# Embedded system Processor

DEVICE DRIVER

# Introduction

- Most Embedded Hardware requires software initialization and management to operate correctly.

- *Device Driver* is a software that directly interfaces with and controls this hardware

- *It* is responsible for configuring the hardware device

- *It* provides an abstraction layer between the hardware and the system software, allowing the software to access the hardware device in a consistent and standardized way.
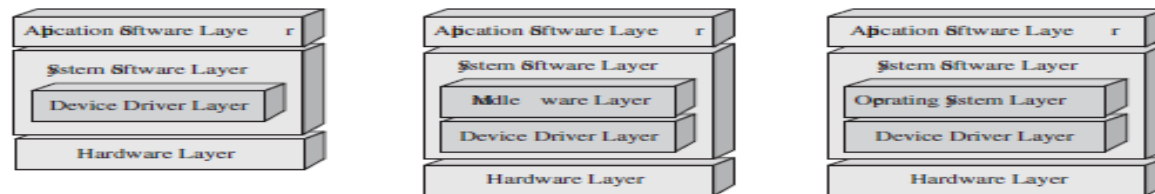


*Figure 8-1: Embedded Systems Model and device drivers*

# Cntd..

- Embedded device drivers are typically written in low-level programming languages such as C or assembly language

- Are tightly coupled with the hardware they control

- They are designed to be efficient and optimized for the specific hardware platform they run on, making the most of the limited resources available in embedded systems.

# What device drivers are required?

- The types of hardware components needing the support of device drivers vary from board to board

- But they can be categorized according to the von Neumann model approach.

- This includes drivers for the master processor architecture-specific functionality, memory and memory management drivers, bus initialization and transaction drivers, and I/O initialization and control drivers (such as for networking, graphics, input devices, storage devices, debugging I/O, etc.)

# Types of Device Drivers

- **Architecture-specific:**
  - It manages the hardware that is integrated into the master processor
  - Initialize and enable components within a master processor
  - Examples - on-chip memory, integrated memory managers (MMUs), and floating point hardware.

- **Generic:**
  - A device driver that is generic manages hardware that is located on the board and not integrated onto the master processor
  - includes code that initializes and manages access to the remaining major components of the board,
  - including board buses (I2C, PCI, PCMCIA, etc.), off-chip memory (controllers, level-2+ cache, Flash, etc.), and off-chip I/O (Ethernet, RS-232, display, mouse, etc.).

# Device Driver Functions

- Regardless of the type of device driver or the hardware it manages, all device drivers are generally made up of all or some combination of the following functions:
  - ❑**Hardware Startup**, initialization of the hardware upon power-on or reset.
  - ❑**Hardware Shutdown**, configuring hardware into its power-off state.
  - ❑**Hardware Disable**, allowing other software to disable hardware on-the-fly.
  - ❑**Hardware Enable**, allowing other software to enable hardware on-the-fly.
  - ❑**Hardware Acquire,** allowing other software to gain singular (locking) access to hardware.
  - ❑**Hardware Release**, allowing other software to free (unlock) hardware.
  - ❑**Hardware Read**, allowing other software to read data from hardware.

# Cntd..

❑**Hardware Write,** allowing other software to write data to hardware.

❑**Hardware Install,** allowing other software to install new hardware on-the-fly.

❑**Hardware Uninstall**, allowing other software to remove installed hardware on-the-fly.

- Device drivers may have additional functions

- These functions are based upon the software's implicit perception of hardware, which is that hardware is in one of three states at any given time—**inactive**, **busy**, or **finished**.

# Cntd..

- Hardware in the **inactive state is** interpreted as being either
  - Disconnected (thus the need for an install function), or
  - Without power (hence the need for an initialization routine) or
  - Disabled (thus the need for an enable routine).
- The **busy and finished states** are active hardware states, as opposed to inactive;
- Thus the need for uninstall, shutdown and/or disable functionality.
- Hardware in a **busy state**
  - actively processing some type of data and is not idle, and thus may require some type of release mechanism.
- Hardware in the finished state is in an **idle state**,
  - Which then allows for acquisition, read, or write requests, for example.

# Device-code layers

- Device drivers may have some functions, and

- It can integrate these functions into single larger functions.

- Each of these driver functions typically has code
  - that interfaces directly to the hardware
  - and code that interfaces to higher layers of software.

- In some cases, the distinction between these layers is clear,

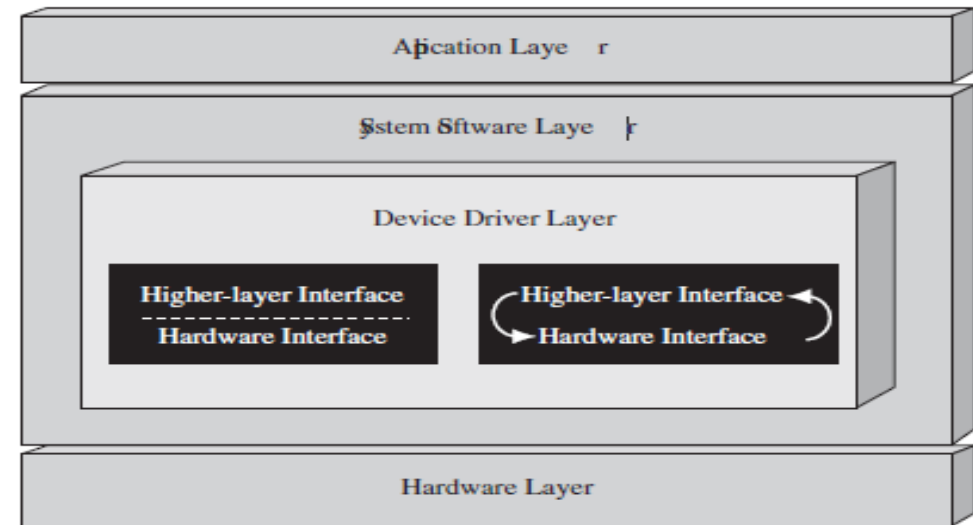- In other drivers, the code is tightly integrated



Figure 8-4: Driver code layers

# Supervisory and User modes

- Depending on the master processor, different types of software can execute in different modes,

- The most common modes - supervisory and user modes.

