



Embedded Based Optimization of CPU for General Purpose Application System

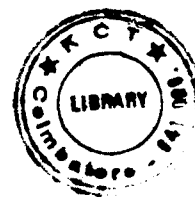


A Project Report

Submitted by

P- 2097

S.Ranjani	-	71203105039
M.Revathi	-	71203105040
T.Vimal Raj	-	71203105056
A.S.Arul Prakash	-	71203105303



*in partial fulfillment for the award of the degree
of*

**Bachelor of Engineering
in
Electrical & Electronics Engineering**

**DEPARTMENT OF ELECTRICAL & ELECTRONICS
ENGINEERING**

**KUMARAGURU COLLEGE OF TECHNOLOGY
COIMBATORE – 641 006**

ANNA UNIVERSITY: CHENNAI 600 025

APRIL– 2007

APRIL 2007

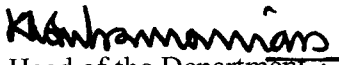
ANNA UNIVERSITY: CHENNAI 600 025

BONAFIDE CERTIFICATE

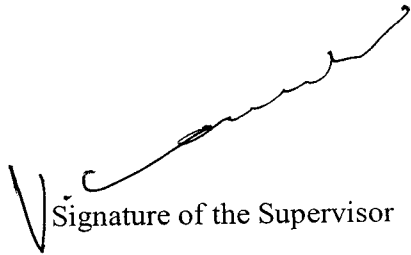
Certified that this project report entitled "Embedded Based Optimization of CPU for General Purpose Application System" is the bonafide work of

Ms.S.Ranjani	-	Register No. 71203105039
Ms.M.Revathi	-	Register No. 71203105040
Mr.T.Vimal Raj	-	Register No. 71203105056
Mr.A.S.Arul Prakash	-	Register No. 71203105303

who carried out the project work under my supervision.



Signature of the Head of the Department

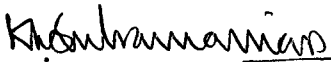


Signature of the Supervisor

PROF.K.REGUPATHY SUBRAMANIAN

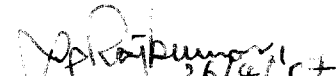
MR.V.KANDASAMY

Certified that the candidate with university register number _____ was examined in project Viva voce examination held on 26/04/07



Internal Examiner

26/4/07



26/4/07

External Examiner

DEPARTMENT OF ELECTRICAL & ELECTRONICS
ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY

COIMBATORE 641 006

, NANJUNDAPURAM ROAD, RAMANATHAPURAM, COIMBATORE - 641 045. INDIA
mail : precitron@vsnl.com • website : www.precitron.co.in

PROJECT COMPLETION CERTIFICATE

This is to certify that the following students who are pursuing final year B.E (Electrical and Electronics Engineering) at Kumaraguru College of Technology, Coimbatore have completed their project under the topic "EMBEDDED BASED OPTIMIZATION OF CPU FOR GENERAL PURPOSE APPLICATION SYSTEM".

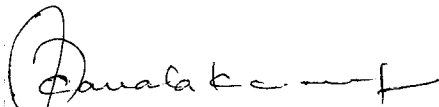
Project Duration : 14/12/2006 – 11/04/2007

Batch Members:

- | | |
|---------------------|-------------|
| 1. S.RANJANI | 71203105039 |
| 2. M.REVATHI | 71203105040 |
| 3. T.VIMALRAJ | 71203105056 |
| 4. A.S.ARUL PRAKASH | 71203105303 |

During this project period their conduct was GOOD.

For PRECITRON *Smart Systems*,



D.Kamalakannan.
Technical Lead



ABSTRACT

The present day computers take more time to boot and to shut down. The speed of operation of the computer, as a whole, is considerably reduced due to the usage of magnetic storage devices, which takes more time for the retrieval of data. Moreover the operation of these magnetic storage devices is not reliable and the size of the present day computer CPU (Central Processing Unit) is not so compact. Our idea 'Embedded Based Optimization of CPU for General Purpose Application System' aims at miniaturizing the motherboard, increasing the speed of operation of the system and makes the motherboard flexible to be used anywhere from industrial automation to server hub .In our proposal we use compact flash memory instead of space occupying magnetic devices, as a result of which the speed of process can be increased. If power goes and comes in-between, due to moving parts, hard disk may crash. But flash disk will never crash. Hence the CPU is safe. Here we use embedded LINUX as an operating system, which is an open source coding and is more secure. We use ARM (Advanced RISC Machine) processor, which requires no fan as required by Intel processors to cool it. This motherboard can be used for various purposes ranging from industrial automation to Internet servers. All the automated industrial processes require a central processing unit for its operation. Our board can be used as the CPU in these applications. After deciding the kind of application for which the board is going to be used, the application program is embedded into the flash memory of the board. Hence our proposal aims at absolute miniaturization, fast and flexible operation of the system CPU.

ACKNOWLEDGEMENT

We take this opportunity to express our gratitude to the people, who have aided us in different ways for the successful completion of our project.

We would like to thank **Prof Dr.K.Arumugam**, Correspondent, Kumaraguru College of Technology for the efforts he had taken to give the students practical experience in current technology. Then we'd like to express our gratitude to **Dr.Joseph. V. Thannikal**, our principal, for his initiative to provide us with fully equipped labs and classrooms.

We are deeply indebted to our Dean, **Prof.K.Regupathy Subramanian**, who generously offered valuable suggestions at different stages of our project. We are very much grateful to our guide, **Mr.V.Kandasamy** and all our staff members, without whose guidance the successful completion of this project would not have been possible.

We thank our sponsors **Precitron Smart Systems** for generously sponsoring the required materials and helping us to complete the project. We sincerely thank our guide in the company, **Mr.D.Kamalakaran** who has been our guiding light, spending time to clarify our doubts in spite of his busy schedule. We thank all the staff in the company for their valuable assistance.

Last but not the least, we'd like to thank our **parents and friends** who provided us with the most needed moral support and helped us to finish the project successfully.

TABLE OF CONTENTS

TITLE	PAGE NO
Bonafide Certificate	ii
Certificate of the Company	iii
Abstract	iv
Acknowledgement	v
Contents	vi
List of figures	xi
List of Photos	xi
List of Tables	xii

CHAPTER NO	TITLE	PAGE NO
1	INTRODUCTION	2
	1.1 System Need	2
	1.2 Objective	2
	1.3 Organization of Report	3
2	EP9302 DEVELOPMENT BOARD	5
	2.1 Introduction	5
	2.2 Physical Structure And Features	5
	2.3 The Schematic Of Processor & Memory	8
	2.4 The Communication Ports	9
	2.5 Memory Devices	10
	2.6 JTAG Interface	10
3	PROCESSOR_EP9302	12
	3.1 Introduction	12
	3.2 EP9302 Features	13
	3.3 EP9302 Applications	14
	3.4 Overview Of EP9302 Features	15
	3.4.1 High Performance ARM920t Processor Core	15
	3.4.2 MaverickKey Unique ID Secures Digital Content	

3.4.3	Integrated Two Port Usb2.0 Full Speed Host With Transceivers	16
3.4.5	Multiple Booting Mechanism Increase Flexibility	16
3.4.6	Abundant General Purpose I/O Build Flexible System	16
3.4.7	General Purpose Memory Interface	17
3.4.8	12Bit Analog-to-Digital Functionality	17
4	ARM920T CORE	19
4.1	Introduction	19
4.2	Overview ARM920T Processor core	19
4.2.1	Features	19
4.2.2	Operations	20
4.2.3	ARM9TDMI Core	22
4.2.3.1	Memory Management Unit	22
4.2.3.2	Address Translation	22
4.2.3.3	Access Permission & Domains	23
4.2.3.4	MMU Enable	24
4.2.4	Cache And Writer Buffer	24
4.2.4.1	Instruction Cache Enable	25
4.2.4.2	Data Cache Enable	25
4.2.4.3	Write Buffer Enable	25
4.3	Advanced High Speed Bus	25
4.3.1	AMBA AHB Bus Interface Overview	25
4.3.2	EP9301 AHB Implementation Details	27
4.3.3	Memory And Bus Access Errors	28
4.3.4	Bus Arbitration	29
4.3.4.1	Main AHB Bus Arbiter	29
4.3.4.2	EBI Bus Arbiter	30
4.4	AHB Decoder	30
4.4.1	AHB Bus Slave	31
4.4.2	AHB to APB Bridge	31
4.4.2.1	Function & Operation Of APB Bridge	32
4.4.3	APB Bus Slave	32

5	BOOT ROM	34
	5.1 Introduction	34
	5.1.1 Boot ROM Hardware Operational Overview	34
	5.1.1.1 Memory Map	34
	5.1.2 Boot Rom Software Operational Overview	35
	5.1.2.1 Flow Chart	35
	5.2 Boot Options	36
6	6.1/10/100 MBPS ETHERNET LAN CONTROLLER	39
	6.1 Introduction	39
	6.1.1 Host Interface & Description Processor	40
	6.1.2 Address Space	41
	6.1.3 Data Encapsulation	42
	6.2 Packet Transmission Process	43
	6.3 Transmit Back-Off	44
	6.3.1 Transmission	45
	6.3.2 Bit Order	46
	6.3.3 Flow Control	46
	6.3.4 Receive Flow Control	46
	6.3.5 Transmit Flow Control	47
7	UNIVERSAL SERIAL BUS HOST CONTROLLER	49
	7.1 Introduction	49
	7.1.1 Features	49
	7.2 Overview	49
	7.2.1 Data Transfer Types	50
	7.2.2 USB Host Controller	52
	7.2.3 AHB Master	52
	7.2.4 HCI Slave Block	52
	7.2.5 HCI Master Block	52
	7.2.6 USB State Controller	53
	7.2.7 Data FIFO	53
	7.2.8 List Processor	53
	7.2.9 Root Hub & Host SIE	53

8	SDRAM, SYNCROM, & SYNCFLASH CONTROLLER	55
	8.1 Introduction	55
	8.1.1 Booting	55
	8.1.2 Address Pin Usage	56
	8.1.3 Synchronous FLASH Programming	57
9	GPIO INTERFACE	59
	9.1 Introduction	59
	9.1.1 Memory Map	60
	9.1.2 Functional description	61
	9.1.3 GPIO Pin Map	62
10	LINUX	65
	10.1 Introduction	65
	10.2 Embedded Linux	65
	10.3 Reasons for Choosing Linux	66
	10.3.1 Quality & Reliability of Code	66
	10.3.2 Availability of Code	67
	10.3.3 Hardware Support	68
	10.3.4 Communication Protocol & Software Standards	68
	10.3.5 Players of The Embedded Linux Scene	69
	10.4 Host: Linux Workstation	69
	10.5 Host/Target Development Setups	69
11	BUILDING UP OF KERNEL WITH TOOLCHAIN	72
	11.1 Linux Kernel	72
	11.1.1 Configuring the Kernel	73
	11.1.2 Building Dependencies	73
	11.1.3 Building the Kernel	74
	11.2 Building the Modules	74
	11.3 GNU Tool-chain Basics	75
	11.3.1 Component Versions	75
	11.3.2 Building the Tool-chain	76
	11.3.2.1 Binutils	76
	11.3.2.2 Binutils Components	76
		78

11.3.3 Gcc	79	
11.3.4 C Library Setup	80	
11.3.5 Full Compiler Setup	82	
12	PROCESS OF BUILDING LINUX OPERATING	
	SYSTEM ON ARM PROCESSOR	85
12.1	Introduction	85
12.2	Boot Loaders	85
12.2.1	Running Red boot	86
13	USING BOARD PERIPHERALS	92
13.1	Flash	92
13.2	SDRAM	93
13.3	Serial Port	94
13.4	USB Host	95
13.4.1	USB 802.11b	95
13.5	Ethernet Device	96
13.6	Mass Storage	96
14	CONCLUSION AND FUTURE ENHANCEMENTS	99
	APPENDIX A (Pin Diagram)	100
	APPENDIX B (Pin configuration)	101
	REFERENCES	103

LIST OF FIGURES

S.NO	TITLE	PAGE NO
1	2.1 Basic Block Diagram	7
2	2-1 The schematic of processor and memory	8
3	3.1 Block Diagram	12
4	4.1 ARM920T Block Diagram	20
5	4.2 Typical AMBA AHB System	26
6	4.3 EP9301 Main Data Paths	27
7	5.1 Flow Chart For Operation Of The Boot ROM Software	36
8	6.1 Block Diagram Of Ethernet LAN controller	39
10	6.2 Ethernet Frame/Packet Format	42
11	6.3 Packet Transmission Process	43
12	6.4 Data Bit Transmission Order	46
13	7.1 USB focus areas	50
14	7.2 USB Host Controller Block Diagram	51
15	9.1 System Level GPIO Set Up	60
16	10.1 Host/Target Removable Storage Setup	69

LIST OF PHOTOS

S.NO	TITLE	PAGE NO
1	EP9302 Processor Based Board Top View	6

LIST OF TABLES

S.NO	TITLE	PAGE NO
1	4.1 AHB Arbiter Priority Scheme	30
2	4.2 AHB Peripheral Address Range	31
3	5.1 Show Configuration Settings That Are Common to all Boot modes	37
4	6.1 FIFO RAM address map	41
5	8.1 Book Device Selection	56
6	9.1 GPIO Port to Pin Map	62

CHAPTER 1

INTRODUCTION

1.INTRODUCTION

1.1 SYSTEM NEED

The size of the central processing unit in a general PC (Personal Computer) is determined by the hard disk and the motherboard. The hard disk drive used in general PC is a space occupying magnetic storage device. If there occurs a bad sector in this kind of device then it would be impossible to retrieve stored data from that part of the device. Moreover the storage and retrieval of information from these storage devices is not an easy task and it is time consuming. The usage of fan is another reason for the large size of general PC motherboards. Moreover in industrial automation applications the processing unit's design must be in an optimized manner for its better performance. In order to satisfy both the needs we go in for a motherboard design, which can be used for any of the general-purpose application systems.

1.2 OBJECTIVE

The aim of our project is to minimize the motherboard size. Instead of hard disk, flash memory device is used which reduces the size of the CPU to a large extent. Then the usage of ARM processor, instead of the regular processors, would reduce the size of the motherboard, as it does not require a fan to cool it. The following are the components that are present in the minimized motherboard (1) Flash Memory IC, (2) ARM processor IC, (3) D-RAM IC, (4) USB, A-V and VGA Ports, (5) Ethernet card, and (6) A PCI slot. The presence of these essential components is enough for the effective operation of the PC. Moreover this board can be used for a variety of applications right from industrial automation to server hubs. This is a generalized board having its base being suited to any kind of applications where the processing unit is required. Our project is mainly divided under three heads, (1) Board design, (2) Embedding linux into the flash memory and (3) Developing an application.

1.3 ORGANISATION OF THE REPORT

Chapter 1 provides the background information and objectives of the project.

Chapter 2 describes about the various components of the board. The design of the board is based entirely on the processor.

Chapter 3 describes about the processor and its features.

Chapter 4 deals with processor core. The core of the processor is the central part which determines the various operations of the processor.

Chapter 5 gives a view about the internal boot ROM that is available in the processor.

Chapter 6 describes about the Ethernet LAN controller that is present in the architecture of the processor and the means of accessing it.

Chapter 7 deals with the Universal Serial Bus host controller and its operation.

Chapter 8 describes about the SDRAM, Synchronous ROM and Synchronous Flash controllers and their means of access.

Chapter 9 gives an overview of the General Purpose Input\Output interface of the processor.

