactFlash_module :: "     "checking if there is a master boot
record ... \n");


    ReadSectorW((unsigned short *)Sector_Buff, 0);

    if(Get8Bits(0)==0xeb)

    {

    printf("no MBR\n");

    BPB_LBA=0;

    }

    else if(Get8Bits(0)==0xe9)

    {

    printf("no MBR\n");

    BPB_LBA=0;
```

```
    }
    else
    {
    printf("processing MBR\n");
    tmp=Get8Bits(0x1c2);
        if(tmp==0x01)
    {
        FATtype=12;
        printk("CompactFlash_module :: "     "it is FAT12\n");
    }
    else if(tmp == 0x04)
    {
        FATtype=16;
        printk("CompactFlash_module :: "     "it is FAT16 (smaller than 32MB)\n");
    }
    else if( (tmp == 0x06)  || (tmp == 0x0e))
    {
        FATtype=16;
        printk("CompactFlash_module :: "     "it is FAT16 (larger than 32MB)\n");
    }
    else if( (tmp == 0x0b) || (tmp == 0x0c))
    {
        FATtype=32;
        printk("CompactFlash_module :: "     "it is FAT32\n");
    }

    BPB_LBA=Get32Bits(0x1C6);
    Waiting_RDY();
    ReadSectorW((unsigned short *)Sector_Buff, BPB_LBA);     // read bootsector of
first partition
    }
    printk("CompactFlash_module :: "     "BPB_LBA=%d\n",BPB_LBA);
}
```

```
void entryInfo(unsigned long ent)

{

    printk("CompactFlash_module :: "     "Clus=%d\n",getFileFstClus(ent));

    printk("CompactFlash_module :: "
"SecNum=%d\n",getClusFstSet(getFileFstClus(ent)));

    printk("CompactFlash_module :: "     "FileSize=%d\n",getFileSize(ent));

}


void Fat16Init(void)

{

    unsigned long ent, clus, set;


    checkMBR();

    ProcessBPB();// extract info from Bios Parameter Block

    changeToRootDirectory(0);


    // testing

    getRootFileEnt(1);

    getRootFileEnt(2);

    getRootFileEnt(4);

    getRootFileEnt(5);


    ent=getRootFileEnt(2);

    clus=getFileFstClus(ent);

    set=getClusFstSet(clus);

    printk("CompactFlash_module :: "     "file ent=%d, 1st clus=%d, 1st data
set=%d\n", ent,clus,set);

    while(!isEndOfclusterChain(clus))

    {

    clus=getNextFATentry(clus);

    printk("CompactFlash_module :: "     "next clus=%d\n",clus);

    }


    ent=getRootFileEnt(0);

    if(isItDir(ent))
```

```
    changeToDirectory(ent,0);
    else
    readFile(ent);



    listCurrentDirectory();
    ent=getFileEnt(1);
    if(isItDir(ent))
    changeToDirectory(ent,0);
    else
    readFile(ent);

}



void changeToUpperDir(unsigned long changeFlag)
{
    unsigned long ent;

    if(currentDirfstclus==0)
        listCurrentDirectory();
    else
    {
        listCurrentDirectory();
        ent=getFileEnt(1);
        if(isItDir(ent))
        changeToDirectory(ent, changeFlag);
    }
}

void changeToDir(char *str, unsigned long changeFlag)
{
    unsigned long ent;
```

```
        listCurrentDirectory();


    ent=getFileEntByName(str);
    if((int)ent<0)
    {
        printf("can not found the dir\n");
        return;
    }

    if(isItDir(ent))
        changeToDirectory(ent, changeFlag);
}

void readFileByName(char *str)
{
    unsigned long ent;
    ent=getFileEntByName(str);
    if((int)ent<0)
    {
        printf("can not found the file\n");
        return;
    }
    if(!isItDir(ent))
        readFile(ent);
}
```

A.7 CompactFlashFat16.h

```
#ifndef __COMPACTFLASHFAT16_H
#define __COMPACTFlashFAT16_H

void strcpyW(char *dst, char *src);
void printfW(char *src);
```

```
unsigned long isItDir(unsigned long ent);
unsigned long getFileEnt(unsigned long fileIndex);
unsigned long getFileSize(unsigned long ent);
unsigned long isEndOfclusterChain(unsigned long clus);
unsigned long  getNextFATentry(unsigned long clus);
unsigned long getFileFstClus(unsigned long ent);
unsigned long getClusFstSet(unsigned long clus);

void ListRootDirectory();
void listFileEntryInOneSector(short *fileNamePtr);
void ChangeToRootDirectoryFAT16(unsigned long changeFlag);