- These modes essentially differ in terms of what system components the software is allowed access to,

- Software running in a supervisory mode has more access (privileges) than software running in user mode.

- Device driver code typically runs in supervisory mode.

# Device Drivers for Interrupt-Handling

- Interrupts are signals triggered by events during the execution of an instruction stream.

- Interrupts can be initiated asynchronously, for external hardware devices, resets, power failures etc., or

- Synchronously, for instruction-related activities such as system calls or illegal instructions.

- These signals cause the master processor to halt execution of the current instruction stream and begin handling (processing) the interrupt.

# Device driver functions for interrupt handling

❏ **Interrupt-Handling Startup**, initialization of the interrupt hardware (i.e., interrupt controller, activating interrupts, etc.) upon power-on or reset.

❏ **Interrupt-Handling Shutdown**, configuring interrupt hardware (i.e., interrupt controller, deactivating interrupts, etc.) into its power-off state.

❏ **Interrupt-Handling Disable**, allowing other software to disable active interrupts on the-fly (not allowed for Non-Maskable Interrupts (NMIs), which are interrupts that cannot be disabled).

❏ **Interrupt-Handling Enable**, allowing other software to enable inactive interrupts on-the-fly.

❏ **Interrupt-Handler Servicing**, the interrupt-handling code itself, which is executed after the interruption of the main execution stream

# How can interrupts be initiated?

- The software that handles interrupts on the master processor and manages interrupt hardware mechanisms (i.e., the interrupt controller) consists of the device drivers for interrupt handling.

# Types of Interrupt

- The three main types of interrupts are software, internal hardware, and external hardware.
- **Software interrupts**
  - Triggered internally by some instruction within the current instruction stream being executed by the master processor
- **Internal hardware interrupts**
  - Initiated by an event that is a result of a problem with the current instruction stream that is being executed by the master processor
- **External hardware interrupts**
  - Interrupts initiated by hardware other than the master CPU—board buses and I/O for instance.

# Cntd..

- Software and internal hardware interrupts—are also commonly referred to as exceptions or traps.

- **Exceptions**
  - Internally generated hardware interrupts triggered by errors that are detected by the master processor during software execution, such as invalid data or a divide by zero.

- **Traps**
  - Software interrupts specifically generated by the software, via an exception instruction

# Level-triggered and Edge-triggered interrupt

- IRQ (Interrupt Request Level)
  - An input pin or port through which the master processor is wired to outside intermediary hardware for interrupts that are raised by external events
- External interrupts are triggered in one of two ways:
  - Level-triggered or
  - Edge-triggered.
- A level-triggered interrupt is initiated when its interrupt request (IRQ) signal is at a certain level
- These interrupts are processed when the CPU finds a request for a level-triggered interrupt when sampling its IRQ line, such as at the end of processing each instruction.

# Cntd..

- Edge-triggered interrupts trigger when a change occurs on its IRQ line (from LOW to HIGH/rising edge of signal or from HIGH to LOW/falling edge of signal

- Once triggered, these interrupts latch into the CPU until processed.

# Drawbacks of Level-triggered interrupt

- **Disadvantage of Level-triggered interrupt**
    - if the request is being processed and has not been disabled before the next sampling period, the CPU will try to service the same interrupt again.
    - elseif interrupts were triggered and then disabled before the CPU's sample period, the CPU would never note its existence and would therefore never process it.
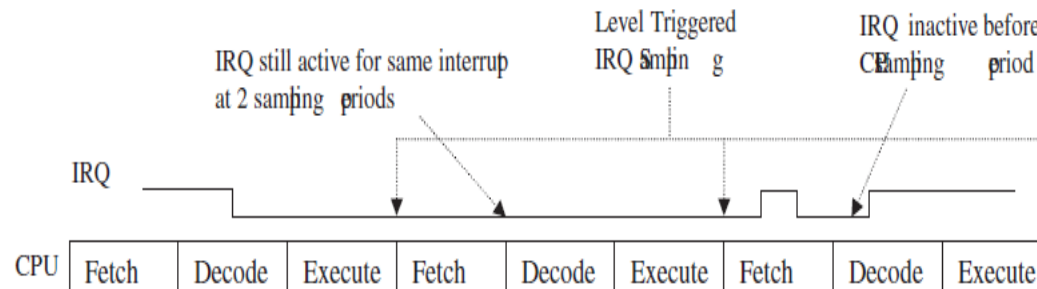


Figure 8-6a: Level-triggered interrupts drawbacks [8-3]

# Drawbacks of Edge-triggered interrupts

- **Edge-triggered interrupts** could have problems
    - if 2 interrupts share the same IRQ line, if they were triggered in the same manner at about the same time, resulting in the CPU being able to detect only one of the interrupts