Chapter 10 describes about the LINUX operating system and Embedded Linux

Chapter 11 deals with the process of building the Linux tool chain.

Chapter 12 gives an overview of how to build the operating system on the processor.

Chapter 13 is about how to use the various board peripherals

Chapter 14 talks about conclusion and future enhancements

CHAPTER 2

EP9302 DEVELOPMENT BOARD

2. EP9302 DEVELOPMENT BOARD

2.1 INTRODUCTION

This chapter describes in brief about all the components available on the board and its functions and features.

2.2 PHYSICAL STRUCTURE AND FEATURES

The EP9302 board is a convenient and easy-to-operate evaluation platform. It has been designed to provide the majority of the EP9302 functions on a small 6" x 4" base board. These features include:

- EP9302 Processor Running at 166MHz
- 64MByte SDRAM
- 16MByte NAND Flash Memory
- Two USB 2.0 Full-speed Host Ports
- USB 2.0 High-speed Device Interface
- Audio Out
- Audio In
- Two UARTs
- 10/100 Ethernet Interface
- JTAG
- Expansion Connectors
- 5-channel ADC

Two high-density connectors have been provided to allow for daughter card expansion. The full memory bus is connected to one of the connectors and any peripherals not on the development board are attached to the other connector. In addition, some features such as Ethernet MII interface have been brought out to the peripheral connector as well.

EP9302 Processor Based Board Top View



- A. EP9302 Processor
- B. Processor Status LEDs - one Red and one Green
- C. USB 2.0 High-speed Device IC (ISP1581)
- D. SDRAM - 16-bit device
- E. Memory Bus Expansion Connector
- F. Main Power Switch, S2
- G. 12V Power Connector
- H. 3.3V Voltage Regulator - switching, 3A
- I. Serial Boot Pushbutton - labeled "SERIAL BOOT"

- K. UART1 Header - 5x2
- L. UART0, DB9 Male
- M. USB Device Connector
- N. Ethernet Connector
- O. Dual, Stacked USB Host Connector
- P. Audio Out Connector
- Q. Audio In Connector
- R. Peripheral Bus Expansion Connector
- S. ADC Connector
- T. JTAG Connector
- U. Commercial IR
- V. RTC and Battery Backup
- W. Power Status LEDs

One item not listed above is the Flash device. It is located on the bottom side of the board; under item C. There are no jumpers on the board. The only "jumper" is the serial boot pushbutton. This button is used when the developer wants to use the Cirrus Logic download utility to put new code into Flash memory.

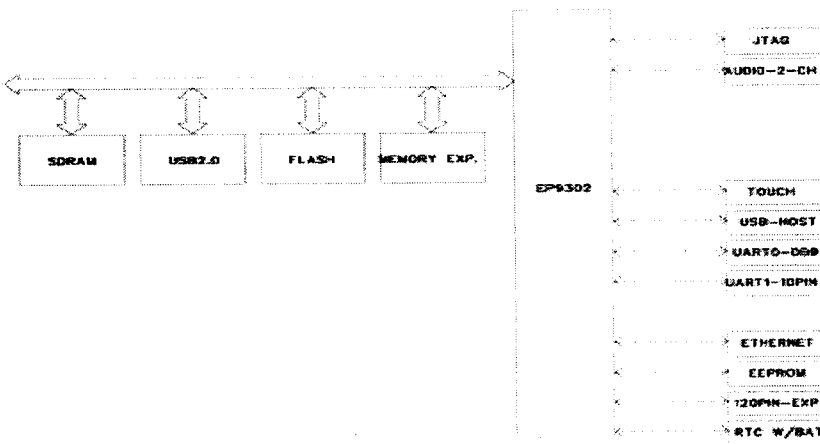


Figure 2-1 Basic Block Diagram

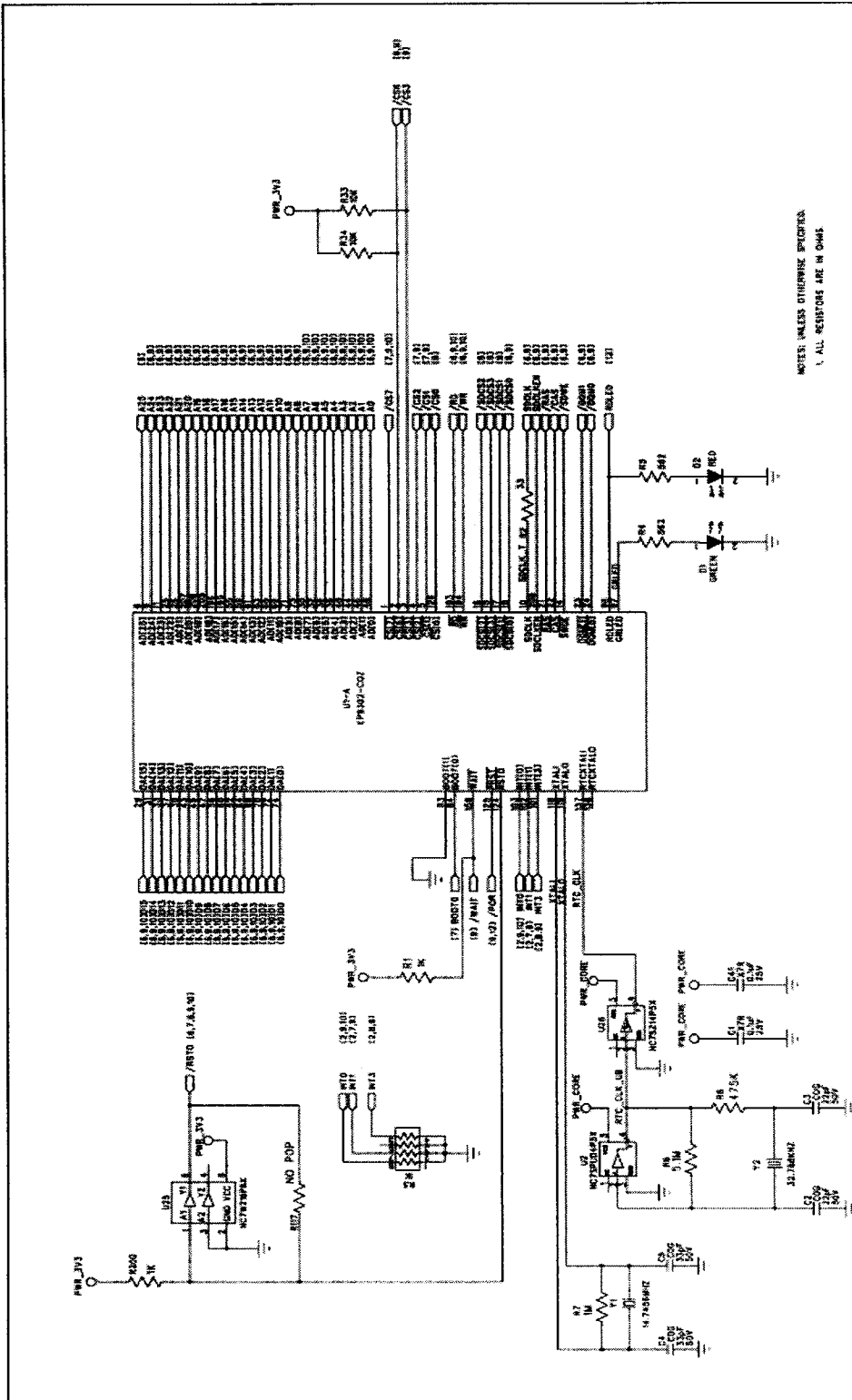


Figure 2-2 The schematic of processor and memory

2.3 CLOCKS AND OSCILLATORS

There are two main clock inputs to the EP9302 device. One is the 14.7456 MHz crystal oscillator and the other is the 32.768 kHz real time clock (RTC).

The 14.7456 MHz clock can be generated from a crystal circuit as shown in the schematics, or optionally from a 14.7456 MHz oscillator. If a 14.7456 MHz oscillator is used, then only the XTALI pin is driven and the drive level of the clock must be 3.3 V.

The RTC clock may be generated by the circuit shown in the schematics, or optionally an RTC oscillator. Due to the cost of RTC oscillators, the circuit shown in the schematics is used. An external oscillator is made by using an unbuffered '04 inverter. It is very important to use an unbuffered inverter in this application. Using a buffered inverter may make the circuit oscillate in the ~MHz range or not start at all.

The two LEDs connected to the EP9302 device are used to indicate processor health during boot and general status of the board. The reset output, RSTON (active-low signal) is buffered by U25. There is a resistor option for the RSTON signal to be either driven by this buffer or bypassed. If the reset signal is not going to be buffered, U25 must be removed and R117 installed. By default, RSTON is buffered.

2.4 THE COMMUNICATION PORTS (USB, UART AND GPIO)

The USB Host circuits are connected directly from the EP9302 to U3 and U4 and then on to the stacked USB connector, J1. U3 and U4 provide termination for the USB signals and ESD protection. Power for the USB Host ports comes from the +5V switching regulator and is protected with poly fuses rated for 0.5 A each.

The board has both UARTs brought out. The main UART interface, UART0, is connected to a standard male DB9 connector and provides for full modem control. The other UART interface is connected to a 5x2 header. There is an option for installing a zero-ohm resistor if the developer needs to provide +5V power to an external device. The pinout of the headers matches common IDC 10-to-DB9 cables. One such cable is included with the kit. All UART signals are level shifted from TTL to RS-232.

The CIR uses an enhanced GPIO (EGPIO) line to communicate to the EP9302 device. This is the power section for the EP9302 device. The ADC and PLL supplies are

filtered. There is no reason to filter the 3.3 and the 1.8V power rails. An external USB 2.0 High-speed Peripheral device is provided. The USB device allows a Host to see the board as a Mass Storage device. The USB interface chip is connected to the lower 16-bits of the memory bus.

2.5 MEMORY DEVICES (SDRAM & EEPROM)

The SDRAM interface is comprised of one 16-bit SDRAM device to form a 16-bit SDRAM bus. The SDRAM is connected to /SDCS0 and is located at physical memory address 0xC000_0000. The Flash interface is made from a single 16-bit device. It uses a "multi-cell" Flash device. The Flash device is connected to /CS6 and is located at physical memory address 0x6000_0000. The referenced design uses only one Flash device. The Flash device installed is a 128 Mbit, 16 Mbyte, device.

The serial EEPROM is a 4 Mbit device and may be used for serial "SPI Boot" or for storing the Ethernet MAC address. SPI Boot is not used as the default boot method but may be used by the developer if it fits his/her application. Details about SPI Boot are in the EP9302 User's Guide. The serial EEPROM is accessed only if the EGPIO7 line is configured to the proper level. The SPI™ frame signal and the EGPIO7 "enable" signal are OR'ed to create the enable for the serial EEPROM device. EGPIO7 must be low in order to communicate to the serial EEPROM. Other devices on the SPI bus must be enabled in a similar manner but not enabled simultaneously.

2.6 JTAG INTERFACE

The JTAG interface is connected to a 2x10 header, JP3. This connector is wired for the Multi-ICE debugger. There are 10 signals that determine how the EP9302 will boot and operate. BOOT1 is connected to GND. BOOT1 is used for factory testing only. The other nine signals are either pulled up or down external to the EP9302 device. The boot configuration shown sets the EP9302 device to perform an Internal Async boot from a 16-bit-wide memory.

The following chapter describes in detail about the processor.

CHAPTER 3

PROCESSOR-EP9302



P-2097

3. PROCESSOR-EP9302

3.1 INTRODUCTION

The EP9302 is a highly integrated system-on-chip processor that paves the way for a multitude of next-generation consumer and industrial electronic products. Designers of digital media servers and jukeboxes, telematic control systems, thin clients, set-top boxes, point-of-sale terminals, industrial controls, biometric security systems, and GPS devices will benefit from the EP9302's integrated architecture and advanced features. In fact, with amazingly agile performance provided by a 166 MHz ARM920T processor, and featuring an incredibly wide breadth of peripheral interfaces, the EP9302 is well suited to an even broader range of high volume applications. Furthermore, by enabling or disabling the EP9302's peripheral interfaces, designers can reduce development costs and accelerate time-to-market by creating a single platform that can be easily modified to deliver a variety of differentiated end products.

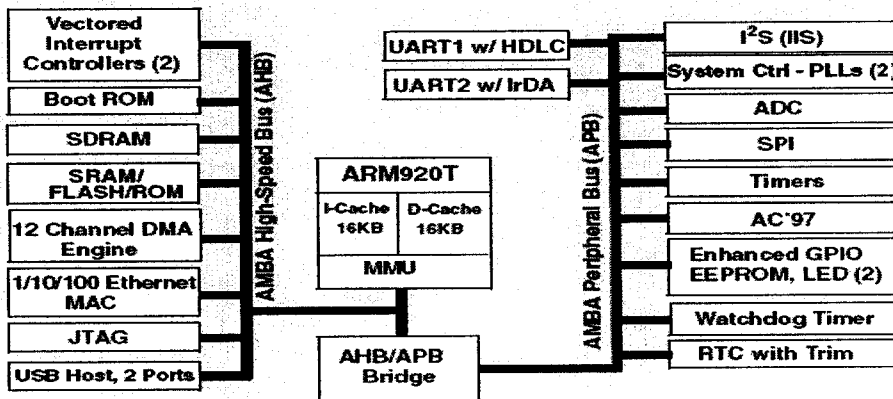


Figure 3.1 Block diagram

3.2 EP9302 FEATURES

The **EP9302** processor **200-MHz ARM920T** Processor with the following

- It has a 16-kbyte Instruction Cache
- It has a 16-kbyte Data Cache
- It supports Linux®, Microsoft® Windows® CE-enabled Memory Management Unit
- It has a 100-MHz System Bus

It has a **MaverickCrunch™ Math Engine** which supports the following

- Floating point, Integer and Signal Processing Instructions
- Optimized for digital music compression and decompression algorithms.
- Hardware interlocks allow in-line coding.

It has a **MaverickKey™ IDs** which is a 32-bit unique ID can be used for DRM-compliant, 128-bit random ID.

It has the following **Integrated Peripheral Interfaces**

- 16-bit SDRAM Interface (up to 4 banks)
- 16-bit SRAM / FLASH / ROM
- Serial EEPROM Interface
- 1/10/100 Mbps Ethernet MAC
- Two UARTs
- Two-port USB 2.0 Full-speed Host (OHCI) (12 Mbits per second)
- IrDA Interface
- ADC
- Serial Peripheral Interface (SPI) Port
- 6-channel Serial Audio Interface (I2S)
- 2-channel, Low-cost Serial Audio Interface (AC'97)

It has the following **Internal Peripherals**

- 12 Direct Memory Access (DMA) Channels
- Real-time Clock with Software Trim
- Dual PLL controls all clock domains.

- Watchdog Timer
- Two General-purpose 16-bit Timers
- One General-purpose 32-bit Timer
- One 40-bit Debug Timer
- Interrupt Controller
- Boot ROM

The processor is available in a **208-pin LQFP package**

3.3 EP9302 APPLICATIONS

The EP9302 is an ARM920T-based system-on-a-chip design with a large peripheral set targeted to a variety of applications:

- Industrial controls
- Digital media servers
- Integrated home media gateways
- Digital audio jukeboxes
- Streaming audio players
- Set-top boxes
- Point-of-sale terminals
- Thin clients
- Biometric security systems
- GPS & fleet management systems
- Educational toys
- Industrial computers
- Industrial hand-held devices
- Voting machines
- Medical equipment

3.4 OVERVIEW OF EP9302 FEATURES

3.4.1 High-Performance Arm920t Processor Core

The EP9301 features an advanced ARM920T processor design with an MMU that supports Linux[®], Windows[®] CE, and many other embedded operating systems. The ARM920T's 32-bit microcontroller architecture, with a five-stage pipeline, delivers impressive performance at very low power. The included 16 KByte instruction cache and 16 KByte data cache provide zero-cycle latency to the current program and data, or can be locked to provide guaranteed no-latency access to critical instructions and data. For applications with instruction memory size restrictions, the ARM920T's compressed Thumb[®] instruction set provides a space-efficient design that maximizes external instruction memory usage.

