FORCE COMPUTERS
A SOLECTRON COMPANY

VxWorks/Tornado ™

BSP Rel. 3.x

for PPMC-280

Programmer's Guide

P / N 221088 Revision AC
November 2003

Copyright

A SOLECTRON COMPANY

World Wide Web: www.forcecomputers.com
24-hour access to on-line manuals, driver updates, and application notes is provided via SMART, our SolutionsPLUS customer support program that provides current technical and services information.

Headquarters

The Americas

Corporate Headquarters/CA
Force Computers
4211 Starboard Drive
Fremont, CA 94538

Tel.: +1 510 624-8274
Fax: +1 510 445-6007
Email: support@fci.com

Europe

Force Computers GmbH
Lilienthalstr. 15
D-85579 Neubiberg / München
Germany

Tel.: +49 (89) 608 14-0
Fax: +49 (89) 609 77 93
Email: support-de@fci.com

Asia

Force Computers Japan KK
Shibadaimon MF Building 4F
2-1-16 Shiba Daimon
Minato-ku, Tokyo 105-0012 Japan

Tel.: +81 (03) 3437 6221
Fax: +81 (03) 3437 6223
Email: support-de@fci.com

221088 410 000 AC

Contents

3 Dual CPU Configuration

4 Software Basics

5 API Call Reference

A Appendix

Index of Functions

Product Error Report

Tables

Using This Guide

This Programmer's Guide is intended for software developers writing applications to run on PPMC-280. This manual describes board specific information necessary to run VXWorks on PPMC-280.

This guide is to be referenced for PPMC-280 VxWorks BSP Rel. 3.x.

Throughout this guide, it is assumed that you are generally familiar with C programming, VxWorks, and the Tornado Development Environment.

Conventions

Notation	Description
57	All numbers are decimal numbers except when used with the following notations:
0x0000000	Typical notation for hexadecimal numbers (digits are 0 through F), e.g. used for addresses and offsets.
Bold	Character format used to emphasize a word
`Courier`	Character format used for on-screen output, user input/output
Italics	Character format for references and for table and figure descriptions.
Note:	No danger encountered. Pay attention to important information marked using this layout.

Abbreviations

API	Application Program Interface
BRG	Baud Rate Generator
BSP	Board Support Package
CPD	(DMA) Current Descriptor Pointer
CPU	Central Processing Unit
DMA	Dynamic Memory Access

DRAM	Dynamic Random Access Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory
FDP	(DMA) First Descriptor Pointer
FIFO	First In First Out
GPP	General Purpose Port
HDLC	High-level Data Link Control
I^2C	Inter Integrated Circuit
ISR	Interrupt Service Routine
LANHDLC	Local Area Network HDLC
LSB	Least Significant Bit
MAC	Media Access Control
MPSC	Multi Protocol Serial Controller
NIC	Network Interface Card
PCI	Peripheral Connect Interface
RISC	Reduced Instruction Set Computer
RMII	Reduced Media-Independent Interface
RTC	Real Time Clock
SDMA	Serial Dynamic Memory Access
SDRAM	Synchronous DRAM
SFM	Single Frame Mode
SIO	Serial Input Output
UART	Universal Asynchronous Receiver Transmitter
VPD	Vital Product Data
VTS	Validation Test Suite
WRS	Wind River Systems

Revision History

Order Number	Revision	Date	Description
221088 410 000	AA	May 2003	Release 1.0 Release for Tornado 1.0.1
221088 410 000	AB	July 2003	Release 2.0 Release for Tornado 2.2 Editorial Changes Support for two Ethernet ports added Modified section “Ethernet Driver” page 5-86.
221088 410 000	AC	November 2003	Rel. 3.x Release for Tornado 2.2

Other Sources of Information

For further information, refer to the following documents:

Company	Web Address	Document
Marvell Technology Group Ltd.	www.marvell.com	MV-64360 Datasheet
Atmel Corporation	www.Atmel.com	ATMEL AT24C02A Serial EEPROM Datasheet ATMEL AT24C02A and AT24C64A Serial EEPROM
Motorola	www.motorola.com	MPC7447 datasheet and user manual
Maxim Integrated Products	www.maxim-ic.com	MAX6900 Datasheet

In addition, refer to PCI Local Bus Specifications Revision 2.2 for VPD Data Structure.

1

Introduction

System Architecture

The PPMC-280 board supports the following features:

- Motorola PowerPC® 7447 processor (single or dual depending on the variant of the board).
  - (1 GHz core) in a "loosely-coupled symmetric multi-processing (SMP)" environment (applicable for dual CPU variants only).
  - 133 MHz front-side bus
- Marvell MV64360/362 system controller
- Upto 1GB Dual Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM)
  - Up to 133 MHz bus frequency
- PCI2.2 interface
  - Universal signaling
  - 64-bit, 66 MHz
- PCI boot
- Two Gigabit Ethernet ports
  - Accessible through PMC I/O (P4) connector.
- Two RS232 serial ports
  - Accessible through PMC I/O (P4) connector.
- Y2K-compliant Real Time Clock (RTC)
- Serial E2PROMs for board configuration and identification
- Upto 64 MB of User flash
- BIB
- RTC
- Debug support through COP and JTAG ports

The function block diagram is shown in Figure 1.

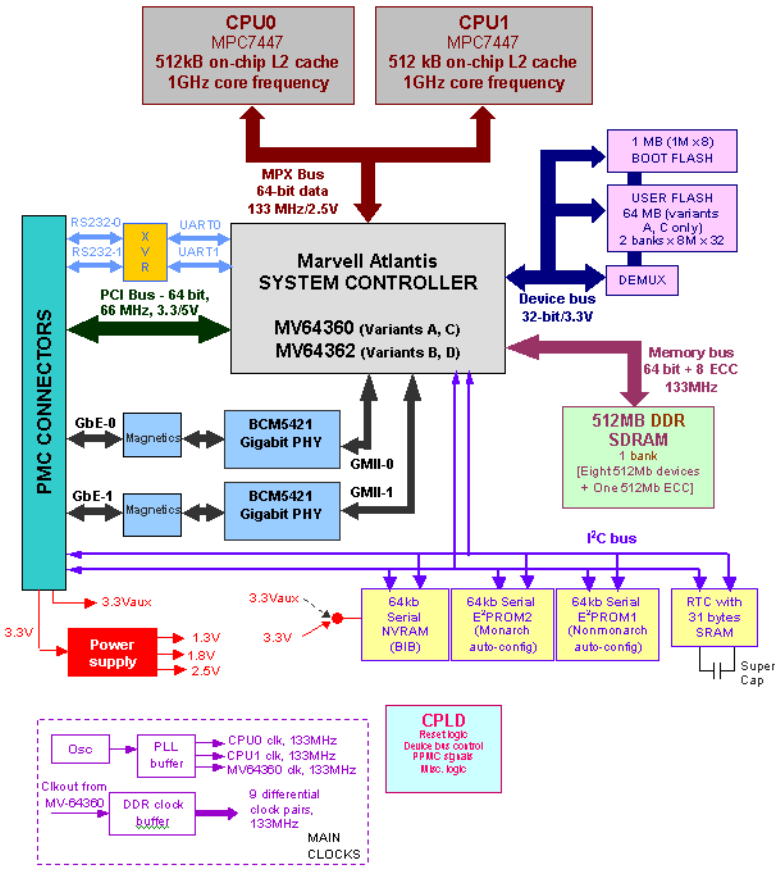

Figure 1: *Functional Block Diagram*

System Environment

PPMC-280 Board Support Package (BSP) supports VxWorks Operating system. The BSP Rel. 3.x has been built under Tornado ™ 2.2 on Windows and Sun™ Solaris ™ host machines.

The Tornado configuration `setup.log` file which is automatically generated while installing the tools is given below for reference.

Note: The BSP Rel. 3.x can be used with Tornado ™ 2.2 in the Solaris™ or Windows NT environment.

Setup.log file:

```
07-Mar-03.13:53SETUP detected the following warning, and
installation was continued:
"WARNING: SETUP has detected that this machine is running on Solaris
2.5.x. Tornado does not officially support Solaris 2.5.x and Solaris
2.6."
07-Mar-03.14:00CD manufacturing time: Thu Oct 03 16:55:33 PDT 2002
07-Mar-03.14:00TDK-14620-ZC-01SETUP-2.2/home3/champ/tor2_2Ppc
07-Mar-03.14:00Tornado 2.2/VxWorks 5.5 for PowerPC
07-Mar-03.14:00SunOS surya 5.5.1 Generic_103640-20 sun4m sparc
SUNW,SPARCstation-5
07-Mar-03.14:00100-22651-30Back End Developer's Toolkit
07-Mar-03.14:00100-22700-30Compiler - GNU: solaris x ppc
07-Mar-03.14:01100-22566-30Tornado Setup SDK
07-Mar-03.14:02100-22531-30Tornado Tools: solaris x ppc
07-Mar-03.14:35100-23549-30VxWorks: ppc40x
07-Mar-03.14:37100-23550-30VxWorks: ppc44x
07-Mar-03.14:37100-22535-30VxWorks: ppc6xx
07-Mar-03.14:38100-22537-30VxWorks: ppc74xx
07-Mar-03.14:39100-22536-30VxWorks: ppc7xx
07-Mar-03.14:40100-22539-30VxWorks: ppc82xx
07-Mar-03.14:40100-22538-30VxWorks: ppc8xx
07-Mar-03.14:41100-22533-30WindView: solaris x ppc
07-Mar-03.14:42
07-Mar-03.14:42licensed product: tornado 310
07-Mar-03.14:42licensed product: windview 320
07-Mar-03.14:42
```

Board Settings

Refer to the respective Installation Guides of PPMC-280 and carrier card for board configuration.

Devices

The various hardware devices and their description are provided in the following table.

Table 1: *Hardware Devices*

HW Device	Description
System Clocks	PPMC-280 features two clocks: TCLK and SysCLK. SysCLK is used for the CPU, while TCLK is used for the MV-64360. Both clocks are described as TCLK_RATE and SYSCLK_RATE macros in the pmc280.h file. Make sure that these frequencies are described correctly. (SYSCLK_RATE = 133000000; TCLK_RATE = 133000000)
Serial Ports	PPMC-280 supports two serial ports (Port0, Port1). For a single CPU BSP Serial Port 0 is used, while in case of a Dual CPU BSP, Serial Port 0 is the console for CPU0 and Serial Port 1 is the console for CPU1
PCI Bus	PPMC-280 supports a 64-bit bus running at 33/66 MHz.
Ethernet Interfaces	PPMC-280 contains one Ethernet controller, which can be configured to the GMII interface.

VxWorks BSP I/O Interface

PPMC-280 provides two ways to connect VxWorks to the network interface for network operations.

Table 2: *I/O Interfaces for VxWorks BSP*

Interface	Description
mgi (Marvel Gigabit interface)	The “mgi” interface utilizes a GMII Ethernet port.

2

PPMC-280 BSP Features

BSP Features

This chapter details all the supported BSP features.

Supported Features

The supported features are detailed in the following sections.

Board Initialization

The board-specific initializations performed by the BSP are listed below:

- Initialize all the CPU registers (such as IBAT, DBAT, LR, MSR)
- Initialize CPU interface registers of MV-64360.
- Automatically configure on-board memory and Memory Controller registers of MV64360/362.
- Set up initial stack pointer
- Sets up the board memory map by configuring MV-64360 decode registers
- Configure PCI interface registers of MV-64360
- Detect PCI devices and assign memory and I/O resources as well as interrupt numbers (depending upon whether the board operates as a monarch or a non-monarch)
- Initialize and configure MV-64360 Interrupt controller to route PCI and other device interrupts
- Initialize all on-board devices by calling the driver initialization functions. This would include (but not limited to) initialization of MPSC ports, and Gigabit Ethernet.
- Initialize and enable L1/L2 cache
- Enable bus pipelining on CPU0 and CPU1
- Enable Interrupts

CPU0 performs most of the system controller (MV-64360) initialization while some initialization such as enabling cache, setting up stack pointer, are performed by both CPU0 and CPU1.

Interrupt Routing

The BSP will support masking-specific device interrupts from CPU0 or CPU1. It is possible to enable PCI interrupts for CPU1 and mask it for CPU0 (by default, CPU0 receives all the PCI interrupts). MV64360/362 resources such as Timers, DMA Engines, and MPSC ports are shared between the CPUs and therefore need not be routed. Similarly, both the Gigabit Ethernet ports on MV64360/362 are handled by CPU0 and need not be handled by CPU1.

Boot from PCI

The BSP provides support for booting the board from the PCI. As this is not a feature of the BSP, a feature supported by MV64360/362 is used, which is the self configuration of registers by reading (register offset: data) pairs from serial EEPROM on I^2C.

Monarch and Non-Monarch operation

The BSP supports detection of its monarch/non-monarch status (through MONARCH# pin of the PMC slot). The board configured to be the monarch will enumerate the PCI bus devices (assigns memory, I/O and interrupt numbers) and also configures itself to route PCI interrupts lines (INTA#, INTB#, INTC# and INTD#) to the CPU. The non-monarch will not perform PCI enumeration.

Memory Partitioning

The system memory will be partitioned into three regions, one for CPU0, one for CPU1, and the third partition shared between the two CPUs for inter-processor communication.

Loosely coupled Symmetric Multi-Processing operation

Each CPU will run a copy of the VxWorks independent of the other. Multiple VxWorks images running on each of the CPUs on board will not mean that there will be two different VxWorks images (one for each CPU). The VxWorks image will be built from the same source tree (by a single compilation) just as one would build the image for a board with a single CPU. .

Although by definition “loosely coupled” in multiprocessing systems means each processor is assigned its own resources such as memory, I/O, interrupts (processors do not share system resources), it is not so in the case of PPMC-280. There are resources that belong to each CPU (like MPSC port

or system memory portion identified as belonging to the CPU) while there are resources that are shared (such as the partition identified as “shared” or the registers of MV64360/362, etc.) The resource allocation or partitioning is purely in software and there is no restriction from hardware, in other words all the resources are accessible by both the CPUs and there is no physical limitation imposed by hardware.

By definition, in a Symmetric Multi-Processing (SMP) system any processor can run any kind of process (operating system as well as applications) while ASMP means that one CPU is selected to run the operating system while the other CPU would run all the user applications. In our implementation for PPMC-280, both CPUs run their own copies of the operating system as well as user applications (in this sense, we have an SMP but with two images of OS).

The case in PPMC-280 is a pseudo-loosely coupled, pseudo-SMP system.

Serial Console

Each CPU will have its own serial console for user interaction.

Board Information

Board related information such as revision information, processor information, size of the on-board SDRAM is provided in a form as according to the required specifications. The BSP will provide programming interfaces to read and write the BIB information to the BIB device (which will be a serial EEPROM on I^2C bus).

Support for On-board Devices

The BSP supports the following on-board devices:

I^2C

The driver for I^2C bus supports reading from and writing to any of the devices on I^2C especially serial EEPROM and RTC.

Serial EEPROM

The Serial EEPROM driver supports read, write and erase operations to the device based on the support provided by the I^2C driver.

Real Time Clock

The RTC driver supports reading from and writing to RTC registers as well as scratch memory.

PCI

The PCI driver for MV64360/362 supports read and write PCI configuration space registers (of MV64360/362 as well as other devices on PCI bus), scan the bus to detect devices and configure them by assigning memory and I/O resources. The driver will support the above functionality beyond a PCI-to-PCI bridge also.

MPSC

The driver for MPSC ports supports read and write to the console as well as functions to change the properties of the serial port such as baud rate, stop bits. The driver will operate in the interrupt mode.

Gigabit Ethernet

Wind River END style driver for MV64360/362 Gigabit Ethernet(s) is supported by the BSP.

3

Dual CPU Configuration

Dual CPU Configuration

VxWorks support for Dual CPU in PPMC-280 varies from the normal (single CPU) VxWorks. The areas of difference are:

- SDRAM Partitioning
- Setting up of BAT registers
- Setting up of Page Table Entries
- MV64360/362 resource partitioning
- Exception handling

SDRAM Partitioning

In a Single CPU system, the entire SDRAM is accessible through BSP. However in a Dual CPU system, it is necessary to partition the SDRAM into three regions:

- Region belonging to CPU0
- Region belonging to CPU1
- Shared memory region (seen by both CPUs)

Although both the CPUs have the same virtual address map, the configuration of the Memory Management Unit has to be done in such a way that it will map it to different physical regions.

Single CPU BSP Defines

In a single CPU BSP, the following defines in `config.h` are usually used to declare the amount of memory on-board.

LOCAL_MEM_SIZE	Total memory available on-board
USER_RESERVED_MEM	Memory reserved for serial/MPSC drivers.

Dual CPU BSP Defines

The following definitions are used:

BOARD_MEM_SIZE	Total memory size, if it is hard-coded. However, if auto-sizing is enabled, it is not used.
APP_SHMEM_SIZE	Memory reserved for shared memory applications
SYS_PGT_SIZE	Memory reserved for the page table
SYS_DRV_SIZE	Memory reserved for drivers like MPSC, etc.
SYS_SHMEM_SIZE	System Shared memory (SYS_PGT_SIZE + SYS_DRV_SIZE)
USER_RESERVED_MEM	Total memory reserved for drivers, page table entries and shared memory application (SYS_SHMEM_SIZE + APP_SHMEM_SIZE)

These defines are critical for the boot to happen. It is recommended that these be changed with utmost care.

Setting up of BAT registers

Normally, in a Single CPU BSP, `sysBatDesc` (defined in `sysLib.c`) is used to by `usrMmuInit()` (called from `usrConfig.c`) to set up the Block Address Translation (BAT) registers of CPU. However in the Dual CPU BSP this is done in `romInit.s` using the `earlysysBatDescCPU0` and `earlysysBatDescCPU1` defined in `frcmmu.c`. The default is to map PCI memory regions and the MV64360/362 internal register space BATs.

Setting up of these tables is critical for the boot to happen. It is recommended that these be changed with utmost care.

Setting up of Page Table Entries

Normally, in a Single CPU BSP, `sysMemPhysDesc` (defined in `sysLib.c`) is used to by `usrMmuInit()` (called from `usrConfig.c`) to set up the page table in memory. However in the Dual CPU BSP this is done in `bootInit.c` (`romStart`) using `earlysysPhysMemDescCPU0` and `earlysysPhysMemDescCPU1` defined in `frcmmu.c`. The default is to map in SDRAM and Internal SRAM through page tables.

Setting up of these tables is critical for the boot to happen. It is recommended that these be changed with utmost care.

MV64360/362 Resource Partitioning

In a single CPU BSP, all the resources on the system controller such as PCI, DMA engines, timers, serial ports, Ethernet ports belong to the CPU. However in the dual CPU BSP these common resources are partitioned. This list below provides details of the partitioning:

- Serial Ports: MPSC0 belongs to CPU0 and MPSC1 belongs to CPU1
- DMA Engines: Two DMA engines belong to CPU0 and two to CPU1
- Timers: Timers are split between the two CPUs
- Ethernet: Both the Ethernet Ports belong to CPU0
- PCI: PCI belongs to CPU0

All the files related to these units such as `sysLib.c`, `vxDmaIntCtrl.c`, `vxCntmrIntCtrl.c`, etc. handle partitioning.

Exception handling

Although exception-handling code is not included as a part of the BSP, dual CPU BSP handles exception is a different way. It is necessary to know that in this BSP, MMU is turned on as soon as the exception handler is called (as against enabling MMU only before calling the interrupt service routine). This causes a latency of a few instructions.

4

Software Basics

System Software Preparation

To perform System Software Preparation, do the following:

- Install the BSP
- Compile source code to build binaries

Installing the BSP

The default packaging of the BSP is a compressed tar file or zip file. You can download the zip file or tar file from the Force Computers SMART™ page at *http://splus.forcecomputers.com/cgi-bin/user/account/services*.

The BSP Rel. 3.x has been built under Tornado™ 2.2 on Windows and Sun Solaris ™ host machines. The tar file has the following directory:

- **PMC280**: this is the main target BSP directory

Installation Procedure for Solaris and Windows NT

To install the BSP, use the compressed tar file.

The following procedure explains how to install a BSP contained in a file named, for example, `bspFile.tar`:

1. Uncompress the tar file to a temporary directory. For example, to uncompress the tar file to a temporary directory in a Solaris environment, use the following command.

```
tar -xvf <bspFile>.tar.
```

2. Copy the `PMC280` directory to `$(WIND_BASE)/target/config` where `WIND_BASE` is the directory where Tornado is installed.

Setup.log file

The Tornado configuration setup.log file which is automatically generated while installing the tools is shown below for reference.

```
07-Mar-03.13:53SETUP detected the following warning, and
installation was continued:
"WARNING: SETUP has detected that this machine is running on Solaris
2.5.x. Tornado does not officially support Solaris 2.5.x and Solaris
2.6."
07-Mar-03.14:00CD manufacturing time: Thu Oct 03 16:55:33 PDT 2002
07-Mar-03.14:00TDK-14620-ZC-01SETUP-2.2/home3/champ/tor2_2Ppc
07-Mar-03.14:00Tornado 2.2/VxWorks 5.5 for PowerPC
```

```
07-Mar-03.14:00SunOS surya 5.5.1 Generic_103640-20 sun4m sparc
SUNW,SPARCstation-5
07-Mar-03.14:00100-22651-30Back End Developer's Toolkit
07-Mar-03.14:00100-22700-30Compiler - GNU: solaris x ppc
07-Mar-03.14:01100-22566-30Tornado Setup SDK
07-Mar-03.14:02100-22531-30Tornado Tools: solaris x ppc
07-Mar-03.14:35100-23549-30VxWorks: ppc40x
07-Mar-03.14:37100-23550-30VxWorks: ppc44x
07-Mar-03.14:37100-22535-30VxWorks: ppc6xx
07-Mar-03.14:38100-22537-30VxWorks: ppc74xx
07-Mar-03.14:39100-22536-30VxWorks: ppc7xx
07-Mar-03.14:40100-22539-30VxWorks: ppc82xx
07-Mar-03.14:40100-22538-30VxWorks: ppc8xx
07-Mar-03.14:41100-22533-30WindView: solaris x ppc
07-Mar-03.14:42
07-Mar-03.14:42licensed product: tornado 310
07-Mar-03.14:42licensed product: windview 320
07-Mar-03.14:42
```

Compile Source Code to Build Binaries

1. Compile the source code to build binaries. Ensure that you have made the necessary changes in the Makefile as mentioned in the Installation guide for the PCI bootable image.
2. Download the PCI bootable image using the procedure as mentioned in the appendix.

5

API Call Reference

List of APIs

This chapter provides a detailed description of all functions.

Note:

- **The term Input used in this chapter is a parameter that must be passed to the function.**
- **The term Output used in this chapter is the result of the function.**
- **The term Return used in this chapter is the value returned by the function.**

MV-64360 General Driver

This driver functions as the lowest software interface to the MV-64360 registers and SDRAM accesses. All hardware register accesses and SDMA readings are completed by this API. To achieve better performance, most accesses are implemented using macro definition

Software Modules

The driver is implemented in:

- `gtCore.c`: Data block read / write and register write mask bits.
- `gtCore.h`: Macro for read / write cacheable / non-cacheable char / short / word.