Problem if Falling Edge Trigger *or* Rising Edge Trigger for *both* Edge Triggered Interrupts around the same time with the CPU only processing one

| | |
|---|---|
| IRQ & IRQ 2 | |

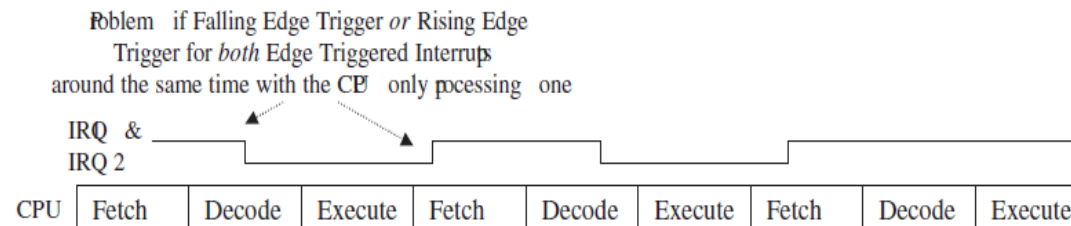| CPU | Fetch | Decode | Execute | Fetch | Decode | Execute | Fetch | Decode | Execute |
|---|---|---|---|---|---|---|---|---|---|

*Figure 8-6b: Edge-triggered interrupts drawbacks* [8-3]

- Because of these drawbacks, level-triggered interrupts are generally recommended for interrupts that share IRQ lines,

- Edge-triggered interrupts are typically recommended for interrupt signals that are very short or very long.

# Auto-vectored vs. interrupt-vectored scheme

- **ISR(Interrupt Service Routine)**
  - The software that the master CPU executes after the triggering of an interrupt
- **Interrupt vector**
  - A place in memory that holds the address of an interrupt's ISR
- **Auto-vectored interrupt scheme**
  - One ISR is shared by the non-vectored interrupts; determining which specific interrupt to handle, interrupt acknowledgment, etc., are all handled by the ISR software
- **Interrupt vectored scheme**
  - Implemented to support peripherals that can provide an interrupt vector over a bus, and where acknowledgment is automatic

# Non-maskable Interrupt

- When an interrupt is triggered, lower priority interrupts are typically **masked**,
  - meaning they are not allowed to trigger when the system is handling a higher-priority interrupt.
- The interrupt with the highest priority is usually called a non-maskable interrupt (NMI).

# Memory device drivers

- The software involved in managing the memory on the master processor and on the board, consists of the device drivers

- **Memory Map**
  - According to the master processor and programmers physical memory is a large one-dimensional array, commonly called memory map.
  - Each cell of the array is a row of bytes (8 bits) and
  - The number of bytes per row depends on the width of the data bus (8-bit, 16-bit, 32-bit, 64-bit,etc.).
  - Width of the data bus depends on the width of the registers of the master architecture.

# Device driver functions for memory

- All or some combination of six device driver functions are commonly implemented, including:
  - ❑**Memory Subsystem Startup**, initialization of the hardware upon power-on or reset (initialize TLBs for MMU, initialize/configure MMU).
  - ❑**Memory Subsystem Shutdown,** configuring hardware into its power-off state
  - ❑Memory Subsystem Disable, allowing other software to disable hardware on-the-fly (disabling cache).
  - ❑Memory Subsystem Enable, allowing other software to enable hardware on-the-fly (enable cache).
  - ❑Memory Subsystem Write, storing in memory a byte or set of bytes (i.e., in  cache, ROM, and main memory).
  - ❑Memory Subsystem Read, retrieving from memory a "copy" of the data in the form of a byte or set of bytes (i.e., in cache, ROM, and main memory)

# Memory addressing

- All data within memory is managed as a sequence of bytes.
- The **order** in which **bytes are retrieved** or **stored** in memory depends on the **byte ordering scheme** of an architecture.
- The two possible byte ordering schemes are little endian and big endian.
- In *little-endian mode*, bytes are retrieved and stored in the order of the lowest byte first, meaning the lowest byte is furthest to the left.
- In *big-endian mode* bytes are accessed in the order of the highest byte first, meaning that the lowest byte is furthest to the right

# Example

| Odd Bank | | Even Bank | |
|---|---|---|---|
| F | 90 | 87 | E |
| D | E9 | 11 | C |
| 8 | F1 | 24 | A |
| 9 | 01 | 46 | 8 |
| 7 | 76 | DE | 6 |
| 5 | 14 | 33 | 4 |
| 3 | 55 | 12 | 2 |
| 1 | AB | FF | 0 |
| ⇓ | | ⇓ | |
| Data Bus (15:8) | | Data Bus (7:0) | |

In little-endian mode if a byte is read from address "0", an "FF" is returned, if 2 bytes are read from address 0, then (reading from the lowest byte which is furthest to the LEFT in little-endian mode) an "ABFF" is returned. If 4 bytes (32-bits) are read from address 0, then a "5512ABFF" is returned.

In big-endian mode if a byte is read from address "0", an "FF" is returned, if 2 bytes are read from address 0, then (reading from the lowest byte which is furthest to the RIGHT in big-endian mode) an "FFAB" is returned. If 4 bytes (32-bits) are read from address 0, then a "1255FFAB" is returned.

Figure 8-17: Endianess [8-4]

- Memory is either soldered into or plugged into an area on the embedded board, called **memory banks**.

# Cntd..

- how the software actually accesses memory depends on the programming language used to write the software.

- For example, assembly language has various architecture-specific addressing modes that are unique to an architecture,

- And Java allows modifications of memory through objects.