#endif
```

A.8 CompactFlashApp.c

```
#include <pthread.h>

#include <string.h>
#include <unistd.h>
#include <sched.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/uio.h>
#include <sys/sendfile.h>
#include <sys/stat.h>

#include <stdio.h>



#include "CompactFlash.h"
#include "CompactFlashIDE.h"
```

```
#define IX_CF_CODELET_STRLEN 50


void testDriver(void);

void TestMenu(void);

void TestFileSystemMenu(void);

int CompactFlashGetNum(char *str);



static int inMsgQue, CFdriver;


int main(void)

{


    testDriver();

    exit(0);

}



// little endian to big endian

unsigned short byteSwap(unsigned char *addr)

{

    //convert from little to big endian

    unsigned short tmp;

    tmp=(addr[1]<<8)|addr[0];

    return tmp;

}




void testDriver(void)

{

    int rc;

    int passedArg;

    int CntCd;
```

```
    int i, N;
    unsigned char data[512];

    /* open the driver */
    CFdriver = open("/dev/CompactFlashModule",O_RDWR);

    if(!CFdriver )
    {
        printf("Open failed. Ensure module CompactFlashModule is inserted "
               "and /dev/CompactFlashModule exists\nIf necessary, create "
               "with: mknod /dev/CompactFlashModule c 253 0\n");
        exit(0);
    }

    // initialize IDE
    rc = ioctl(CFdriver,IX_CF_CODELET_INIT_IDE, &passedArg);
    if (rc != 0)
    {
        printf("CompactFlash ioctl rc is failed to init IDE .. %d \n", rc);
        exit(1);
    }

    //TestMenu();
    TestFileSystemMenu();
}

int CompactFlashGetNum(char *str)
{
    int c;
    int i = 0;
    char input[IX_CF_CODELET_STRLEN];
```

```
    if(str && *str) printf("%s", str);
    do
    {
        c = getc(stdin);

        if (c == 0x08)
        {
            if(i) i--;
        }
        else
        {
            input[i++] = c;
        }

        /* exception if x entered, jump to main menu */
    } while(i<IX_CF_CODELET_STRLEN && c!='\r' && c!='\n');

    input[i] = '\0';

    return atoi(input);
}


void TestMenu(void)
{
    int selectedItem=0;
    int passedArg, val, i, rc;
    int cureentIndex, lastIndex;
    unsigned char data[512];
    unsigned short *wPtr;

    wPtr=(unsigned short *)data;

    do
```

```
{
    /* print the test menu */
        printf(     "\n----------------------------------------\n"
                    "- IxCompactFlashCodelet Demo Menu      -\n"
                    "----------------------------------------\n");


                printf("\nRead/Write to CF Registers:\n");
            printf("%d: check if the flash card is ready\n", Check_Card);
            printf("%d: read all the CF registers\n", Read_All_Regs);
    printf("%d: read one CF register\n", Read_One_Reg);
            printf("%d: write to one CF register\n", Write_One_Reg);
            printf("%d: show Exp Bus Regs\n", Show_Exp_Regs);

            printf("\nView CF Identify Information:\n");
    printf("%d: read the identify sector\n", Read_Identify_Sector);

                printf("\nRead/Write to sectors:\n");
            printf("%d: read from one sector\n", Read_From_One_Sector);
            printf("%d: write to one sector\n", Write_To_One_Sector);


                printf("\nDisplay data:\n");
            printf("%d: show one byte in a sector\n", Show_One_Byte);
            printf("%d: show next 10 bytes\n", Show_Next_10_Bytes);

            printf("%d: show one word in a sector\n", Show_One_Word);
            printf("%d: show next 10 words\n", Show_Next_10_Words);

                printf("\nDisplay MBR & BPB:\n");
            printf("%d: find MBR and BPB\n", FindMBRandBPB);
            printf("%d: Process BPB data\n", ProcessBPBdata);