External Interface

The APIs for the external interface are listed here.

Table 3:*General Driver API Synopsis*

Macro	Description
REG_ADDR(offset)	Returns the full address of a given register offset.
REG_CONTENT(offset)	Returns the register's content.
VIRTUAL_TO_PHY(address)	Meaningless for PPC CPUs.

Table 3:*General Driver API Synopsis*

Macro	Description
PHY_TO_VIRTUAL(address)	Meaningless for PPC CPUs.
GT_REG_READ(offset, pData)	Reads and byte swaps an MV64360/362 internal register into pData.
GT_REG_WRITE(offset, data)	Writes and byte swaps data into an MV64360/362 internal register.
WRITE_CHAR(address, data)	Writes a character into an address.
GT_WRITE_SHORT(address, data)	Writes a short into an address.
GT_WRITE_WORD(address, data)	Writes a word into an address.
Writes a word into an address.	Same as WRITE_CHAR for PPC CPUs
GT_WRITE_SHORT_CACHEABLE(address, data)	Same as WRITE_SHORT for PPC CPUs.
GT_WRITE_WORD_CACHEABLE(address, data)	Same as WRITE_WORD for PPC CPUs.
GT_READ_CHAR(address, pData)	Reads a character from address into pData.
GT_READ_SHORT(address, pData)	Reads a short from address into pData.
GT_READ_WORD(address, pData)	Reads a word from address into pData.
GT_READ_CHAR_CACHEABLE(address, pData)	Same as READ_CHAR for PPC CPUs.
GT_READ_SHORT_CACHEABLE(address, pData)	Same as READ_SHORT for PPC CPUs.
GT_READ_WORD_CACHEABLE(address, pData)	Same as READ_WORD for PPC CPUs.
GT_READCHAR(address)	Returns a char from an address.

Table 3:*General Driver API Synopsis*

Macro	Description
GT_READSHORT(address)	Returns a short from an address.
GT_READWORD(address)	Returns a word from an address.
GT_READCHAR_CACHEABLE(address)	Same as READCHAR for PPC CPUs.
GT_READSHORT_CACHEABLE(address)	Same as READSHORT for PPC CPUs.
GT_READWORD_CACHEABLE(address)	Same as READWORD for PPC CPUs.
GT_SET_REG_BITS(offset, bits)	Sets the specified bits in the given register.
GT_RESET_REG_BITS(offset, bits)	Resets the specified bits in the given register.
GT_REGREAD(offset)	Returns a swapped value of a register.
GT_WORD_SWAP(32bitWord)	Changes the endianess of a given word.
GT_SHORT_SWAP(16bitShort)	Changes the endianess of a given short.
GT_LONG_SWAP(64bitDword)	Changes the endianess of a given long word.

MV-64360 INTERRUPT CONTROLLER

An interrupt controller is necessary because numerous MV-64360 interrupts share the same physical line. Hooking an Interrupt Service Routine (ISR) to this controller requires knowledge of the various GT interrupt causes. This Interrupt Controller supports a connection to the CPU interrupt line.

The MV-64360 Interrupt Controller system introduces two layers of Cause registers:

- First Layer
- Second Layer

FIRST LAYER

This layer includes the Main High Interrupt Cause and Main Low Interrupt Cause registers. (The Select register reflects both High and Low Cause registers). This layer summarizes the interrupts generated by each MV-64360 subunit. Each bit set in these registers implies that a non-masked interrupt has occurred in a subunit.

SECOND LAYER

This layer includes all MV-64360 subunit's Cause registers. Each subunit has its own Cause and Mask registers. Once an interrupt event occurs, its corresponding bit in the cause register is set to "1". If the interrupt is not masked, it is also marked in the Main Interrupt Cause register (First layer). This architecture implies a unique interrupt controller for each MV-64360 subunit.

Note: This interrupt controller introduces support only for First Layer interrupts and consists of three phases. The controller interface also provides MV64360/362 interrupt masking ability for the First Layer.

INTERRUPT CONTROLLER PHASES

Table 4:*Interrupt Controller Phases*

Phase	Description
Driver Initialization	This phase includes hooking the driver's ISR to the CPU interrupt vector.

Table 4:*Interrupt Controller Phases*

Phase	Description
ISR Connecting	This phase includes the gathering of information about user / subunit ISR and interrupt priority.
Interrupt handling	This includes the handling of an interrupt by the Interrupt Handlers (driver's ISR).

This controller prevents halt of the CPU caused by an interrupt that was enabled while no service routine was connected. In addition, there is full interrupt masking control over the MV First Layer interrupts.

Software Modules

The software modules are:

- `gtIntControl.c`
- `gtIntControl.h`

Software Requirements

The software requirements are:

- WindRiver VxWorks Operating System, Version 5.3.1 or later
- MV General Driver

Restrictions

This Interrupt Controller driver supports the First layer of the MV-64360 Interrupt Controller. Thus each of the Second Layer interrupts should be handled in its own unit.

Interrupt acknowledgement is NOT the responsibility of this driver. The First Layer Cause registers are Read Only. To acknowledge an interrupt, the software needs to clear (write 0) the active bit(s) in the Second Layer Cause register.

Execution Flow

This interrupt controller introduces full support for the CPU interrupt lines driven by the MV-64360. It also supports the Select Register used by the MV-64360 for minimizing the interrupt identification process to a single read cycle. This support is provided by an individual ISR for each MV-64360 interrupt output pin. As the Power PC architecture is restricted to only one external interrupt pin, this driver has the following handlers:

Table 5:*Handlers*

Handler	Description
`GtIntCpuHigh()`	Handles the interrupts asserted by the CPU pin and interrupt events which are generated by the CPU High Interrupt Cause register.
`GtIntCpuLow()`	Handles the interrupts asserted by the CPU pin and interrupt events which are generated by the CPU Low Interrupt Cause register.
`GtIntCpuSelect()`	Handles the interrupts asserted by the CPU pin and interrupt events which are generated by the CPU High or the Low Interrupt Cause registers (or both), using one read cycle.

Each of these handlers has its own data structure, which holds a list of ISRs to invoke in case of an interrupt pending on its corresponding pin. As the Power PC architecture restrict only one external interrupt pin, this driver uses the gtIntCpuSelect() handler.

Driver Initialization

The driver's handlers are connected to the Power PC external exception vector 0x500 using the VxWorks connecting routine excIntConnect(). The user can decide which ISR to connect to the CPU interrupt pin (gtIntCpuHigh, gtIntCpuLow or gtIntCpuselect). As the Power PC architecture is restricted to only one external interrupt pin, and this driver handles all MV interrupts (High and Low), this driver makes use of the gtIntCpuSelect() handler.

ISR Connection

This stage fills each driver's ISR data structure with the information regarding the connected user ISR. The Interrupt Controller driver decides which user ISR to connect to which data structure according to its enumerated macro that defines the interrupt cause distribution (High or Low Cause registers).

Interrupt Handling

When an interrupt is pending, the connected driver's ISR (connected in initialization phase) is invoked to search its data structure for the interrupt cause which initiated the interrupt. After the initiating interrupt has been identified, the appropriate user ISR is invoked.

External Interface Data Structure

The external interface data structure is provided here.

ENUM GT_INT_CAUSE {The list of High and Low Interrupt causes (Total of 64 causes)} This enumerator creates the Global Cause register out of Main High and Low Interrupt Cause registers. When the High Cause register is first, each interrupt cause is represented by an integer. To hook a C routine to the MV Interrupt Controller, use this enum type to describe the MV cause that you would like to hook to (this is done for improved code readability).

CAUSE DISTRIBUTOR MACROS This driver defines Cause distribution macros. Each MV-64360 interrupt pin is represented by a macro (or two). This macro defines which cause bits are active in each interrupt pin:

- CPU_INT_HIGH_CAUSE CPU interrupt pin
- CPU_INT_LOW_CAUSE CPU interrupt pin

For example:
#define CPU_INT_LOW_CAUSE (cause >=00 && cause <=31)

CPU INT[0/1]* MASK SELECTION Not Applicable

External Interface APIs

The external interface APIs are detailed here.

STATUS gtIntCntrlInit ()

DESCRIPTION As the Power PC architecture is restricted to only one external interrupt pin, and this driver handles all MV interrupts (High and Low), this driver makes use of the gtIntCpuSelect() handler. The driver connects to the CPU external interrupt vector (0x500) by using VxWorks excIntConnect() routine.

INPUT Not Applicable

OUTPUT Attaches the interrupt handler to the CPU external interrupt vector (by using VxWorks routine excIntConnect()).

RETURN

`OK` If the output was successful.

`ERROR` If the output failed.

STATUS gtIntConnect(GT_INT_CAUSE cause, VOIDFUNCPTR ISRptr, int ISRarg, int prio)

DESCRIPTION Hooks a user's C routine to one of the GT interrupt causes specified by cause (use GT_INT_CAUSE enumerated type). The user's C routine is given by ISRptr and the ISRarg is an argument to this routine. This connection can be given a priority in case of simultaneous multiple interrupts. The highest priority is 0.

INPUT

`GT_INT_CAUSE` Cause Interrupt cause as defined in CAUSE data structure.

`VOIDFUNCPTR ISRptr` Pointer to User ISR.

`int ISRarg A` Parameter to the user ISR.

`int prio` Interrupt priority

OUTPUT Addresses the infrastructure of the driver and creates the connection according to its given priority.

RETURN

`OK`	If the output was successful.
`ERROR`	If the output failed.

STATUS gtIntEnable(GT_INT_CAUSE cause)

DESCRIPTION Enables a given interrupt cause described by cause.

INPUT

`UINT cause`	Description of interrupt cause (See enum GT_INT_CAUSE).

OUTPUT Changes the corresponding bits in the Mask registers according to the cause bit distribution macros.

RETURN

```
OK      If the output was successful.

ERROR   If the output failed.
```

STATUS gtIntDisable(GT_INT_CAUSE cause)

DESCRIPTION Disables a given interrupt cause described by cause.

INPUT

`UINT cause`	Description of interrupt cause (See enum GT_INT_CAUSE).

OUTPUT Changes the corresponding bits in the Mask registers according to the cause bit distribution macros.

RETURN

```
OK      If the output was successful.

ERROR   If the output failed.
```

System Interrupt Controller

A system interrupt controller is necessary because of the Power PC architecture restriction concerning interrupts. Since the Power PC has only one external interrupt exception (vector 0x500), it is essential to have an interrupt controller on board. This interrupt controller provides the support for the following routines:

- intConnect()
- intDisable()
- intenable()

The system interrupt controller uses the General Purpose Port (GPP) Interrupt Controller services to implement those functionalities.
This driver is fully compatible with VxWorks. Thus, to connect an interrupt routine to one of the above external interrupts, use the standard VxWorks intConnect() routine.

Supported Features

This driver is fully compatible with the VxWorks interrupt API. For example, to connect an interrupt routine to an external interrupts event, use the standard VxWorks intConnect() routine.

Software Modules

The following software modules are available:

- `sysIntCtrl.c`
- `sysIntCtrl.h`

Software Requirements

The software requirements are:

- WindRiver VxWorks Operating system, Version 5.3.1 or later
- GPP Interrupt Controller driver.

System Resource Usage

System interrupt sources are connected to a GPP pin. Make sure the GPP pin is configured to act as interrupt.

Restrictions

Interrupt acknowledgement is completed by the device driver that triggered the interrupt (not by the System Interrupt Controller).

External Interface

GPP pin descriptions

Use the following macros as interrupt vectors intConnect routine (located in pmc280.h file)

- CARRIER_INT0 Describes interrupt input from carrier card (GPP pin 6)
- CARRIER_INT1 Describes interrupt input from carrier card (GPP pin 7)
- WD_NMI Describe watchdog NMI interrupt (GPP pin 18)
- PHY0_INT Describes Interrupt from Ethernet PHY0 (GPP pin 12)
- PHY0_INT Describes Interrupt from Ethernet PHY0 (GPP pin 13)
- PCI_INTA Describes PCI interrupt A (GPP pin 27)
- PCI_INTB Describes PCI interrupt B (GPP pin 29)
- PCI_INTC Describes PCI interrupt C (GPP pin 16)
- PCI_INTD Describes PCI interrupt D (GPP pin 17)

Driver's API

This driver provides function pointers to VxWorks, thus interrupt control (Connect, Enable and Disable) is performed using the VxWorks interface.

STATUS gtIntCtrlInit ()

DESCRIPTION

This driver initializes the GPP Interrupt Controller and assigns the VxWorks interrupt control routines pointers to the system interrupt controller. This function is called in the system initialization routine sysHwInit2() of sysLib.c.

INPUT

Not Applicable

OUTPUT	Driver's routines are connected to the Vxworks Interrupt control pointers.
RETURN	Not Applicable

General Purpose Port Interrupt Controller

The General Purpose Port (GPP) input pins can be used to register external interrupts. An assertion of a GPP input pin (toggle from "0" to "1" in case of active high pin, from high to low in case of active low pin), results in setting the corresponding bit in the GPP Interrupt Cause register.
This VxWorks driver has full control over the GPP interrupt system:

- User Interrupt Service Routine connection for each GPP pin interrupt.
- Enable / Disable a GPP pin interrupt.

Note: The GPP cause bit must be unmasked prior to receiving an interrupt.

Supported Features

The supported features are listed here:

- The controller provides an easy way to hook a C Interrupt Service Routine (ISR) to a specific interrupt caused by the GPP.
- The controller interrupt mechanism provides a way for the programmer to set an interrupt priority.
- Full interrupt control over the GPP Interrupt facility.
- This driver auto acknowledges interrupts and you are not required to acknowledge the interrupt in ISR.

Software Modules

The software modules are:

`vxGppIntCtrl.c`	GPP interrupt controller implementation file
`vxGppintCtrl.h`	GPP interrupt controller header file

Requirements

The requirements are listed here:

- GT General Driver
- GT Interrupt Controller

Restrictions

This driver does not auto-acknowledge GPP interrupts. You must acknowledge the initiating interrupt in the hooked ISR. Use vxGppIntAck (GPP_CAUSE cause) for this purpose.

External Interface Data Structures

The external interface data structures are listed here.

GPP_CAUSE ENUMERATOR This enumerator describes the GPP interrupt causes to which the user attaches the ISR. For example, to connect an ISR to watchdog NMI (GPP pin 18) event, use:

```
frcGppIntConnect (GPP_PIN6_WD_NMI, ISRptr,
ISRparameter, ISRpriority)
```

This enumerator is defined in frcGppIntCntl.h file.

External API

The external APIs are detailed here.

STATUS frcGppCPU1IntEnable(GPP_CAUSE cause)

DESCRIPTION This routine makes a specified GT GPP interrupt cause available to the CPU.

INPUT

cause GPP interrupt cause (0-31).

OUTPUT The appropriate bit in GPP mask register is reset.

RETURN OK

STATUS frcGppCPU1IntDisable(GPP_CAUSE cause)

DESCRIPTION	This routine makes a specified GT GPP interrupt cause unavailable to the CPU.
INPUT	
cause	GPP interrupt cause (0-31).
OUTPUT	The appropriate bit in GPP mask register is reset.
RETURN	OK.

void frcGppIntCtrlInit ()

DESCRIPTION	This routine connects the driver's interrupt handlers, each to its corresponding bit in the GT Interrupt Con-troller using the gtIntConnect() routine.
INPUT	Not Applicable
OUTPUT	Driver's ISRs are connected to the GT Interrupt Controller and interrupts are unmasked.
RETURN	Not Applicable

STATUS frcGppIntConnect (GPP_CAUSE cause, VOIDFUNCPTR routine, int parameter, int prio)

DESCRIPTION	This routine connects a specified user ISR to a specified GPP interrupt cause.
INPUT	
GPP_CAUSE cause	GPP interrupt cause.
VOIDFUNCPTR routine	User ISR.
int parameter	User ISR parameter.

`int prio`	Interrupt handling priority where 0 is highest.
OUTPUT	An internal data structure is filled with the connection details.

UINT 32 frcGppIntDisable (GPP_CAUSE cause)

DESCRIPTION	This routine masks a specified GPP interrupt cause on the GPP mask register.
INPUT	
`GPP_CAUSE cause`	GPP interrupt cause.
OUTPUT	The appropriate bit in the GPP mask register is reset (0xf10c).
RETURN	The former GPP interrupt mask register value.

UINT 32 frcGppIntEnable (GPP_CAUSE cause)

DESCRIPTION	This routine unmasks a specified GPP interrupt cause on the GPP mask register.
INPUT	
`GPP_CAUSE cause`	GPP interrupt cause.
OUTPUT	The appropriate bit in the GPP mask register is set (0xf10c).
RETURN	The former GPP interrupt mask register value.

PCI Scan Driver

This driver includes routines that execute the PCI scanning and basic initialization of the PCI devices -Network Interface Card NIC and Galileo's GalNet PCI devices for any future use. Moreover, to comply with Galileo's GalNet drivers, the driver delivers an array Gal-NetMappingArray in which each entry describes a GalNet device's PCI information, including:

- Device and Vendor ID
- Internal register base address
- IDSel

- PCI number (0)

The driver flow of execution is divided into four phases:

- Driver initialization
- PCI0 scanning
- PCI0 MEM1 address spacing remap
- PCI devices configuration

Driver Initialization

In this phase, the driver:

- initializes the pciConfigLib with PCI read / write routines.
- scans for Monarch or Non-Monarch Mode and performs initialization accordingly.
- cleans both GalNetMappingArray and pciDeviceArray data structures.

PCI0 Scanning

If the Monarch pin is asserted then the driver waits till EREADY Signal goes to HIGH and then does PCI Scan by the pciGoScan() routine. The EREADY signal assures all other PCI devices are ready for initialization. In case of Monarch the pciGoScan routine fills up the GalNet device table's `GalNetMappingArray`, with information on the GalNet devices in PCI0 and the pciDeviceArray table with information on any other PCI devices located in PCI0 (NIC).
In Non-Monarch mode it pulls down the EREADYOUT signal of the MV-64360, which in turn causes the EREADY signal to be pulled high so that an external monarch can initiate PCI Bus enumeration.

Debugging Facilities

Add the -DDEBUG_PCI flag to compilation flags to obtain the debug information.
The debugging information includes:

- A list of the PCI scan, for each IDSel on the PCI bus, regarding the device and vendor ID found.
- A full detailed print of each PCI configuration write and read whenever performed.

Software Modules

The software modules are detailed here.

pciScan The main PCI scan routine, including some read / write PCI and internal registers functions.

pciConfigLib Based on a the pciConfigLib module provided by WindRiver, includes a library containing device configuration and search functions for PCI bus.

Note: To use this library's functions, the pciConfigLibInit function must be called, with the PCI_MECHANIZM0 flag. This allow the usage of a user defined read/write routines. Those user routines are delivered in the `pciConfigLib` **initialization routine.**

Structures

pciDeviceStruct This struct is delivered to GalNet drivers for further processing.

```
typedef struct
{
UINT32 type; /* Device and Vendor id */
UINT32 InternalRegistersBaseAddress; /* Internal Register
Base Address */
int IDSel; /* IDsel */
int pciNo; /* PCI number (0 or 1) */
} pciDeviceStruct;
```

Variables

GalNetMappingArray pciDeviceStruct GalNetMappingArray[MAX_DEV_NUM]
This variable holds the GalNet devices information collected in the PCI scanning.

pciDeviceArray pciDeviceStruct pciDeviceArray[MAX_DEV_NUM]
This variable holds the other PCI devices information collected in the PCI scanning.

Driver APIs

The APIs are listed here.

void frcPciShow(void)

DESCRIPTION This displays information such as device number, device ID, Vendor ID and other resource related to all the PCI devices found.

unsigned int frcPci0ReadConfigReg (unsigned int regOffset,unsigned int pciDevNum)

DESCRIPTION The GT holds two registers to support configuration accesses as defined in the PCI Specifications Rev 2.2: Configuration Address and Configuration Data registers. The mechanism for accessing configuration space is to write a value into the Configuration Address register that specifies the PCI bus number (this function use the value of 0 by default for this parameter), Device number on the bus, Function number within the device (will be combined with the register offset) and Configuration register offset within the device / function being accessed. A subsequent read to the PCI Configuration Data register causes the GT to translate that Configuration Address value to the requested cycle on the PCI bus (in this case - read) or internal configuration space. This function reads from an agent's configuration register at any of the eight possible function in its Configuration Space Header.

EXAMPLE The value 0x004 is combined from the function number (bits[11:8]) and the register offset (bits[7:0]) in the Configuration Space Header. In this case, the fuction number is 0 and the register offset is 0x04.

```
...
        data = frcPci0ReadConfigReg(0x004,6);
        ...
The configuration address register (0xCF8) fields are:

      31 30    24 23  16 15  11 10     8 7      2  0     <=bit Number
 |congif|Reserved|  Bus |Device|Function|Register|00|
 |Enable|        |Number|Number| Number | Number |  |    <=field Name
```

INPUT

regOffset The register offset PCI configuration Space Header combined with the function number as shown in the example above.

pciDevNum The agent's device number.

OUTPUT PCI write configuration cycle.

RETURN 32 bit read data from the agent's configuration register. if the data = 0xffffffff check the master abort bit in the cause register to make sure the data is valid.

void frcPci0WriteConfigReg(unsigned int regOffset,unsigned int pciDevNum, unsigned int data)

DESCRIPTION The MV holds two registers to support configuration accesses as defined in the PCI spec Rev 2.2: Configuration Address and Configuration Data registers. The mechanism for accessing configuration space is to write a value into the Configuration Address register that specifies the PCI bus number (this function use the value of 0 by default for this parameter), Device number on the bus, Function number within the device (will be combined with the register offset) and Configuration register offset within the device / function being accessed. A subsequent write to the PCI Configuration Data register causes the MV to translate that Configuration Address value to the requested cycle on the PCI bus (in this case - write) or internal configuration space. This function writes to an agent's configuration register at any of the 8 possible function in its Configuration Space Header.

EXAMPLE The value 0x004 is combined from the function number (bits[11:8]) and the register offset (bits[7:0]) in the Configuration Space Header. In this case, the fuction number is 0 and the register offset is 0x04.

```
...
      frcPci0WriteConfigReg(0x004,6,PCI_MASTER_ENABLE);
        ...
```

The configuration address register (0xCF8) fields are:

```
     31 30    24 23  16 15  11 10     8 7      2  0     <=bit Number
|congif|Reserved|  Bus |Device|Function|Register|00|
|Enable|        |Number|Number| Number | Number |  |    <=field Name
```

INPUT

regOffset The register offset PCI configuration Space Header combined with the function number as shown in the example above.

pciDevNum The agent's device number.

data The data to be written.

OUTPUT PCI write configuration cycle.

RETURN None.