3.4.2 MaverickKey[™] Unique ID Secures Digital Content and OEM Designs

MaverickKey unique hardware programmed IDs provide an excellent solution to the growing concern over secure Web content and commerce. With Internet security playing an important role in the delivery of digital media such as books or music, traditional software methods are quickly becoming unreliable. The MaverickKey unique IDs provide OEMs with a method of utilizing specific hardware IDs for DRM (Digital Rights Management) mechanisms.

MaverickKey uses a specific 32-bit ID and a 128-bit random ID that are programmed into the EP9301 through the use of laser probing technology. These IDs can then be used to match secure copyrighted content with the ID of the target device that the EP9301 is powering, and then deliver the copyrighted information over a secure connection. In addition, secure transactions can benefit by matching device IDs to server IDs.

MaverickKey IDs can also be used by OEMs and design houses to protect against design piracy by presetting ranges for unique IDs. For more information on securing your design using MaverickKey, please contact your Cirrus Logic sales representative.

3.4.3 Integrated Two-port USB 2.0 Full Speed Host with Transceivers

The EP9301 integrates two USB 2.0 Full Speed host ports. Fully compliant to the OHCI USB 2.0 Full Speed specification (12 Mbps), the host ports can be used to provide connections to a number of external devices including mass storage devices, external portable devices such as audio players or cameras, printers, or USB hubs. Naturally, the two-port USB host also supports the USB 2.0 Low Speed standard. This provides the opportunity to create a wide array of flexible system configurations.

3.4.4 Integrated Ethernet MAC Reduces BOM Costs

The EP9301 integrates a 1/10/100 Mbps Ethernet Media Access Controller (MAC) on the device. With a simple connection to an M II-based external PHY, an EP9301-based system has easy, high-performance, cost-effective Internet capability.

3.4.5 Multiple Booting Mechanisms Increase Flexibility

The processor includes a 16 KByte boot ROM to set up standard configurations. Optionally, the processor may be booted from FLASH memory, over the SPI serial interface, or through the UART. This boot flexibility makes it easy to design user-controlled, field-upgradable systems. See Chapter 3 on page 59, for additional details.

3.4.6 Abundant General Purpose I/Os Build Flexible Systems

The EP9301 includes both enhanced and standard general-purpose I/O pins (GPIOs). The 16 different enhanced GPIOs may individually be configured as inputs, outputs, or interrupt-enabled inputs. There are an additional 31 standard GPIOs that may individually be used as inputs, outputs, or open-drain pins. The standard GPIOs are multiplexed with peripheral function pins, so the number available depends on the utilization of peripherals. Together, the enhanced and standard GPIOs facilitate easy system design with external peripherals not integrated on the EP9301.

3.4.7 General-Purpose Memory Interface (SDRAM, SRAM, ROM and FLASH)

The EP9301 features a unified memory address model in which all memory devices are accessed over a common address/data bus. Memory accesses are performed via the high-speed processor bus. The SRAM memory controller supports 8 and 16-bit devices and accommodates an internal boot ROM concurrently with a 16-bit SDRAM memory.

3.4.8 12-Bit Analog-to-Digital Converter (ADC) Functionality

The EP9301 includes a 12-bit ADC, which can be used for General ADC functionality. The interface performs all sampling, averaging, ADC range checking, and control for a wide variety of applications. To improve system performance, the converter only interrupts the processor when a meaningful change occurs. The core of the processor is the most important part of processor. The following chapter describes about the core and the bus which connects the processor.

CHAPTER 4

ARM 920T CORE

4. ARM 920T CORE

4.1 INTRODUCTION

This section discusses the ARM920T processor core and the Advanced High-Speed Bus (AHB).

4.2 OVERVIEW: ARM920T PROCESSOR CORE

The ARM920T is a Harvard architecture processor core with separate 16 kbyte instruction and data caches with an 8-word line length used in the EP9301. The processor core utilizes a five-stage pipeline consisting of fetch, decode, execute, data memory access, and write stages.

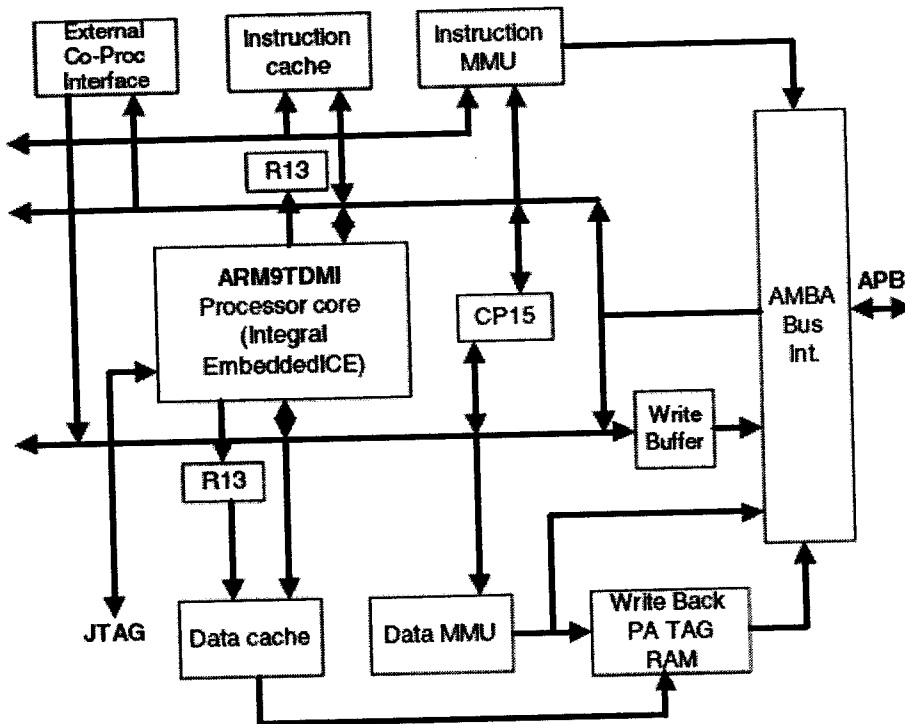
4.2.1 Features

Key features include:

- ARM V4T (32-bit) and Thumb (16-bit compressed) instruction sets
- 32-bit Advanced Micro-Controller Bus Architecture (AMBA)
- 16 kbyte Instruction Cache with lockdown
- 16 kbyte Data Cache (programmable write-through or write-back) with lockdown
- Write Buffer
- MMU for Microsoft Windows CE and Linux operating systems
- Translation Look-aside Buffers (TLB) with 64 Data and 64 Instruction Entries
- Programmable Page Sizes of 64 kbyte, 4 kbyte, and 1 kbyte
- Independent lockdown of TLB Entries
- JTAG Interface for Debug Control

4.2.2 Operations

Figure 4.1 ARM920T Block Diagram



The ARM920T core follows Harvard architecture and consists of an ARM9TDMI core, MMU, instruction and data cache. The core supports both the 32-bit ARM and 16-bit Thumb instruction sets. The internal bus structure (AMBA) includes both an internal high speed and external low speed bus. The high speed bus AHB (Advanced High-performance Bus) contains a high speed internal bus clock to synchronize coprocessor, MMU, cache, DMA controller, and memory modules. AMBA includes a AHB/APB bridge to the lower speed APB (Advanced Peripheral Bus). The APB bus connects to lower speed peripheral devices such as UARTs and GPIOs. The MMU provides memory address translation for all memory and peripherals designed to remap memory devices and peripheral address locations. Sections, large, small and tiny pages are programmable to map memory in 1 Mbyte, 64 kbyte, 4 kbyte, 1 kbyte size blocks. To increase system performance, a 64-entry translation look-aside buffer will cache 64 address locations before a TLB miss occurs.

A 16 kbyte instruction and a 16 kbyte data cache are included to increase performance for cache-enabled memory regions. The 64-way associative cache also has lock-down capability. Cached instructions and data also have access to a 16-word data and 4-word instruction write buffer to allow cached instructions to be fetched and decoded while the write buffer sends the information to the external bus. The ARM920T core supports a number of coprocessors by means of a specific pipeline architecture interface.

4.2.3 ARM9TDMI Core

ARM9TDMI core is responsible for executing both 32-bit ARM and 16-bit Thumb instructions. Each provides a unique advantage to a system design. Internally, the instructions enter a 5-stage pipeline. These stages are:

- Instruction Fetch
- Instruction Decode
- Execute
- Data Memory Access
- Register Write

All instructions are fully interlocked. This mechanism will delay the execution stage of a instruction if data in that instruction comes from a previous instruction that is not available yet. This simply insures that software will function identically across different implementations. For memory access instructions, the base register used for the access will be restored by the processor in the event of an Abort exception. The base register will be restored to the value contained in the processor register before execution of the instruction. The ARM9TDMI core memory interface includes a separate instruction and data interface to allow concurrent access of instructions and data to reduce the number of CPI (cycles per instruction). Both interfaces use pipeline addressing. The core can operate in big and little endian mode. Endianess affects both the address and the data interfaces. The memory interface executes four types of memory transfers: sequential,

word, burst, and coprocessor. It will also support uni- and bidirectional

transfer modes. The core provides a debug interface called JTAG (Joint Testing Action Group). This interface provides debug capability with five external control signals:

- **TDO** - Test Data Out
- **TDI** - Test Data In
- **TMS** - Test Mode Select
- **TCK** - Test Clock
- **nTRST** - Test Reset

There are six scan chains (0 through 5) in the ARM9TDMI controlled by the JTAG Test Access Port (TAP) controller. Details on the individual scan chain function and bit order can be found in the ARM920T Technical Reference Manual.

4.2.3.1 Memory Management Unit

The MMU provides the translation and access permissions for the address and data ports for the ARM9TDMI core. The MMU is controlled by page tables stored in system memory and accessed using the CP15 register 1. The main features of the MMU are as follows:

- Address Translation
- Access Permissions and Domains
- MMU Cache and Write Buffer Access

4.2.3.2 Address Translation

The virtual address from the ARM920T core is modified by R13 internally to create a modified virtual address. The MMU then translates the modified virtual address from R13 by the CP15 register 3 into a physical address to access external

memory or a device. The MMU looks for the physical address from the Translation Table Base (TTB) in system memory. It will also update the TLB cache.

The TLB is two 64-entry caches, one for data and one for instruction. If the physical address for the current virtual address is not found in the TLB (miss), the processor will go to external memory and look for the TTB in system memory. The internal translation table walks hardware steps through the page table setup in external memory for the appropriate physical address. When the physical address is acquired, the TLB is updated. When the address is found in the TLB, system performance will increase since it will take additional cycles to access memory and update the TLB. Translation of system memory is done by breaking up the memory into different size blocks called sections, large pages, small pages, and tiny pages. System memory and registers can be remapped by the MMU. The block sizes are as follows:

- Section - 1 Mbyte
- Large Page - 64 kbyte
- Small Page - 16 kbyte
- Tiny Page - 1 kbyte

4.2.3.3 Access Permission and Domains

Access to any section or page of memory is dependent on its domain. The page table in external memory also contains access permissions for all subdivisions of external memory. Access to specific instructions or data has three possible states, assuming access is permitted:

- *Client* Access permissions based on the section or page table descriptor
- *Manager*. Ignore access permissions in the section or page table descriptor
- *No access*: any attempted access generates a domain fault

4.2.3.4 MMU Enable

Enabling the MMU allows for system memory control, but is also required if the data cache and the write buffer are to be used. These features are enabled for specific memory regions, as defined in the system page table. MMU enable is done via CP15 register 1. The procedure is as follows:

1. Program the Translation Table Base (TTB) and domain access control registers.
2. Create level 1 and level 2 pages for the system and enable the cache and the write buffer.
3. Enable MMU - bit 0 of CP15 register 1.

4.2.4 Cache and Write Buffer

Cache configuration is 64-way set associative. There is a separate 16 kbyte instruction and data cache. The cache has the following characteristics:

- 8 words per line with 1 valid bit and 2 dirty bits per line for allowing half-line write-backs.
- Write-through and write-back capable, selectable per memory region defined by the MMU.
- Pseudo random or round robin replacement algorithms for cache misses. This is determined by the RR bit (bit 14 in CP15 register 1). An 8-word line is reloaded on a cache miss.
- Independent cache lock-down with granularity of 1/64th of total cache size or 256 bytes for both instructions and data. Lock-down of the cache will prevent an eight-word cache line fill of that region of cache.

For compatibility with Windows CE and to reduce latency, physical addresses stored for data cache entries are stored in the PA TAG RAM to be used for cache linewrite-back operations without need of the MMU, which prevents a possible TLB miss that would degrade performance.

- Write Buffer is a 4-word instruction x 16-word data buffer. If enabled, writes are sent to buffer directly from cache or from the CPU in the event of a cache miss or cache not enabled.

4.2.4.1 Instruction Cache Enable

- At reset, the cache is disabled.
- A write to CP15 register 1, bit 12, will enable or disable the Instruction Cache. If the Instruction Cache (I-Cache) is enabled without the MMU enabled, all accesses are treated as cacheable.
- If disabled, current contents are ignored. If re-enabled before a reset, contents will be unchanged but may not be coherent with main memory. If so, contents must be flushed before re-enabling.

4.2.4.2 Data Cache Enable

- A write to CP15 register 1, bit 0, will enable or disable the Data Cache (DCache)/Write Buffer.
- D-Cache must only be enabled when the MMU is enabled. All data accesses are subject to MMU and permission checks.
- If disabled, current contents are ignored. If re-enabled before a reset, contents will be unchanged but may not be coherent with main memory. Depending on system software, a clean and invalidate action may be required before re-enabling.

4.2.4.3 Write Buffer Enable

- The Write bugger is enabled by the page table entries in the MMU. The Write buffer is not enabled unless MMU is enabled.

4.3. ADVANCED HIGH-SPEED BUS (AHB)

4.3.1 AMBA AHB Bus Interface Overview

The AMBA AHB is designed for use with high-performance, high clock frequency system modules. The AHB acts as the high-performance system backbone bus.

AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces with low-power peripheral functions. AHB is also specified to ensure ease of use in an efficient design flow using synthesis and automated test techniques. The figure shows a typical AMBA AHB System. AHB (Advanced High-Performance Bus) connects with devices that require greater bandwidth, such as DMA controllers, external system memory, and coprocessors. The AMBA AHB bus has the following characteristics:

- Burst Transactions
- Split Transactions
- Bus Master hand-over to devices, that is, DSP or DMA controller
- Single clock edge operations

APB (Advanced Peripheral Bus) is a lower bandwidth lower power bus which provides the following:

- Low Power Operations
- Latched address and control
- Simple Interface

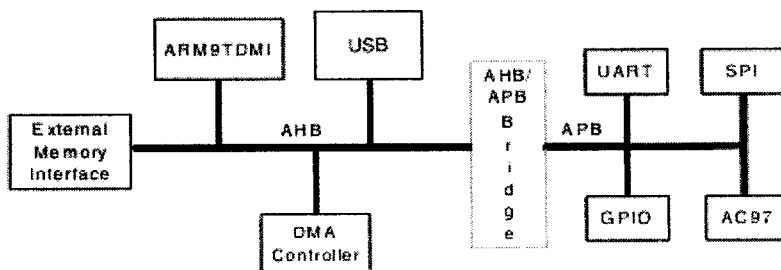


Figure4.2 Typical AMBA AHB System

4.3.2 EP9301 AHB Implementation Details

Peripherals that have high bandwidth or latency requirements are connected to the EP9301 processor using the AHB bus. These include the external memory interface, Vectored Interrupt Controllers (VIC1, VIC2), DMA, USB host, Ethernet MAC and the bridge to the APB interface. The AHB/APB Bridge transparently converts the AHB access into the slower speed APB accesses. All of the control registers for the APB peripherals are programmed using the AHB/APB bridge interface. The main AHB data and address lines are configured using a multiplexed bus. This removes the need for three state buffers and bus holders and simplifies bus arbitration. Figure 2-3 shows the main data paths in the EP9301 AHB implementation. Before an AMBA-to-AHB transfer can commence, the bus master must be granted access to the bus. This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master will be granted use of the bus. A granted bus master starts an AMBA-to-AHB transfer by driving the address and control signals. These signals provide information on the address, direction and width of the transfer, as well as indicating whether the transfer forms part of a burst.

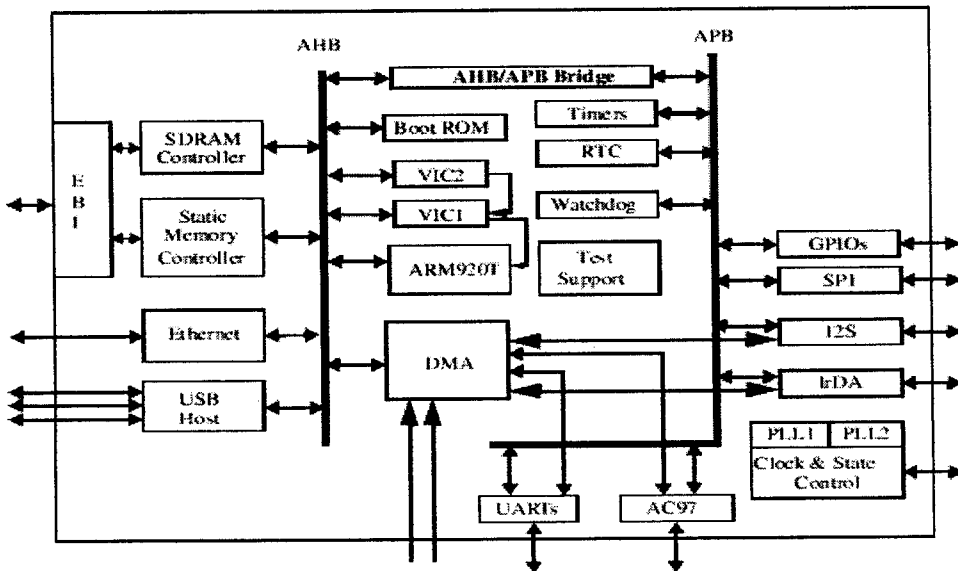


Figure 4.3 EP9302 Main Data Path

Two different forms of burst transfers are allowed:

- Incrementing bursts, which do not wrap at address boundaries
- Wrapping bursts, which wrap at particular address boundaries.

A write data bus is used to move data from the master to a slave, while a read data bus is used to move data from a slave to the master. Every transfer consists of:

- An address and control cycle
- One or more cycles for the data.

In normal operation a master is allowed to complete all the transfers in a particular burst before the arbiter grants another master access to the bus. However, in order to avoid excessive arbitration latencies, it is possible for the arbiter to break up a burst, and, in such cases, the master must re-arbitrate for the bus in order to complete the remaining transfers in the burst.

4.3.3 Memory and Bus Access Errors

There are several possible sources of access errors.

- Reads to reserved or undefined register memory addresses will return indeterminate data. Writes to reserved or undefined memory addresses are generally ignored, but this behavior is not guaranteed. Many register addresses are not fully decoded, so aliasing may occur. Addresses and memory ranges listed as **Reserved** should not be accessed; access behavior to these regions is not defined.
- Access to non-existent registers or memory may result in a bus error.
- Any access in the APB control register space will complete normally, as these devices have no means of signaling an error.
- Access to non-existent AHB/APB registers may result in a bus error, depending on the device and nature of the error. Device specific access rules are defined in the device descriptions.
- External memory access is controlled by the Static Memory Controller

(SMC) and the Synchronous Dynamic RAM (SDRAM) controller. In general, access to non-existent external memory will complete normally, with reads returning random false data.

4.3.4 Bus Arbitration

The arbitration mechanism is used to ensure that only one master has access to the bus at any one time. The arbiter performs this function by observing a number of different requests to use the bus and deciding which is currently the highest priority master requesting the bus.

The arbitration scheme can be broken down into three main areas:

- The main AHB system bus arbiter
- The SDRAM slave interface arbiter
- The EBI bus arbiter

4.3.4.1 Main AHB Bus Arbiter

This arbiter controls the bus master arbitration for the AHB bus. The AHB bus has several Master interfaces. These are:

- ARM920T
- DMA controller
- USB host (USB1, 2)
- Ethernet MAC

These interfaces have an order of priority that is linked closely with the power saving modes. The power saving modes of Halt and Standby force the arbiter to grant the default bus master, in this case, the ARM920T.

In summary, the order of priority of the bus masters, from highest to lowest, is shown in Table 4-1.

Table 4-1: AHB Arbiter Priority Scheme

Priority Number	PRIOR 00 (Reset value)	PRIOR 01	PRIOR 10	PRIOR 11
1	MAC	MAC	DMA	DMA
2	USB	USB	USB	MAC
3	DMA	ARM920T	MAC	USB
4	ARM920T	DMA	ARM920T	ARM920T

The priority of the Arbiter can be programmed in the BusMstrArb register in the Clock and State Controller. The Arbiter can also be programmed to degrant one of the following masters: DMA, USB Host or Ethernet MAC, if an interrupt (IRQ or FIQ) is pending or being serviced. This prevents one of these masters from blocking important interrupt service routines. These masters are prevented from accessing the bus, and their bus requests are masked, until the IRQ/FIQ is removed (by the Interrupt Service Routine), at which point their bus requests will be recognized. The default is to program the Arbiter so that it does *not* degrant any of these masters.

In normal operation, when the ARM920T is granted the bus and a request to enter Halt mode is received, the ARM920T is de-granted from the AHB bus. Any other master requesting the bus in Halt mode (according to the priority) will be granted the bus. In the case of the entry into Standby, the dummy master will be granted the bus, which simply performs IDLE transfers. In this way, **all the masters except the ARM920T can be used during Halt mode, but are shutdown during an entry into Standby.**

4.3.4.2 EBI Bus Arbiter

This arbiter is used to arbitrate between accesses from the SDRAM controller and the Static Memory controller. The priority is given to accesses from the SDRAM controller.

4.4 AHB Decoder

The AHB decoder contains the memory map for all the AHB masters/slaves and the APB bridge. When a particular address range is selected, the appropriate signal is generated. It is defined in Table 2-2.

Table 4-2: AHB Peripheral Address Range

Address Range	Register	Peripheral	Peripheral
0x800D_0000 - 0x800F_FFFF	-	-	Reserved
0x800C_0000 - 0x800C_FFFF	32	AHB	VIC2
0x800B_0000 - 0x800B_FFFF	32	AHB	VIC1
0x800A_0000 - 0x800A_FFFF	-	-	Reserved
0x8009_0000 - 0x8009_FFFF	32	AHB	Boot ROM
0x8008_0000 - 0x8008_FFFF	32	AHB	SRAM
0x8007_0000 - 0x8007_FFFF	-	-	Reserved
0x8006_0000 - 0x8006_FFFF	32	AHB	SDRAM
0x8005_0000 - 0x8005_FFFF	-	-	Reserved
0x8004_0000 - 0x8004_FFFF	-	-	Reserved
0x8003_0000 - 0x8003_FFFF	-	-	Reserved
0x8002_0000 - 0x8002_FFFF	32	AHB	USB Host
0x8001_0000 - 0x8001_FFFF	32	AHB	Ethernet MAC
0x8000_0000 - 0x8000_FFFF	32	AHB	DMA

4.4.1 AHB Bus Slave

An AHB slave responds to transfers initiated by bus masters within the system. The slave uses signals from the decoder to determine when it should respond to a bus transfer. All other signals required for the transfer, such as the address and control information, are generated by the bus master.

4.4.2 AHB to APB Bridge

The AHB to APB bridge is an AHB slave, providing an interface between the high-speed AHB and the low-power APB. Read and write transfers on the AHB are converted into equivalent transfers on the APB. As the APB is not pipelined. Wait states are added during transfers to and from the APB when the AHB is required to wait for the APB.

The main sections of this module are:

- AHB slave bus interface
- APB transfer state machine, which is independent of the device memory map

4.4.2.1 Function and Operation of APB Bridge

The APB bridge responds to transaction requests from the currently granted AHB master. The AHB transactions are then converted into APB transactions.

4.4.3 APB Bus Slave

An APB slave responds to transfers initiated by bus masters within the system. The slave uses signals from the decoder to determine when it should respond to a bus transfer. All other signals required for the transfer, such as the address and control information, are generated by the APB bridge. If an undefined location is accessed, operation of the system continues as normal, but no peripherals are selected. The APB bridge acts as the only master on the APB.

4.4.4 Register Definitions

ARM has thirty seven 32-bit internal registers, some are modal, some are banked. If operating in Thumb mode, the processor must switch to ARM mode before taking an exception. The return instruction will restore the processor to Thumb state. Most tasks are executed out of User mode.

User:	Unprivileged normal operating mode
FIQ:	Fast interrupt (high priority) mode when FIQ is asserted
IRQ:	Interrupt request (normal) mode when IRQ is asserted
Supervisor:	Software interrupt instruction (SWI) or reset will cause entry into this mode
Abort:	Memory access violation will cause entry into this mode
Undef:	Undefined instructions
System:	Privileged mode. Uses same registers as user mode

CHAPTER 5

BOOT ROM

5. BOOT ROM

5.1 INTRODUCTION

This chapter deals with the Boot ROM used for the board. The Boot ROM allows a program or OS to boot from the following devices:

- SPI EEPROM
- FLASH/SyncFLASH or SROM
- Serial port

5.1.1 Boot ROM Hardware Operational Overview

The Boot ROM is an AHB slave device containing a 16 kbyte mask-programmed ROM. The AHB slave always operates with one wait state, so all data reads from the ROM use 2 HCLK cycles.

The ROM contains 3 code sections. The lower 8 kbytes contain the system boot code. The next 4 kbytes contain the first secure code block, and the top 4 kbytes contain the second secure code block. In non-secure boot, the lower 8 kbytes are accessible. In secure boot, one of the two secure code blocks is accessible. See Chapter 22, "Security," for details.

On system reset, the ARM920T begins executing code at address zero. The system follows the Hardware Configuration controls to select the boot device that appears at address zero. If Internal Boot is selected, the Boot ROM is mapped to address zero and the ARM920T will execute the Boot ROM code.

5.1.1.1 Memory Map

The Boot ROM base address (ROM base) is fixed in the EP9301 at 0x8009_0000. It will alias on 16 kbyte intervals. When internal boot is active, the Boot ROM is double decoded and appears at its normal address space and at address zero. **(The Boot ROM writes the BootModeClr in order to remap address 0x0 to be external memory while the Boot ROM code continues execution at 0x8009_0000.)**

5.1.2 Boot ROM Software Operational Overview

The Boot ROM is a 16 kbyte mask-programmed ROM that controls the source of the first off-chip code executed by the EP9301. The code within the Boot ROM supports the following sources for the EP9301 initialization program

- FLASH: Code present in FLASH memory is executed directly.

Note that the code retrieved via UART1 and the SPI Serial ROM is not intended to be a complete operating system image. It is intended to be a small (up to 2 kbyte) loader that will, in turn, retrieve a complete operating system image. This small loader can retrieve this complete image through UART1 or the SPI Serial ROM (just as the Boot ROM did) or it can be more sophisticated and retrieve it through the IrDA, USB, or Ethernet interfaces.

The Boot ROM code disables the ARM920T's MMU, so any loader program that is downloaded sees physical addresses. The loader is free to initialize the page tables and start the MMU and caches if needed.

The Boot ROM code also does not enable interrupts or timers, so that the system delivered to the user is in a known safe state and is ready for an operating system or for user code to be loaded.

5.1.2.1 Flowchart

Figure 5-1 provides a flow chart for operation of the Boot ROM software.

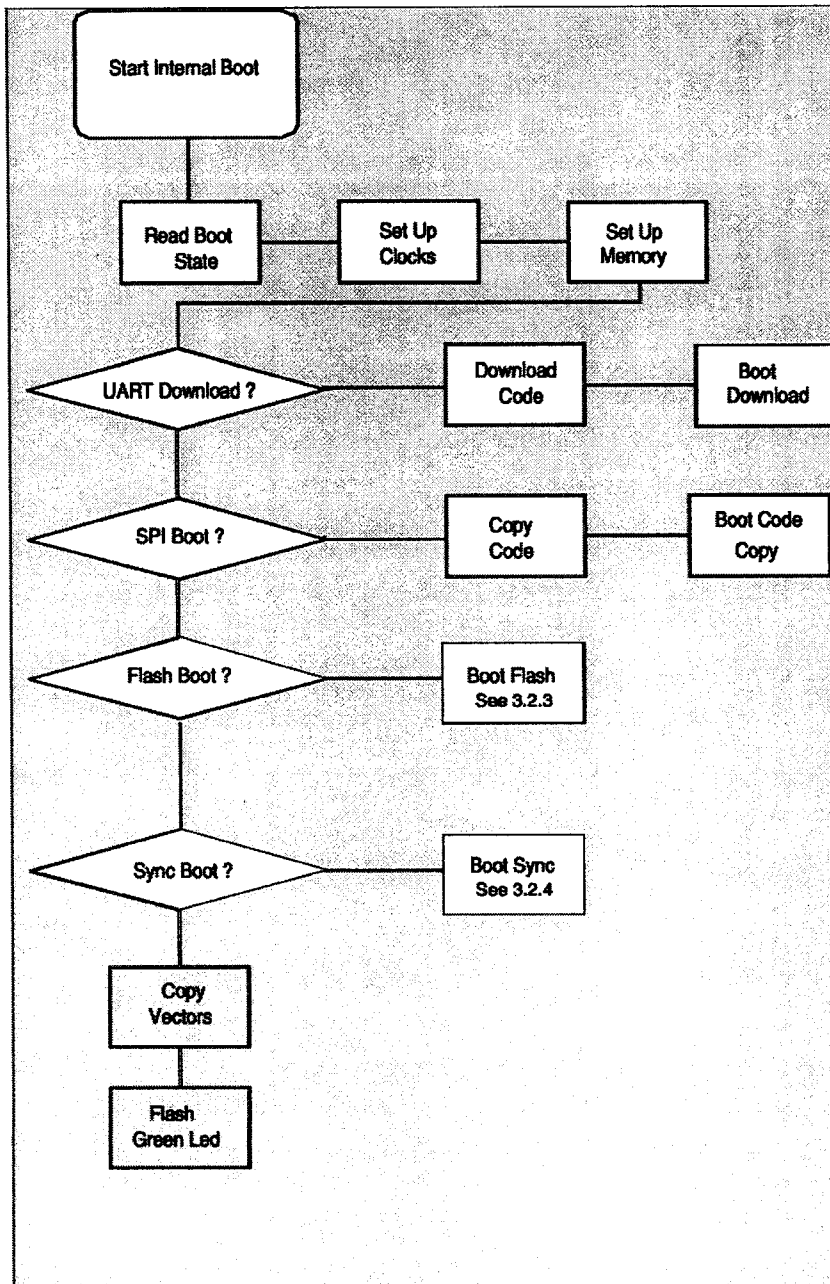


Figure 5.1 Flow Chart for Operation of the Boot ROM Software

5.2 Boot Options

Table 5-1 shows configuration settings that are common to all boot modes.