# Memory Management Device Driver Pseudocode Examples-MPC860

- The MPC860 memory controller is responsible for the control of up to eight memory banks,
    - interfacing to SRAM, EPROM, flash EPROM, various DRAM devices, and other peripherals (i.e., PCMCIA).
- For every chip select (CS), there is an associated memory bank.
- The memory controller has two different types of subunits, the
    - ❖general-purpose chip-select machine (GPCM)
        - o interface to SRAM,EPROM, Flash EPROM, and other peripherals
    - ❖user-programmable machines (UPMs)
        - o interface to a wide variety of memory, including DRAMs.

# Base and option registers

- Each memory bank has a pair of base and option registers

- With every new access request to external memory,

- The memory controller determines whether the associated address falls into one of the eight address ranges (one for each bank) defined by
  - Base registers
    - specify the start address of each bank
  - Option registers
    - specify the bank length

BRx - Base Register

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| BA0- BA15 | | | | | | | | | | | | | | | |

| 16 | 17 | 18 | 19 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 |
|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|---|
| BA16 | AT0A T2 | | PS0PS1 | | PARE | | WP | MS0MS1 | | | Reserved | | | | V |

ORx - Option Register

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| AM0- AM15 | | | | | | | | | | | | | | | |

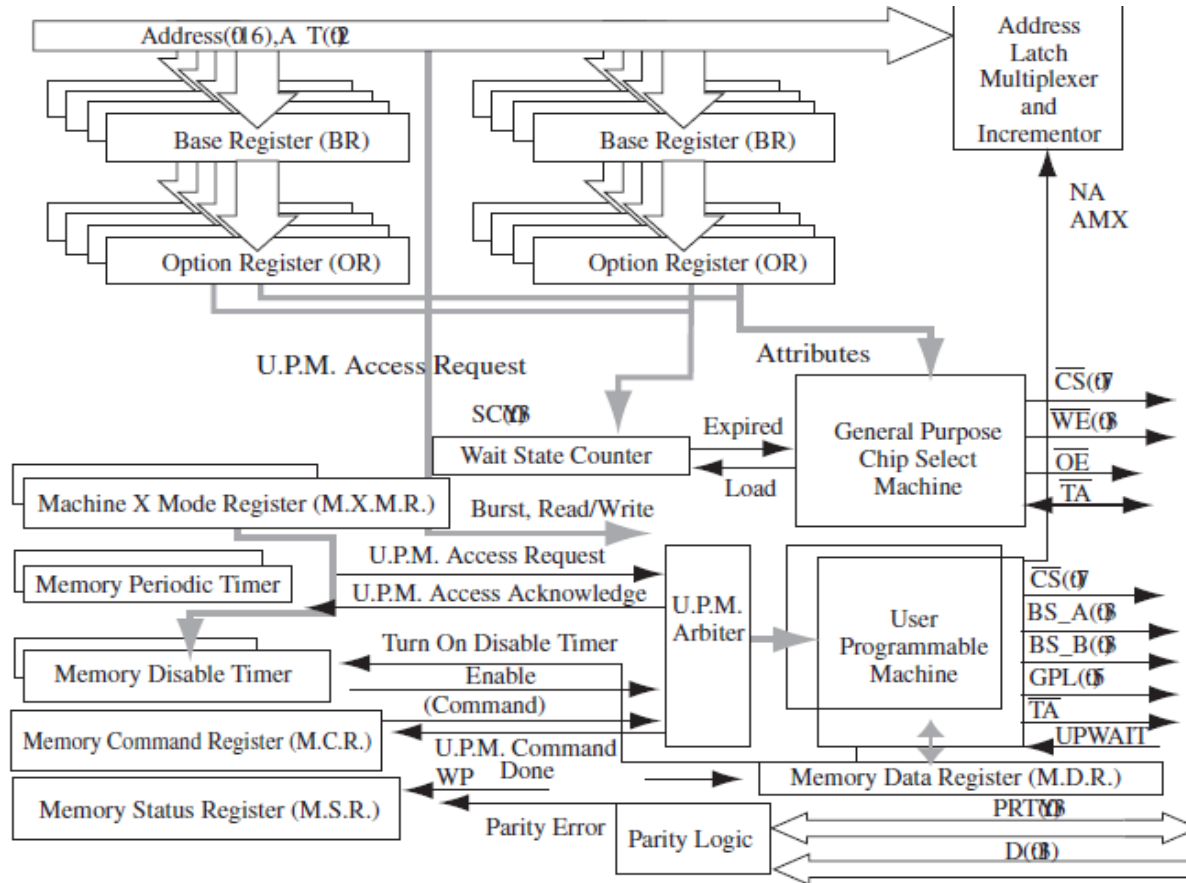| 16 | 17 | 18 | 19 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 |
|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|---|
| AM16 | ATM0A TM2 | | CSNT/SAM | ACS0A CS1 | | BI | SC0SCY | | | SETA | TRLX | EHTR | Res | | |

Figure 8-19: MPC860 Integrated memory controller [8-4]