STATUS frcPciConfigRead (int bus, int dev, int func, int RegNum, UINT32 *RegData)

DESCRIPTION The function makes a PCI configuration register read. It reads the data from a register number (offset) -RegNum, of device number - dev, on the active PCI bus number -bus, and puts the data into `RegData`. The function locks interrupts before reading, and unlocks them after reading, using the OS functions intLock() and intUnlock(). The reading is 32 bit wide (long).

INPUT

`int bus` PCI bus number.

`int dev` PCI device number.

`int func` Device's function (must be NULL).

`int RegNum` PCI configuration register's offset.

`UINT32 *RegData` A pointer to a variable in which the read data must be returned.

OUTPUT Not Applicable

RETURN

```
OK      On success.

Error   On failure.
```

STATUS frcPciConfigWrite (int bus, int dev, int func, int RegNum, UINT32 RegData)

DESCRIPTION The function makes a PCI configuration write (PCI0 or PCI1 depending on which is the active). It writes the data in RegData into a register number (offset) -RegNum, of device number -dev, on PCI bus number -bus. The function locks interrupts before writing, and unlocks them after writing, using the OS functions intLock() and intUnlock(). The writing is 32 bit wide (long).

INPUT

`int bus`	PCI bus number.
`int dev`	PCI device number.
`int func`	Device's function (must be NULL).
`int RegNum`	PCI configuration register's offset.
`UINT32 RegData`	Data to write into the register.

OUTPUT Not Applicable

RETURN

`OK`	`On success.`
`Error`	`On failure.`

Communication Unit Management Driver

The BSP includes a Communication Unit Management driver that utilizes the MV-64360 Communication Unit's basic drivers to dominate
the flow of data in the various ports in the MV-64360. This driver also determines the Communication Unit configuration and initializes the MV-64360 accordingly. This driver uses the Communication
Unit driver's APIs to operate the MV-64360 Communication Unit. This application layer is responsible for the initialization and configuration
of the various communication units as defined by the user. This application layer is also responsible for handling all events (interrupts) generated by the communication units. This driver can be replaced by a user-defined management unit driver that manages the communication units in a different way. The driver also includes a switching table that determines where to switch the packets received in each communication port (Repeater functionality).

Software Modules

This driver is implemented in the following:

- Ethernet.c
- Ethernet.h

Supported Features:

- This low level driver is OS independent. Allocating memory for the descriptor rings and buffers are not within the scope of this driver.
- The user is free from Rx/Tx queue managing.
- This low level driver introduce functionality API that enable the to operate Marvell's Gigabit Ethernet Controller in a convenient way.
- Simple Gigabit Ethernet port operation API.
- Simple Gigabit Ethernet port data flow API.
- Data flow and operation API support per queue functionality.
- Support cached descriptors for better performance.
- Enable access to all four DRAM banks and internal SRAM memory spaces.
- PHY access and control API.
- Port control register configuration API.
- Full control over Unicast and Multicast MAC configurations.

Operation flow

Initialization phase

This phase complete the initialization of the ETH_PORT_INFO struct. User information regarding port configuration has to be set prior to calling the port initialization routine. For example, the user has to assign the portPhyAddr field which is board depended parameter. In this phase any port Tx/Rx activity is halted, MIB counters are cleared, PHY address is set according to user parameter and access to DRAM and internal SRAM memory spaces.

Driver ring initialization

Allocating memory for the descriptor rings and buffers is not within the scope of this driver. Thus, the user is required to allocate memory for the descriptors ring and buffers. Those memory parameters are used by the Rx and Tx ring initialization routines in order to curve the descriptor linked list in a form of a ring.

Note: Pay special attention to alignment issues when using cached descriptors/buffers. In this phase the driver store information in the ETH_PORT_INFO struct regarding each queue ring.

Driver start

This phase prepares the Ethernet port for Rx and Tx activity. It uses the information stored in the ETH_PORT_INFO struct to initialize the various port registers.

Data flow

All packet references to/from the driver are done using PKT_INFO struct. This struct is a unified struct used with Rx and Tx operations. This way the user is not required to be familiar with neither Tx or Rx descriptors structures.The driver's descriptors rings are management by indexes. Those indexes control the ring resources and used to indicate a SW resource error

`current` This index points to the current available resource for use. For example in Rx process this index will point to the descriptor that will be passed to the user upon calling the receive routine.In Tx process, this index will point to the descriptor that will be assigned with the user packet info and transmitted.

`used` This index points to the descriptor that need to restore its resources. For example in Rx process, using the Rx buffer return API will attach the buffer returned in packet info to the descriptor pointed by 'used'. In Tx process, using the Tx descriptor return will merely return the user packet info with the command status of the transmitted buffer pointed by the 'used' index. Nevertheless, it is essential to use this routine to update the 'used' index.

`first` This index supports Tx Scatter-Gather. It points to the first descriptor of a packet assembled of multiple buffers. For example when in middle of Such packet we have a Tx resource error the 'curr' index get the value of 'first' to indicate that the ring returned to its state before trying to transmit this packet.

Receive operation

The ethPortReceive API set the packet information struct, passed by the caller, with received information from the 'current' SDMA descriptor. It is the user's responsibility to return this resource back to the Rx descriptor ring to enable the reuse of this source. Return Rx resource is done using the ethRxReturnBuff API.

Transmit operation

The ethPortSend API supports Scatter-Gather which enables to send a packet spanned over multiple buffers. This means that for each packet info structure given by the user and put into the Tx descriptors ring, will be transmitted only if the 'LAST' bit will be set in the packet info command status field. This API also consider restriction regarding buffer alignments and sizes.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires the following external support:

D_CACHE_FLUSH_LINE (ADDRESS, ADDRESS OFFSET)

This macro applies assembly code to flush and invalidate cache line.

- address - address base.
- address offset - address offset

External Interface -Api's

void ethPortInit(ETH_PORT_INFO *pEthPortCtrl)

DESCRIPTION

This function prepares the ethernet port to start its activity:

- Completes the ethernet port driver struct initialization toward port start routine.
- Resets the device to a quiescent state in case of warm reboot.
- Enables SDMA access to all four DRAM banks as well as internal SRAM.
- Cleans MAC tables. The reset status of those tables is unknown.
- Sets PHY address.

Note: Call this routine prior to ethPortStart routine and after setting user values in the user fields of Ethernet port control struct (portPhyAddr).

INPUT

ETH_PORT_INFO *pEthPortCtrl Ethernet port control struct

OUTPUT See description.

RETURN None.

bool ethPortStart(ETH_PORT_INFO *pEthPortCtrl)

DESCRIPTION This routine prepares the Ethernet port for Rx and Tx activity:

- Initialize Tx and Rx Current Descriptor Pointer for each queue that has been initialized a descriptor's ring (using etherInitTx-DescRing for Tx and etherInitRxDescRing for Rx)
- Initialize and enable the Ethernet configuration port by writing to the port's configuration and command registers.
- Initialize and enable the SDMA by writing to the SDMA's configuration and command registers.

After completing these steps, the ethernet port SDMA can starts to perform Rx and Tx activities.

Note: Each Rx and Tx queue descriptor's list must be initialized prior to calling this function (use etherInitTxDescRing for Tx queues and etherInitRxDescRing for Rx queues).

INPUT

ETH_PORT_INFO *pEthPortCtrl Ethernet port control struct

OUTPUT Ethernet port is ready to receive and transmit.

RETURN

```
False   If the port PHY is not up.

True    Otherwise.
```

void ethPortUcAddrSet(ETH_PORT ethPortNum, unsigned char *pAddr,ETH_QUEUE queue)

DESCRIPTION This function Set the port Ethernet MAC address.

INPUT

ETH_PORT ethPortNum Port number.

char * pAddr Address to be set

ETH_QUEUE queue **Rx queue number for this MAC address.**

OUTPUT Set MAC address low and high registers. Also calls ethPortUcAddr() to set the unicast table with the proper information.

RETURN Not Applicable

static bool ethPortUcAddr(ETH_PORT ethPortNum, unsigned char ucNibble, ETH_QUEUE queue, int option)

DESCRIPTION This function sets the Port Unicast address table and locates the proper entry in the Unicast table for the specified MAC nibble and sets its properties according to function parameters.

INPUT

ETH_PORT ethPortNum Port number.

unsigned char ucNibble Unicast MAC Address last nibble.

ETH_QUEUEqueue Rx queue number for this MAC address.

int option 0 = Add
1 = remove address.

OUTPUT This function add / removes MAC addresses from the port unicast addresstable.

RETURN

`True` `If output succeeded.`

`False` `If option parameter is invalid.`

void ethPortMcAddr(ETH_PORT ethPortNum, unsigned char *pAddr, ETH_QUEUE queue, int option)

DESCRIPTION This API controls the MV device MAC multicast support. The MV device supports multicast using two tables:

1. Special Multicast Table for MAC addresses of the form 0x01-00-5E-00-00-XX (where XX is between 0x00 and 0xFF). The MAC DA[7:0] bits are used as a pointer to the Special Multicast Table entries in the DA-Filter table. In this case, the function calls ethPortSmcAddr() routine to set the Special Multicast Table.
2. Other Multicast Table for multicast of another type. A CRC-8bit is used as an index to the Other Multicast Table entries in the DA-Filter table. In this case, the function calculates the CRC-8bit value and calls ethPortOmcAddr() routine to set the Other Multicast Table.

INPUT

ETH_PORT ethPortNum Port number.

unsigned char *pAddr Unicast MAC Address.

ETH_QUEUE queue Rx queue number for this MAC address.

int option 0 = Add,
1 = remove address.

OUTPUT Not Applicable

RETURN

```
True    If output succeeded

False   If addAddressTableEntry( ) failed.
```

static bool ethPortSmcAddr(ETH_PORT ethPortNum, unsigned char mcByte, ETH_QUEUE queue, int option)

DESCRIPTION This routine controls the MV device special MAC multicast support. The Special Multicast Table for MAC addresses supports MAC of the form 0x01-00-5E-00-00-XX (where XX is between 0x00 and 0xFF). The MAC DA[7:0] bits are used as a pointer to the Special Multicast Table entries in the DA-Filter table.This function set the Special Multicast Table appropriate entry according to the argument given.

INPUT

ETH_PORT ethPortNum Port number.

unsigned char mcByte Multicast addr last byte (MAC DA[7:0] bits).

ETH_QUEUE queue Rx queue number for this MAC address.

int option 0 = Add, 1 = remove address.

OUTPUT See description.

RETURN

```
True    If output succeeded.

False   If option parameter is invalid.
```

static bool ethPortOmcAddr(ETH_PORT ethPortNum, unsigned char crc8, ETH_QUEUE queue, int option)

DESCRIPTION This routine controls the MV device Other MAC multicast support. The Other Multicast Table is used for multicast of another type. A CRC-8bit is used as an index to the Other Multicast Table entries in the DA-Filter table.The function gets the CRC-8bit value from the calling routine and set the Other Multicast Table appropriate entry according to the CRC-8 argument given.

INPUT

ETH_PORT ethPortNum Port number.

unsigned char crc8 A CRC-8bit (Polynomial: x^8+x^2+x^1+1).

ETH_QUEUE queue Rx queue number for this MAC address.

int option 0 = Add,
1 = remove address.

OUTPUT See description.

RETURN

```
True If output succeeded.

False If option parameter is invalid.
```

void ethPortInitMacTables(ETH_PORT ethPortNum)

DESCRIPTION Go through all the DA filter tables (Unicast, Special Multicast & Other Multicast) and set each entry to 0.

INPUT

ETH_PORT ethPortNum Ethernet Port number. See ETH_PORT enum.

OUTPUT Multicast and Unicast packets are rejected.

RETURN None.

void ethClearMibCounters (ETH_PORT ethPortNum)

This function clears all MIB counters of a specific ethernet port. A read from the MIB counter will reset the counter.

INPUT

ETH_PORT ethPortNum Ethernet Port number. See ETH_PORT enum.

OUTPUT After reading all MIB counters, the counters resets.

RETURN MIB counter value.

static void ethernetPhySet(ETH_PORT ethPortNum, int phyAddr)

ethernetPhySet Set the ethernet port PHY address.

This routine set the ethernet port PHY address according to given parameter.

INPUT

ETH_PORT ethPortNum Ethernet Port number. See ETH_PORT enum.

OUTPUT Set PHY Address Register with given PHY address parameter.

RETURN None.

static int ethernetPhyGet(ETH_PORT ethPortNum)

This routine returns the given ethernet port PHY address.

INPUT

ETH_PORT ethPortNum Ethernet Port number. See ETH_PORT enum.

OUTPUT None.

RETURN PHY address.

bool ethernetPhyReset(ETH_PORT ethPortNum)

This routine utilize the SMI interface to reset the ethernet port PHY. The routine waits until the link is up again or link up is time-out.

INPUT

ETH_PORT ethPortNum Ethernet Port number. See ETH_PORT enum.

OUTPUT The ethernet port PHY renew its link.

RETURN None.

void ethPortReset(ETH_PORT ethPortNum)

This routine resets the chip by aborting any SDMA engine activity and clearing the MIB counters. The Receiver and the Transmit unit are in idle state after this command is performed and the port is disabled.

INPUT

ETH_PORT ethPortNum Ethernet Port number. See ETH_PORT enum.

OUTPUT Channel activity is halted.

RETURN None.

void ethernetSetConfigReg(ETH_PORT ethPortNum, unsigned int value)

This function sets specified bits in the given ethernet configuration register.

INPUT

ETH_PORT ethPortNum Ethernet Port number. See ETH_PORT enum.

unsigned int value 32 bit value.

OUTPUT The set bits in the value parameter are set in the configuration register.

RETURN None.

void ethernetResetConfigReg(ETH_PORT ethPortNum, unsigned int value)

This function resets specified bits in the given Ethernet configuration register.

INPUT

ETH_PORT ethPortNum Ethernet Port number. See ETH_PORT enum.

unsigned int value 32 bit value.

OUTPUT The set bits in the value parameter are reset in the configuration register.

RETURN None.

unsigned int ethernetGetConfigReg(ETH_PORT ethPortNum)

This function returns the configuration register value of the given ethernet port.

INPUT

ETH_PORT ethPortNum Ethernet Port number. See ETH_PORT enum.

OUTPUT None.

RETURN Port configuration register value.

bool etherInitRxDescRing(ETH_PORT_INFO *pEthPortCtrl, ETH_QUEUE rxQueue,int rxDescNum, int rxBuffSize,unsigned int rxDescBaseAddr,unsigned int rxBuffBaseAddr)

This function prepares a Rx chained list of descriptors and packet buffers in a form of a ring. The routine must be called after port initialization routine and before port start routine. The Ethernet SDMA engine uses CPU bus addresses to access the various devices in the system (i.e. DRAM). This function uses the ethernet struct 'virtual to physical' routine (set by the user) to set the ring with physical addresses.

INPUT

ETH_PORT_INFO *pEthPortCtrl Ethernet Port Control struct.

ETH_QUEUE rxQueue Number of Rx queue.

int rxDescNum Number of Rx descriptors

int rxBuffSize Size of Rx buffer

unsigned int rxDescBaseAddr Rx descriptors memory area base addr.

unsigned int rxBuffBaseAddr Rx buffer memory area base addr.

OUTPUT The routine updates the Ethernet port control struct with information regarding the Rx descriptors and buffers.

RETURN

```
False  If the given descriptors memory area is
not aligned according to Ethernet SDMA
specifications.

True   Otherwise.
```

bool etherInitTxDescRing(ETH_PORT_INFO *pEthPortCtrl, ETH_QUEUE txQueue, int txDescNum,int txBuffSize,unsigned int txDescBaseAddr,unsigned int xBuffBaseAddr)

This function prepares a Tx chained list of descriptors and packet buffers in a form of a ring. The routine must be called after port initialization routine and before port start routine. The Ethernet SDMA engine uses CPU bus addresses to access the various devices in the system (i.e. DRAM). This function uses the ethernet struct 'virtual to physical' routine (set by the user) to set the ring with physical addresses.

INPUT

ETH_PORT_INFO *pEthPortCtrl	Ethernet Port Control struct.
ETH_QUEUE txQueue	Number of Tx queue.
int txDescNum	Number of Tx descriptors
int txBuffSize	Size of Tx buffer
unsigned int txDescBaseAddr	Tx descriptors memory area base addr.
unsigned int txBuffBaseAddr	Tx buffer memory area base addr.

OUTPUT The routine updates the Ethernet port control struct with information regarding the Tx descriptors and buffers.

RETURN

False If the given descriptors memory area is not aligned according to Ethernet SDMA specifications.

True Otherwise.

ETH_FUNC_RET_STATUS ethPortSend(ETH_PORT_INFO *pEthPortCtrl, ETH_QUEUE txQueue, PKT_INFO *pPktInfo)

This routine send a given packet described by pPktinfo parameter. It supports transmitting of a packet spanned over multiple buffers. The routine updates 'curr' and 'first' indexes according to the packet segment passed to the routine. In case the packet segment is first, the 'first' index is update. In any case, the 'curr' index is updated. If the routine get into Tx resource error it assigns 'curr' index as 'first'. This way the function can abort Tx process of multiple descriptors per packet.

INPUT

ETH_PORT_INFO *pEthPortCtrl Ethernet Port Control struct.

ETH_QUEUE txQueue Number of Tx queue.

PKT_INFO *pPktInfo User packet buffer.

OUTPUT Tx ring 'curr' and 'first' indexes are updated.

RETURN

ETH_QUEUE_FULL In case of Tx resource error.

ETH_ERROR In case the routine can not access Tx desc ring.

ETH_QUEUE_LAST_RESOURCE If the routine uses the last Tx resource.

ETH_OK Otherwise.

ETH_FUNC_RET_STATUS ethTxReturnDesc(ETH_PORT_INFO *pEthPortCtrl, ETH_QUEUE txQueue, PKT_INFO *pPktInfo)

This routine returns the transmitted packet information to the caller. It uses the 'first' index to support Tx desc return in case a transmit of a packet spanned over multiple buffer still in process. In case the Tx queue was in "resource error" condition, where there are no available Tx resources, the function resets the resource error flag.

INPUT

ETH_PORT_INFO *pEthPortCtrl Ethernet Port Control struct.

ETH_QUEUE txQueue Number of Tx queue.

PKT_INFO *pPktInfo User packet buffer.

OUTPUT Tx ring 'first' and 'used' indexes are updated.

RETURN

ETH_ERROR In case the routine can not access Tx desc ring.

ETH_RETRY In case there is transmission in process.

ETH_END_OF_JOB If the routine has nothing to release.

ETH_OK Otherwise.

ETH_FUNC_RET_STATUS ethPortReceive(ETH_PORT_INFO *pEthPortCtrl, ETH_QUEUE rxQueue, PKT_INFO *pPktInfo)

This routine returns the received data to the caller. There is no data copying during routine operation. All information is returned using pointer to packet information struct passed from the caller. If the routine exhausts Rx ring resources then the resource error flag is set.

INPUT

ETH_PORT_INFO *pEthPortCtrl Ethernet Port Control struct.

ETH_QUEUE rxQueue Number of Rx queue.

PKT_INFO *pPktInfo User packet buffer.

OUTPUT Rx ring current and used indexes are updated.

RETURN

ETH_ERROR In case the routine can not access Rx desc ring.

ETH_QUEUE_FULL If Rx ring resources are exhausted.

ETH_END_OF_JOB if there is no received data.

ETH_OK Otherwise.

ETH_FUNC_RET_STATUS ethRxReturnBuff(ETH_PORT_INFO *pEthPortCtrl, ETH_QUEUE rxQueue, PKT_INFO *pPktInfo)

This routine returns a Rx buffer back to the Rx ring. It retrieves the next 'used' descriptor and attached the returned buffer to it. In case the Rx ring was in "resource error" condition, where there are no available Rx resources, the function resets the resource error flag.

INPUT

ETH_PORT_INFO *pEthPortCtrl Ethernet Port Control struct.

ETH_QUEUE rxQueue Number of Rx queue.

PKT_INFO *pPktInfo Information on the returned buffer.

OUTPUT New available Rx resource in Rx descriptor ring.

RETURN

ETH_ERROR In case the routine can not access Rx desc ring.

ETH_OK Otherwise.

void ethPortSetRxCoal (ETH_PORT ethPortNum, unsigned int tClk, unsigned int linkSpeed, unsigned int numPackets)

This routine sets the RX coalescing interrupt mechanism parameter. This parameter is a timeout counter, that counts in 64 tClk-chunks ; that when timeout event occurs a maskable interrupt occurs.The parameter is calculated using the tCLK frequency of the MV-643xx chip, the interface speed (10 / 100 / 1000 MBps)and the required number of 64 bytes packets the RX SDMA has received.

INPUT

ETH_PORT ethPortNum	Ethernet port number
unsigned int tClk	tClk of the MV-643xx chip in HZ units
unsigned int linkSpeed	Can get values of 10 / 100 / 1000 which is the link speed in MBps units.
unsigned int numPackets	Number of packets required to be seen on RX queue upon receiving the coalescing interrupt.
OUTPUT	Interrupt coalescing mechanism value is set in MV-643xx chip.
RETURN	None.

void ethPortSetTxCoal (ETH_PORT ethPortNum, unsigned int tClk, unsigned int linkSpeed, unsigned int numPackets)

This routine sets the TX coalescing interrupt mechanism parameter.This parameter is a timeout counter, that counts in 64 tClk-chunks ; that when timeout event occurs a maskable interrupt occurs.The parameter is calculated using the tCLK frequency of the MV-643xx chip, the interface speed (10 / 100 / 1000 MBps) and the required number of 64 bytes packets the TX SDMA has transmitted.

INPUT

ETH_PORT ethPortNum	Ethernet port number
unsigned int tClk	tClk of the MV-643xx chip in HZ units
unsigned int linkSpeed	Can get values of 10 / 100 / 1000 which is the link speed in MBps units.
unsigned int numPackets	Number of packets required to be seen on TX queue upon receiving the coalescing interrupt.
OUTPUT	Interrupt coalescing mechanism value is set in MV-643xx chip.
RETURN	None.

void ethBCopy(unsigned int srcAddr, unsigned int dstAddr, int byteCount)

This function supports the eight bytes limitation on Tx buffer size. The routine will zero eight bytes starting from the destination address followed by copying bytes from the source address to the destination.

INPUT

unsigned int srcAddr	32 bit source address.
unsigned int dstAddr	32 bit destination address.
int byteCount	Number of bytes to copy.
OUTPUT	See description.
RETURN	None.

Communication Unit Serial Dynamic Memory Access Driver

There are two Serial Dynamic Memory Access (SDMA) channels on the MV-64360 that are dedicated to moving data between the serial communications channels (MPSCs) and memory buffers. Each SDMA channel consists of one Dynamic Memory Access (DMA) engine for receiving and one for transmitting. Each SDMA channel has two dedicated First In First out (FIFOs) for data buffering (for a total of four FIFOs). All FIFOs are 256 bytes deep. The SDMA channel descriptors use a chained data structure. They work without CPU interference after appropriate initialization. SDMA channels can be programmed to generate interrupts on buffer or frame boundaries. The SDMA low level driver supplies the Current Descriptor Pointer (CDP) and First Descriptor Pointer (FDP) initialization routines for the Rx and Tx descriptors per communication port (MPSC, Ethernet). The driver allocates a chain of packet descriptors in system memory for each of the ports. Each chain's starting pointer is stored in a table which is used by the low level driver in order to initiate the DMA.