Table 5-1: Boot Configuration Options (Normal Boot)

EECLK	EEDAT	LBOOT1	LBOOT0	ASDO	CSn[7:6]	Boot Configuration
0	1	0	0	1	00 01	External boot from Sync memory space selected by DevCfg3 through the SDRAM Controller. The media type must be either SROM or SyncFLASH. The selection of the SRAM width is determined by latched CSn[7:6] value: 16-bit SFLASH 16-bit SROM
0	1	0	0	0	00 01	External boot from Async memory space selected by nCS0 through Synchronous Memory Controller. The selection of the SRAM width is determined by latched CSn[7:6] value: 8-bit SRAM 16-bit SRAM
1	1	0	1	x	01	16-bit serial boot
1	1	0	0	1	00 01	Internal boot from on-chip ROM. The selection of the ROM width is determined by latched CSn[7:6] value: 16-bit 16-bit
1	1	0	0	0	00 01	Internal boot from on-chip ROM. The selection of the ROM width is determined by latched CSn[7:6] value: 8-bit 16-bit

The Processor is well connected to ROM and other peripherals through an Ethernet interface and the forthcoming chapter describes about the Ethernet interface.

CHAPTER 6

1/10/100 MBPS ETHERNET LAN

CONTROLLER

6.1/10/100 Mbps Ethernet LAN Controller

6.1 INTRODUCTION

The Ethernet LAN Controller incorporates all the logic needed to interface directly to the AHB and to the Media Independent Interface (MII). It includes local memory and DMA control, and supports full duplex operation with flow control support. Figure 6-1 shows a simplified block diagram.

This block was designed with a RAM of 544 words, each word containing 33 bits. These RAMs are used for packet buffering and controller data storage. One RAM is dedicated to the receiver, and one dedicated to the transmitter. These RAMs are mapped into the register space and are accessible via the AHB.

Block Diagram

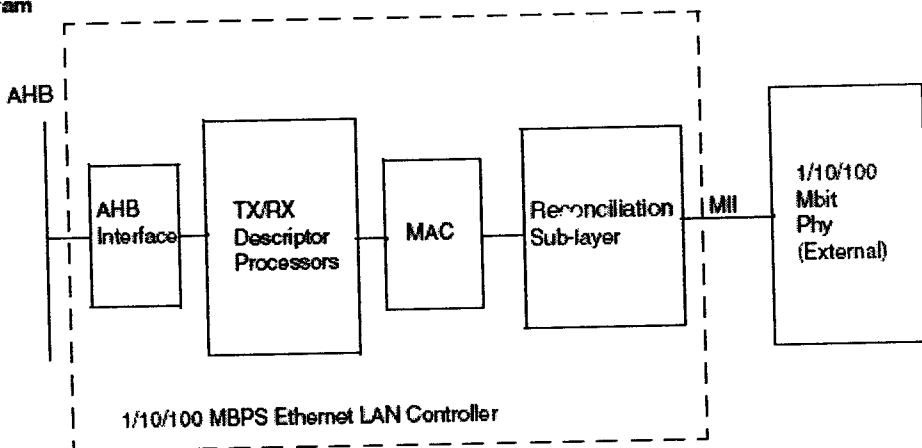


Figure 6.1 1/10/100 Mbps Ethernet LAN Controller

6.1.1 Host Interface and Descriptor Processor

The Host Interface can be functionally decomposed into the AHB Interface Controller and the Descriptor Processor. The AHB Interface Controller implements the actual connection to the AHB. The controller responds as a AHB bus slave for register programming, and acts as an AHB bus master for data transfers. The Descriptor Processor implements the Hardware Adapter Interface Algorithm and generates transfer requests to the AHB Interface Controller. The back-end interfaces to the MAC controllers and services MAC requests to run accesses to the FIFO and update queue status. The descriptor processor also generates internal requests for descriptor fetches. A priority arbiter arbitrates among the various requests and generates transfer requests to the AHB Interface Controller. There are 6 queues that require service in system memory:

- RxData: Write received frame data to host memory.
- RxStatus: Write received frame status to host memory.
- TxData: Read frame data from host memory.
- TxStatus: Write transmitted frame status to host memory.
- RxDescriptor: Read descriptors from host memory.
- TxDescriptor: Read descriptors from host memory.

Each queue generates a hard request (for urgent service) and a soft request (not urgent, but queue can run transfers). The priority assigned to the queues varies depending on the state of the system, but hard requests are prioritized over soft requests, and AHB write requests are prioritized over AHB read requests to allow faster

6.1.2 Address Space

The Address space is mapped as:

MACBase + 0x0000 - MACBase + 0x00FF: MAC setup registers.

MACBase + 0x0100 - MACBase + 0x011F: MAC configuration registers, only first 4 words used.

The RAM blocks are interleaved in the AHB address space. AHB address bits 0 and 1 are byte selects and must be zero for direct access. AHB address bit 2 selects the left or right RAM array, which is the Transmit or Receive array. AHB address bits 3,4, and 5 perform a 1-of-8 column select. Address bit 6 selects the even or odd row address. Address bits 7, 8, 9, and 10 decode the rows. Thus from an AHB addressing perspective, the MAC FIFOs are one large RAM array.

Table 6-1 FIFO RAM Address Map

FIFO RAM Address Map	Usage
0x8001_4000 to 0x8001_47FF	Rx Data
0x8001_4800 to 0x8001_4FFF	Tx Data
0x8001_5000 to 0x8001_503F	Rx Status
0x8001_5040 to 0x8001_507F	Tx Status
0x8001_5080 to 0x8001_50BF	Rx Descriptor
0x8001_50C0 to 0x8001_50FF	Tx Descriptor

The following table defines the FIFO RAM address map as it appears in the address space. Address are in byte units. All data transfers to the FIFO RAM are restricted to words. Accessing the FIFO RAM while the MAC is operating will likely cause a malfunction. There is no arbitration logic between direct AHB access and MAC descriptor processor access.

Table 6-2: FIFO RAM Address Map

FIFO RAM Address Map	Usage
0x8001_4000 to 0x8001_47FF	Rx Data
0x8001_4800 to 0x8001_4FFF	Tx Data
0x8001_5000 to 0x8001_503F	Rx Status
0x8001_5040 to 0x8001_507F	Tx Status
0x8001_5080 to 0x8001_50BF	Rx Descriptor
0x8001_5000 to 0x8001_50FF	Tx Descriptor

6.1.3 Data Encapsulation

In transmission, the MAC automatically prepends the preamble, and computes and appends the FCS. The data after the SFD and before the FCS is supplied by the host as the transmitted data. FCS generation by the MAC may be disabled by setting InhibitCRC bit in the Transmit Frame Descriptor. Refer to Figure 6-2.

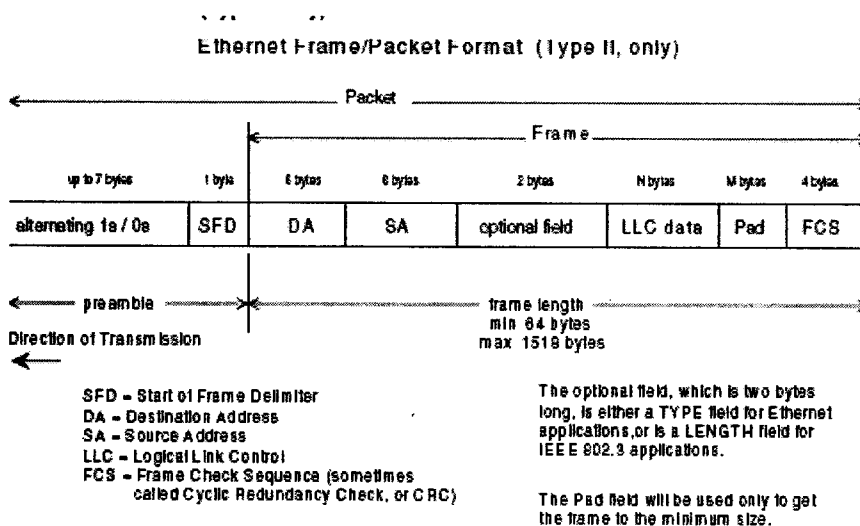


Figure 6.2 Ethernet Frame/Packet Format

In the receiver, the MAC detects the preamble and locks onto the embedded clock. The MAC performs destination address filtering (individual, group, broadcast, promiscuous) on the DA. The MAC engine computes the correct FCS, and reports if the received FCS is "good" or "bad". The data after the SFD and before the FCS is supplied to the

host as the received data. The received FCS may also be passed to the host by setting RXCtl.BCRC.

6.2 Packet Transmission Process

This section explains the complete packet transmission process as seen on the Ethernet line. This process includes: carrier deference, back-off, packet transmission, transmission of EOF, and SQE test. Refer to Figure 6-3.

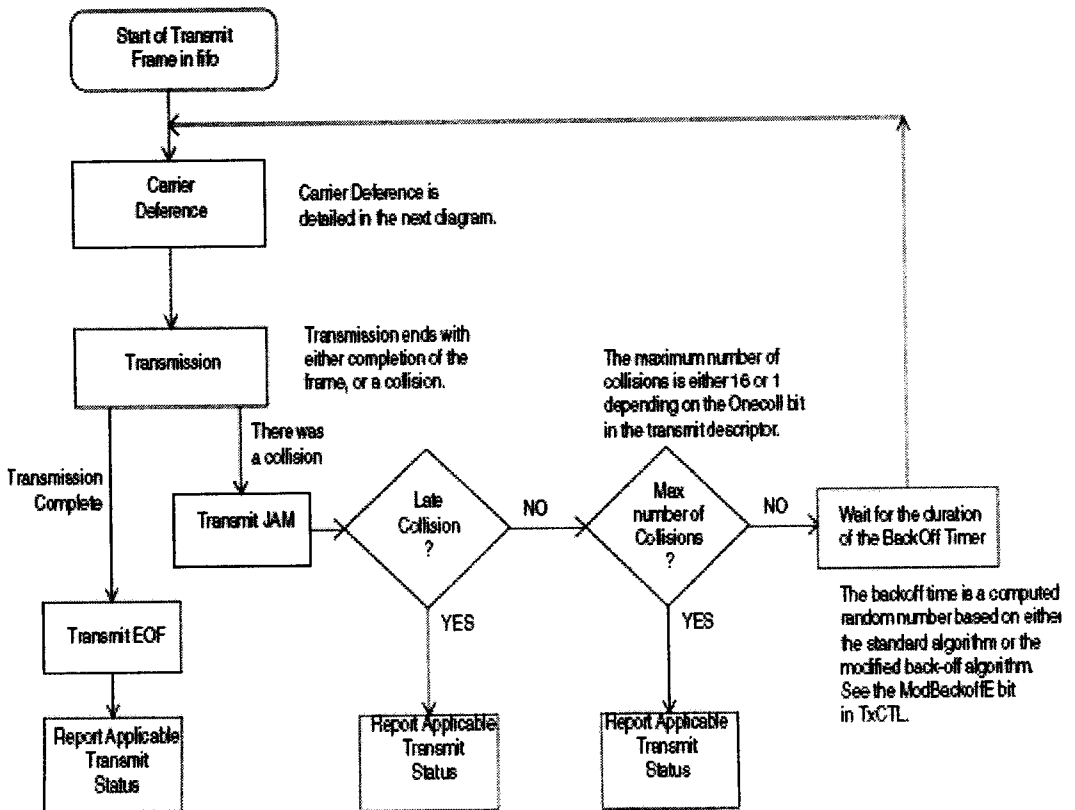


Figure 6.2 Packet transmission status

The Ethernet/ISO/IEC 8802-3 topology is a single shared medium with several stations. Only one station can transmit at a time. The access method is called Carrier Sense Multiple Access with Collision Detection (CSMA/CD). This method is a "listen before talk" mechanism that has an added feature to end transmissions when two, or more, stations start transmissions at nearly the same time.

The CSMA portion of this method provides collision avoidance. Each station monitors its receiver for carrier activity. When activity is detected, the medium is busy, and the MAC defers (waits) until the medium no longer has a carrier.

6.3 Transmit Back- Off

Refer to Figure 6-3. Once transmission is started, either the transmission is completed, or there is a collision. There are two kinds of collision: normal collision (one that occurs within the first 512 bits of the packet) and late collision (one that occurs after the first 512 bits). In either collision type, the MAC engine always sends a 32 bit jam sequence, and stops transmission.

After a normal collision and the jam, transmission is stopped, or "backed-off". The MAC attempts transmission again according to one of two algorithms. The ISO/IEC standard algorithm or a modified back-off algorithm may be used, and the host chooses which algorithm through the ModBackoffE control bit (TXCtI). The standard algorithm from ISO/IEC paragraph 4.2.3.2.5 is called the "truncated binary exponential backoff" and is shown below:

$$0 \leq r \leq 2^k$$

where r is a random integer for the number of slot times the MAC waits before attempting another transmission, and a slot time is time of 512 bits, $k = \text{minimum}(n, 10)$ and n is the n th retransmission attempt. The modified back off algorithm uses delays longer than the ISO/IEC standard after each of the first three transmit collisions as shown below:

$$0 \leq r \leq 2^k$$

Where $k = \min(n, 10)$ but less than 3, and n is the n th retransmission attempt

The advantage of the modified algorithm over the standard algorithm is that the modification reduces the possibility of multiple collisions on any transmission attempt. The disadvantage is that the modification extends the maximum time needed to acquire access to the medium.

The host may choose to disable the back-off algorithm altogether. This is done through the control bit DisableBackoff (TestCtl). When set, the MAC transmitter waits for the Inter Frame Gap time before starting transmission. There is no back-off algorithm employed. When clear, the MAC uses either the standard or the modified algorithm.

6.3.1 Transmission

After the transmission has passed the time for a normal collision (512 bits), then transmission is either completed, or aborted due to a late collision. For a late collision, the transmitter sends the 32 bit jam sequence, but does not back-off and try again. When a late collision occurs, Out-of-wdw collision (XStatQ) is set. A late collision is not retried, because the first 64 bytes of the FIFO are freed after the normal collision window, and will likely be refilled by a following packet. Driver intervention is needed to reconstruct the FIFO data.

6.3.2 Bit Order

In compliance with ISO/IEC 8802-3 section 3.3, each byte is transmitted low order bit first, except for the CRC, as noted in "The FCS Field" on page 126.