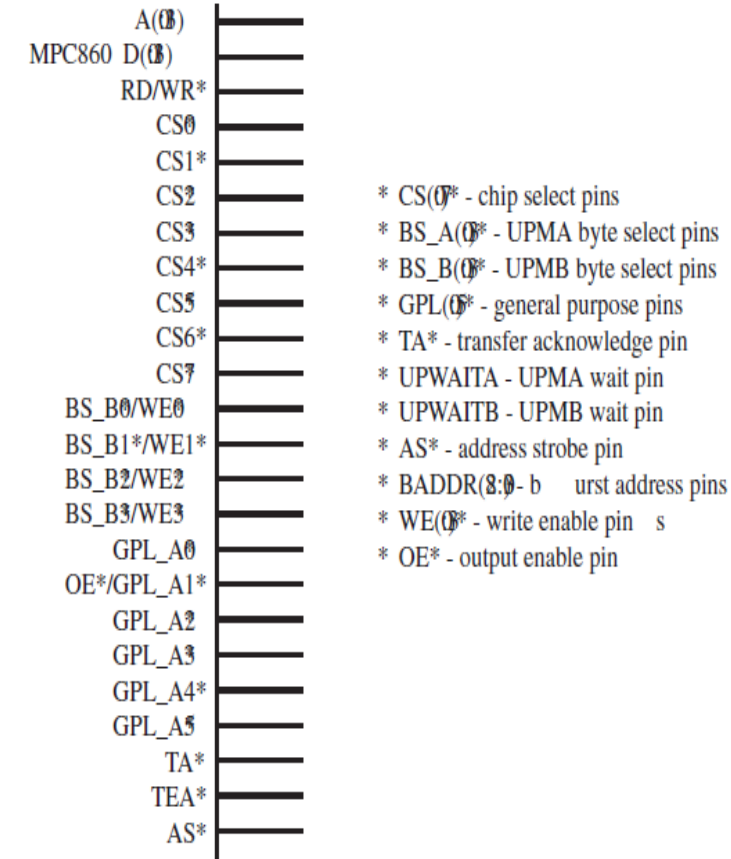Copyright of Freescale Semiconductor, Inc. 2004. Used by permission.



Figure 8-20a: Memory controller pins [8-4]

* CS(0)* - chip select pins
* BS_A(0)* - UPMA byte select pins
* BS_B(0)* - UPMB byte select pins
* GPL(0)* - general purpose pins
* TA* - transfer acknowledge pin
* UPWAITA - UPMA wait pin
* UPWAITB - UPMB wait pin
* AS* - address strobe pin
* BADDR(8:0) - burst address pins
* WE(0)* - write enable pins
* OE* - output enable pin

# Initializing the Memory Controller and connected ROM/RAM

| Address Range | Accessed Device | Port Width |
|---|---|---|
| 0x00000000 - 0x003FFFFF | Flash PROM Bank 1 | 32 |
| 0x00400000 - 0x007FFFFF | Flash PROM Bank 2 | 32 |
| 0x04000000 - 0x043FFFFF | DRAM 4 Mbyte (1Meg × 32-bit)it) | 32 |
| 0x09000000 - 0x09003FFF | MPC Internal Memory Map | 32 |
| 0x09100000 - 0x09100003 | BCSR - Board Control & Status Register | 32 |
| 0x10000000 - 0x17FFFFFF | PCMCIA Channel | 16 |

*Figure 8-18: Sample memory map* [8-4]

- Pseudocode for configuring the first two banks (of 4 MB of Flash each), and the third bank (4 MB of DRAM) would be as follows:

```
...
// OR for Bank 0 – 4 MB of flash , 0x1FF8 for bits AM (bits 0-16) OR0 = 0x1FF80954;
// Bank 0 – Flash starting at address 0x00000000 for bis BA(bits 0-16), configured for
// GPCM, 32-bit
BR0 = 0x00000001;

// OR for Bank 1 – 4 MB of flash , 0x1FF8 for bits AM (bits 0-16) OR1 = 0x1FF80954;
// Bank 1 - 4 MB of Flash on CS1 starting at address 0x00400000, configured for GPCM,
// 32-bit
BR1 = 0x00400001;

// OR for Bank 2 – 4 MB of DRAM , 0x1FF8 for bits AM (bits 0-16) OR2 =
// 0x1FF80800; Bank 2 - 4 MB of DRAM on CS2 starting at address 0x04000000,
// configured for UPMA, 32-bit
BR2 = 0x04000081;

// OR for Bank 3 for BCSR OR3 = 0xFFFF8110; Bank 3 – Board Control and Status
// Registers from address 0x09100000
BR3 = 0x09100001;
....
```

# Cntd..

- The memory periodic timer prescaler register (MPTPR) is initialized for the required refresh timeout (i.e., for DRAM),

- And the related memory mode register (MAMR or MBMR), for configuring the UPMs, needs initialization.

- The core of every UPM is a (64 × 32 bit) RAM array

```
…
// set periodic timer prescaler to divide by 8
MPTPR = 0x0800; // 16 bit register

//periodic timer prescaler value for DRAM refresh period (see the PowerPC manual for calculation),
timer enable,..
MAMR = 0xC0A21114;

// 64-Word UPM RAM Array content example --the values in this table were generated using the
// UPM860 software available on the Motorola/Freescale Netcomm Web site.

UpmRamARRY:
 // 6 WORDS - DRAM 70ns - single read. (offset 0 in upm RAM)
 .long 0x0fffcc24, 0x0fffcc04, 0x0cffcc04, 0x00ffcc04, 0x00ffcc00, 0x37ffcc47
```