Supported features

- The driver practices zero copy where no data copying is done during receive operation.
- The driver supports Tx Scatter-Gather where no copying is done when transmitting a packet that is spanned over multiple descriptors.
- Self data structure built and management. The driver carves the Rx and Tx descriptor linked list (in a form of a ring) according to parameters passed by the SDMA struct.
- The driver is protocol oriented. It considers the protocols restrictions for Tx buffer alignments and size.

Operation

The driver curves the Tx/Rx descriptor linked list in a form of a ring using the parameters in the SDMA channel struct. Those parameters are filled by the user prior to calling the sdmaChanInit() routine which initiates the drivers descriptors rings in memory. The driver's descriptors rings are managed by indexes. Those indexes controls the ring resources and are used to indicate a SW resource error:'current'. This index points to the current available resource for use. For example in Rx process this will be

the descriptor passed to the sdmaChanReceive caller containing received information from the MPSC channel. In Tx process, this will be the descriptor that will be assigned with the user packet info and transmitted.

'used' This index points to the descriptor that need to restore its resources. For example in Rx process, using sdmaRxReturnBuff will attach the buffer returned in packet info to the descriptor pointed by 'used'. In Tx process, using sdmaTxReturnDesc will merely return the user packet info with the command status of the transmitted buffer pointed by the 'used' index. Nevertheless, it is essential to use this routine to update the 'used' index.

'first' This index supports Tx Scatter-Gather. It points to the first descriptor of a packet assembled of multiple buffers. For example when in middle of such packet we have a Tx resource error the 'curr' index get the value of 'first' to indicate that the ring returned to its state before trying to transmit this packet.

Receive operation

The sdmaChanReceive API return the caller the packet information struct pointer describing the current SDMA descriptor containing the received information. It is the user responsibility to return this resource back to the Rx descriptor ring to enable the reuse of this source. Return Rx resource is done using the sdmaRxReturnBuff API.

Transmit operation

The sdmaChanSend API supports Scatter-Gather which enables to send a packet spanned over multiple buffers. This means that for each packet info structure given by the user and put into the Tx descriptors ring, will be transmitted only if the 'LAST' bit will be set in the packet info command status field. This API also considers the protocol used and its restriction regarding buffer alignments and sizes. The user must return a Tx resource after ensuring the buffer has been transmitted to enable the Tx ring indexes to update.

Software Modules

This driver is implemented in:

- sdma.c
- sdma.h

sdma.c SDMA register manipulation.

sdma.h SDMA function and structure declaration.

SDMA Low Level Driver Features

The SDMA Low Level Driver Features are:

- Full control over SDMA configuration registers
- Changing ports configuration before and after initialization
- SDMA engines initialization
- The driver is Operating System Independent

System Resource Usage

The driver uses the two SDMA channels.

External Interface (Low Level Driver)

This section details the External Interface (Low Level Driver).

Low Level Driver Data Structure

The following tables details the Low Level Driver Data structure.

Table 6:*SDMA Driver Structure:RX_COMMAND*

Macro	Field	Description
OWNER_BY_GT	O	When set to ‘1’ the buffer is is “owned” by the device. When set to ‘0’ the buffer is owned by the CPU.
	AM	Auto Mode When set, the DMA does not clear the Ownership bit at the end of buffer processing.
	reserved1	

Table 6:*SDMA Driver Structure:RX_COMMAND*

Macro	Field	Description
ENABLE_INTERRUPT	EI	The device generates a maskable interrupt upon closing the descriptor. NOTE: To limit the number of interrupts and prevent an interrupt per buffer situation, set the EI bits in all the Rx descriptors and set RIFB bit in the DMA Configuration register. The RxBuffer interrupt is set only on frame (rather than buffer) boundaries.
	reserved2	
FIRST	F	Indicates first buffer of a packet.
LAST	L	Indicates last buffer of a packet.
ERROR_SUMMARY	ES	ES = CE or COL or LC or OR or MFL or SF NOTE: Valid only if F (bit 17) is set. These bits have different meanings for each protocol. See the MV-64360 datasheet for details.
	bites14_0	

Table 7:*SDMA Driver Structure: TX_COMMAND*

Macro	Field	Description
OWNER_BY_GT	O	When set to ‘1’ the buffer is is “owned” by the device. When set to ‘0’ the buffer is owned by the CPU. Buffers owned by the CPU are not processed by the DMA.
	AM	Auto Mode When set, the DMA does not clear the Ownership bit at the end of buffer processing.

Table 7:*SDMA Driver Structure: TX_COMMAND (cont.)*

Macro	Field	Description
	reserved1	
ENABLE_INTERRUPT	EI	The device generates a maskable interrupt upon closing the descriptor. **Note: To limit the number of interrupts and prevent an interrupt per buffer situation, set this bit only in descriptors associated with LAST buffers. If this is done, Tx Buffer interrupt is set only when transmission of a frame is completed.**
	GC	Generate CRC When set, CRC is generated and appended to this packet. **Note: Valid only if L (bit16) is set.**
	reserved2	
PADDING	P	Padding When set, zero bytes are appended to the packet if the packet is smaller than 60 bytes. Use this feature to prevent transmission of fragments. NOTE: Valid only if L (bit 16) is set.
FIRST	F	Indicates first buffer of a packet.

Table 7:*SDMA Driver Structure: TX_COMMAND (cont.)*

Macro	Field	Description
LAST	L	Indicates last buffer of a packet.
ERROR_SUMMARY	ES	ES = LC or UR or RL Set by the device to indicate an error event that occurred during packet transfer. NOTE: Valid only if L (bit 16) is set. These bits have different meanings for each protocol. See the MV-64360 specifications for details.
	bites14_0	

Table 8:*SDMA Driver Structure: RX_DESC*

Macro	Field	Bit Width	Description
UINT	Bufsize	16	When set to '1' the buffer is "owned" by the device. When set to '0' the buffer is owned by the CPU.
UINT	bytecnt	16	When the descriptor is closed this field is written by the device with a value indicating number of bytes actually written by the DMA in to the buffer.
RX_COMMAND	cmd_sts	32	
UINT	next_desc_ptr	32	32-bit Next Descriptor Pointer to the Beginning of Next Descriptor Bits [3:0] must be set to 0. DMA operation is stopped when a NULL value in the Next Descriptor Pointer field is encountered.

Table 8:*SDMA Driver Structure: RX_DESC (cont.)*

Macro	Field	Bit Width	Description
UINT	buf_ptr	32	32-bit Pointer to The Beginning of the Buffer Associated with The Descriptor RX buffers has to be 64-bit aligned, so bits [2:0] must be set to 0.
UINT	Index-ToRx Queue	4	An extra field that determines the queue number to release a descriptor after it has been used.

Table 9:*SDMA Driver Structure: TX_DESC*

Macro	Field	Bit Width	Description
UINT	bytecnt	16	Byte count is the number of bytes to be transmitted. Zero byte counters are not supported with retransmission. Do not use zero byte buffers with LAP-D protocol.
UINT	Shadow	16	The CPU must initialize this field with a value identical to the Byte Count field. The MV-64360 subtracts the number of bytes actually transmitted from this parameter. Usually the MV-64360 writes “0” in this field when closing a descriptor. However, when the transmit SDMA halts due to a transmit error, this number can be used to determine the number of bytes that were fetched into the MV-64360. Setting both the Byte Count and Shadow Byte Count to “0” will cause the SDMA to close the descriptor and move to the next descriptor, if both or neither of the F and L bits are set. Setting Byte Count and Buffer Size to “0” in transmit descriptors with one of the F or L bits set will lead to unpredictable behavior.
TX_COMMAND	cmd_sts	32	

Table 9:*SDMA Driver Structure: TX_DESC (cont.)*

Macro	Field	Bit Width	Description
UINT	next_desc_ptr	32	32-bit pointer that points to the beginning of next descriptor. Bits [3:0] must be set to 0. DMA operation is stopped when a NULL (all zero) value in the Next Descriptor Pointer field is encountered.UINT buf_ptr 32 32-bit pointer to the beginning of the buffer associated with this descriptor. **Note: The alignment restrictions for buffers that have Byte-Count smaller than 8 bytes**
UINT	Pointer to Rx Queue	32	An extra field to determine the source queue where the packet was taken from. Release the source descriptor after receiving TxEnd Interrupt.
UINT	shadowOwner	1	An extra field to help manage the Tx descriptors.

Table 10:*SDMA Driver Structure: MPSC_SDCMR*

Macro	Field	Description
	AT	Abort Transmit The CPU sets the TRD bit to '1' when it needs to abort a transmit SDMA channel operation. When the TRD bit is set, the SDMA aborts its operation and goes to IDLE state. No descriptor is closed. The MV-64360 clears both the TRD and TXD bits when entering IDLE state. The CPU must poll bit 31. When it is '0' the MV-64360 has completed the abort sequence. After an abort, the CPU must write the first descriptor address and than set TXD bit to '1'.
	reserved4	
TX_DEMAND	TXD	Tx Demand When this bit is set to '1', the Tx DMA will fetch the first descriptor and will start the transmission process. The MV-64360 clears TXD when it successfully ends an SDMA transmit process. It also clears TXD when a resource error occurs, when the transmit process is halted due to channel error (i.e. CTS# lost), or when the CPU issues an abort command.
	reserved3	

Table 10:*SDMA Driver Structure: MPSC_SDCMR (cont.)*

Macro	Field	Description
	STD	Stop Tx The SDMA stops transmission at the end of frame (i.e. at the end of buffer with L bit set to '1'). After transmitting the last buffer, the transmit SDMA goes to IDLE state. The MV-64360 clears the TXD bit when entering IDLE state. After the SDMA stops, the CPU must write the first descriptor address and than set the TXD bit to '1'. The MV-64360 signals the CPU with Interrupt when the stop procedure is accomplished.
ABORT_RECEIVE	AR	Abort Receive The CPU sets the AR bit when it needs to abort a receive SDMA channel operation. When the AR bit is set, the SDMA aborts its operation and goes to IDLE state. No descriptor is closed. The MV-64360 clears both the AR and ERD bits when entering IDLE state. The CPU must poll bit 15. When it is '0', the MV-64360 has completed the abort sequence. After an abort the CPU should write the 1st descriptor address and then set ERD bit to '1'.
	reserved2	

Table 10:*SDMA Driver Structure: MPSC_SDCMR (cont.)*

Macro	Field	Description
ENABLE_RX_DMA	ERD	Enable Rx DMA When set to '1', the Rx SDMA will fetch the 1st descriptor and will be ready for a receive frame. The MV-64360 clears ERD when the MV-64360 receive SDMA has a resource error or when the CPU issues an abort command.
	reserved1	

Table 11:*SDMA Driver Structure: SDMA_CONFIGURATION*

Macro	Field	Description
	reserved0	
BURST_SIZE	BSZ	Burst Size Sets the maximum burst size for SDMA transactions: 00 - Burst is limited to 1 64bit words. 01 - Burst is limited to 2 64bit words. 10 - Burst is limited to 4 64bit words. 11 - Burst is limited to 8 64bit words.
	reserved1	
	RIFB	RIFB Receive Interrupt on Frame Boundaries When set, the SDMA Rx generates interrupts only on frame boundaries (i.e. after writing the frame status to the descriptor).
	POVR	PCI Override When set, causes the SDMA to direct all its accesses in PCI_0 direction, overriding normal address decoding process.

Table 11:*SDMA Driver Structure: SDMA_CONFIGURATION (cont.)*

Macro	Field	Description
	BLMT	Tx Big / Little Endian Transmit Mode The MV-64360 supports big or little endian configuration per channel for maximum system flexibility. The BLMT bit only affects data movements. 0 - Big Endian convention. 1 - Little Endian convention.
	BLMR	Rx Big Little / Endian Receive Mode The MV-64360 supports big or little endian configuration per channel for maximum system flexibility. The BLMR bit only affects data movements. 0 - Big Endian convention. 1 - Little Endian convention
	RC	Retransmit Count In collision modes (LAP-D), after executing a backoff procedure RC times, the Tx SDMA will close the buffer with a Retransmit Limit (RL) error, a maskable interrupt will be generated, and the SDMA will go to OFF state. A new Transmit Demand command should be issued in order to start a new transmission process. When RC field is 0000, the MV-64360 tries to retransmit forever. The CPU needs to issue an abort command in order to stop the retransmit process.

Table 11:*SDMA Driver Structure: SDMA_CONFIGURATION (cont.)*

Macro	Field	Description
SINGLE_FRAME_MODE	SFM	0 - Multi frame mode 1 - Single frame mode **Note:** • **The SFM bit must be set to '1' for HDLC Collision mode, and BISYNC protocols. It is also recommended for UART.** • **When the SFM bit is set to '0', CTS Lost cannot be reported in the correct descriptor/frame. In LAN-HDLC mode SFM must be set for proper operation.**
RECEIVE_FIFO_THRESHOLD	RFT	0 - 8 bytes 1 - Half FIFO (128 bytes) **Note: When working with an 8-bit data path, the threshold is always one byte regardless of the RFT value. It is recommended that RFT bit be set to '0' in this case. When RFT is set to '0', the SDMA will not burst. It will transfer one word (64 bits) on each transfer.**

External Interface APIs

void sdmaChanInit(SDMA_CHANNEL *sdmaChan)

DESCRIPTION	This function completes the SDMA channel struct initialization and carves the Rx and Tx descriptor / buffers data structures in memory according to user parameters passed by the SDMA channel struct.
INPUT	
SDMA_CHANNEL	
***sdmaChan**	SDMA channel struct
OUTPUT	SDMA channel struct initialized and SDMA channel descriptors are ready.
RETURN	None.

void sdmaChanStart(SDMA_CHANNEL *sdmaChan)

DESCRIPTION	This function start the SDMA activity. The routine assigns the SDMA struct values to the SDMA channel registers and enables the Rx.
INPUT	
SDMA_CHANNEL	
***sdmaChan**	SDMA channel struct
OUTPUT	SDMA channel register are initiated and Rx is enabled.
RETURN	None.

void sdmaChanStopRx(SDMA_CHANNEL *sdmaChan)

DESCRIPTION	This function halts SDMA Rx activity.

INPUT

SDMA_CHANNEL

***sdmaChan** SDMA channel struct

OUTPUT If Rx enabled is set in SDMA command register, an abort Rx command is issued and register is polled until the abort command is complete.

RETURN None.

void sdmaChanStopTx(SDMA_CHANNEL *sdmaChan)

DESCRIPTION This function halts SDMA Tx activity.

INPUT

SDMA_CHANNEL

***sdmaChan** SDMA channel struct

OUTPUT If Tx demand is set in SDMA command register, a stop Tx command is issued and register is polled until the stop command is complete.

RETURN None.

void sdmaChanStopTxRx(SDMA_CHANNEL *sdmaChan)

This routine halts both Rx and Tx activities of a given SDMA channel.

INPUT

SDMA_CHANNEL

***sdmaChan** SDMA channel struct

OUTPUT Rx and Tx activity are stopped.

RETURN None.

bool sdmaInitRxDescRing(SDMA_CHANNEL *sdmaChan, int rxDescNum, int rxBuffSize, unsigned int rxDescBaseAddr, unsigned int rxBuffBaseAddr)

DESCRIPTION	This function prepares a Rx chained list of descriptors and packet buffers in a form of a ring.
INPUT	
SDMA_CHANNEL	
***sdmaChan**	SDMA channel struct
int rxDescNum	Number of Rx descriptors
int rxBuffSize	Size of Rx buffer
unsigned int rxDescBaseAddr	Rx descriptors memory area base addr.
unsigned int rxBuffBaseAddr	Rx buffer memory area base addr.
OUTPUT	The routine updates the SDMA channel struct with the information regarding the Rx descriptors and buffers.
RETURN	
	`False If the given descriptors memory area is not aligned according to SDMA specifications.`
	`True When given descriptors memory area is aligned according to SDMA specifications`

bool sdmaInitTxDescRing(SDMA_CHANNEL*sdmaChan, int txDescNum, int txBuffSize, unsigned int txDescBaseAddr, unsigned int txBuffBaseAddr)

DESCRIPTION	This function prepares a Tx chained list of descriptors and packet buffers in a form of a ring.

INPUT

SDMA_CHANNEL

***sdmaChan**	SDMA channel struct
int txDescNum	Number of Tx descriptors
int txBuffSize	Size of Tx buffer
unsigned int txDescBaseAddr	Tx descriptors memory area base addr.
unsigned int txBuffBaseAddr	Tx buffer memory area base addr
OUTPUT	The routine updates the SDMA channel struct with the information regarding the Tx descriptors and buffers.

RETURN

`False If the given descriptors memory area is not aligned according to SDMA specifications.`

`True If the given descriptors memory area is aligned according to SDMA specifications.`

SDMA_STATUS sdmaChanSend(SDMA_CHANNEL *sdmaChan, PKT_INFO *pPktInfo)

DESCRIPTION	This routine send a given packet described by pPktinfo parameter. It support a packet span over multi buffers. The routine updates 'curr' and 'first' indexes according to the descriptor given. In case the descriptor is "first", the 'first' index is update. In any case, the 'curr' index is updated. If the routine get into Tx resource error it assigns 'curr' index as 'first'. This way the function can abort Tx process of multiple descriptors per packet.

INPUT

SDMA_CHANNEL *sdmaChan	SDMA channel struct pointer

PKT_INFO *pPktInfo Packet struct pointer.

OUTPUT Tx ring 'curr' and 'first' indexes are updated.

RETURN

SDMA_QUEUE_FULL In case of Tx resource error.

SDMA_ERROR In case the routine can not access Tx desc ring.

SDMA_QUEUE_LAST_RESOURCE If the routine uses the last Tx resource.

SDMA_OK Otherwise.

SDMA_STATUS sdmaTxReturnDesc(SDMA_CHANNEL *sdmaChan, PKT_INFO *pPktInfo)

DESCRIPTION This routine returns the transmitted packet information to the caller. It updates the 'used' Tx ring index. In case the Tx queue was in "resource error" condition, where there are no available Tx resources, the function resets the resource error flag.

INPUT

SDMA_CHANNEL *sdmaChan SDMA channel struct pointer

PKT_INFO *pPktInfo Packet struct pointer.

OUTPUT Tx ring current and used indexes are updated.

RETURN

SDMA_ERROR In case the routine can not access Tx desc ring.

SDMA_RETRY In case the routine could not release descriptor.

SDMA_END_OF_JOB`If the routine has nothing to release.

SDMA_OK Otherwise.

SDMA_STATUS sdmaChanReceive(SDMA_CHANNEL *sdmaChan, PKT_INFO *pPktInfo)

This routine returns the received data to the caller. There is no data copying during routine operation. All information is returned using pointer to packet information struct. If the routine exhausts the Rx ring resources the resource error flag is set.

INPUT

SDMA_CHANNEL *sdmaChan SDMA channel struct pointer

PKT_INFO *pPktInfo Packet struct pointer.

OUTPUT Rx ring current and used indexes are updated.

RETURN

SDMA_ERROR In case the routine can not access Rx desc ring.

SDMA_QUEUE_FULL If Rx ring resources are exhausted.

SDMA_END_OF_JOB If there is no received data.

SDMA_OK Otherwise.

SDMA_STATUS sdmaRxReturnBuff(SDMA_CHANNEL *sdmaChan, PKT_INFO *pPktInfo)

DESCRIPTION This routine returns a Rx buffer back to the Rx ring. It retrieves the next 'used' descriptor and attached the returned buffer to it. In case the Rx queue was in "resource error" condition, where there are no available Rx resources, the function resets the resource error flag.

INPUT

SDMA_CHANNEL *sdmaChan SDMA channel struct pointer

PKT_INFO *pPktInfo Packet struct pointer.

OUTPUT New available Rx resource in Rx descriptor ring.

RETURN

SDMA_ERROR In case the routine can not access Rx desc ring.

SDMA_OK Otherwise.

Driver Introduction

This module allocates the descriptors for each MPSC and each Ethernet port (separate chain for each priority) depending on the protocol programmed for this port. The descriptors allocating method defines the minimum and maximum number of descriptors per port protocol. Each protocol allows a limited number of descriptors to avoid one port consuming all the resources. Each descriptor chain allocation results in the addition of the chain's first pointer to a table of pointers that is used by the Low Level Driver to initialize the DMA.
Tx descriptors are allocated for each port (four Tx DMA exist) with Buffer Pointer NULL.

Software Modules

This driver is implemented in:

- sdma.c
- sdma.h

sdma.c SDMA memory allocation and transmit packet routines.

sdma.h SDMA structs and function declaration.

Restriction

The memory is limited in size and hence there is a limit to the number of descriptors.

System Resource Usage

This driver uses the user memory pool for descriptors and buffers allocation.

External Interface

This section details the external interface.

Driver Data Structure

After system initialization, the sdmaAllocationTable structure has the first allocation of the first descriptor of each queue.

Driver External Interface APIs

The following tables detail the SDMA channel structures.

Table 12:*SDMA Channel Structure: PortAllocStruct*

Type	Field	Bit Width	Description
UINT32	portNumber	4	The number of the port.
UINT32	protocolType	3	ETHERNET_PROTOCOL or MPSC_PROTOCOL.
UINT32	priority	2	The queue priority PRIO0-PRIO3.
UINT32	numberOfDescriptors		The number of descriptors to allocate

Table 12:*SDMA Channel Structure: PortAllocStruct*

Type	Field	Bit Width	Description
UINT32	bufferSize		The size of the buffer to allocate.
UINT32	rx Or Tx	1	The allocation for RX_DESCRIPTOR or TX_DESCRIPTOR.

Table 13:*SDMA Channel Structure: TX_PACKET*

Type	Field	Bit width	Description
UINT32	portNumber	4	The Port number.
UINT32	PacketSize	12	Packet Size
UINT32	appendCrc	1	
UINT32	priority	1	
UINT32	destHigh	16	
UINT32	destLow		
UINT32	srcHigh	16	
UINT32	srcLow		
UINT32	pattern		

STATUS sdmaAllocateDescriptorsForOnePort (PORT_ALLOCATION_STRUCT* portAllocationStruct)

DESCRIPTION Allocates descriptors for each port. It writes the first descriptor pointer to a table that is used by the low level for DMA initialization.

INPUT

PORT_ALLOCATION_STRUCT* portAllocationStruct PORT_ALLOCATION_STRUCT*

OUTPUT Writes the first allocated descriptor pointer to a table that is used by the low level for DMA initialization

RETURN

OK On success.

ERROR On failure to make the assignment.

STATUS sdmaSendPackets (UINT32 protocolType,UINT32 priority,UINT32 portNumber, UINT32*cmdAddress)

DESCRIPTION Sends packets from the CPU to any port in any MPSC protocol (or ethernet). Before calling this function, you must point to a valid buffer and set the byte count and packet description in the linked descriptors.