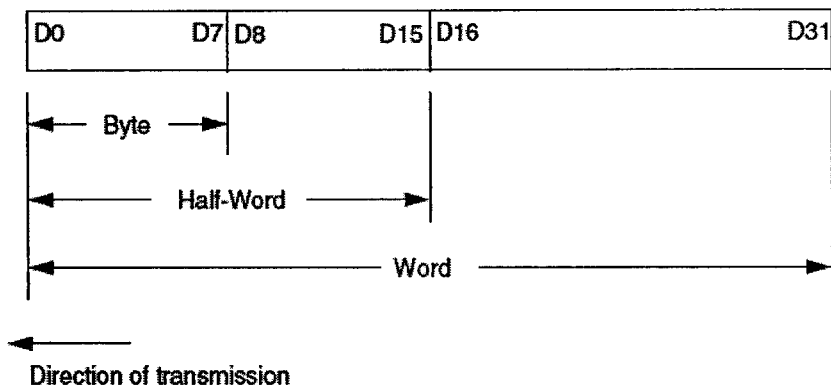


Figure 6-4. Data Bit Transmission Order

6.3.3 Flow Control

The MAC provides special support for flow control by the transmission and reception of pause frames. A pause frame is a specific format of a MAC control frame that defines an amount of time for a transmitter to stop sending frames. Sending pause frames thereby reduces the amount of data sent by the remote station.

6.3.4 Receive Flow Control

The MAC can detect receive pause frames and automatically stop the transmitter for the appropriate period of time. To be interpreted as a pause frame the following conditions must be met:

- Destination address accepted by one of the first two individual address filters, with the appropriate RXctl.RxFCE bit set.
- The Type field must match that programmed in the Flow Control Format register.
- The next two bytes of the frame (MAC Control Opcode) must equal 0x0001.
- The frame must be of legal length with a good CRC.

If accepted as a pause frame, the pause time field is transferred to the Flow Control Timer register. The pause frame may be optionally passed on to the Host or

discarded by the MAC. Once the Flow Control Timer is set to a nonzero value, no new transmit frames are started, until the count reaches zero.

6.3.5 Transmit Flow Control

When receive congestion is detected, the driver may want to transmit a pause frame to the remote station to create time for the local receiver to free resources. As there may be many frames queued in the transmitter, and there is a chance that the local transmitter is itself being paused, an alternative method is provided to allow a pause frame to be transmitted. Setting the Send Pause bit in the Transmit Control register causes a pause frame to be transmitted at the earliest opportunity. This occurs either immediately, or following the completion of the current transmit frame. If the local transmitter is paused, the pause frame will still be sent, and the pause timer will still be decremented during the frame transmission.

To comply with the standard, pause frames should only be sent on full duplex links. The MAC does not enforce this, it is left to the driver. If a pause frame is sent on a half duplex link, it is subject to the normal half duplex collisions rules and retry attempts.

The format of a transmit pause frame is:

Bytes 17-18 - Pause time - this is defined in the Flow Control Format register

Once the Host sets the Send Pause bit in TXCtI, it will remain set until the pause frame starts transmission. Then the Send Pause clears and the Pause Busy bit is set and remains set until the transmission is complete. No end of frame status is generated for pause frames.

Along with Ethernet port, the board has USB ports also helping for better communication. The following chapter deals with the USB connectivity of the board.

CHAPTER 7

UNIVERSAL SERIAL BUS HOST CONTROLLER

7. UNIVERSAL SERIAL BUS HOST CONTROLLER

7.1 INTRODUCTION

The EP9302 Universal Serial Bus (USB) Host Controller enables communication to USB 2.0 low-speed (1.2 Mbps) and full-speed (12 Mbps) devices. The controller supports two root hub ports and complies with the Open Host Controller Interface (OpenHCI) specification, version 1.0a. (For additional information, see "Reference Documents", item 9, on page 5.)

7.1.1 FEATURES

The features of the USB Host Controller are:

- Open Host Controller Interface Specification (OpenHCI) Rev 1.0 compliant.
- Universal Serial Bus Specification Rev. 2.0 compliant.
- Support for both low speed and full speed USB devices.
- Root Hub has two downstream ports
- Master and Slave AHB interfaces
- USB Host test register block for test and software use.
- DMA functionality

The USB Host Controller is partitioned into the key sub blocks as indicated in Figure 8-7.

7.2 OVERVIEW

Figure 8-1 shows four main focus areas of a USB system. These areas are:

- Client Software/USB Driver
- Host Controller Driver (HCD)
- Host Controller (HC)
- USB Device.

The Client Software/USB Device and Host Controller Driver are implemented in software. The Host Controller and USB Device are implemented in hardware. OpenHCI specifies the interface between the Host Controller Driver and the Host Controller and describes the fundamental operation of each.

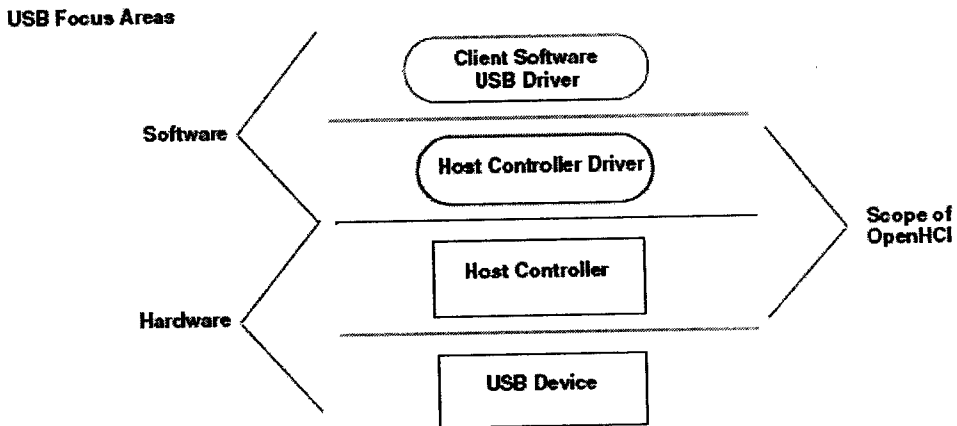


Figure 7.1 USB Focus areas

The Host Controller Driver and Host Controller work in tandem to transfer data between client software and a USB device. Data is translated from shared-memory data structures at the client software end to USB signal protocols at the USB device end, and vice-versa.

7.2.1 Data Transfer Types

There are four data transfer types defined in USB. Each type is optimized to match the service requirements between the client software and the USB device. The four types are:

- Interrupt Transfers - Small data transfers used to communicate information from the USB device to the client software. The Host Controller Driver polls the USB device by issuing tokens to the device at a periodic interval sufficient for the requirements of the device.

- Isochronous Transfers - Periodic data transfers with a constant data rate.

Data transfers are correlated in time between the sender and receiver.

- Control Transfers - Nonperiodic data transfers used to communicate configuration/command/status type information between client software and the USB device.

In OpenHCI the data transfer types are classified into two categories: periodic and nonperiodic. Periodic transfers are interrupt and isochronous since they are scheduled to run at periodic intervals. Nonperiodic transfers are control and bulk since they are not scheduled to run at any specific time, but rather on a time-available basis.

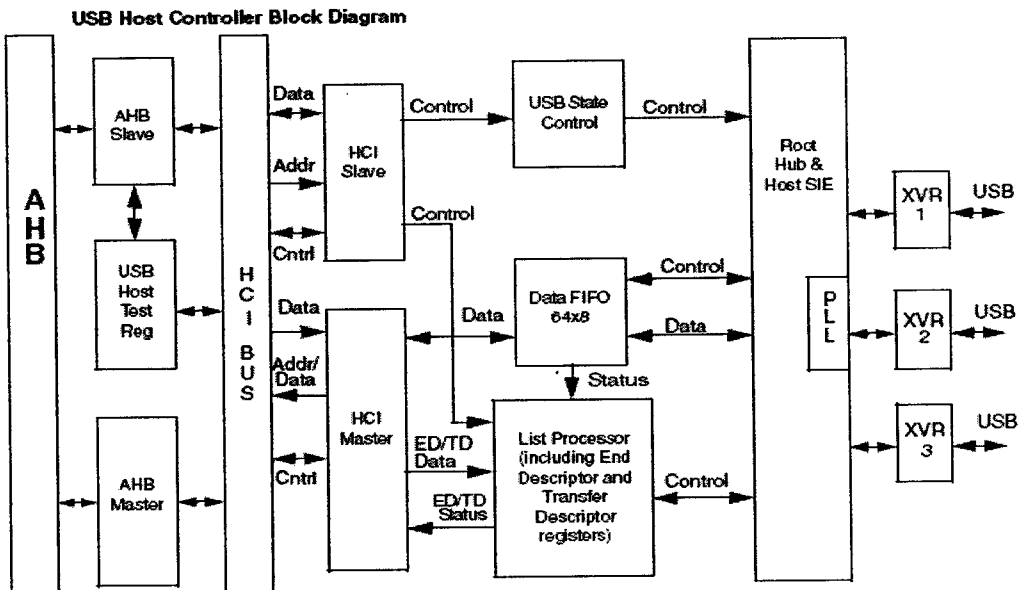


Figure7.2 USB Host controller

7.2.2 USB Host Controller

Blocks 8.2.5.1 AHB Slave

This block allows access to the OHCI operational registers from/to the AHB via the HCI Bus.

7.2.3 AHB Master

This block enables the USB Host Controller to be an AHB Master peripheral and interfaces with the HCI Master block via the HCI Bus.

The AHB Master includes a Data FIFO which will use a 44x37 bit Data FIFO. 32-bit data, 4-bit HCI_MBeN[3:0] (byte lane enables) and HCI_MWBstOnN (burst on) make up the width of the Data FIFO.

7.2.4 HCI Slave Block

This block contains the OHCI operational registers, which are programmed by the Host Controller Driver (HCD).

7.2.5 HCI Master Block

The HCI Master Block handles read/write requests to system memory that are initiated by the List Processor while the Host Controller (HC) is in the operational state and is processing the lists queued in by HCD. It generates the addresses for all the memory accesses, which is the DMA functionality. The major tasks handled by this block are:

- Fetching Endpoint Descriptors (ED) and Transfer Descriptors (TD)
- Read/Write endpoint data from/to system memory
- Accessing HC Communication Area (HCCA)
- Write Status and Retire TDs

7.2.6 USB State Control

This block implements:

- The USB operational states of the Host Controller, as defined in the OHCI Specification.
- It generates SOF tokens every 1 ms
- It triggers the List Processor while HC is in the operational states.

7.2.7 Data FIFO

This block contains a 64x8 FIFO to store the data returned by endpoints on IN tokens, and the data to be sent to the endpoints on OUT Tokens. The FIFO is used as a buffer in case the HC does not get timely access to the host bus.

7.2.8 List Processor

The List Processor processes the lists scheduled by HCD according to the priority set in the operational registers.

7.2.9 Root Hub and Host SIE

The Root Hub propagates Reset and Resume to downstream ports and handles port connect and disconnect. The Host Serial Interface Engine (HSIE) converts parallel to serial, serial to parallel, Non-Return to Zero Interface (NRZI) encoding/decoding and manages USB serial protocol.

CHAPTER 8

SDRAM, SYNCROM, AND SYNC FLASH CONTROLLER

8.SDRAM, SyncROM, and Sync FLASH Controller

8.1 INTRODUCTION

The SDRAM controller provides a high speed memory interface to a wide variety of Synchronous Memory devices, including SDRAMs, Synchronous FLASH, and Synchronous ROMs.

The Key Features of this block are as follows:

- Includes special configuration bits for Synchronous ROM operation.
- Data is transferred between the controller and the SDRAM in quad word bursts. Longer transfers within the same page are concatenated, forming a seamless burst.
- 16-bit data bus.
- SDRAM contents are preserved over a "soft" reset.
- Power saving Synchronous Memory CKE and external clock modes

8.1.1 Booting (from SROM or SyncFLASH)

If the system is booting up using a Synchronous ROM (on SDRAMDevCfg3 register), then a short configuration sequence is activated before releasing the processor from reset. This ensures that a known configuration (RAS=2, CAS=5, and Burst Length=4) is set for whatever device may be attached, as different SROM's default to different Burst Lengths. It is not possible to reconfigure other SDRAM memory banks when running code from SDRAM. Attempting to do this may cause the system to lock-up. It is advised that the boot code copy the SDRAM configuration code to some non-SDRAM memory space, and then set up the SDRAM from that space. A similar automatic boot-up sequence will be initiated when booting from Synchronous FLASH, (WBL=1, CAS=3, Burst Length=4).

Table 8-1: Boot Device Selection

Boot modes	CSn7	CSn6	ASDO	EECLK
8-bit ROM	0	0	0	0
16-bit ROM	0	1	0	0
16-bit SFLASH (Initializes Mode	0	0	1	0
16-bit SRAM (Initializes Mode	0	1	1	0

The following power-up sequence is executed by an internal state machine after power on reset:

1. Power is applied to the circuit with inputs CKE and DQM pulled high so that they rise with the supply VDD and VDDQ.
2. Following power-up, the processor is held in the reset state while the clock runs. The Command pins are put in the NOP condition for 200 is by setting both the Initialize and MRS bits to 1, while the SRAM device is clocked.
3. The default settings are written to the Mode register by setting the Initialize and MRS bits to 0, 1 respectively and reading the appropriate address.
4. After 3 clock cycles from the mode register set cycle, the device is ready for power-up, and all data outputs are in a high impedance state. The processor is released from the reset state.

8.1.2 Address Pin Usage

Each of the four SDRAMDevCfg domains can be fitted with a variety of device types, provided the total capacitance on any address/control/data line does not exceed the specified operating limit. Because of the row/column/bank architecture of Synchronous Memory devices, the mapping of these memories into the EP9301 memory space is not always obvious. Typically, the memory in a SDRAM device will not appear continuous to the EP9301 chip. For example, a 32-Mbyte SDRAM device may be visible as eight 4-Mbyte blocks. In order to understand the reason for this non-continuous memory it is necessary to understand the address pin usage of the EP9301 device. The top row in this table shows the address lines connected to the memory device. The remaining rows show how the device's linear address space is mapped into the

SDRAM address lines. For each memory device configuration, that is, 16-bit wide SDRAM, SROM etc., there is a row and column line. These lines show the addresses presented to the Synchronous Memory device for row and column access. By taking the number of row and columns in a Synchronous memory device (available in the device datasheet), the actual address lines used in addressing the device can be determined. Because some address lines are not used, the memory appears as non-continuous. By using address bits A26 and A27 as the Bank control pins, the memory map can look the same whether a FLASH device or SROM device is attached. This also helps reduce power consumption by ensuring that multiple banks within a device need not always be activated.

8.1.3 Synchronous FLASH Programming

The Synchronous FLASH Command Register enables erasing and writing of Synchronous FLASH attached to the system. It operates in a similar way to programming the Mode register in that once the LCR bit is set, subsequent read addresses are placed onto the Synchronous Address bus and act as the command word. Thus, the address must match the command word specified in the device datasheet. This cycle happens immediately before the usual ACTIVE command in a synchronous instruction sequence and can only be delayed by NOP cycles. Note the following about Synchronous FLASH devices

- They cannot be written in bursts, but only one word at a time. Hence the WBL bit is set to 1, allowing burst Reads and Single Writes. When in this mode, no Auto Refresh cycle will occur to this Chip Select, as it will be assumed that the device is a Synchronous FLASH device
- Synchronous FLASH can be set by either programming the Mode Register with a known configuration mode before releasing the processor from reset, or by using the contents of its NonVolatileMODE register, (which must have been previously set).

CHAPTER 9

GPIO INTERFACE

9.GPIO INTERFACE

9.1 INTRODUCTION

The General Purpose Input/Output (GPIO) is an Advanced Peripheral Bus (APB) slave module. The GPIO block is the primary controller for the **EGPIO**, **RDLED**, **GRLED**, **EECLK**, and **EEDAT** pins.