# Cntd..

```
// 2 WORDs - offsets 6-7 not used
.long 0xffffffff, 0xffffffff
// 14 WORDs - DRAM 70ns - burst read. (offset 8 in upm RAM)
.long 0x0fffcc24, 0x0fffcc04, 0x08ffcc04, 0x00ffcc04,0x00ffcc08, 0x0cffcc44,
.long 0x00ffec0c, 0x03ffec00, 0x00ffec44, 0x00ffcc08, 0x0cffcc44,
.long 0x00ffec04,0x00ffec00, 0x3fffec47
// 2 WORDs - offsets 16-17 not used
.long 0xffffffff, 0xffffffff
// 5 WORDs - DRAM 70ns - single write. (offset 18 in upm RAM)
.long 0x0fafcc24, 0x0fafcc04, 0x08afcc04, 0x00afcc00,0x37ffcc47
// 3 WORDs - offsets 1d-1f not used
.long 0xffffffff, 0xffffffff, 0xffffffff
// 10 WORDs - DRAM 70ns - burst write. (offset 20 in upm RAM)
.long 0x0fafcc24, 0x0fafcc04, 0x08afcc00, 0x07afcc4c, 0x08afcc00, 0x07afcc4c,
.long 0x08afcc00, 0x07afcc4c, 0x08afcc00, 0x37afcc47
// 6 WORDs - offsets 2a-2f not used
.long 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
// 7 WORDs - refresh 70ns. (offset 30 in upm RAM)
.long 0xe0ffcc84, 0x00ffcc04, 0x00ffcc04, 0x0fffcc04, 0x7fffcc04, 0xffffcc86,
.long 0xffffcc05
// 5 WORDs - offsets 37-3b not used
.long 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,0xffffffff
// 1 WORD - exception. (offset 3c in upm RAM)
.long 0x33ffcc07
// 3 WORDs - offset 3d-3f not used
.long 0xffffffff, 0xffffffff, 0x40004650
UpmRAMArrayEnd:

// Write To UPM Ram Array
Index = 0
Loop While Index < 64
{
MDR = UPMRamArray[Index]; // Store data to MDR
MCR = 0x0000; // Issue "Write" command to MCR register to store what is in MDR in RAM Array
Index = Index + 1;
} //end loop
.....
```

# Initializing the Internal Memory Map on the MPC860

- The MPC860's internal memory map contains
  - ✓ special purpose registers (SPRs),
    - ○ Internal Memory Map Register (IMMR) is one of these SPRs
  - ✓ Dual-port RAM
    - ○ contain the buffers of the various integrated components, such as Ethernet or I2C,
- Here the internal memory map starts at 0x09000000, so in pseudocode form, the IMMR would be set to this value via the "mfspr" or "mtspr" commands:

```
mtspr 0x090000FF // the top 16 bits are the address, bits 16–23 are the part number
// (0x00 in this example), and bits 24–31 is the mask number
// (0xFF in this example).
```

# Initializing the MMU on the MPC860

- MPC860 uses the **MMUs** to manage the board's virtual memory management scheme, address translations, cache control and memory access protections.

- MPC860 MMU allows support for a 4 GB uniform (user) address space that can be divided into pages of a variety of sizes,
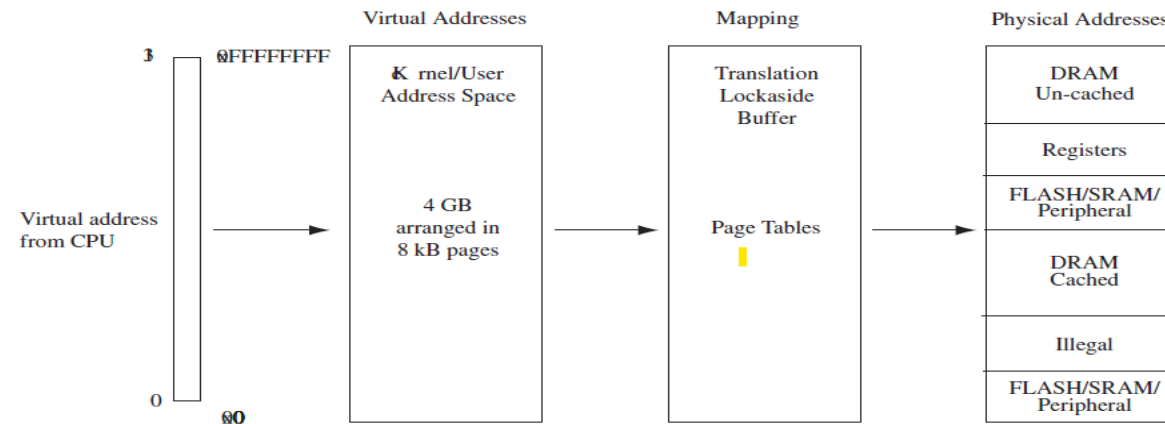


Figure 8-23a: TLB within VM scheme [8-4]

# TLB

- The MPC860 MMU does not manage the entire translation table at one time

- Since embedded boards do not typically have 4 GB of physical memory that needs to be managed at one time.

- It would be very time-consuming for an MMU to update a million entries with every update to virtual memory by the software

- So MPC860 MMU contains small caches within it to store a subset of this memory map.
  - ❖These caches are referred to as **translation lookaside buffer**
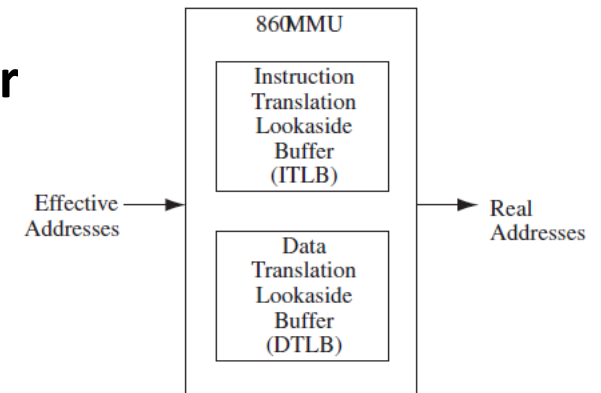    - o Contains one instruction and one data



Figure 8-23b: TLB [8-4]

# Memory Segment

- All data within memory is managed as a sequence of bytes.