                printf("%d: Exit\n", Exit_Now);
```

```
    /* select a menu item */
    selectedItem = CompactFlashGetNum(NULL);

printf("*****************************************************\n\n");
switch(selectedItem)
{
    case Check_Card:
        ioctl(CFdriver,Check_Card, &passedArg);
        if(passedArg)
            printf("Flash ccard is ready\n");
        else
            printf("Flash ccard is not ready\n");

        break;

    case Read_All_Regs:
        printf("\n");
        rc = ioctl(CFdriver,Read_All_Regs, passedArg);
        break;

    case Read_One_Reg:
        printf("CF_ERROR     %d\n",CF_ERROR);
        printf("CF_SECT_CNT  %d\n",CF_SECT_CNT);
        printf("CF_SECT_NUM  %d\n",CF_SECT_NUM);
        printf("CF_CYL_L     %d\n",CF_CYL_L );
        printf("CF_CYL_H     %d\n",CF_CYL_H);
        printf("CF_DRV_HEAD  %d\n",CF_DRV_HEAD);
        printf("CF_STATUS    %d\n",CF_STATUS);
        printf("CF_ALTSTATUS %d\n",CF_ALTSTATUS);

        passedArg=CompactFlashGetNum("choose a reg to read: ");
        rc = ioctl(CFdriver,Read_One_Reg, passedArg);
        break;
```

```
            case Write_One_Reg:
                printf("CF_COMMAND   %d\n",CF_COMMAND);
                printf("CF_DEV_CTR   %d\n",CF_DEV_CTR);
                printf("CF_SECT_CNT  %d\n",CF_SECT_CNT);
                printf("CF_SECT_NUM  %d\n",CF_SECT_NUM);
                printf("CF_CYL_L     %d\n",CF_CYL_L );
                printf("CF_CYL_H     %d\n",CF_CYL_H);
                printf("CF_DRV_HEAD  %d\n",CF_DRV_HEAD);
                printf("CF_FEATURES  %d\n",CF_FEATURES);

                passedArg=CompactFlashGetNum("choose a reg to write: ");
                passedArg=(passedArg<<8)|CompactFlashGetNum("input a value: ");
                rc = ioctl(CFdriver,Write_One_Reg, passedArg);
                break;

            case Read_Identify_Sector:
                printf("reading words from identify sector\n");
                ioctl(CFdriver,Read_Identify_Sector, passedArg);
                rc = read(CFdriver,data, passedArg);
                if (rc == -1)
                {
                    printf("CompactFlash read failed .. %d \n", rc);
                    exit(1);
                }
                printf("done with reading identify sector\n");
                printf("bytes:\n");
                lastIndex=0;
                for(i=0; i<10; i++)
                   printf("data[%d]=0x%x (%d)\n",lastIndex+i, data[lastIndex+i],
data[lastIndex+i]);
                lastIndex=lastIndex+10;
                break;
```

```
        case Read_From_One_Sector:
            passedArg=CompactFlashGetNum("enter sector number to read: ");
            printf("reading words from flash\n");
            ioctl(CFdriver,Word_Access, passedArg);
            rc = read(CFdriver,data, passedArg);
            if (rc == -1)
            {
                printf("CompactFlash read failed .. %d \n", rc);
                exit(1);
            }
            printf("done with reading from flash\n");
            printf("bytes:\n");
            lastIndex=0;
            for(i=0; i<10; i++)
                printf("data[%d]=0x%x (%d)\n",lastIndex+i, data[lastIndex+i],
data[lastIndex+i]);
            lastIndex=lastIndex+10;
            break;


        case Write_To_One_Sector:
            passedArg=CompactFlashGetNum("enter sector number to write (prefer
1000): ");
            if(passedArg==0)
            {
                printf("you are not allowed to write to master sector\n");
                break;
            }



            printf("1:  write one value to each byte in the sector\n");
            printf("2:  write 0, 1, 2, ....to the sector\n");
            printf("3:  input values byte by byte for the sector\n");
            rc=CompactFlashGetNum(" ");
            switch(rc)
            {
```