The GPIO block has seven ports, named Port A through Port C and Port E through Port H. Ports C, E, G, and H are standard GPIO ports. Ports A, B, and F are enhanced GPIO ports.

GPIO ports control up to eight individual pins. Each port has an 8-bit data register and an 8-bit data direction register. The EGPIO ports each have additional 8-bit registers for interrupt configuration and status. The control of an individual pin is determined in a bit-slice fashion across all registers for that port; only a single bit at a particular index from each register affects or is affected by that pin.

Port sizes are as follows:

Port A - 8 bits

Port B - 8 bits

Port C - 1 bits

Port E - 2 bits

Port F - 3 bits

Port G - 2 bits

Port H - 4 bits

System Level GPIO Connectivity

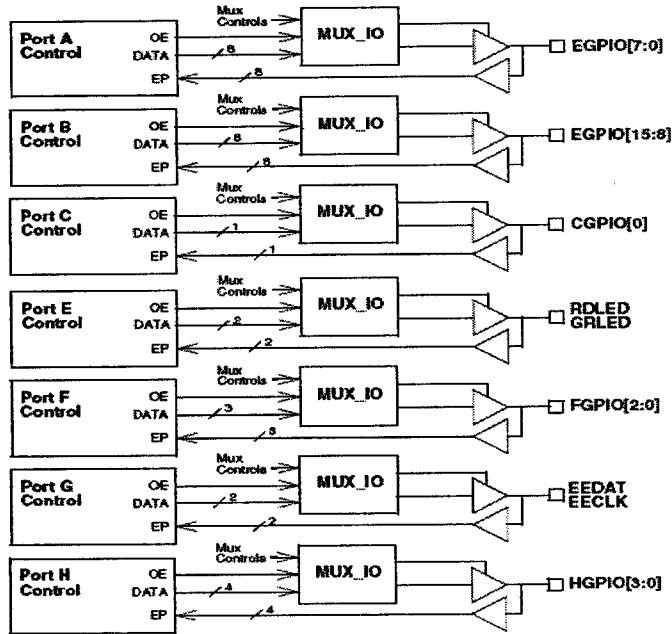


Figure 9.1 System Level GPIO Set Up

9.1.1 Memory Map

The GPIO base address is 0x8084_0000. All registers are 8 bits wide and are aligned on word boundaries. For all registers, the upper 24 bits are not modified when written and always read zeros.

9.1.2 Functional Description

Each port has an 8-bit data register and an 8-bit direction register. The data direction register controls whether each individual GPIO pin is an input or output. Writing to a data register only affects the pins that are configured as outputs. Reading a data register returns the value on the corresponding GPIO pins.

Ports A, B, and F also provide interrupt capability. The 16 interrupt sources from Ports A and B are combined into a single signal **GPIOINTR** which is connected to the system

interrupt controller. All three individual interrupt signals on Port F are available to the system interrupt controller as **GPIO0INTR** through **GPIO2INTR**.

The interrupt properties of each of the 19 GPIO pins on ports A, B, and F are individually configurable. Each interrupt can be either high or low level sensitive or either positive or negative edge triggered. It is also possible to enable debouncing on the Port A, B, and F interrupts. Debouncing is implemented using a 2-bit shift register clocked by a 128 Hz clock.

There are seven additional registers for ports A, B and F:

- *GPIO Interrupt Enable* registers (GPIOAIntEn, GPIOBIntEn, GPIOFIntEn) control which bits are to be configured as interrupts. Setting a bit in this register configures the corresponding pin as an interrupt input.
- *GPIO Interrupt Type 1* registers (GPIOAIntType1, GPIOBIntType1, GPIOFIntType1) determines interrupt type. Setting a bit in this register configures the corresponding interrupt as edge sensitive; clearing it makes it level sensitive.
- *GPIO Interrupt Type 2* registers (GPIOAIntType2, GPIOBIntType2, GPIOFIntType2) determines interrupt polarity. Setting a bit in this register configures the corresponding interrupt as rising edge or high level sensitive; clearing it configures the interrupt as falling edge or low level sensitive.
- *GPIO End-Of-Interrupt* registers (GPIOAEOI, GPIOBEOI, GPIOFEOI) are used to clear specific bits in the interrupt Status Register. Writing a one to a bit will clear the corresponding interrupt; writing a zero has no effect.
- *GPIO Debounce* registers (GPIOADB, GPIOBDB, GPIOFDB) enable debouncing of specific interrupts signals.
- *Interrupt Status* registers (IntStsA, IntStsB, IntStsF) provide the status of any pending unmasked interrupt.
- *Raw Interrupt Status* registers (RawIntStsA, RawIntStsB, RawIntStsF) provide the status of any pending interrupt regardless of masking.

In order to stop any spurious interrupts that may occur during the programming of the GPIOxINTTYPE_x registers, the following sequence should be observed:

1. Disable interrupt by writing to GPIO Interrupt Enable register.
2. Set interrupt type by writing GPIOxINTTYPE1/2 register.
3. Clear interrupt by writing to GPIOxEOI register.
4. Enable interrupt by writing to GPIO Interrupt Enable register.

9.1.3 GPIO Pin Map

All GPIO signals are mapped to device pins. The following table shows how the GPIO ports map to EP9301 pins.

Table 9-1: GPIO Port to Pin Map

Pin Name	Default Function
EGP10[7:0]	Port A
EGP10[15:8]	Port B
GRLED	Port E[0]
RDLED	Port E[1]
EECLK	PortG[0]
EEDAT	PortG[1]

GRLED is the Green LED pin.

RDLED is the Red LED pin.

EECLK is the EEPROM clock pin.

EEDAT is the EEPROM data pin.

When the GPIO port signals are not explicitly mapped to a device pin, as defined in Table 21-1, the inputs will continue to monitor the pin while outputs are disconnected.

Another level of functional muxing is applied to several EGPIO pins. The Syscon DeviceCfg register bits RonG, MonG, HC1 EN, and map different functionality to the EGPIO pins:

- MonG maps **RI** (modem Ring Indicator) onto **EGP10[0]**.
- RonG maps **CLK32K**, the 32 KHz clock monitor output for RTC calibration, onto **EGPIO[1]**.
- HC1 EN maps the synchronous HDLC clock onto **EGPIO[3]**.

Some GPIO signals are used as inputs by other functional blocks. **EGPIO[2:1]** are routed to the DMA controller to allow for external DMA requests.

CHAPTER 10

LINUX

10.LINUX

10.1 INTRODUCTION

Linux is interchangeably used in reference to the Linux kernel, a Linux system, or a Linux distribution. The broadness of the term plays in favor of the adoption of Linux. Strictly speaking, Linux refers to the kernel maintained by Linus Torvalds and distributed under the same name through the main repository and various mirror sites. The kernel provides the core system facilities. It may not be the first software to run on the system, as a bootloader may have preceded it, but once it is running, it is never swapped out or removed from control until the system is shut down. In effect, it controls all hardware and provides higher-level abstractions such as processes, sockets, and files to the different software running on the system. As the kernel is constantly updated, a numbering scheme is used to identify a certain release. This numbering scheme uses three numbers separated by dots to identify the releases. The first two numbers designate the version, and the third designates the release.

10.2 EMBEDDED LINUX

There is no such thing as an embedded version of the kernel distributed by Linus. This doesn't mean the kernel can't be embedded. It only means you do not need a special kernel to create an embedded system. An embedded Linux system simply designates an embedded system based on the Linux kernel and does not imply the use of any specific library or user tools with this kernel. An embedded Linux distribution may include: a development framework for embedded linux systems, various software applications tailored for usage in an embedded system, or both. Development framework distributions include various development tools that facilitate the development of embedded systems. This may include special source browsers, cross-compilers, debuggers, project management software, boot image builders, and so on. These distributions are meant to be installed on the development host.

10.3 REASONS FOR CHOOSING LINUX

There are various motivations for choosing Linux over a traditional embedded OS.

10.3.1 Quality and reliability of code

Quality and reliability are subjective measures of the level of confidence in the code. Although an exact definition of quality code would be hard to obtain, there are properties common programmers come to expect from such code:

Modularity and structure

Each separate functionality should be found in a separate module, and the file layout of the project should reflect this. Within each module, complex functionality is subdivided in an adequate number of independent functions.

Ease of fixing

The code should be (more or less) easy to fix for whoever understands its internals.

Extensibility

Adding features to the code should be fairly straightforward. In case structural or logical modifications are needed, they should be easy to identify.

Configurability

It should be possible to select which features from the code should be part of the final application. This selection should be easy to carry out.

The properties expected from reliable code are:

Predictability

Upon execution, the program's behavior is supposed to be within a defined framework and should not become erratic.

Error recovery

In case a problematic situation occurs, it is expected that the program will take steps to recover from the problem and alert the proper authorities, usually the system administrator, with a meaningful diagnostic message.

Longevity

The program will run unassisted for long periods of time and will conserve its integrity regardless of the situations it encounters.

10.3.2 Availability of code

Code availability relates to the fact that Linux's source code and all build tools are available without any access restrictions. The most important Linux components, including the kernel itself, are distributed under the GNU General Public License (GPL). Access to these components' source code is therefore compulsory. When source access problems arise, the open source and free software community seeks to replace the "faulty" software with an open source version providing similar capabilities. This contrasts with traditional embedded OSes, where the source code isn't available or must be purchased for very large sums of money. The advantages of having the code available are the possibility of fixing the code without exterior help and the capability of digging into the code to understand its operation.

10.3.3 Hardware support

Broad hardware support means that Linux supports different types of hardware platforms and devices. Although a number of vendors still do not provide Linux drivers, considerable progress has been made and more is expected. Broad hardware support also means that Linux runs on dozens of different hardware architectures. No other OS provides this level of portability

10.3.4 Communication protocol and software standards

Linux also provides broad communication protocol and software standards support. This is where the spirit of the free software and open source community can most be felt. As with application needs, it is likely that someone has encountered the same problems as you in similar circumstances.

The development and support mailing lists are the best place to find this community support. With Linux, an email to the appropriate mailing list will often get you straight to the person who wrote the software. Pointing out a bug and obtaining a fix or suggestions is thereafter a rapid process.

Licensing

Licensing enables programmers to do with Linux what they could only dream of doing with proprietary software. In essence, you can use, modify, and redistribute the software with only the restriction of providing the same rights to your recipients.

Vendor independence

Vendor independence, as was demonstrated by the polls presented earlier, means that you do not need to rely on any sole vendor to get Linux or to use it.

All development tools and OS components are available free of charge, and the licenses under which they are distributed prevent the collection of any royalties on these core components.

10.3.5 Players of the Embedded Linux Scene

Unlike proprietary Operating Systems, Linux is not controlled by a single authority who dictates its future, its philosophy, and its adoption of one technology or another.

Free software and open source community

The free software and open source community is the basis of all Linux development and is the most important player in the embedded Linux arena. It is made up of all the developers who enhance, maintain, and support the various software components that make up a Linux system. There is no central authority within this group. Rather, there is a loosely tied group of independent individuals, each with his specialty.

10.4 Hosts : Linux Workstation

This is the most common type of development host for embedded Linux systems.

10.5 Host/Target Development Setups :Removable Storage Setup

In this setup, there are no direct physical links between the host and the target. Instead, a storage device is written by the host, is then transferred into the target, and is used to boot the device. Figure 10-1 illustrates this setup.

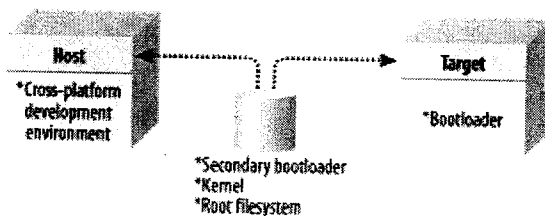


Figure 10-1. Host/target removable storage setup

As with the previous setup, the host contains the cross-platform development environment. The target, however, contains only a minimal bootloader. The rest of the

components are stored on a removable storage media, such as a Compact Flash IDE device or any other type of drive, which is programmed on the host and loaded by the target's minimal bootloader upon startup.

It is possible, in fact, that the target may not contain any form of persistent storage at all. Instead of a fixed flash chip, for instance, the target could contain a socket where a flash chip could be easily inserted and removed. The chip would be programmed by a flash programmer on the host and inserted into the socket in the target for normal operation. This setup is mostly popular during the initial phases of embedded system development. You may find it more practical to move on to a linked setup once the initial development phase is over, so you can avoid the need to physically transfer a storage device between the target and the host every time a change has to be made to the kernel or the root file system.

CHAPTER 11

BUILDING UP OF KERNEL WITH TOOLCHAIN

11.BUILDING UP OF KERNEL WITH TOOLCHAIN

11.1 LINUX KERNEL.

The setup of the kernel headers is the first step in building the toolchain. In this case, we are using kernel Version 2.4.18, Having selected a kernel, the first thing you need to do is download a copy of that kernel into the directory in which you have chosen to store kernels.

We use the following command:

```
$ tar xvzf linux-2.4.18.tar.gz
```

With the kernel now extracted, we proceed to configuring it:

11.1.1 Configuring the Kernel

Configuration is the initial step in the build of a kernel for our target. There are many ways to configure the kernel, and there are many options from which to choose .

Regardless of the configuration method we use or the actual configuration options we choose, the kernel will generate a *.config* file at the end of the configuration and will generate a number of symbolic links and file headers that will be used by the rest of the build.

The kernel supports four main configuration methods:

`make config`

Provides a command-line interface where you are asked about each option one by one. If a *.config* configuration file is already present, it uses that file to set the default values of the options it asks you to set.

make oldconfig

Feeds config with a an existing *.config* configuration file, and prompts you to configure only those options you had not previously configured. This contrasts with `make config`, which asks you about all options, even those you may have previously configured.

make menuconfig

Displays a curses-based terminal configuration menu. If a *.config* file is present, it uses it to set default values, as with `make config`.

make xconfig

Displays a Tk-based X Window configuration menu. If a *.config* file is present, it uses it to set default values, as with `make config` and `make menuconfig`.

Any of these can be used to configure the kernel. They all generate a *.config* file in the root directory of the kernel sources.

```
$ cd linux-2.4.18
```

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

Compiling the kernel involves a number of steps: building the kernel dependencies, building the kernel image, and building the kernel modules.

11.1.2 Building Dependencies

Most files in the kernel's sources depend on a number of header files. To build the kernel adequately, the kernel's Makefiles need to know about these dependencies. For each subdirectory in the kernel tree, a hidden *.depend* file is created during the dependencies build. This contains the list of header files that each file in the directory depends on. As

with other software that relies on make, only the files that depend on a header that changed since the last build will need to be recompiled when a kernel is rebuilt.