Note: Set the descriptors before calling this function.

INPUT

UINT32 ProtocolType ETHERNET_PROTOCOL or MPSC_PROTOCOL.

UINT32 Priority 0 for low, 1 for high.

UINT32 PortNumber The number of the port, ETHERNET(0,1,2), MPSC(0-1).

OUTPUT Packet sending.

RETURN

OK On success.

ERROR On failure to make the assignment.

STATUS sdmaReleaseRxDesc (RX_DESC* rxDescStruct)

DESCRIPTION Called by the application after a packet is received and was processed. Return the used descriptor back to the descriptors chain and puts it at the end.

INPUT

RX_DESC* rxDescStruct Pointer to descriptor struct to be released.

OUTPUT Descriptor is returned to the end of descriptor chain.

RETURN

OK On success.

ERROR On failure to make the assignment.

STATUS sdmaTransmitPackets (TX_PACKET packetDetails, int numberofPackets)

DESCRIPTION Sends the programmed number of MAC packets to the port specified in packetsDetails.

INPUT

TX_PACKET Packet Details

OUTPUT Packets sent out.

RETURN

OK On success.

ERROR On failure to make the assignment.

Communication Unit MPSC Driver

The MV-64360 system controller includes two MPSCs that support:

- Bit oriented protocols (for example HDLC)
- Byte oriented protocols (for example BISYNC)
- Transparent protocols
- The Universal Asynchronous Receiver Transmitter (UART)
- (Start / Stop) mode

- All two MPSCs can operate simultaneously and can be routed out via serial interface ports which implement interfaces such as EIA-232 and V.34.

Low Level Driver Introduction

The MPSC Low Level Driver routines initialize the MPSC main configuration registers, channel registers, and protocol configuration registers.

Software Modules

This driver is implemented in:

- mpsc.c
- mpsc.h

Low Level Driver External Interface- Data Structure

MPSC Main Structure: MPSC_MAIN_STRUCT

The table details the MPSC Main structure.

Note: This structure consists of two UINT32s (High and Low) because it combines the Low and High registers. The manipulated bits are identified through macros.

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT*

Bits	Field	Bit Width	Description
Low			
31	GDE	1	Glitch Detect Enable 0 - Normal mode. No glitch detect. (Default) 1 - When glitch is detected, a maskable interrupt is generated. When this bit is set the MPSC looks for glitches in the external receive and transmit clocks. **Note: The MV-64360 tries to clean the input clocks by receiving them via a Schmitt trigger input buffer.**
30	reserved3	1	
29	RRVD	1	Receive Reverse Data 0 - Normal Mode. (Default) 1 - Reverse Data Mode. MSB is shifted in first.
28	TRVD	1	Transmit Reverse Data 0 - Normal Mode. (Default) 1 - Reverse Data Mode. MSB is shifted out first.
27:26	reserved4	2	
25:23	CRCM	3	CRC Mode 000 - CRC16-CCITT (HDLC based protocols, e.g. X.25) (Default) 001 - CRC-16 (BISYNC) 010 - CRC32-CCITT (HDLC based protocols, e.g. LAP-D. Identical to the Ethernet CRC) 011 - Reserved 1XX- Reserved

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT (cont.)*

Bits	Field	Bit Width	Description
22	CDM	1	CD* Operating Mode 0- Normal mode. (Envelop Mode). CD* should envelop the frame. Deassertion of CD* during reception will cause a CD lost error. 1- Pulse Mode. Once CD* is sampled low, synchronization has been achieved. Further transitions of CD* have no effect.
21	CTSM	1	CTS* Operating Mode 0 - Normal mode. (Envelop Mode). CTS* should envelop the frame. Deassertion of CTS* during transmission will cause a CTS lost error. 1- Pulse Mode. Once CTS* is sampled low, synchronization has been achieved. Further transitions of CTS* have no effect. CTS* synchronization will be lost when RTS* is deasserted.
20	CDS	1	CD* Sampling mode 0 - Asynchronous CD*. CD* is synchronized internally in the MV-64360 and then data is received. (Default) 1 - Synchronous CD*. CD* is synchronized to the Rx clock. This mode is recommended when connecting an MPSC to a Flex-TDM.
19	CTSS	1	CTS* Sampling Mode 0 - Asynchronous CTS*. CTS* is synchronized inside the MV-64360. Transmission starts after synchronization is achieved with a few cycles delay to the external CTS*. (Default) 1 - Synchronous CTS*. CTS* is synchronized to the Rx clock. This mode is recommended when connecting an MPSC to a FlexTDM. Synchronous CTS* must be used for ISDN D channels.
18	reserved5	1	

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT (cont.)*

Bits	Field	Bit Width	Description
17	RTSM	1	RTS* Mode This bit may be changed on the fly. 0 - Send IDLE between frames. RTS* is negated between frames. IDLE pattern is defined by the protocol and TIDL bit. 1- Send flags / syncs between frames according to the protocol. RTS* is always asserted.
16	TIDL	1	Transmit Idles 0 - TxD is encoded during data transmission (including preamble and flags / sync patterns). TxD is in MARK during idle. (Default.) 1 - TxD is encoded all the time, even when idles are transmitted.
15:14	TSNS	2	Transmit Sense. Defines the number of bit times the internal sense signal will stay active after last transition on the RXD line occurs. It is useful for AppleTalk protocol to avoid the spurious CD* change interrupt that would otherwise occur during the frame synchronization sequence that precedes the opening flag. The delay is a function of RCDV (clock divider) setting. 00 (RCDV = 0) - Infinite (Carrier Sense is always active - default) 00 (RCDV _ 0) - Infinite (Carrier Sense is always active - default) 01 (RCDV = 0) - 14 bit times 01 (RCDV _ 0) - 6.5 bit times 10 (RCDV = 0) - 4 bit times (normal AppleTalk) 10 (RCDV _ 0) - 2.5 bit times (normal AppleTalk) 11 (RCDV = 0) - 3 bit times 11 (RCDV _ 0) - 1 bit time
13	reserved6	1	

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT (cont.)*

Bits	Field	Bit Width	Description
12	TSYN	1	Transmitter Synchronize to Receiver Setting this bit synchronizes the transmitter to receiver byte boundaries. This is particularly important in the X.21 protocol. 0 - No synchronization assumed. 1 - Transmit bit stream is synchronized to the receive bit stream. This bit affects only a transparent transmitter. Transmitter will start transmission nx8 bit period after the receive data arrives. If CTS* is already asserted, the transparent transmitter will start transmit 8 clocks after the receiver starts to receive data. **Note: Only this bit when transmit and receive clocks are equal and TCDV and RCDV are set to '00'.**
11	reserved7	1	
10	NLM	1	Null Modem 0 - Normal operation. The MPSC uses the CD* and CTS* inputs to control the data flow 1 - Null Modem. The MPSC CD* and RTS* internal signals are always asserted. The external pin status can be still read from the Event Register.

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT (cont.)*

Bits	Field	Bit Width	Description
9:8	LPBK	2	Loop Back (for diagnostic) mode 00 -Normal Operation, no loopback (Default) 01 -Loopback 10 -Echo 11 -Loop Back + Echo In loopback mode, transmitted data on TxD is also fed into RxD, mainly for diagnostic purposes. In this mode, the same clock source should be used for both Rx and TX. Echo mode re-transmits received data on RxD (with one clock delay) on TxD. If CD* is asserted, the receiver also receives the incoming data.
7	ER	1	Enable Receive 0 - Disabled. The Rx channel is in Low Power Mode. 1 - Enable. The Rx controller is ready to receive data.
6	ET	1	Enable Transmit 0 - Disabled. The Tx channel is in Low Power Mode. 1 - Enable. The Tx controller is ready for data. When the SDMA has data to transmit it loads the data to the Tx controller, that will transmit the data in the selected protocol.
5	reserved8	1	
4	TRX	1	Transparent Receiver 0 - Normal Mode (default) 1 - Transparent Mode. (Transparent Mode overrides the program mode in MODE bits)
3	TTX	1	Transparent Transmitter 0 - Normal Mode. (default) 1 - Transparent Mode. (Transparent Mode overrides the program mode in MODE bits)

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT (cont.)*

Bits	Field	Bit Width	Description
2:0	MODE	3	Mode 000 -HDLC (default) 001 -Reserved 010 -Reserved 011 -Reserved 100 -UART 101 -BISYNC 110 -Reserved 111 -Reserved
High			
31:30	SEDG	2	Synchronization Clock Edge The clock edge used by the DPLL for adjusting the receive sample point due to drift in the receive signal. 00 - Both rising and falling edges. (Default.) 01 - Rising edge 10 - Falling edge 11 - No adjustment
29:27	RENC	3	Receive Encoder Specifies the encoding method for the dedicated Rx channel DPLL. 000 - NRZ (default) 001 - NRZI (Mark, can be set to Space by setting RINV bit) 010 - FM0 (can be set to FM1 by setting the RINV bit) 011 - Reserved 100 - Manchester 101 - Reserved 110 - Differential Manchester 111 - Reserved

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT (cont.)*

Bits	Field	Bit Width	Description
26:25	RCDV	2	Receive Clock Divider Defines the receive clock divider. The receive bit rate is the rate of the clock entering the MPSC Rx machine (from external pin or a BRG) divided by the RCDV field. For FM0, FM1, Manchester, and Differential Manchester, one of the 8x, 16x, or 32x options must be set. 00 - 1x clock mode (Default. For NRZ and NRZI only.) 01 - 8x clock mode 10 - 16x clock mode 11 - 32x clock mode
24:23	RSYL	2	Receive Sync Length (BISYNC and Transparent Modes) 00 - External sync (CD* assertion) 01 - 4-bit sync 10 - 8-bit sync (MonoSYNC) 11 - 16-bit sync (BISYNC)

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT (cont.)*

Bits	Field	Bit Width	Description
22	RDW	1	Receive Data Width 0 - Normal mode. The MPSC data path is 16 bits wide. Upon receiving 16 bits, the data is transferred into the SDMA FIFOs. Buffers must be 64-bit word aligned. DMA bursts are enabled. **Note:** • **Normal Mode must be used for HDLC based protocols. 1 - Low latency operation. Data is transferred to the FIFOs after 8 bits are received. Logical FIFO width is one byte.** • **This mode allows byte aligned buffers. This mode must be chosen for BISYNC and UART modes. DMA bursts are disabled. The SDMA writes one byte per DRAM access. Setting RDW also bypasses the receive FIFO threshold. The SDMA arbitrates for DMA access as soon as the FIFO has one byte in it.**
21	reserved1	1	
20:17	GDW	4	Clock Glitch Width When the GDE bit is set, the MPSC will consider Tx/Rx clock pulses that are narrower than GDW system clocks as a glitch.

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT (cont.)*

Bits	Field	Bit Width	Description
16	RINV	1	Receive Bit Stream Inversion. 0 - No invert. 1 - Inverts the data before it is sent from the DPLL to the MPSC data path. Setting RINV to '1' decodes FM1 and NRZI mark when the RENC field is programed to FM0 and NRZI space etc. It also inverts the received bit stream in NRZ mode.
15:14	reserved2	2	
13:11	TDEC	3	Transmit Encoder Specifies the encoding method for the dedicated Tx channel DPLL. 000 - NRZ (default) 001 - NRZI (mark, can be set to Space by setting TINV bit) 010 - FM0 (can be set to FM1 by setting the TINV bit) 011 - Reserved 100 - Manchester 101 - Reserved 110 - Differential Manchester 111 - Reserved
10:9	TCDV	2	Transmit Clock Divider Defines the transmit clock divider. The transmit bit rate is the rate of the clock entering the MPSC Tx machine (from external pin or a BRG) divided by the TCDV field. For FM0, FM1, Manchester, and Differential Manchester, one of the 8x, 16x, or 32x options must be set. 00 - 1x clock mode (Default. For NRZ and NRZI only.) 01 - 8x clock mode 10 - 16x clock mode 11 - 32x clock mode
8:5	TPPT	4	TPPT 4 Transmit Preamble Pattern Defines a character sent as a preamble sequence. Two TPPT characters form a preamble byte. The number of preamble bytes sent is defined by the TPL field. The receiving DPLL uses the preamble pattern to lock on the receiving signal.

Table 14:*MPSC Main Structure: MPSC_MAIN_STRUCT (cont.)*

Bits	Field	Bit Width	Description
4:2	TPL	3	Transmit Preamble Length Determines the number of preamble bytes the transmitter sends before it starts to transmit data. The send pattern is defined by the TPPT bits. 000 - No Preamble (Default) 001 - 1 byte 010 - 2 bytes 011 - 4 bytes 100 - 6 bytes 101 - 8 bytes 110 - 16 bytes 111 - Reserved
1	TINV	1	1 TINV 1 Transmit bit stream inversion 0 - No invert. 1 - Invert the data before it is sent to the DPLL. Setting TINV to '1' generates FM1 from FM0, NRZI mark from NRZI space, etc. It also inverts the bit stream in NRZ mode.
0	TCL	1	Transmit Clock Invert 0 - Normal operation - Data is shifted out on the falling edge. (Default.) 1 -The internal transmit clock is inverted by the MPSC before it is used. This allows the MPSC to clock data out half a cycle earlier on the rising edge of the clock.

Table 15:*MPSC Channel Structures: MPSC_Channel_Chr 1 to 10*

Macro	Field	Description
CHR1		
	reserved	
	abort	
	reserved2	
	SYNC	Holds the synchronization pattern for the receive machine and opening / closing flag / sync-pattern for the transmit machine. The abort pattern is transmitted upon receiving an abort command. This is an HDLC flag so no additional programing is needed for the HDLC protocol. After reset it holds the value of 7E in the SYNC field
CHR2		
ENTER_HUNT	EH	Upon receiving the Enter Hunt command, the receive machine moves to HUNT state and continuously searches for an opening flag. If the enter hunt mode command is issued during frame reception, the current descriptor is closed with CRC error 1. The EH bit is cleared upon entering Hunt state.
	reserved1	
ABORT_RECEPTION	AbortReception	Abort receive immediately and go to IDLE. The descriptor is not closed or incremented. The processor must issue enter hunt command after abort command in order to enable reception. The bit is cleared upon entering IDLE state.
	reserved2	
	A	
	reserved3	
CHR3		
	reserved1	

Table 15:*MPSC Channel Structures: MPSC_Channel_Chr 1 to 10 (cont.)*

Macro	Field	Description
	FLBR	Frame Length Buffer Register Holds the maximum allowed frame length. When a frame exceeds the number written in the FLBR, the remainder of the frame is discarded. The HDLC controller waits for a closing flag and then Return the frame status with bit 7 (MFLE) set to '1'.
CHR4		
	B	Broadcast Enable Enables the reception of HDLC broadcast address (0xFFFF or 0xFF, depending on the BCE setting).
	reserved1	
	N	Null Enable Enables the reception of HDLC NULL address (0x0000 or 0x00 depending on the BCE setting)
	reserved	
	BCE	BCE Bit Comparison Enable Bits Setting '1' in one of the BCE bits enables the address comparison for this bit: ² For 16-bit LAP-D like address recognition, write 0xFFFF in ADFR. ² For 8-bit HDLC/LAP-B like address recognition, write 0x00FF in ADFR. ² For reception of a predefined address group, write '0' to the appropriate bits to disable address comparison on these bits.
CHR5		
	reserved	
	SHFR	SHFR Short Frame Register Setting SHFR to '1' enables the Short Frame Error report. Short Frames are frames with byte count less than 3+SHFR.

Table 15:*MPSC Channel Structures: MPSC_Channel_Chr 1 to 10 (cont.)*

Macro	Field	Description
CHR6		
	AD2	Address 2 A 16-bit address used for receive address recognition.
	AD1	Address 1 A 16-bit address that can be used for receive address recognition.
CHR7		
	AD4	Address 4 A 16-bit address used for receive address recognition.
	AD3	Address 3 A 16-bit address that can be used for receive address recognition.
CHR8		
	reserved1	
CHR9		
	reserved1	
CHR10		
	reserved1	
	RRF	1 = Rx Receiving Flags.
	DPCS	1 = DPLL Carrier Sense.
	RLIDL	1 = Rx IDLE Line
	reserved2	
RX_ENTER_HUNT_STATE	RHS	Rx in HUNT state.
	reserved3	
	TIDLE	Tx in IDLE state. An interrupt is generated upon entering IDLE state.

Table 15:*MPSC Channel Structures: MPSC_Channel_Chr 1 to 10 (cont.)*

Macro	Field	Description
	reserved4	
	CD	Carrier Detect Signal An interrupt is generated when this signal is deasserted during receive.

Table 16:*MPSC Channel Structure: MPSC_CHANNEL_STRUCTURE*

Type	Field
CHR1	chr1
CHR2	chr2
CHR3	chr3
CHR4	chr4
CHR5	chr5
CHR6	chr6
CHR7	chr7
CHR8	chr8
CHR9	chr9
CHR10	chr10

Driver Introduction

This module initializes the MPSC according to the user demand and the protocol assigned to the port. If a BRG facility is in use, BRG initialization is completed. The driver supports all MPSC working configurations. The default is HDLC protocol.

Implementation Files

This driver is implemented in:

- mpsc.c
- mpsc.h

mpsc.c Initialize MPSC port engines.

Restriction

Setting the MPSC protocol to UART restricts the network bandwidth that can be used. This is a result of the UART protocol design.

Driver External Interface- Data Structure

Driver Data Structures: MPSC_PORT_CONFIG

Table 17:*Driver Data Structures: MPSC_PORT_CONFIG*

Type	Description
UINT protocol Bit Width: 3	Protocol Type
MPSC_MAIN_STRUCT main	Main structure as defined in the low level.
UINT32 portconf	Protocol Configuration - union of all protocols
MPSC_CHANNEL_STRUCT channel	MPSC Channel Register

External Interface APIs

void mpscChanInit(MPSC_CHANNEL *mpscChan)

This function completes the MPSC SW struct initialization towards the mpscChanStart() phase. It also initiates the MPSC interrupt controller.

INPUT

`MPSC_CHANNEL *mpscChan` MPSC channel struct.

OUTPUT MPSC Interrupt controller initiated.

RETURN N / A.

bool mpscChanStart(MPSC_CHANNEL*mpscChan)

DESCRIPTION	This function starts the MPSC channel activity by writing to the MPSC registers the values needed for its operation. Some of the values are given by the user and some are calculate in the mpsc-ChanInit() phase.
INPUT	
MPSC_CHANNEL *mpscChan	MPSC channel struct.
OUTPUT	All MPSC register are assigned. The channel enters hunt state.
RETURN	
	`False If the MPSC channel fails to enter hunt state within a given timeout.`
	`True If the MPSC channel enters hunt state within a given timeout.`

void mpscChanStopTx(MPSC_CHANNEL *mpscChan)

This function stops any Tx activity of a given MPSC channel.

Note: In order to stop a complete serial channel Tx, the MPSC channel Tx should be stopped only after the SDMA channel Tx already stopped.

INPUT	
MPSC_CHANNEL *mpscChan	MPSC channel struct.
OUTPUT	MPSC Tx activity is stopped. MPSC channel register 10 (Event Status Register) should mark that the Tx is in idle state.
RETURN	Not Applicable.

void mpscChanStopRx(MPSC_CHANNEL *mpscChan)

This function stops any Rx activity of a given MPSC channel.

Note: In order to stop a complete serial channel Rx, the MPSC channel Rx should be stopped before the SDMA channel Rx.

INPUT

`MPSC_CHANNEL *mpscChan` MPSC channel struct.

OUTPUT MPSC Rx activity is stopped. MPSC channel register 2 (Command Register) should clear the Rx abort bit upon entering Rx IDLE state.

RETURN Not Applicable.

void mpscChanStopTxRx(MPSC_CHANNEL *mpscChan)

This function stops Tx and Rx activity of a given MPSC channel simultaneously.

INPUT

`MPSC_CHANNEL *mpscChan` MPSC channel struct.

OUTPUT MPSC Tx and Rx activity are stopped. MPSC channel register 10 (Event Status Register) should mark that the Tx and Rx are in idle state.

RETURN Not Applicable

bool mpscChanSetCdv(MPSC_CHANNEL *mpscChan, int mpscCdv)

This function sets the MPSC Tx and Rx clock divider according to a given parameter.

Note: This function sets TCDV and RCDV with the same value.

INPUT

MPSC_CHANNEL *mpscChan	MPSC channel struct.
int mpscCdv	Clock divider. Can be 1, 8, 16, 32.
OUTPUT	Set MPSC Main configuration register fields TCDV and RCDV.

RETURN

False If the mpscCdv argument is neither of the following 1, 8, 16, 32

True otherwise.

Ethernet Driver

This driver implements a Gigabit Ethernet Controller network interface driver. It utilize the Gigabit Ethernet Controller low level driver tointroduce VxWorks END interface driver.

Supported Features

- Zero Copy Buff. This driver implements the Zero Copy Buff methodology. This means no data copy is done during either Rx nor Tx process.
- Scatter-Gather. When the driver gets a chain of mBlks to transmit it is able to perform Gather-write. It does not need to do any data copying.
- Supports cached buffers and descriptors for better performance. This driver utilize the internal device SRAM for descriptor memory area. This reduce access time to the descriptor thus increase overall performance.
- This driver supports multicasting.
- Easy to use API to manipulate the MAC address (using bootline).

Software Modules

- mgiEnd.c
- mgiEnd.h

Operation Flow

This driver establishes a shared memory for the communication system, which is divided into two parts:

- Descriptors area (Tx and Rx)
- Receive buffer area.

The descriptors area consists of a linked list of descriptors through which packet are received and transmitted. Both Tx and Rx linked lists are created using the low level driver in a form of a ring.

The receive buffer area is where the device received packet data is stored. This area is curved by the netBufLib to be the END pool of clusters. Those clusters pointers are set to be the Rx descriptors buffers pointers. This means that the received Rx packet is stored directly into the netBufLib clusters with no need to copy data (See netBufLib).

Upon receive event, the driver creates a mBlk-clBlk-cluster tuple using netBuffLib API and low level receive routine to get a cluster pointer. As the driver manage cluster by itself it assigns release information to the clBlk structure. The mBlk is then sent to the upper layers. This driver manages the cluster memory pool by itself. netBuffLib 'get' and 'free' routine are not used in case of cluster return. Release of cluster by upper layers will result calling to local free routine. This saves expensive pool management time. The clusters and descriptors must comply to alignment restriction and CPU data cache line alignment restrictions. Thus the cluster and descriptors size are calculated in such way the base address is 32bytes aligned.

External Interface

The driver provides the standard external interface, mgiEnd-Load(), which takes a string of colon separated parameters. The parameters should be specified in hexadecimal, optionally preceded by "0x" or a minus sign "-". The parameter string is parsed using strtok_r() and each parameter is converted from a string representation to binary by a call to strtoul(parameter, NULL, 16). The format of the parameter string is:
0
"<portNum>:<portMacAddr>:<memBase>:<memSize>:<nCFDs>:<nRFDs>:<flags>"

In addition, the two global variables 'bspEndIntConnect' and 'bspEndIntDisconnect' specify respectively the interrupt connect routine and the interrupt disconnect routine to be used depending on the BSP. The former defaults to intConnect() and the user can override this to use any other interrupt connect routine such as pciIntConnect()) in sysHwInit() or any device specific initialization routine called in sysHwInit(). Likewise, the latter is set by default to NULL, but it may be overridden in the BSP in the same way.