```
                    case 1:
                    val=CompactFlashGetNum("enter a byte to write to the whole sector:
");
                        for(i=0; i<512; i++)
                            data[i]=val;
                        break;
                    case 2:
                        for(i=0; i<512; i++)
                            data[i]=i&0x0ff;
                        break;
                    case 3:
                        for(i=0; i<512; i++)
                        {
                            printf("byte[%d]=\n",i);
                            val=CompactFlashGetNum("   (555 to exit) ");
                            if(val==555)
                                break;
                            else
                                data[i]=val;
                        }
                        break;
                    default:
                        break;
                }
                printf("writing words to flash\n");
                ioctl(CFdriver,Word_Access, passedArg);
                rc = write(CFdriver,data, passedArg);
                if (rc == -1)
                {
                    printf("CompactFlash read failed .. %d \n", rc);
                    exit(1);
                }
                printf("done with writing to the flash\n");
                break;
```

```
        case Show_One_Byte:

            cureentIndex=CompactFlashGetNum("byte index (0...511): ");

            printf("bytes:\n");

            printf("data[%d]=0x%x (%d)\n",cureentIndex, data[cureentIndex],
data[cureentIndex]);

            lastIndex=cureentIndex;

            break;

        case Show_Next_10_Bytes:

            printf("bytes:\n");

            for(i=0; i<10; i++)

                printf("data[%d]=0x%x (%d)\n",lastIndex+i, data[lastIndex+i],
data[lastIndex+i]);

            lastIndex=lastIndex+10;

            break;


        case Show_One_Word:

            cureentIndex=CompactFlashGetNum("word index (0...255): ");

            printf("words:\n");

            printf("data[%d]=0x%x (%d)\n",cureentIndex,
byteSwap(data+cureentIndex*2), byteSwap(data+cureentIndex*2));

            lastIndex=cureentIndex;

            break;


        case Show_Next_10_Words:

            printf("words:\n");

            for(i=0; i<10; i++)

                printf("data[%d]=0x%x (%d)\n",lastIndex+i,
byteSwap(data+(lastIndex+i)*2), byteSwap(data+(lastIndex+i)*2));

            lastIndex=lastIndex+10;

            break;


        case Show_Exp_Regs:

            for (i=0; i<12; i++)

            {

                passedArg=IXP425_EXP_CS0_OFFSET+i*4;
```

```
            printf("reg offset: 0x%x:   ",passedArg);
        rc = ioctl(CFdriver,Show_Exp_Regs, &passedArg);
            if (rc != 0)
            {
            printf("CompactFlash ioctl Show_Exp_Regs is failed .. %d \n", rc);
            exit(1);
            }
            printf("reg value:=0x%x\n",passedArg);
        }
    break;



    case FindMBRandBPB:
        passedArg=0;
        rc = ioctl(CFdriver,FindMBRandBPB, passedArg);
        if (rc != 0)
        {
            printf("CompactFlash ioctl FindMBRandBPB is failed .. %d \n", rc);
        exit(1);
        }
    break;
    case ProcessBPBdata:
        passedArg=0;
        rc = ioctl(CFdriver,ProcessBPBdata, passedArg);
        if (rc != 0)
        {
            printf("CompactFlash ioctl ProcessBPBdata is failed .. %d \n", rc);
        exit(1);
        }
    break;


    case Exit_Now:selectedItem=-1;
    break;
}
```

```
    } while(selectedItem != -1);

}



void CompactFlashGetString(char *inputString)
{

    int c;
    int i = 0;

    do
    {
        c = getc(stdin);

        if (c == 0x08)
        {
            if(i) i--;
        }
        else
        {
            inputString[i++] = c;
        }

        /* exception if x entered, jump to main menu */
    } while(i<IX_CF_CODELET_STRLEN && c!='\r' && c!='\n');

    inputString[i] = '\0';

    //printf("inputString=%s\n",inputString);

}
```

```
void trimSpace(char *inputString, char *dst)
{
    int i;
    char *ptr;

    ptr=inputString;

    while(*ptr==' ') ptr++;
    i=0;
    while (*ptr!=0)
    {
        dst[i++]=*ptr++;
    }
    dst[i]=0;
    while((dst[i-1]==' ')||(dst[i-1]=='\n')||(dst[i-1]=='\r'))
    {
        i--;
        dst[i]=0;
    }
}



int processCommand(char *inputString)
{
    int selectedItem=0;
    char *ptr=NULL;
    int i;

    trimSpace(inputString, inputString);

    ptr=strstr(inputString,"cd");
    if(ptr!=NULL)
    {
```

```
    if(ptr[2]==0)
    {
        selectedItem=ChangeToDir;
        inputString[0]=0;
    }
    else if(ptr[2]==' ')
    {
        trimSpace(ptr+3, inputString);
        selectedItem=ChangeToDir;
    }
    else
        selectedItem=ReadFile;