From the kernel source's root directory, the following command builds the kernel's dependencies:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- clean dep
```

11.1.3 Building the Kernel

With the dependencies built, we can now compile the kernel image:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- zImage
```

The zImage target instructs the Makefile to build a kernel image that is compressed using the gzip algorithm

11.2 BUILDING THE MODULES

With the kernel image properly built, we can now build the kernel modules:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- modules
```

The duration of this stage depends largely on the number of kernel options we chose to build as modules instead of having linked as part of the main kernel image.

The Linux operating system cannot work independently on the board. It needs to build a tool chain, containing of various components that are needed for the operating system. First the kernel needs to be compiled and built. Then a tool chain must be built over it. Setting up a tool chain creates a build environment on a host machine for compiling the kernel and those applications that are to be executed on the target -- this is because the target hardware may not have binary execution-level compatibility with the host.

A tool chain consists of a set of components used for compiling, assembling, and linking the kernel and applications. These components include:

- **Binutils** -- A collection of utilities for manipulating binary files. They include utilities like `ar`, `as`, `objdump`, `objcopy`, and so on.
- **Gcc** -- The GNU C compiler.
- **Glibc** -- The C library that all user applications will link to. The kernel and other things that avoid using any C library functions can be compiled without it.

Building a tool chain establishes a cross-compiler environment. A *native compiler* compiles instructions for the same sort of processor as the one it is running on. A *cross-compiler* runs on one type of processor, but compiles instructions for another. Setting up a cross-compiler tool chain from scratch is not an easy task: it involves downloading the sources, patching, configuring, compiling, setting up headers, installation, and much, much more. In addition, the memory and hard disk requirements for such an exhaustive build procedure are huge.

11.3 GNU TOOLCHAIN BASICS

Configuring and building an appropriate GNU toolchain is a complex and delicate operation that requires a good understanding of the dependencies between the different software packages and their respective roles. This knowledge is required, because the GNU toolchain components are developed and released independently from one another.

11.3.1 Component versions

The first step in building the toolchain is selecting the component versions we will use. This involves selecting a `binutils` version, a `gcc` version, and a `glibc` version. Because these packages are maintained and released independently from one another, not all versions of one package will build properly when combined with different versions of the other packages.

To select the appropriate versions, you have to test a combination tailored to your host and target.

Host	Target	Kernel	binutils	gcc	glibc
i386	ARM	2.4.1-rmk1	2.10.1	2.95.3	2.1.3

11.3.2 Building the Toolchain

In outline what you need to do is:

- Compile binutils first;
- Then compile gcc;
- Produce gLibc last.

11.3.2.1 Binutils

Binutils is a collection of utilities, for doing things with binary files.

11.3.2.2 Binutils components

addr2line

Translates program addresses into file names and line numbers. Given an address and an executable, it uses the debugging information in the executable to figure out which file name and line number are associated with a given address.

ar

The GNU **ar** program creates, modifies, and extracts from archives. An archive is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called members of the archive).

as

GNU **as** is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the

others, including object file formats, most assembler directives (often called pseudo-ops) and assembler syntax.

as is primarily intended to assemble the output of the GNU C compiler **gcc** for use by the linker **ld**. Nevertheless, we've tried to make **as** assemble correctly everything that the native assembler would. This doesn't mean **as** always uses the same syntax as another assembler for the same architecture.

c++filt

The **c++filt** program does the inverse mapping: it decodes (demangles) low-level names into user-level names so that the linker can keep these overloaded functions from clashing.

gasp

Gnu Assembler Macro Preprocessor.

ld

The GNU linker **ld** combines a number of object and archive files, relocates their data and ties up symbol references. Often the last step in building a new compiled program to run is a call to **ld**.

nm

GNU **nm** lists the symbols from object files.

objcopy

The GNU **objcopy** utility copies the contents of an object file to another. **objcopy** uses the GNU BFD library to read and write the object files. It can write the destination object file in a format different from that of the source object file. The exact behavior of **objcopy** is controlled by command-line options.

objdump

objdump displays information about one or more object files. The options control what particular information to display.

ranlib

ranlib generates an index to the contents of an archive, and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file. You may use ``nm -s'` or ``nm --print-arnmap'` to list this index.

readelf

readelf Interprets headers on elf files.

size

The GNU **size** utility lists the section sizes and the total size for each of the object files objfile in its argument list. By default, one line of output is generated for each object file or each module in an archive.

strings

GNU **strings** prints the printable character sequences that are at least 4 characters long (or the number given with the options below) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.

strip

GNU **strip** discards all symbols from the target object file(s). The list of object files may include archives. At least one object file must be given. **strip** /modifies the files named in its argument, rather than writing modified copies under different names.

11.3.2.3 Unpacking and patching

The first thing you need to build is GNU binutils. 2.10.1 version. We can download the binutils from <ftp://ftp.gnu.org/gnu/binutils/>;

Unpack the archive to /usr/src:

```
cd /usr/src  
tar -xzf ../binutils-2.10.1.tar.gz
```

As we are building Linux on another machine, we use

```
./configure --target=arm-linux --prefix=/usr
```

Invoke **make** in the binutils directory:

```
make install
```

This should proceed without incident. Now we move on to the compiler.

11.3.3 Gcc

In contrast to the binutils package, the gcc package contains only one utility, the GNU compiler, along with support components such as runtime libraries. We divide the installation of GCC into two steps. First we build the bootstrap compiler, which will support only the C language. Later, once the C library has been compiled, we will recompile gcc with full C++ support. We start by extracting the gcc package from the archive

```
$ cd ${PRJROOT}/build-tools
```

```
$ tar xvzf gcc-2.95.3.tar.gz
```

This will create a directory called *gcc-2.95.3* with the package's content. We can now proceed to the configuration of the build in the directory we had prepared for the

```
$ cd build-boot-gcc
```

```
$ ../gcc-2.95.3/configure --target=$TARGET --prefix=${PREFIX} \  
> --without-headers --with-newlib --enable-languages=c
```

With the Makefiles ready, we can now build the compiler:

```
$ make all-gcc
```

The compile time for the bootstrap compiler is comparable to that of the binutils. With the compilation complete, we can now install gcc:

```
$ make install-gcc
```

The bootstrap compiler is now installed alongside the binutils. The full compiler is set up after installation of glibc.

11.3.4 C Library Setup(glib C)

The glibc package is made up of a number of libraries and is the most delicate and lengthy package build in our cross-platform development toolchain.

As with the previous packages, we start by extracting the C library from the archive we downloaded earlier:

```
$ cd ${PRJROOT}/build-tools
```

```
$ tar xvzf glibc-2.2.3.tar.gz
```

This will create a directory called *glibc-2.2.3* with the package's content. In addition to extracting the C library, we extract the linuxthreads package in the glibc directory for the reasons stated earlier in the chapter:

```
$ tar -xvzf glibc-linuxthreads-2.2.3.tar.gz --directory=glibc-2.2.3
```

We can now proceed to preparing the build of the C library in the *build-glibc* directory:

```
$ cd build-glibc
```

```
$ CC=i386-linux-gcc ../glibc-2.2.3/configure --host=$TARGET \
```

```
> --prefix="/usr" --enable-add-ons \
```

```
> --with-headers=${TARGET_PREFIX}/include
```

During the actual build of the library, three sets of libraries are built: a shared set, a static set, and a static set with profiling information. If you do not intend to use the profiling version of the library, you may instruct the configuration script not to include it as part of the build process by using the `--disable-profile` option. The same applies to the shared set, which can be disabled using the `--disable-shared` option. If you have chosen not to download the `linuxthreads` package, or the `crypt` package if you were using `glibc 2.1.x`, you may try to compile the C library by removing the `--enable-add-ons` option and adding the `--disable-sanity-checks` option. Otherwise, the configuration script will complain about the missing `linuxthreads`. Note, however, that although `glibc` may build successfully without `linuxthreads`, it is possible that the full compiler's build will fail when including C++ support later.

With the configuration script done, we can now compile `glibc`:

```
$ make
```

The C library is a very large package and its compilation may take several hours, depending on your hardware.

Once the C library is built, we can now install it:

```
$ make install_root=${TARGET_PREFIX} prefix="" install
```

In contrast to the installation of the other packages, the installation of the C library will take some time. It won't take as much time as the compilation, but it may take between 5 and 10 minutes, again depending on your hardware. There is one last step we must carry out to finalize `glibc`'s installation: the configuration of the `libc.so` file. This file is used

during the linking of applications to the C library and is actually a link script. It contains references to the various libraries needed for the real linking. The installation carried out by our `make install` above assumes that the library is being installed on a root filesystem and hence uses absolute pathnames in the `libc.so` link script to reference the libraries. Since we have installed the C library in a nonstandard directory, we must modify the link script so that the linker will use the appropriate libraries. Along with the other components of the C library, the link script has been installed in the ``${TARGET_PREFIX}/lib` directory.

In its original form, `libc.so` looks like this:

```
/* GNU ld script
  Use the shared library, but some functions are only in
  the static library, so try that secondarily. */
GROUP ( /lib/libc.so.6 /lib/libc_nonshared.a )
```

```
$ cd `${TARGET_PREFIX}/lib
$ cp ./libc.so ./libc.so.orig
```

11.3.5 Full Compiler Setup

Since we had already extracted the compiler from its archive we will not need to repeat this step. Overall, the build of the full compiler is much simpler than the build of the bootstrap compiler.

From the `build-tools/build-gcc` directory enter:

```
$ cd `${PRJROOT}/build-tools/build-gcc
$ ../gcc-2.95.3/configure --target=${TARGET} --prefix=${PREFIX} \
> --enable-languages=c,c++
```

With the full compiler properly configured, we can now build it:

\$ make all

This build will take slightly longer than the build of the bootstrap compiler

With the full compiler now built, we can install it:

\$ make install

This will install the full compiler over the bootstrap compiler we had previously installed.

Thus we can build the tool chains required for the Kernel to operate efficiently on the board.

CHAPTER 12

PROCESS OF BUILDING LINUX OPERATING SYSTEM ON ARM PROCESSOR

12.PROCESS OF BUILDING LINUX OPERATING SYSTEM ON ARM PROCESSOR (steps to be followed)

12.1 INTRODUCTION

The source code Linux 2.4 must be rebuilt on the processor core. To rebuild the system from source, an arm-linux and arm-elf tool chain are required. Pre-built tool chains, along with the source required to rebuild them, are available in the files arm-linux-gcc-3.3.tar.bz2 .These tarballs should be unpacked into the root directory of our system, and will place files into "/usr/local/arm/3.3" and "/usr/local/arm/3.2.1-elf" arm-elf-gcc-3.2.1.tar.bz2 The pre-built tool chains run on glibc-2.3.2 based Linux systems

Once installed the tool chains must be made accessible in our search path. The "/usr/local/arm/3.3/bin" and "/usr/local/arm/3.2.1-elf/bin" directories must be in our search path; how this is accomplished is dependent upon the shell you use. For a bourne-like shell (such as bash), use the following:

```
PATH=/usr/local/arm/3.3/bin:/usr/local/arm/3.2.1-elf/bin:$PATH
```

Once fully built, this package requires almost 1GB of disk space and almost 70,000 inodes; If adequate amount of storage is unavailable seemingly strange errors will occur.

12.2 BOOT LOADERS

The boot loader is usually the **first piece of code** that will be executed on any hardware.

The boot loader performs the following types of functions:

- Initialize CPU speed
- Initialize memory, which includes enabling memory banks, initializing memory configuration registers, and so on
- Initialize serial port (if present on the target)
- Enable instruction/data caches
- Set up stack pointer

- Set up parameter area and construct parameter structures and tags (this is an important step, as boot parameters are used by the kernel in identifying root device, page size, memory size and more)
- Perform POST (Power On Self Test) to identify the devices present and to report any problems
- Provide support for suspend/resume for power management
- Jump to start of kernel

In conventional systems like desktops, the bootloader is normally loaded into the MBR (Master Boot Record) or the first sector of the disk where Linux resides. Normally, BIOS will transfer control to the bootloader in the case of desktops or other systems. But in embedded systems we don't have the BIOS. Hence we use RedBoot loader. RedBoot is a complete bootstrap environment for embedded systems. RedBoot allows download and execution of embedded applications via serial or Ethernet ports.

12.2.1 Running Redboot

There is a pre-built RedBoot image for each board type in the "redboot.bin" file of each board specific directory. We use the download application to program this image into the NOR FLASH of the board to be used and to program the ethernet MAC address so that the ethernet interface can be used by RedBoot. Once programmed, this is the sequence to run RedBoot for our board.

- Run "minicom" (a terminal emulation program).
- Set the serial port to 57600, 8-N-1. Also, select the same serial port as used to download RedBoot to the board.
- Make sure that the TEST0 jumper
- Press and release the RESET button.

After this process, the first thing to do is to initialize the NOR FLASH filesystem (this only needs to be done once, unless we wish to destroy the contents of your FLASH).

Type "fis init -f" (Initialize Flash Image System (FIS)) and wait for it to complete.

Then type "fconfig -i". (If the optional flag -i is specified, then the configuration database will be reset to its default state.) This is also needed the first time RedBoot is installed on the target to configure the board.

RedBoot uses TFTP (Trivial File Transfer Protocol) to retrieve images from the network. Therefore, a TFTP server must be running on a machine on our network, and the images to be loaded onto the board must reside on that machine. Once RedBoot (with networking) is running on the board, Linux can be loaded and run. The images to be loaded by RedBoot must be placed into the area used by the TFTP server on your host machine, typically "/tftpboot"

Initially, we type the following.

```
load -r -v -b 0x800000 ramdisk.gz
load -r -v -b 0x80000 zImage
exec -r 0x800000 -s 0x600000
```

This will load the minimal ramdisk image, the Linux kernel, and then start it running. The "-s" argument to "exec" is the size of the ramdisk image; it must be greater than or equal to the size of the ramdisk.gz loaded. If the size specified is too small, the kernel will report "RAMDISK: ran out of compressed data" and then raise an alarm.

While the SDRAM on the boards appears as multiple discontinuous regions of memory in the physical address space, it is mapped into a single contiguous logical region of memory by RedBoot. But, images must be loaded into a physically contiguous region of memory since Linux is unable to correctly handle images that are not physically contiguous. This means there are artificial limitations on the locations to which images

can be loaded. For our board, the memory is segmented four equal sized blocks of 8MB each.

Then we will be able to see "Uncompressing linux.....done, booting the kernel." on the serial port. The LCD will then become the Linux console and the typical startup messages will be displayed there. If there is no LCD so the serial port will be the console).

After a little bit, board will display "Press enter to activate this console" on the LCD and the serial connection. We start by pressing enter. We'll see a welcome banner, followed by a command prompt:

```
BusyBox v1.00 (2004.02.20-18:43+0000) Built-in shell (ash)
```

```
Enter 'help' for a list of built-in commands.
```

```
~ #
```

Type some typical Linux commands, such as ls. Some commands of special interest on the minimal ramdisk are (not all applications will be on the ramdisk for every target):

```
bplay/brec
```

Playback and record wave files.

```
cmix
```

Mixer application for setting audio volumes.

dosfsck

Same as on desktop Linux. Verifies the consistency of a MS-DOS VFAT filesystem.

e2fsck

Same as on desktop Linux. Verifies the consistency of an EXT2 or EXT3 filesystem.

eraseall

This application erases the entire contents of a MTD partition on the NOR FLASH. Care should be taken in using it; if run on /dev/mtd0 the entire FLASH will be erased, including RedBoot!

fdisk

Same as on desktop Linux. A hard drive can be partitioned with this application.

gdbserver

This application runs another application as a subprocess and responds to the GDB debug protocol to allow the subprocess to be debugged.

gpm

Same as the gpm application on a desktop Linux system; it is run by the init script. If a mouse is plugged into the mouse port on the board, it can be used to move a cursor around on the LCD, and to cut and paste text.

mkdosfs

Same as on desktop Linux. A hard drive partition, or a ramdisk, can be formatted with this application using the MS-DOS VFAT filesystem.

mke2fs

Same as on desktop Linux. A hard drive partition, or a ramdisk, can be formatted with this application using the EXT2 or EXT3 filesystems.

CHAPTER 13

USING BOARD PERIPHERALS