Target-specific Parameters

<portNum> Describes the MV-643xx Ethernet port number (integer).

<portMacAddr> This is a default MAC address to assign the MV-643xx Ethernet port . The MAC address is given in the following form: 11-22-33-44-55-66. One can override this MAC address using the bootline which finally saved in the system NVRAM.

<memBase> This parameter is used to inform the driver about the shared memory region. This parameter should always be NONE to inform the driver to allocate the shared memory from the system. Any other value for this parameter is not supported.

<memSize> This parameter is used to check that this region is large enough with respect to the provided values of both transmit/receive frames. Because the <memBase> parameter is always NONE, this parameter should always be 0.

<nTfds> This parameter specifies the number of transmit descriptors to be allocated. If this parameter is less than two, a default of 0x32 is used.

<nRfds> This parameter specifies the number of receive descriptor / buffers to be allocated. If this parameter is less than two, a default of0x32 is used. In addition, the number of 10 loaning buffers are created. These buffers are loaned up to the network stack.

External Interface-APIs

END_OBJ* mgiEndLoad(char *initString)

DESCRIPTION This routine initializes both, driver and device to an operational state using device specific parameters specified by <initString>. The parameter string, <initString>, is an ordered list of parameters each separated by a colon. The format of <initString> is, "<port-Num>:<portMacAddr>:<memBase>:<mem-Size>:<nCFDs>:<nRFDs>:<flags>"

<portNum> The Ethernet port number.

<portMacAddr> A MAC address to assign the Ethernet port. The MAC address is given in the following form: 11-22-33-44-55-66

<memBase> This parameter is used to inform the driver about the shared memory region. This parameter should always be NONE to inform the driver to allocate the shared memory from the system. Any other value for this parameter is not supported.

<memSize> This parameter is used to check that this region is large enough with respect to the provided values of both transmit / receive frames. Because the <memBase> parameter is always NONE, this parameter should be always 0.

INPUT

char *initString Parameter string

OUTPUT

- Ethernet Port data structures are initialized.
- The driver memory pool is ready.
- Ethernet devices driver is loaded to the MUX.

RETURN An END object pointer, or NULL on error.

LOCAL STATUS mgiUnload(DRV_CTRL *pDrvCtrl)

DESCRIPTION This function unloads the ethernet port END driver from the MUX.

INPUT

DRV_CTRL *pDrvCtrl Pointer to DRV_CTRL structure

OUTPUT
- Driver memory allocated is freed.
- Driver is disconnected from the MUX.

RETURN OK.

LOCAL STATUS memoryInit(DRV_CTRL *pDrvCtrl)

DESCRIPTION This function allocates the necessary memory space for the Tx/Rx descriptors as well as Rx buffers. Both Tx/Rx descriptor memory space and Rx buffers allocated using the malloc() routine which defines this memory space cacheable as the low level driver supports cacheable descriptors. This is done to allow better packet process performance. The function assigns the Rx buffer memory space to be the network pool cluster memory space which means that the network pool clusters and the Gigabit Ethernet Controller buffers are the same. This allow the zero copy in Rx where the Rx data is placed into the network pool clusters with no copying. This function also makes sure the descriptor and cluster addresses are cache line size aligned in order to avoid data loss when performing data cache flush or invalidate.

After memory allocation the driver creates and initializes netBufLib, the pool of mBlk and clBlk is created. The pool of clusters is defined to be the Rx buffers. This way the driver can practice the Zero Copy Buff methodology.

Tx/Rx descriptors Call low level routine to create Tx descriptor data structure.
Call low level routine to create Rx descriptor data structure.
Assign each Rx descriptor with netBufLib cluster.

INPUT

DRV_CTRL *pDrvCtrl Pointer to DRV_CTRL structure

OUTPUT Driver memory is ready for Rx and Tx operation.

RETURN

OK If output succeeded.

ERROR If driver failed to allocate one of the address spaces.

<memBase>
Parameter was not NONE.
- failure to initialize netBufLib pools.
- fail to retrieve cluster from pool for all Rx descriptors.

LOCAL STATUS mgiStart(DRV_CTRL *pDrvCtrl)

DESCRIPTION This routine prepare the ethernet port and system for operation:

- Connects the driver ISR.
- Enables interrupts.
- Start the Gigabit ethernet port using the low level driver.
- Marks the interface as active.

INPUT

DRV_CTRL *pDrvCtrl Pointer to DRV_CTRL structure

OUTPUT See description.

RETURN OK always.

LOCAL STATUS mgiStop(DRV_CTRL * pDrvCtrl)

DESCRIPTION This routine marks the interface as inactive, disables interrupts and resets the Gigabit Ethernet Controller port. It brings down the interface to a non-operational state. To bring the interface back up, mgiStart() must be called.

INPUT None.

OUTPUT

DRV_CTRL *pDrvCtrl Pointer to DRV_CTRL structure

RETURN

OK Always.

LOCAL STATUS mgiSend(DRV_CTRL *pDrvCtrl, M_BLK *pMblk)

DESCRIPTION This routine takes a M_BLK and sends off the data using the low level API. The buffer must already have the addressing information properly installed in it. This is done by a higher layer. The routine calls low level Tx routine for each mblk possibly reside in the M_BLK struct passed as a parameter.
The routine has to be familiar with Tx command status field of the low-level Tx descriptor in order to signal the low level driver on the location of the transmitted buffer in the packet (for packet spanned over multiple buffers). This way the routine supports Scatter-Gather. When the driver gets a chain of mBlks it is able to perform Gather-write where it does not need to do any data copying. muxSend() calls this routine each time it wants to send a packet.

INPUT

DRV_CTRL *pDrvCtrl Pointer to DRV_CTRL structure

M_BLK * pMblk Pointer to the mBlk/cluster pair

OUTPUT	The packet described by pMblk is sent to the Ethernet low level driver for transmission.
RETURN	OK, END_ERR_BLOCK in case there are no Tx descriptors available.

LOCAL void rxInt(DRV_CTRL *pDrvCtrl)

DESCRIPTION	This routine is the Rx interrupt handler. When this routine is called (by the Ethernet interrupt controller handler) the Rx interrupt event are already acknowledged, so that the device will deassert its interrupt signal. The amount of work done here is kept to a minimum; the bulk of the work is deferred to the netTask.
INPUT	
DRV_CTRL *pDrvCtrl	Pointer to DRV_CTRL structure
OUTPUT	Activating netJobAdd to registrate the Rx events.
RETURN	None.

LOCAL void recvIntHandle(DRV_CTRL *pDrvCtrl)

DESCRIPTION	Service task-level interrupts for receive frames.This routine is run in netTask's context. The ISR scheduled this routine so that it could handle receive packets at task level.
INPUT	Pointer to DRV_CTRL structure
RETURNS	None

LOCAL void mgiReceive(DRV_CTRL * pDrvCtrl, ETH_RX_DESC * pRxDesc)

DESCRIPTION	This passes a received frame to the upper next layer.
INPUT	Pointer to DRV_CTRL structure pointer to a RFD
RETURNS	None

LOCAL void rxRsrcReturn(DRV_CTRL *pDrvCtrl, char *pCluster, int dataSize)

DESCRIPTION	This routine returns the Rx resource back to the low level driver
INPUT	
DRV_CTRL *pDrvCtrl	Pointer to DRV_CTRL structure
ETH_RX_DESC * pRxDesc	Pointer to a RFD
OUTPUT	Rx resource is cache invalidated and returned to low level driver.
RETURN	None.

LOCAL void txRsrcReturn(DRV_CTRL *pDrvCtrl)

DESCRIPTION	This routine runs in netTask's context. The ISR scheduled this routine so that it could handle Tx packet resource release at task level. This routine free used Tx descriptors as well as mBlks. mBlks to release are located in the PKT_INFO returnInfo field. The mgiSend routine places the mBlk pointer in that field only in case the Tx desc is a packet last buffer. In case the Tx process was in 'stall' situation, the routine returns the Tx process to normal and notified the upper layers that the 'stall' situation is over. This routine is active as long as there are Tx resources to release.
INPUT	
DRV_CTRL *pDrvCtrl	Pointer to DRV_CTRL structure
OUTPUT	Return Tx resources to low level driver and release mBlk struct.

RETURN None.

LOCAL int mgiIoctl(DRV_CTRL *pDrvCtrl, int cmd, caddr_t data)

DESCRIPTION Process an interface ioctl request.

INPUT

DRV_CTRL *pDrvCtrl Pointer to DRV_CTRL structure

int cmd Command to process

caddrt data Pointer to data

OUTPUT None.

RETURN

OK Upon success.

ERROR If the config command failed.

LOCAL STATUS mgiMCastAddrAdd(DRV_CTRL *pDrvCtrl, char *pAddr)

DESCRIPTION This routine adds a multicast address to whatever the driver is already listening for.

INPUT

DRV_CTRL *pDrvCtrl Pointer to DRV_CTRL structure

char * pAddr Address to be added

OUTPUT Multicast address will be accepted.

RETURN OK or ERROR.

LOCAL STATUS mgiMCastAddrDel(DRV_CTRL *pDrvCtrl, char *pAddr)

DESCRIPTION This routine deletes a multicast address from the current list of multicast addresses.

INPUT

DRV_CTRL *pDrvCtrl Pointer to DRV_CTRL structure

char * pAddr Address to be added

OUTPUT Multicast address will be rejected.

RETURN OK or ERROR.

BRG Driver

Introduction

This driver implements the low level BRG engine. The MPSC ports can operate over a range of clock frequencies. The ports exploit the MV-64360 BRG facility for that purpose.

Software Modules

This driver is implemented in:

- brg.c BRG Register manipulation.
- brg.h BRG function and structure declaration.

brg.c BRG Register manipulation.

brg.h BRG function and structure declaration.

void brgInit(BRG_ENGINE *pBrg)

DESCRIPTION Initiate the BRG engine SW struct.

INPUT

`BRG_ENGINE *pBrg` Pointer to BRG struct.

OUTPUT None.

RETURN Not Applicable

void brgStart(BRG_ENGINE *pBrg)

DESCRIPTION This routine starts the BRG engine.

INPUT

`BRG_ENGINE *pBrg` Pointer to BRG struct.

OUTPUT None.

RETURN Not Applicable.

void brgDbg(BRG_ENGINE *pBrg)

DESCRIPTION Display the BRG engine struct.

INPUT

`BRG_ENGINE *pBrg` Pointer to BRG struct.

OUTPUT None.

RETURN Not Applicable.

void brgSetCdv(BRG_ENGINE *pBrg, unsigned short brgCdv)

DESCRIPTION This function Set CDV field in the BRG Control register.

INPUT

`BRG_ENGINE *pBrg` Pointer to BRG struct.

`unsigned short brgCdv` New CDV value.

OUTPUT Set Only CDV value in BRG config register.

RETURN Not Applicable

UART Over MPSC Port Driver

The PPMC-280 board offers MV-64360 MPSC port configured as UART for serial communication that is supported by this BSP.This UART Over MPSC Port driver (vxMpscUart.c) is VxWorks compliant and functions as a standard VxWorks SIO driver. This section describes how to install / uninstall the UART Over MPSC feature, which uses a MV-64360 MPSC port as UART.

Supported Features

This driver supports Baud rates 9600,19200,38400,57600 and 115200.

Software Modules

This driver is implemented in:

- vxMpscUart.c (UART Over MPSC Port drive)
- vxMpscUart.h

External Interface -APIs

void vxMpscUartDevInit(MV_SIO_CHAN *pChan)

DESCRIPTION This routine fills most of the channel's data structure in order for each of the channel components to be initiated properly. It also allocates memory areas for the driver operation in case the caller defined the Tx number of descriptors as zero. This function initialize the various interrupt controllers.

Note: The driver allocates memory from the USER_RESERVED_MEM area in order to be able to operate the serial channel prior to VxWorks memory initialization.

INPUT

MV_SIO_CHAN *pChan MV serial channel struct.

OUTPUT See description.

RETURN None.

void vxMpscUartDevReset(MV_SIO_CHAN *pChan)

DESCRIPTION This function resets the UART channel. It halts any of the channel Tx/Rx operation, clears and mask any pending interrupts.

INPUT

MV_SIO_CHAN *pChan MV serial channel struct.

OUTPUT See description.

RETURN None.

LOCAL void vxMpscUartStartChannel(MV_SIO_CHAN *pChan)

DESCRIPTION This routine resets the serial channel and start it.

INPUT

MV_SIO_CHAN *pChan MV serial channel struct.

OUTPUT The serial channel is ready to operate.

RETURN None.

LOCAL STATUS vxMpscUartIoctl(MV_SIO_CHAN *pChan, int request, int arg)

DESCRIPTION This routine controls the device basic functionality like baud rate and mode (polling or interrupt).

INPUT

***pChan**	MV serial channel struct, device to control.
request	Request code
arg	Some argument
OUTPUT	See description.

RETURN

OK	On success.
EIO	On device error.
ENOSYS	On unsupported request.

void vxMpscUartRxInt(MV_SIO_CHAN *pChan)

DESCRIPTION This routine is called when there is an Rx packet ready to be processed. The data with in the Rx packet is then passed to the VxWorks IO buffers using the Rx callback routine.

INPUT

***pChan**	MV serial channel struct.
OUTPUT	Received data is transferred to the IO layer.
RETURN:	None.

void vxMpscUartTxInt(MV_SIO_CHAN *pChan)

DESCRIPTION This routine transmits characters taken from the VxWorks IO buffer by Tx callback function. The characters are copied into the driver Tx buffer. This buffer and other information creates the packet information passed to the channel transmitting routine. This routine continues the Tx process which started by vxMpscUartStartup () routine and ends when there are no more characters to transmit.

INPUT

`*pChan` MV serial channel struct.

OUTPUT Calling the channel transmit routine with the proper packet information.

RETURN None.

LOCAL void vxMpscUartStartup(MV_SIO_CHAN *pChan)

DESCRIPTION This routine transmits characters taken from the VxWorks IO buffer by Tx callback function. The characters are copied into the driver Tx buffer. This buffer and other information creates the packet information passed to the channel transmitting routine. This routine invokes Tx process which continues in vxMpscUartTxInt() and ends when there are no more characters to transmit.

INPUT

`*pChan` MV serial channel struct.

OUTPUT Calling the channel transmit routine with the proper packet information.

RETURN None.

LOCAL int vxMpscUartPollInput(SIO_CHAN *pSioChan, char *thisChar)

DESCRIPTION This function receives a character using the given serial channel. The routine fills the given char buffer with the received char.

INPUT

***pSioChan** SIO channel struct.

***thisChar** Rx character buffer.

OUTPUT Calling the channel receive routine with the proper packet information.

RETURN

```
OK      If a character arrived

ERROR   On device error

EAGAIN  If the output buffer if full.
```

LOCAL int vxMpscUartPollOutput(SIO_CHAN *pSioChan, char outChar)

DESCRIPTION This function transmits a character using the given serial channel. The routine fills the packet information struct with the proper information for the given character.

INPUT

***pSioChan** SIO channel struct.

outChar The transmitted character.

OUTPUT Calling the channel transmit routine with the proper packet information.

RETURN

```
OK      If a character arrived

ERROR   On device error

EAGAIN  If the output buffer if full.
```

LOCAL int vxMpscUartCallbackInstall(SIO_CHAN *pSioChan, intcallbackType,STATUS (* callback)(), void *callbackArg)

DESCRIPTION This function installs ISR callbacks that transfers the data to and from the OS serial buffers.

INPUT

`*pSioChan` SIO channel struct.

`callbackType` Rx or Tx type of callback routine.

`(* callback)()` Callback routine.

`*callbackArg` Callback routine argument.

OUTPUT Calling Rx callback will deliver the received data to the OS layer. Calling Tx callback will retrieve data from the OS layer.

RETURN

```
OK      If callback installation succeeded

ENOSYS On wrong callbackType.
```

Serial EEPROM Driver

This driver supports the reading and writing into on-board Serial Electrically Erasable Programmable Read Only Memory (EEPROM) Device through I2C Bus. ATMEL AT24C64 Serial EEPROM device that supports 8192 Bytes or 64 Kbits is used in PPMC-280 board. The ATMEL AT24C64 device provides 256 pages each of 32 bytes length thus making it as 256*32 = 8192 Bytes. The end locations of the third EEPROM device is used

To store the MAC addresses of the MGI ports ,the starting locations of which are used to store the BIB related information.

Note: The first 128 bytes EEPROM1 and EEPROM2 are reserved for the storage of PCI Boot Data and the first 512 bytes and the last 12 bytes are reserved for the storage of BIB information and MAC addresses respectively on the BIBEEPROM.

Supported Features

The details of supported features are:

- The page rollover that is a device feature of ATMEL AT24C64 has been excluded to prevent page-rollover.
- When the offset for the data write is not at the starting location of the page and the number of data bytes is equal to or greater-than the page size, then the data is written in the sequence of incrementing memory locations without page-rollover.
- If the data to be written or read is more than 8 bytes for a 2 KBEEPROM or 32 bytes for a 64 KB EEPROM length then it automatically goes to next page and fetches data.

Software Modules

This driver is implemented in:

- i2cDrv.c: Located in the BSP directory.
- I2cDrv.h

Software Requirements

WindRiver VxWorks Operating System, Version 5.4 or higher.

External Interface - External APIs

bool frcEEPROMWrite16(UINT8 devAdd, UINT16 writeoffset, unsigned int noBytes, UINT8 *regFile1)

DESCRIPTION This routine does a slave write to the slave EEPROM. The EEPROM is divided into 256 pages of 32 bytes each. For example: 0x0000, 0x0020,0x0040 etc.

INPUT

devAdd Slave EEPROM device's hardware address.
UINT16

writeoffset A -byte offset in the EEPROM slave where the data has to be written.

noBytes Number of bytes to be written.

regFile Array of data that has to be written to the EEPROM.

OUTPUT Not Applicable

RETURN

`True If the write was successful.`

`False If the write was not successful.`

bool frcEEPROMRead16(UINT8 devAdd, UINT16 readoffset, unsigned int noBytes, UINT8*regFile1)

DESCRIPTION This routine does a slave write to the slave EEPROM. The EEPROM is divided into 256 pages of 32 bytes each. For example: 0x0000, 0x0020,0x0040 etc.

INPUT

devAdd Slave EEPROM device's hardware address.

UINT16

readoffset A 2-Byte offset in the EEPROM slave where the data has to be read from.

noBytes Number of bytes to be read.

regFile Array where the read data that has to be stored.

OUTPUT Not Applicable.

RETURN

```
True If the read was successful.

False If the read was not successful.
```

bool frcEEPROMWrite8(UINT8 devadd, UINT8 writeoffset,unsigned int noBytes, UINT8*regfile1)

DESCRIPTION This routine does a write to the slave EEPROM specified. The EEPROM is divided into number of pages each of size 8 bytes. A page's address is 8 bytes aligned. For example, 0x00,0x08,0x10 .. etc.

INPUT

devAdd Slave EEPROM device's hardware address.

writeoffset Byte offset in the EEPROM slave where the data has to be written. noBytes Number of bytes to be written.

regFile Array of data that has to be written to the EEPROM.

OUTPUT Not Applicable

RETURN

```
True If the write was successful.

False If the write was not successful.
```

bool frcEEPROMRead8(UINT8 devadd, UINT8 readoffset,unsigned int noBytes, UINT8*regfile1)

DESCRIPTION This routine does a read from the slave EEPROM specified. The EEPROM is divided into number of pages each of size 8 bytes. A page's address is 8 bytes aligned. For example, 0x00,0x08,0x10 .. etc.

INPUT

devAdd Slave EEPROM device's hardware address.

readoffset	Byte offset in the EEPROM slave where the data has to be read from.
noBytes	Number of bytes to be read.
regFile	Array where the read data that has to be stored.
OUTPUT	Not Applicable.
RETURN	
	`True If the read was successful.`
	`False If the read was not successful.`

Real Time Clock Driver

The driver supports reading from and writing into the RTC device through an I2C bus. The MAXIM MAX6900 that supports realtime clock counts seconds, minutes, hours, date, month day and year is present on the PPMC-280.

Supported Features

The driver supports read and write from and to the RTC device

Software Modules

The software modules supported are:

- i2cDrv.c
- i2cDrv.h
- rtcsupport.c

External API's

void frcRTCWrite(UINT8 *buffer)

This command writes to the RTC device, the contents of buffer that is passed to it.

INPUT	Contents of the buffer which indicate the time.
OUTPUT	Not Applicable
RETURNS	Not Applicable

void frcRTCRead(UINT8 *buffer)

This command reads the contents of the RTC device and writes into the buffer.

INPUT	Not Applicable
OUTPUT	Contents of the buffer which indicate the time.

Board Information Block Driver

The Board Information Block (BIB) is a data structure that allows storing information of all hardware devices in a compact manner in a non-volatile memory usually a serial I2C bus EEPROM called the ID-ROM which is mounted on the board.

Supported Features

Displays the entire contents of the BIB.

Note: For more details on how data structure for the BIB is defined refer to the "BIB Specifications 2.0" from Force Computers.

Software Modules

The driver is implemented in:

- i2cDrv.c
- frcBibDrv.c
- frcBibDrv.h
- frcBibShow.c
- frcBib.c
- frcBIB.h

External Interfaces- External APIs

void frcBibDataWrite(char *FTPSERVER_IP_ADRS,char *directory,char *filename)

DESCRIPTION This routine writes the hardware related information in the format required by the BIB driver into the EEPROM. Tools are available to convert the BIB definition file (as mentioned in the BIB specification) into the required format.

INPUT

char *FTPSERVER_IP_ADRS The server IP Address from where the file has to be downloaded

char *directory The directory within the FTP root where the file in s-record format is stored char *filename The filename of the file in s-record format that has to be programmed into the EEPROM

OUTPUT Not Applicable

RETURNS Not Applicable

The file that contains BIB information can be defined using the The BIB description language section in the BIB specifications. Then the definition file which has an extension .bib and the frcBib.h file are provided as input to the mk_bib tool .The output of which is fed along with the product specific information in the SAP file to the upd_bib which gives the final bib image that should be stored on the network.
The tool chain is as follows.

```
+ Tool Chain
file.bib, frcBIB.h -> mk_bib : rawBIB.x -> upd_bib :
finalBIB.x
^
BarCode -> SerialNum -> SAP : sap_info.dat ____|
The tools in detail
mk_bib : BIB Compiler, generates a "raw" BIB S-
record file.
upd_bib : Updates the "raw" BIB with board-specific data
(from file "sap_info.dat") and recalculates the checksum.
Generates
a "final" BIB image (S-record file).
```

GLOBAL STATUS frcBibAttach(UINT32 handle)

DESCRIPTION This routine must be called once to attach a handle to a BIB device, usually an I2C-bus EEPROM (ID-ROM). The handle is used as a unique identifier for the internal driver structures, and it is passed to the interface routines bibReadIntfRtn() and bib-WriteIntfRtn(), which are specified in the structure BIB_INTF, referred by pIntf. As described in section "3.1 Data Block Structure" of The Board Information Block (BIB) Version 2.0
document by Force Computers, more than one data block can be stored in the same IDROM by building a linked list. The driver searches this linked list of data blocks for a valid BIB image, verifies its checksum and copies the BIB image into an internal memory buffer. The routine returns ERROR in case of insufficient memory, if the device cannot be read, no BIB is found, or the BIB image is not valid.