    //printf("selectedItem=%d, inputString=%s\n",selectedItem,inputString);
    return selectedItem;
}

ptr=strstr(inputString,"dir");
if(ptr!=NULL)
{
    if(ptr[3]==0)
    {
        selectedItem=ShowDir;
        inputString[0]=0;

    }
    else if(ptr[3]==' ')
    {
        trimSpace(ptr+4, inputString);
        selectedItem=ShowDir;
    }
    else
        selectedItem=ReadFile;
```

```
        //printf("selectedItem=%d, inputString=%s\n",selectedItem,inputString);
        return selectedItem;
    }

    ptr=strstr(inputString,"test!");
    if(ptr!=NULL)
    {
        selectedItem=GoTestMenu;
        //printf("selectedItem=%d, inputString=%s\n",selectedItem,inputString);
        return selectedItem;
    }

    ptr=strstr(inputString,"exit!");
    if(ptr!=NULL)
    {
        selectedItem=Exit_Now;
        //printf("selectedItem=%d, inputString=%s\n",selectedItem,inputString);
        return selectedItem;
    }

    if(inputString[0]!=0)
        selectedItem=ReadFile;

    //printf("selectedItem=%d, inputString=%s\n",selectedItem,inputString);
    return selectedItem;
}



void TestFileSystemMenu(void)
{
    unsigned char inputString[100];
    unsigned long rc;
```

```
    int selectedItem=0;



    do
    {
        /* print the test menu */
            printf(     "\n-----------------------------------------------\n"
                        "- IxCompactFlashCodelet File System Demo -\n"
                        "-----------------------------------------------\n");



            printf("\nCommands: cd/dir [/][.][..][dir name]; file or dir name;
test!, exit!\n");



            /* get a command */
            CompactFlashGetString(inputString);
            selectedItem=processCommand(inputString);



        printf("*****************************************************\n\n");
    switch(selectedItem)
    {
        case ChangeToDir:
            rc = ioctl(CFdriver,ChangeToDir, inputString);
            if (rc != 0)
            {
                printf("CompactFlash ioctl: ChangeToDir is failed .. %d \n", rc);
            exit(1);
            }


        break;


        case ShowDir:
```

```
                rc = ioctl(CFdriver,ShowDir, inputString);
                if (rc != 0)
                {
                    printf("CompactFlash ioctl: ShowDir is failed .. %d \n", rc);
                exit(1);
                }
            break;

            case ReadFile:
                rc = ioctl(CFdriver,ReadFile, inputString);

                if (rc != 0)
                {
                    printf("CompactFlash ioctl: ReadFile is failed .. %d \n", rc);
                exit(1);
                }
            break;

            case GoTestMenu:
                TestMenu();
            break;

            case Exit_Now:
                selectedItem=-1;
            break;
        }

        } while(selectedItem != -1);

}
```

A.9 Makefile

```
CC=/opt/hardhat/devkit/arm/xscale_be/bin/xscale_be-gcc
```

```
GPLUS= /opt/hardhat/devkit/arm/xscale_be/bin/xscale_be-g++

# Override standard COPTS

CFLAGS+=-mbig-endian -msoft-float -DOS_USRLINUX \

    -I$(IX_XSCALE_SW)/src/include \

    -I../ -I./



TARGET=IxCPCodeletApp


LDFLAGS = -lpthread -ldl


O_OBJS=CompactFlashApp.o


BINS= CompactFlashApp


default: CompactFlashApp


CompactFlashApp: $(O_OBJS) $(O2_OBJS)

    $(CC) $(LDFLAGS) $^ -o $@


# Override .c.o


CompactFlashApp.o:

    $(CC) $(CFLAGS) -c CompactFlashApp.c



depend:

    makedepend -I$(INC_EXPORTPATH) -I$(INC_LIBPATH) -I$(INC_QCOMMON) *.c


clean:

    rm -rf $(BINS) *.o
```