INPUT The address of the EEPROM device that has to be attached.

Note: The device address is 0xa8, where the BIB storage area resides.

OUTPUT Not Applicable

RETURN

```
OK      If the handle is attached.

Error   The routine returns ERROR in case of
 insufficient memory, if the device cannot be
 read, no BIB is found, or the BIB image is not
 valid.
```

GLOBAL STATUS frcBibDrvShow(int infoLvl)

DESCRIPTION This show routine displays all attached BIB handles and the associated information, such as the previous error code, whether a local memory pool is used, and the address of the BIB interface routines. If infoLvl is higher than 1 and an external show routine has been defined via the global function pointer frcBibShowRtn, the contents of the BIB itself is also displayed.

INPUT An integer which indicates the infolvl required:

- 0 describes information about the handle
- 1 describes information about the interface routines and cache
- buffer
- 2 describes the entire information contained in the BIB

OUTPUT The information required.

RETURN

```
OK      If the handle is attached.

Error   The routine returns ERROR in case of
insufficient memory, if the device cannot be
read, no BIB is found, or the BIB image is not
valid.
```

frcBibShowRtn

When frcBibDrvShow( ) is called with an information level above 1, and this function pointer has previously been set to a BIB show routine, this will be called.

The prototype of a BIB show routine is:
STATUS yourBibShow (void *pImage, size_t imgSize, BOOL verbose, FILE *output)
Parameter pImage is the address of the BIB image which should be displayed. It is set to the driver's internal cache buffer.
The imgSize parameter contains the number of bytes of the BIB image.
The verbose flag is set TRUE if frcBibDrvShow( ) is called with an infoLvl greater than 2, otherwise it is FALSE.

The output is not used here, it is set to NULL to specify "standard out". Usually the application sets this function pointer to the frcBibShow( ) routine, which has to be included by the application.
frcBibShowRtn = frcBibShow

STATUS (*bibReadIntfRtn) (BIB_HDL handle, UINT32 offset, UINT32 byteCnt, void *pDstBuf)

DESCRIPTION This function pointer specifies the BIB read interface routine for the respective BIB device. This routine directly passes control to

EEPROM Device which in turn reads from the specified offset and the specified device.

INPUT

handle The BIB device handle specifies the memory device to access. This is usually an I2C-bus EEPROM (ID-ROM). The handle is not defined by the driver, but directly passed from the application to the respective interface routine. If an individual set of interface routines has been defined for each BIB device, this parameter may be discarded.

offset Byte offset within the memory device where to start reading or writing.

byteCnt Byte count, number of bytes to read or write.

pDstBuf Destination buffer, specifies where to store the data which was read.

OUTPUT Not Applicable.

RETURN The interface routines return a status code.

```
OK      If the operation was successful.

ERROR  When an invalid parameter was passed to
the routine, such as when an offset value which
is out of range, or if the read/write operation
failed.
```

STATUS (*bibWriteIntfRtn)(BIB_HDL handle, UINT32 offset, UINT32 byteCnt, void *pSrcBuf)

DESCRIPTION This function pointer specifies the BIB write interface routine for the respective BIB device. For a description of the function parameters see Arguments of the Interface Routines.

INPUT

Handle The BIB device handle specifies the memory device to access. This is usually an I2C-bus EEPROM (ID-ROM). The handle is not defined by the driver, but directly passed from the application to the respective interface routine. If an individual set of interface

routines has been defined for each BIB device, this parameter may be discarded.

offset Byte offset within the memory device where to start reading or writing.

byteCnt Byte count, number of bytes to read or write.

pSrcBuf Source buffer, specifies where to get the write data from.

OUTPUT Not Applicable

RETURN The interface routines return a status code.

```
OK      If the operation was successful,

ERROR   If an invalid parameter was passed to the
routine, for example an offset value which is out
of range, or if theread/write operation failed.
```

VPD Driver

The PPMC-280 BSP supports reading and writing of VPD (vital product data), to and from any PCI 2.2 compliant device.

Supported Features

The driver provides the following features:

- Read
- Write

Software Modules

This driver is implemented in:
vpd.c

Software Requirements

WindRiver VxWorks Operating System, Version 5.5 should be up and running.

External Interface- External APIs

STATUS frcVPDInit()

DESCRIPTION This functions initializes the VPD functionality of the loacl board and hence had to called by the local CPU in pciScan().

INPUT None.

OUTPUT None.

RETURNS OK if successful, else ERROR.

void frcVPDWrite(UINT32 bus, UINT32 dev, UINT32 fun,UINT16 vpdAddr,UINT32 vpdData)

DESCRIPTION This function is to write a 32 bit data into the VPD area of a PCI device.

INPUT

`bus` PCI device bus number.

`dev` PCI device number.

`fun` PCI device function number.

`vpAddr` The VPD address where the VPD has to be written. This is only 15 bits wide.

`vpdData` The VPD to be written.

OUTPUT None.

RETURNS None.

void frcVPDRead(UINT32 bus, UINT32 dev, UINT32 fun,UINT16 vpdAddr,UINT32 *vpdData)

DESCRIPTION	This function is to read a 32 bit VPD from the VPD area of a PCI device.
INPUT	
bus	PCI device bus number.
dev	PCI device number.
fun	PCI device function number.
vpAddr	The VPD address where the VPD has to be read. This is only 15 bits wide.
vpdData	The returned VPD.
OUTPUT	None.
RETURNS	None.

Boot Flash Driver

The PPMC-280 BSP provides a driver for programming onboard Boot Flash AMD AM29LV008. This can be used for programming the boot flash with a new bootable image once VxWorks is up and running with an existing image.

Supported Features

The driver provides the following features:

- Erasing of Sectors
- Verification of the device by writing and reading data from a specified location.
- Programming the flash device from a specified location
- Programming the flash with a file over network.

Software Modules

This driver is implemented in:
bflashdrv.c

Software Requirements

WindRiver VxWorks Operating System, Version 5.5 should be up and running.

External Interface- External APIs

short frcBootFlashSectorErase (volatile unsigned char *addr);

DESCRIPTION This function erases the contents with "FF" from the specified address till end of the sector. The function replaces the contents of the flash with data "FF".

INPUT

Unsigned char *addr Sector Address

OUTPUT Not Applicable

RETURNS

FLASH_SUCCESS If OK

short frcBootFlashProgram (volatile unsigned char *Address, unsigned char *data, unsigned int size)

DESCRIPTION This function programs the Flash device from a specified address. The address, data and size are input parameters of this routine as mentioned below.

INPUT

unsigned char *address Starting Address of the Flash to be programmed.

Unsigned char*data Starting Address of the data to be written.

Unsigned int size Number of Bytes to be written.

OUTPUT Not Applicable.

RETURN

FLASH_SUCCESS If OK.

FLASH_TIMEOUT If any error.

short frcBootFlashVerify (volatile unsigned char *Address, unsigned char *data, unsigned intsize)

DESCRIPTION This function verifies the Flash device from a given location by writing specified data for given size. It reads the contents of written data from the specified location and compares it with actual data, and reports if there are any mis-matches.

INPUT

Unsigned char *Address Starting Address of the Flash to be Programmed / Verified.

Unsigned char *data Starting Address of the data to be Written / Verified.

Unsigned int size
Number of Bytes to be verified.

OUTPUT Not Applicable.

RETURN

SUCCESS If OK.

void frcBootFlashFile (char * FTPSERVER_IP_ADRS ,char * DIRECTORY,char * filename)

DESCRIPTION This function loads the file from the Network and then programs the Boot Flash device with that file.

INPUT

char *FTPSERVER_IP_ADRS The server IP Address from where the file has to be downloaded

char *directory	The directory within the FTP root where the file in s-record format is stored
char *filename	The name of the file that has to programmed into the Boot flash device.
OUTPUT	Not Applicable.
RETURN	Not Applicable.

void frcBootFlashFileV(char * filename)

DESCRIPTION	This function verifies the contents of the boot flash with file data.
INPUT	
Unsigned char	
***filename**	Name of the file to be verified with the flash data.
OUTPUT	Not Applicable.
RETURN	Not Applicable.

User Flash Driver

The PPMC-280 BSP provides a driver for reading / writing on-board User Flash 28F128J3A. PPMC-280 has four of these devices. Each device is 8M x 16 (16 bits wide and 8M locations). The four devices are divided into two banks. Each bank has two devices. The size of each bank is 32MB (8M x 32). Each device has 128 blocks each of 128KB. The size of a block in each bank is 256KB. Hence there 128 blocks in each bank , each of size 256KB.

Note: The Bootline parameters are stored in User Flash. During the boot sequence, if an error in Bootline is encountered, then the you must perform a bootChange operation in order to over-come the error.

Supported features

The supported features are:

- Block erase.
- Verification of the device by reading (block wise) from it and
- storing in a defined area (in memory).
- Verification of the device by writing (block wise) defined data.
- Erasing a bank.

Software modules

This driver is implemented in flashDrv.c.

void frcFlashReadRst (unsigned int Flashbase)

DESCRIPTION This routine resets the flash to the read mode. After the execution of this routine the specified user flash would be ready to read.

INPUT

Flashbase The base of the flash device.
0xA0000000 - user flash bank 0
0xA2000000 - user flash bank 1

RETURN None.

int frcFlashAutoSelect (unsigned * Mnfct, unsigned * DevCode, unsigned int Flashbase)

DESCRIPTION This routine returns the manufacturer and device code of the specified user flash.

INPUT Flashbase The base address of the user flash.
0xA0000000 - user flash bank 0.
0xA2000000 - user flash bank 1.

Mnfct Pointer to a variable where the manufacturer code has to be returned.

DevCode Pointer to a variable where the device code has to be returned.

RETURN

FLASH_SUCCESS (0x0)If the routine was successful to get the manufacturer and device code.

FLASH_TIMEOUT (0x100)If the routine was not successful in either finding the flash device or getting the device and manufacturer code of the flash device.

short frcFlashErase (short sector,unsigned int Flashbase)

DESCRIPTION This routine will erase all the blocks of the specified user flash. If successful will then reset the flash device to the read mode.

INPUT

sector Should be 0. (For future use).

Flashbase The base address of the flash device.
0xA0000000 - user flash bank 0.
0xA2000000 - user flash bank 1.

RETURNS

FLASH_SUCCESS (0x00)If successful in erasing the specified flash device.

FLASH_TIMEOUT (0x100)If not successful in erasing the specified flash device.

void frcFlashRead(UINT32 StartAdd, UINT32 noLongs, UINT32 *buffer)

DESCRIPTION This routine reads specified number of bytes from a specified flash device.

INPUT StartAdd - The starting address of the flash where to read from.
noLongs - Number of Longs.
buffer - Destination address for the read data.

RETURNS None

int frcFlashBlockWrite(unsigned int BlockNo,unsigned int Flashbase, unsigned int *buffer))

DESCRIPTION This routine writes into the specified block

INPUT BlockNo - Number of the block to be written.
Flashbase- Base address of the flash device
buffer - Array of data that has to be written. This array should be 256KB in size.

RETURNS FLASH_SUCCESS - If successful
FLASH_TIMEOUT - If not successful

int frcFlashBlockErase(unsigned int BlockNo,unsigned int Flashbase)

DESCRIPTION This routine erases a block in the user specified user flash bank.If successful in erasing, this routine also resets the flash to the read mode.

INPUT

BlockNo The number of the block which has to erased.

Flashbase The base address of the user flash whose block has to be erased.
0xA0000000 - user flash bank 0.
0xA2000000 - user flash bank 1.

RETURNS

FLASH_SUCCESS (0x00)If the erase was successful.

FLASH_TIMEOUT (0x100)If the erase was not successful.

int frcFlashUnlock (unsigned int * flashBase)

DESCRIPTION In PMC-280 Rev. B, the block to which the address belongs will be unlocked. In PMC-280 Rev E0, the flash bank to which the address belongs will be unlocked. If successful in unlocking , this routine returns 1.

INPUT

flashBase The base address of the user flash which has to be unlocked.

0xA0000000 - user flash bank 0.
0xA2000000 - user flash bank 1.

RETURNS

```
1       If flash unlock was successful.

0        If flash unlock was unsuccessful.
```

int frcFlashLock (unsigned int * flashBase)

DESCRIPTION This routine locks a block in the user specified user flash bank.If successful in locking this routine returns 1.

INPUT

`flashBase` The base address of the user flash block which has to be unlocked.

RETURNS

```
1        If flash lock was successful.

0       If flash lock was unsuccessful.
```

Watchdog Timer Driver

The MV internal watchdog timer is a 32-bit count down counter that can be used to generate a non-maskable interrupt or reset the system in the event of unpredictable software behavior. After the watchdog is enabled, it is a free running counter that needs to be serviced periodically in order to prevent its expiration. This driver provides APIs for full domination over the WatchDog.
The API includes:

- frcWatchdogLoad() - Load WatchDog counter with a new value
- frcWatchdogService()- WatchDog service
- frcWatchdogNMILoad()- Load WatchDog value register with NMI_VAL
- frcWatchdogEnable()- Enable WatchDog operation

- frcWatchdogDisable()- Disable WatchDog operation

Note:

- **In order to use the NMI correctly, make sure to service the WD (using the frcWatchdogService routine) in the NMI connected to this driver. Avoiding this service will hang the system.**
- **In order to have the watchDog facility function correctly make sure the value of the watchdog timer is greater than the preset value, which is used for the invocation of the NMI. Avoiding this will cause the watchdog to assert WDE interrupt (reset the system) prior to the NMI event.**
- **In order to be able to receive NMI the user must make HW changes to the development board. The watchdog facility generates the NMI in GPP pin 10 (output). This pin is short to GPP pin 24 (input) using RNC8. Only after making this change the user will be able to see the NMI on GPP pin 24.**

frcWatchdogInit()

This routine initiates the MPP and GPP facilities to have NMI interrupt on GPP pin 24. This routine also connects the user ISR for the NMI using the GPP driver routine frcGppIntConnect()

INPUT None

OUTPUT The MPP + GPP facilities are now ready for WatchDog interrupt reception and user defined NMI service routine is connected to the GPP interrupt controller.

RETURNS Not Applicable

frcWatchdogLoad()

This function loads a value into the 24 most significant bits of the Watchdog Configuration Register, each time it is enabled or serviced.

INPUT

VALUE 24-bit counter

OUTPUT The value is loaded into the 24 Least Significant Bit of Watchdog Configuration Register

RETURNS

```
True    If output succeeds.

False   If argument is invalid.
```

frcWatchdogService()

WatchDog service and NMI interrupt acknowledge. This function service the WD in order to avoid NMI or reset (WDE#). Watchdog service is performed by writing '01' to CTL2, followed by writing '10' to CTL2. Upon watchdog service, the GT clears the NMI bits (if set) and reloads the Preset_VAL into the watchdog counter.

INPUT None.

OUTPUT The Preset_VAL is loaded.

RETURNS Not Applicable.

frcWatchdogNMILoad()

Load WatchDog value register with a new value (NMI_VAL). This function loads a value into the Watchdog Value Register. This value is the 24 least significant bits of a 32 bit value. When the WatchDog counter reaches a value equal to NMI_VAL an NMI interrupt is generated.

INPUT 24-bit wide number.

OUTPUT The value is loaded into the 24 Least Significant Bit of Watchdog Value Register.

RETURN

```
True    If output succeeds.

False   If argument is invalid.
```

frcWatchdogEnable()

Enable WatchDog operation. This function enables the WatchDog operation.A write sequence of '01' followed by '10' into CTL1 disables/enables the watchdog. The watchdog's current status can be read in bit 31 of WDC. When disabled, the GT clears the NMI bits (if set) and reloads the Preset_VAL into the watchdog counter.

INPUT None.

OUTPUT The disabled WatchDog is now enabled.

RETURN Not Applicable.

frcWatchdogDisable()

Disable WD operation. This function Disables the WD operation. A write sequence of '01' followed by '10' into CTL1 disables/enables the watchdog. The watchdog's current status can be read in bit 31 of WDC. When disabled, the GT clears the NMI bits (if set) and reloads the Preset_VAL into the watchdog counter.

INPUT None.

OUTPUT The Enabled Watchdog is now Disabled.

RETURN Not Applicable.

DoorBell Interrupt Support

This driver provides various interface routines to manipulate and connect the hardware interrupts concerning the MV Doorbell facility. The main features are listed here:

- The controller provides an easy way to hook a C Interrupt Service Routine (ISR) to a specific interrupt caused by the Doorbell register

- The controller interrupt mechanism provides a way for the programmer to set the priority of an interrupt
- Full interrupt control over the Doorbell facility

The Interrupt handler has a table which holds information on the connected user ISR. An Interrupt generated by one of the doorbell bits will result a search through this table in order to allocate the generating interrupt cause. After the initiating interrupt cause is identify, the ISR reside in the same table entry is executed. The controller interface also includes interrupt control routines which can enable / disable specific interrupts.

Software Modules

- VxDbIntCtrl.c
- VxDbIntCtrl.h

External Apis

void frcDbIntCtrlInit(void)

This routines connects the drivers interrupt handler, to its corresponding bits in the MV device main Interrupt Controller using the gtIntConnect() routine. It is also cleans and masks interrupts.

INPUT Not Applicable

OUTPUT The Doorbell cause & mask register are initialized (set to zero). Driver's ISR are connected to the main cause register.

RETURNS Not Applicable

STATUS frcDbIntConnect(DB_CAUSE cause,VOIDFUNCPTR routine, int parameter, int prio)

DESCRIPTION This routine connects a specified user ISR to a specified Doorbell interrupt cause. The ISR handler has its own user ISR array. The connection is done by setting the desired routine and parameter in the cause array (dbCauseArray[ ])

- Check for existing connection for the cause bit in the table.
- Connecting the user ISR by inserting the given parameters into an entry according to the user ISR given priority.

INPUT

DB_CAUSE cause Doorbell interrupt cause. See SDMA_DB.

VOIDFUNCPTR routine User ISR.

int parameter User ISR parameter.

int prio Interrupt handling priority where 0 is highest.

OUTPUT A table entry is filled.

RETURN

OK If the table entry of the cause bit, was filled.

ERROR If cause argument is invalid or connected cause is already found in table.

STATUS frcDbIntEnable(DB_CAUSE cause)

DESCRIPTION This routine unmasks a specified Doorbell cause in the mask register. The routine will preform argument validity check.

INPUT

DB_CAUSE cause Doorbell interrupt cause as defined in DB_CAUSE.

OUTPUT The appropriate bit in the Doorbell mask register is set.

RETURN

```
OK      If the bit was set.

ERROR   If the bit was invalid.
```

STATUS frcDbIntDisable(DB_CAUSE cause)

DESCRIPTION This routine masks a specified Doorbell interrupt in the mask register. The routine will preform argument validity check.

INPUT

DB_CAUSE CAUSE Doorbell interrupt cause as defined in DB_CAUSE.

OUTPUT The appropriate bit in the SDMA mask register is reset.

RETURN

```
OK      If the bit was reset.

ERROR   If the bit was invalid
```

STATUS frcDbIntClear(DB_CAUSE cause)

DESCRIPTION This routine clears a specified Doorbell interrupt in the cause register. The routine will preform argument validity check.

INPUT

DB_CAUSE cause Doorbell interrupt cause as defined in DB_CAUSE.

OUTPUT The appropriate bit in the Doorbell Clear register is reset.

RETURN

```
OK      If the bit was reset.

 ERROR  If the bit was invalid.
```

STATUS frcDbIntSend(DB_CAUSE cause)

DESCRIPTION This routine sends a specified Doorbell interrupt in the cause register. The routine will preform argument validity check.

INPUT

`DB_CAUSE cause` Doorbell interrupt cause as defined in DB_CAUSE.

OUTPUT The appropriate bit in the Doorbell Clear register is set.

RETURN

```
OK      If the bit was reset

ERROR   If the bit was invalid
```

void frcDbIntHandler (void)

DESCRIPTION This routine handles the Doorbell interrupts. As soon as the interrupt signal is active the CPU analyzes the Doorbell Interrupt Cause register in order to locate the originating interrupt event. Then the routine calls the user specified service routine for that interrupt cause.The function scans the dbCauseArray[ ] (dbCauseCount valid entries) trying to find a hit in the dbCauseArray cause table.When found, the ISR in the same entry is executed.

Note: The handler automatically acknowledges the generating interrupts.

INPUT None.

OUTPUT If a cause bit is active and it's connected to an ISR function, the function will be called.

RETURN None.

DMA Driver

This Driver gives the user a complete interface to the powerful DMA engines, including functions for controlling the priority mechanism.

Software Modules

- gtDma.c
- gtDma.h

External Apis

bool gtDmaCommand(DMA_ENGINE engine,unsigned int command)

This function writes a command to a specific DMA engine. The command defines the mode in which the engine will work in. There are several commands and that are defined in the MV64360/362 spec. It is possible to combine several commands using the or (|) operation. This function will not enable the DMA transfer unless the CHANNEL_ENABLE command is combined within the command parameter.

INPUT

`engine` One of the possible engines

`command` The command to be written to the given engine number .

OUTPUT None.

RETURN True on success, false on erroneous parameter.

DMA_STATUS gtDmaTransfer (DMA_ENGINE engine, unsigned int sourceAddr, unsigned int destAddr, unsigned int numOfBytes, unsigned int command, char srcIf, char tgtIf, DMA_RECORD *pNextRecordPointer)

DESCRIPTION

This routine transfers data from sourceAddr to destAddr on one of the 4 DMA channels.

When using the chain mode feature, the records must be 16 Bytes aligned. If the records reside on the system local memory, the function will take care of that for you. However, you need to allocate one more record for that; where if you have three records, you must declare four (see the example below) and start using the second one. If the records reside over PCI0/1 , it is the user's responsibility to make sure they are a 16 Bytes aligned.

When using the override feature (source or destination address over PCI) windows 1 and 2 must be allocated for PCI0 and PCI1 respectively before using this function.

DMA_DTL_8BYTES was intentionally defined as BIT1 from backwards compatibility reasons although its defined differently in the datasheet. If using different DTL on source and destination ,use the DMA_DTL_8BYTES (if needed) as is , the function will take care of the rest for you.

INPUT

engine Select one of the four available DMA engines.

sourceAddr The source address on which the transfer will begin.

destAddr The destination address on which the transfer will move the data.

numOfBytes The total number of bytes to transfer.

command The command selects different operation mode of the DMA engine such as different DTL , source/destination address override and so (for more details please refer to the MV's datasheet). The command can be combined with the | operator.

srcIf The source interface either the DRAM,SRAM or PCI

tgtIf The target interface -the DRAM,SRAM or PCI

***pNextRecordPointer** If you are using chain mode DMA transfer, then this pointer should point to the next record, otherwise it should be NULL.

OUTPUT DMA transfer.

RETURN

DMA_NO_SUCH_CHANNEL If channel does not exist.

DMA_CHANNEL_BUSY if channel is active.

DMA_OK If the transfer ended successfully.

DMA_STATUS gtDmaIsChannelActive(DMA_ENGINE engine)

DESCRIPTION This function checks whether a given DMA engine ('engine' parameter) is active or not .Useful for polling on a DMA engine to check if its still working.

INPUT

engine One of the possible engines

OUTPUT None.

RETURN

DMA_CHANNEL_IDLE If the engine is idle.

DMA_CHANNEL_BUSY If the engine is busy.

DMA_NO_SUCH_CHANNEL If there is no such engine.

bool gtDmaEngineDisable(DMA_ENGINE engine)

This function halts a DMA engine number delivered by 'engine' parameter. The engine will abort even if not all the transfer is completed.

INPUT

engine One of the possible engines .

OUTPUT DMA engine aborted.

RETURN True on success, false on erroneous parameter.

bool gtDmaUpdateArbiter(DMA_PIZZA *pPriorityStruct)

This function updates the arbiter`s priority for all the four DMA engines.

INPUT

pPriorityStruct A priority Structure with 16 fields, each field (slice) can be assigned to one of the DMA engines.

OUTPUT None.

RETURN

False If one of the parameters is erroneous, true otherwise.

bool gtDmaSetMemorySpace(DMA_MEM_SPACE memSpace, DMA_MEM_SPACE_TARGET memSpaceTarget, unsigned int memSpaceAttr, unsigned int baseAddress, unsigned int size)

The Atlantis IDMA has its own address decoding map that is de-coupled from the CPU interface address decoding windows. The four DMA channels share eight address windows. Each region can be individually configured by this function by associating it to a target interface and setting base and size values.

INPUT

memSpace One of the possible memory spaces (defined in gtDma.h).

memSpaceTarget The target interface to be associated with the region (DRAM, PCI, devices...).

memSpaceAttr Memory space attributes.The memory space attributes differ in each memory space target (please refer to the IDMA section in the Atlantis specification for more details).

baseAddress Memory space's base address.

size Memory space's size. This function will decrement the 'size' parameter by one and then check if the size is valid. A valid size must be programmed from LSB to MSB as sequence of '1's followed by sequence of '0's. To close a memory window simply set the size to 0.

Note: The size must be in 64Kbyte granularity. The base address must be aligned to the size.

OUTPUT None.

RETURN

```
True   Upon success

False  Otherwise.
```

void gtDmaSetMemorySpaceAttr(DMA_MEM_SPACE memSpace,unsigned int memSpaceAttr)

This function sets attributes for a DMA memory space.

INPUT

memSpace One of the possible memory spaces (defined in gtDma.h).

memSpaceAttr gtMemory space attributes.The memory space attributes differ in each memory space target (please see the IDMA section in the Atlantis specification for more details). Values for each memory space are defined in gtDma.h.Please note that you give the appropriate values to the corresponding interface .

OUTPUT None.

RETURN None.

bool gtDmaSetEngineAccessProtect(DMA_ENGINE engine,DMA_MEM_SPACE memSpace, DMA_MEM_SPACE_ACCESS access)

This function configures access attributes bits for DMA engine. Each engine can be configured with access attributes for each of the gtMemory spaces defined by 'gtDmaSetMemorySpace' function. This function sets access attributes to a given window for the given engine.

INPUTS

`engine` One of the 4 possible engines

`memSpace` One of the 8 possible gtMemory spaces

`access` The access type for the region .

OUTPUT None.

RETURN True for success, false otherwise.

DMA Interrupt Controller

This driver provides various interface routines to manipulate and connect the hardware interrupts concerning the MV DMA facility.

- The controller provides an easy way to hook a C Interrupt Service Routine (ISR) to a specific interrupt caused by the MV DMA engines.
- The controller interrupt mechanism provides a way for the programmer to set the priority of an interrupt.
- Full interrupt control over the MV DMA facility.

The driver's execution flow has three phases:

1. Driver initialization. This initiation includes hooking driver's ISR to the MV Interrupt controller. Composed of frcDmaIntCtrlInit() routine.

2. User ISR connecting. Here information about user ISR and interrupt priority is gathered. Composed of vxDmaIntConnect() routine.
3. Interrupt handler. Here an interrupt is being handle by the Interrupt Handlers (driver's ISR). Composed of vxDmaInt()

void frcDmaIntCtrlInit(void)

This routines connects the drivers interrupt handlers, each to its corresponding bit in the MV main Interrupt Controller using the gtIntConnect() routine.

INPUT None.

OUTPUT The DMA Engine cause & mask register are initiated (set to zero). Driver's ISR are connected to the main cause register. Interrupts are unmasked.

RETURN None.

STATUS frcDmaIntConnect(DMA_CAUSE cause, VOIDFUNCPTR routine, int parameter, int prio)

This routine connects a specified user ISR to a specified MV DMA interrupt cause (0x8c0 or 0x9c0).Each ISR handler has its own user ISR array. The connection is done by setting the desired routine and parameter in the corresponding cause array (i.e. dma0_1Array[ ])

- Locating the correct Array to make the connection. This is made according to the cause. DMA channel completion events has unique ISRs.
- Check for existing connection for the cause bit in the table.
- Connecting the user ISR by inserting the given parameters into an entry according to the user ISR given priority.

INPUT

`cause` DMA interrupt cause. See DMA_INT_CAUSE.

`routine` User ISR.

`parameter` User ISR parameter.

prio Interrupt handling priority where 0 is highest.

OUTPUT An appropriate table entry is filled.

RETURN

OK If the table entry of the cause bit, was filled.

ERROR If cause argument is invalid or connected cause is already found in table.

STATUS frcDmaIntEnable(DMA_CAUSE cause)

This routine unmasks a specified DMA interrupt cause on the appropriate mask register. The routine will perform argument validity check.

INPUT

cause DMA interrupt cause as defined in DMA_CAUSE.

OUTPUT The appropriate bit in the appropriate mask register is set.

RETURN

OK If the bit was unmasked

ERROR If the bit was invalid

STATUS frcDmaIntDisable(DMA_CAUSE cause)

This routine masks a specified DMA interrupt cause on the appropriate mask register. The routine will preform argument validity check.

INPUT

CAUSE DMA interrupt cause as defined in DMA_CAUSE.

OUTPUT The appropriate bit in the appropriate mask register is reset.

RETURN

```
OK      If the bit was masked

ERROR   if the bit was invalid
```

void frcDmaCompIntHandler (void)

This routine handles the DMA 0-3 completion interrupts. As soon as the interrupt signal is active the CPU analyzes the MV DMA 0-3 Interrupt Cause register in order to locate the originating interrupt event. Then the routine calls the user specified service routine for that interrupt cause. The function scans the dma0_3Array[] (dma0_3Count valid entries) trying to find a hit in the dma0_3Array cause table.When found, the ISR in the same entry is executed.

INPUT None.

OUTPUT If a cause bit is active and it's connected to an ISR function, the function will be called.

RETURN None.

void frcDmaErrorIntHandler(void)

This routine handles the DMA 0-3 Error interrupts. As soon as the interrupt signal is active the CPU analyzes the MV DMA 0-3 Interrupt Cause register in order to locate the originating interrupt event. Then the routine calls the user specified service routine for that interrupt cause. The function scans the dmaError0_3Array[] (dmaError0_3count valid entries) trying to find a hit. When found, the ISR in the same entry is executed.

INPUT None.

OUTPUT If a cause bit is active and it's connected to an ISR function, the function will be called.

RETURN None.

PciBoot Feature

The BSP supports the MV64360/362 feature of initialization of all its internal registers through I2C interface. If serial ROM initialization is enabled, the MV64360/362 I2C master starts reading initialization data from serial ROM and writes it to the appropriate registers .

void frcPCIDataWrite(char * FTPSERVER_IP_ADRS,char * filename,UINT8 DevAdd)

This routine burns into the EEPROM device the contents of a file which contain the register offset and register contents that are required for the PCI boot feature.

INPUT

FTPSERVER_IP_ADRS	The FTP server's IP Address from where the file requires to be downloaded.
Filename	The name of the file that has to be downloaded from the FTP server.
Devadd	The device address of the EEPROM to which the data has to be programmed

SMP Driver

This driver gives the user a complete interface to the SMP functionality of MV64360/362.

Software modules

- gtSmp.c
- gtSmp.h

External APIs

Note: The MV Semaphores 0,1,2 and 3 are reserved for use by MPSC, counter and timer facility, GPP and DMA respectively and hence cannot be used for other purposes.

int frcMV64360semGive(unsigned int sem_reg_no, unsigned int timeout)

DESCRIPTION This routine gives a semaphore using the MVs SMP facility.

INPUT

unsigned int sem_reg_no The register number of the semaphore.

unsigned int timeout

1 To indicate wait forever

0 To indicate a wait is not required to take the semaphore.

OUTPUT

OK If the semaphore is successfully taken

ERROR If the semaphore is not taken.

int frcMV64360semTake(unsigned int sem_reg_no)

DESCRIPTION This routine takes a semaphore using the MV's SMP facility.

INPUT

unsigned int sem_reg_no The register number of the semaphore.

OUTPUT

```
OK      If the semaphore is successfully given

ERROR    If the semaphore has not been given.
```

Test Application Support

The PPMC-280 BSP provides test applications to test the main features of the board such as Memory, I2C, Doorbell Interrupt. This includes simple test applications that are written for user to test these features.

Supported Features

The Driver supports test applications for:

- Baud Rate Change
- gettime
- settime

Software Modules

This driver is implemented in:

- rtc.c
- commDrv/vxMpscUart.c

Software Requirements

VxWorks

To Test APIs for RTC

void gettime()

DESCRIPTION Reads the time from the RTC device and displays it in a formatted manner. This is required to test the APIs for the RTC.

INPUT Not Applicable

OUTPUT Displays the time

RETURN Not Applicable

void settime()

DESCRIPTION This command obtains the formatted string from the user and writes the contents into the RTC device. This is required to test the APIs for the RTC.

INPUT The time in a formatted string. The following format is required
WWW-MMM-DD-HH:MM:SS-YYYY
For example
SUN-JUL-14-13:23:56-2002
.

To Test Baud Rate Change

void gtUartBaudRateChange(int portNum,int baudRate)

This routine changes the Baud rate of the specified port number to the required baud rate.

INPUT

`Int portNum` 0 or 1 indicating MPSC0 or MPSC1

`Int baudrate` The baudrate that it requires to be changed to. For example:9600,19200,38400 etc

OUTPUT Changes the baud rate .

RETURNS Not Applicable

A

Appendix

Appendix Overview

This appendix details the following:

Note: Refer to the PCI Local Bus Specifications Revision 2.2 for VPD Data Structure.

Memory Map

The following table describes the default memory map for VxWorks PPMC-280 BSP. Any changes to the address mapping are made by modifying the MV-64360 address decode registers.The memory map for PPMC-280 is given in the following table:

Address Range	Description	Size
FF800000 – FFFFFFFF	Boot Flash (BootCS#)	8MB
F2040000 – FF7FFFFF	Unused	
F2000000 – F203FFFF	Integrated SRAM	256KB
F1010000 – F1FFFFFF	Unused	
F1000000 – F100FFFF	MV64360 / 362 Internal Registers	64KB
A8000000 – F0FFFFFF	Unused	
A4000000 – A7FFFFFF	Reserved for User Flash expansion	Upto 128MB
A2000000 – A3FFFFFF	User Flash 1 (DevCS [1]#)	32MB
A0000000 – A1FFFFFF	User Flash 0 (DevCS [0]#)	32MB
99000000 – 9FFFFFFF	Unused	
98000000 – 98FFFFFF	Reserved for PCI_1 I / O	16MB
96000000 – 97FFFFFF	Reserved for PCI_1 Memory 3	32MB
94000000 – 95FFFFFF	Reserved for PCI_1 Memory 2	32MB
92000000 – 93FFFFFF	Reserved for PCI_1 Memory 1	32MB
90000000 – 91FFFFFF	Reserved for PCI_1 Memory 0	32MB
89000000 – 8FFFFFFF	Unused	
88000000 – 88FFFFFF	PCI_0 I / O	16MB
86000000 – 87FFFFFF	PCI_0 Memory 3	32MB
84000000 – 85FFFFFF	PCI_0 Memory 2	32MB
82000000 – 83FFFFFF	PCI_0 Memory 1	32MB
80000000 – 81FFFFFF	PCI_0 Memory 0	32MB
20000000 – 7FFFFFFF	Reserved for SDRAM expansion	Upto 2GB
00000000 – 1FFFFFFF	On-board SDRAM (CS [0]#)	512MB

Interrupt Routing on PPMC-280

PCI Interrupts

The PCI interrupts INTA#, INTB#, INTC# and INTD# are routed to MV64360 / 362's pins as shown in the table below:

PCI Interrupts	Pin number
PCI_INTA#	MPP27
PCI_INTB#	MPP29
PCI_INTC#	MPP16
PCI_INTD#	MPP17

The mapping between the MPPx to the IRQ number is given by the following formula:
IRQ = 64 + x;

where x = MPP pin number. The offset of 64 is because of the fact that the Main Interrupt Cause registers (Low and High) are 32-bit registers each with every bit corresponding to interrupt from a device (internal or external to MV64360 / 362).

Based on this, the PCI INTA#, INTB#, INTC# and INTD# take IRQ values as shown in the table below:

PCI Interrupts	IRQ Values
PCI_INTA#	27
PCI_INTB#	29
PCI_INTC#	16
PCI_INTD#	17

PCI Boot Procedure on PPMC-280

The procedure depends on whether PPMC-280 is monarch or non monarch. The procedure depends on whether;

a) PPMC-280 is monarch and is required to boot from memory of the non-monarch card

OR

b) PPMC-280 is non-monarch and is required to boot from memory of the monarch card

Case A: PPMC-280 is Monarch

The procedure when PPMC-280 is monarch and is required to boot from memory of the non-monarch card is detailed here:

1. Set up the PPMC-280 image on any FTP server (in its root directory)
2. After the system has powered up, and the non-monarch card has booted up, download thePPMC-280 image into location 0x08F00000 of the monarch's memory. (See Note on page A-3).
3. Set up 0x8000F104 into a variable on the non-monarch card. (See Note on page A-3.)

   y = 0x8000F104;

4. Trigger PPMC-280 to boot by writing 0x0 into the pointer *y

   *y = 0x0;

Immediately after this, you should see boot prints from PPMC-280.

Case B: When PPMC-280 is Non-Monarch

The procedure when PPMC-280 is non-monarch and is required to boot from the memory of the monarch card is detailed here:

1. Set up the PPMC-280 image on any FTP server (in its root directory)
2. After the system has powered up, PPMC-280 automatically releases the EREADY signal to allow the monarch card to boot up. When the monarch card has booted up, download the PPMC-280 image into location 0x08F00000 of the monarch's memory.
3. Identify the PCI device number <DEVNUM> of PPMC-280 by running a PCI scan on the monarch.

4. From the monarch card, read the PCI Internal Registers BAR of PPMC-280 (at offset 0x20). Say the read returns <BADDR>
5. On the monarch, setup <BADDR> + 0xF8 in a variable. For example, if <BADDR> = 0x86F00000,

   y = 0x86F000F8
6. Write 0x80080000 into the pointer *y. This is independent of <BADDR>

   *y = 0x80080000

This sets up the PCI memory address remap register of the MV64360/362 on PPMC-280.

7. On the monarch, setup <BADDR> + 0xF104 in a variable; for example

   y = 0x86F0F104
8. Trigger PPMC-280 to boot by writing 0x0 into the pointer *y

   ***y = 0x0**

You should now see boot prints from PPMC-280.

Note:

- **Download location 0x08F00000 is hard coded in PPMC-280 BSP (as well as the PCI boot serial EEPROM on PPMC-280). If your setup can not accommodate this, you would need to change the PCI boot serial EEPROM content. See "PciBoot Feature" page 5-139.**
- **It is assumed that MV64360/362's internal registers are mapped at 0x80000000 (on the PCI). If your carrier card cannot accommodate this address, you will need to set up an address in PCI Internal Registers Base Address (Low) in MV64360/362's Function 0 PCI configuration.**
- **In case of a watchdog reboot, depending on whether PPMC-280 is in the Monarch, or the Non-Monarch mode, follow the steps listed in the respective sections, "Case A: PPMC-280 is Monarch" page A-6 or "Case B: When PPMC-280 is Non-Monarch" page A-6, of this appendix.**

Using Test Tool

Integrated in BSP Rel. 3.x is a Test Tool. Use this test tool to perform various tests. Some of the tests you can perform are listed here:

- Check Serial Port Settings
- Verify ECC Enabled Mode
- IDMA Scrub Test
- Main Memory Test
- PCI Test
- Dual CPU Tests

This section details how you can use the test tool and the various types of tests that you can perform.

Invoking the Test Tool

Invoke the test tool with the `frcSysTest()` function at the command shell. A main menu is displayed. All menu items are numbered. You can select the menu you want to use by typing the appropriate number. Submenu options are also menu driven.
The following table details the main and sub menu options available.

Main Menu	Use this to:	Sub Menu Item	Sub Menu Option Number	Use this to:
Who Am I	Identify the CPU correctly. CPU-0 is displayed at the console connected to CPU-0 and CPU-1 is displayed at the console for CPU-1	Not Applicable (NA)	NA	NA
Serial Port Settings	Display serial port and baud rate settings	Display serial port settings	1	Display baud rate setting for the given port (either Port0 or Port1)
			2	Change the baud rate for a given port (either Port0 or Port1)

Main Menu	Use this to:	Sub Menu Item	Sub Menu Option Number	Use this to:
ECC Test	Check whether ECC is enabled or not. If ECC is not enabled, a message is printed and you return to the main menu. If ECC is enabled, memory tests are conducted for a given range and given number of cycles. Single bit and double bit errors are checked and their status displayed.	NA	NA	NA
IDMA Scrub Test	Test for proper scrubbing of memory in a given range. **Note: Ensure that you select the range in such a way that the OS will not modify the memory locations.**	NA	NA	NA
Main Memory Tests	Run the memory tests for a given range and for given number of cycles.	Basic Tests	1	Execute basic tests
		Extended Tests	2	Execute extended memory tests
		All Tests	3	Execute basic and extended memory tests together
Main Memory Performance Tests	Calculate the memory performance (Memory Bandwidth)	mam_perf or ASML	1	Measure Memory read Bandwidth with L1 cache, L2 cache and memory

Main Menu	Use this to:	Sub Menu Item	Sub Menu Option Number	Use this to:
		lmbench benchmark tool	2	Measure Memory Read / Write / Read-Write / Copy Bandwidth and also calculate Memory Read Latency. **Note: nbench test is not included in the menu, but is included in tests. Use the command `nbench` at the command prompt to execute the nbench tool.**
PCI Tests	Scan all buses and display information	NA	NA	NA
Configure LAN Ports	Perform various operations with the LAN ports.	Display Ethernet Port Settings	1	This prints mode of operation (10 / 100 / 1000-Full / Half) for the given port. **Note: This support is for On-Board Gigabit Ethernet Ports.**
		Configure Ethernet Port Settings	2	Attach an IP address to a given port.
		Set the mode of operation	3	Force a given port to the required mode (10 / 100 / 1000- Full / Half or Auto negotiation) **Note: This support is for on-board Gigabit Ethernet ports only.**
		Initialize pingLib	4	Initialize the pingLib

Main Menu	Use this to:	Sub Menu Item	Sub Menu Option Number	Use this to:
Register dump (MV64360/362/PCI-0)	Dump the contents of MV64360/362 and PCI-0. This is implemented using a Regparse tool. **Note: Owing to the internal configuration of Regparse tool, for each byte the contents will be printed in the reverse order.**	NA	NA	NA
DMA Tests	Test the DMA operation. This transfers the given size of memory using the user mentioned DMA engine (#0 to #3) and calculate the checksum at source and destination. Comparison of the two is then done. If there is any mismatch, then an error message is displayed. You can also select the option to choose the source/destination as SDRAM/SRAM/PCI.	NA	NA	NA
Dual CPU Tests **Note: Shemem-App address is given as input. The Shemem-App address is obtained by executing the option 24 from the main menu.**	Test Dual CPU features such as Doorbell Interrupts, Shared Memory and MV64360/362 semaphores.	Doorbell Interrupts/Shared Memory Test	1	Read the checksum from a known location and obtain a print message upon error and mismatch. Invoke this first from CPU0 and then CPU1. When this tool is executed from CPU1, data is read from the same shared memory location, the checksum is calculated and stored in the Shared Memory Location A Doorbell Interrupt is given to CPU0. CPU0 then reads the checksum from the known location and performs a comparison. On mismatch, an error is printed.

Main Menu	Use this to:	Sub Menu Item	Sub Menu Option Number	Use this to:
		2	Sema-phore Test	Give or take semaphore. When you take semaphore on CPU0 and try to take the same semaphore on CPU1, the test identify that the semaphore is locked.
I^2C / EEPROM	Test EEPROM Read / Write	EEPROM Write	1	Test EEPROM Write
		EEPROM Read	2	Test EEPROM Read
BIB Tests	Test BIB functionality.	BIB data write	1	Test BIB data write
		Attach BIB device	2	Attach BIB device
		Show BIB info	3	Show BIB info
VPD Tests	Test VPD operation	Program VPD Structure	1	Program VPD Structure
		VPD Write	2	Test VPD Write
		VPD Read	3	Test VPD Read
		VPD Write / Read	4	Test VPD Write / Read
RTC Tests	Set or get the RTC time	NA	NA	NA
Monarch / Non-Mon-arch Tests	Print whether the card is in the Monarch state or Non-Monarch State	NA	NA	NA
Display Memory Map	Display Memory Map	NA	NA	NA
Mac Address Pro-gramming / Display	Display / program MAC addresses	MAC Address Programming	1	To program MAC Address
		MAC Address Display	2	To display MAC Address

Index of Functions

F

Product Error Report

Product:	Serial No.:
Date Of Purchase:	Originator:
Company:	Point Of Contact:
Tel.:	Ext.:
Address:	
Present Date:	
Affected Product: ❑ Hardware ❑ Software ❑ Systems	Affected Documentation: ❑ Hardware ❑ Software ❑ Systems
Error Description:	

This Area to Be Completed by Force Computers:

Date:

PR#:

Responsible Dept.: ❑ Marketing ❑ Production

Engineering ➠ ❑ Board ❑ Systems

☞ Send this report to the nearest Force Computers headquarter listed on the address page.