- While one memory access is limited to the size of the data bus, certain architectures manage access to <u>larger blocks (a contiguous set of bytes) of data</u>, called **segments**,

- Thus implement a more complex address translation scheme
  - The logical address provided via software is made up of a segment number (address of start of segment) and offset (within a segment)
    - ❖Used to determine the physical address of the memory location.

# Tablewalk

- The TLB is how the MMU translates (maps) logical/virtual addresses to physical addresses.
- When the software attempts to access a part of the memory map not within the TLB, a TLB miss occurs,
- which is essentially a trap requiring the system software (through an exception handler) to load the required translation entry into the TLB.
- The system software that loads the new entry into the TLB does so through a process called a **tablewalk**.

# MMU initialization sequence pseudocode

```
// Invalidating TLB entries
tlbia ;              // the MPC860's instruction to invalidate entries within the TLBs, also the
                     // "tlbie" can be used


// Initializing the MMU Instruction Control Register

…
MI_CTR.fld.all = 0; // clear all fields of register so group protection mode =
 // PowerPC mode, page protection mode is page resolution, etc.
MI_CTR.fld.CIDEF = 1; // instructin cache inhibit default when MMU disabled

…
// Initializing the MMU Data Control Register
…
MD_CTR.fld.all = 0; // clear all fields of register so group protection mode =
 // PowerPC mode, page protection mode is page resolution, etc.
MD_CTR.fld.TWAM = 1; // tablewalk assist mode = 4kbyte page hardware assist
MD_CTR.fld.CIDEF = 1; // data cache inhibit default when MMU disabled
…
```

# On-board Bus Device Drivers

- Every bus is

  (1) some type of protocol that defines how devices gain access to the bus (arbitration),

  (2) the rules attached devices must follow to communicate over the bus (handshaking), and

  (3) The signals associated with the various bus lines.

- Bus protocol is supported by the bus device drivers, which commonly include all or some combination of all of the 10 functions:

  ❑**Bus Startup,** initialization of the bus upon power-on or reset.

  ❑**Bus Shutdown,** configuring bus into its power-off state.

# Cntd..

- **Bus Disable,** allowing other software to disable bus on-the-fly.
- **Bus Enable**, allowing other software to enable bus on-the-fly.
- **Bus Acquire**, allowing other software to gain singular (locking) access to bus.
- **Bus Release**, allowing other software to free (unlock) bus.
- **Bus Read**, allowing other software to read data from bus.
- **Bus Write**, allowing other software to write data to bus.
- **Bus Install**, allowing other software to install new bus device on-the-fly for expandable buses.
- **Bus Uninstall**, allowing other software to remove installed bus device on-the-fly for expandable buses.

# Example- Implementing a bus initialization routine on MPC860

- These examples demonstrate how bus management can be implemented on a more complex architecture,

- This can be used as a guide to understand how to write bus management drivers on other processors of equal or lesser complexity than the MPC860 architecture.

  ❖ <u>I2C Bus Startup (Initialization) on the MPC860</u>
  - o a serial bus with one serial data line (SDA) and one serial clock line (SCL)
  - o All devices attached to the bus have a unique address (identifier)
    - o This identifier is part of the data stream transmitted over the SDL line.

# Cntd..

- The master processor components that support the I2C protocol need initialization.
- In the case of the MPC860, there is an integrated I2C controller on the master processor
- The I2C controller is made up
  - ➤ transmitter registers,
  - ➤ receiver registers,
  - ➤ Control unit
  - ➤ Baud rate generator,
    - o It generates the clock signals when the I2C controller acts as the I2C bus master
    - o if in slave mode, the controller uses the clock signal received from the master.
- In reception mode, data is transmitted from the SDA line into the control unit, through the shift register, which in turn transmits the data to the receive data register.
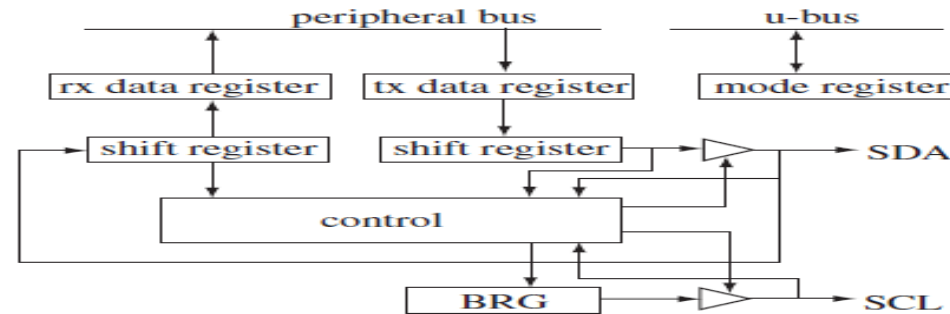


Figure 8-29: I²C controller on MPC860 [8-4]

# SDA and SCL pins initialization

- MPC860 I2C SDA and SCL pins are configured via the Port B general purpose I/O port
- Because the I/O pins can support multiple functions,
- The specific **function** a pin will support, **needs** to be **configured** via **port B's registers**
- Port B has four read/write (16-bit) control registers:
  - ➢ Port B Data Register (PBDAT),
    - o Contains the data on the pin
  - ➢ Port B Open Drain Register (PBODR),
    - o Configures the pin for open drain or active output
  - ➢ Port B Direction Register (PBDIR),
    - o Configures the pin as either an input or output pin,
  - ➢ Port B Pin Assignment Register (PBPAR)
    - o assigns the pin its function (I2C, general purpose I/O, etc.)

| | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PBDAT | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D | D |
| PBODR | OD | OD | OD | OD | OD | OD | OD | OD | OD | OD | OD | OD | OD | OD | OD | OD | OD | OD |
| PBDIR | DR | DR | DR | DR | DR | DR | DR | DR | DR | DR | DR | DR | DR | DR | DR | DR | DR | DR |
| PBPAR | DD | DD | DD | DD | DD | DD | DD | DD | DD | DD | DD | DD | DD | DD | DD | DD | DD | DD |

*Figure 8-30c: MPC860 port B register [8-4]*

# Cntd..

- An example of initializing the SDA and SCL pins on the MPC860 is given in the pseudocode below.

```
...
immr = immr & 0xFFFF0000;    // MPC8xx internal register map
// Configure Port B pins to enable SDA and SCL
immr->pbpar = (pbpar) OR (0x00000030); //set to dedicated I2C
immr->pbdir = (pbdir) OR (0x00000030);  // Enable I2CSDA and I2CSCL as outputs
....
```

# I2C Registers Initialization

- The I2C registers that need initialization include:
    - ✓ I2C Mode Register (I2MOD),
    - ✓ I2C Address Register (I2ADD),
    - ✓ Baud Rate Generator Register (I2BRG),
    - ✓ I2C Event Register (I2CER), and
    - ✓ I2C Mask Register (I2CMR)

```
/* I2C Registers Initialization Sequence */

....

// Disable I2C before initializing it, LSB character order for transmission and reception,
// I2C clock not filtered, clock division factor of 32, etc.
immr->i2mod = 0x00;


immr->i2add = 0x80;    // I2C MPC860 address = 0x80
immr->i2brg = 0x20;    // divide ratio of BRG divider
immr->i2cer = 0x17;    // Clear out I2C events by setting relevant bits to "1"
immr->i2cmr = 0x17;    // Enable interrupts from I2C in corresponding I2CER
immr->i2mod = 0x01;    // Enable I2C bus

....
```

# I2C parameter RAM initialization

- Five of the 15 field I2C parameter RAM need to be configured in the initialization of I2C on the MPC860.
    - ✓Receive function code register (RFCR),
    - ✓Transmit function code register (TFCR),
    - ✓Maximum receive buffer length register (MRBLR),
        - o Maximum number of bytes the I2C receiver writes to a receive buffer before moving to the next buffer.
    - ✓Base value of the receive buffer descriptor array (Rbase),
    - ✓Base value of the transmit buffer descriptor array (Tbase)
        - o Indicate where the BD(buffer descriptor) tables begin in the dualport RAM. Setting

# Cntd..

```
// I2C Parameter RAM Initialization
....

// specifies for reception big endian or true little endian byte ordering and channel # 0
immr->I2Cpram.rfcr = 0x10;

// specifies for reception big endian or true little endian byte ordering and channel # 0
 immr->I2Cpram.tfcr = 0x10;
 immr->I2Cpram.mrblr  = 0x0100; // the maximum length of I2C receive buffer
 immr->I2Cpram.rbase = 0x0400; // point RBASE to first RX BD
immr->I2Cpram.tbase = 0x04F8; // point TBASE to TX BD

....
```

# Board I/O Driver Examples

- The board I/O subsystem components that require software management include components integrated on the master processor, I/O slave controller(Optional)

- The I/O controllers have a set of status and control registers used to control the processor and check on its status

- Depending on the I/O subsystem, commonly all or some combination of all of the 10 functions from the list of device driver functionality including:
  - ❑ **I/O Startup**,
    - o Initialization of the I/O upon power-on or reset
  - ❑ **I/O Shutdown**
    - o Configuring I/O into its power-off state
  - ❑ **I/O Disable**
    - o Allowing other software to disable I/O on-the-fly

# Cntd..

❑ **I/O Enable**
  o Allowing other software to enable I/O on-the-fly

❑ **I/O Acquire**
  o Allowing other software gain singular (locking) access to I/O

❑ **I/O Release**
  o Allowing other software to free (unlock) I/O

❑ **I/O Read**
  o Allowing other software to read data from I/O

❑ **I/O Write**
  o Allowing other software to write data to I/O.

❑ **I/O Install**
  o Allowing other software to install new I/O on-the-fly.

❑ **I/O Uninstall**
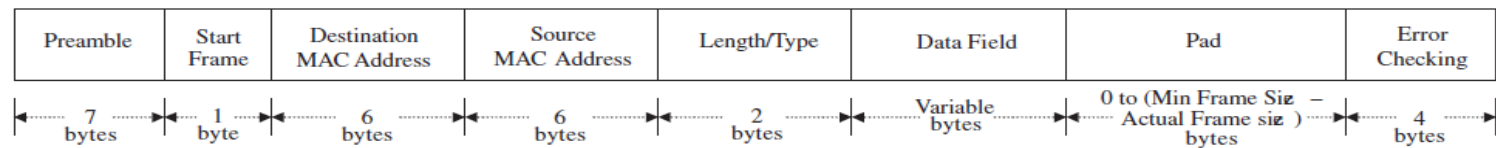  o Allowing other software to remove installed I/O on-the-fly.

# Examples of I/O startup device drivers

- Example- **Ethernet initialization routines**
  - Demonstrate how I/O can be implemented on more complex architectures
- Ethernet component that can be integrated onto the master processor is called the **Ethernet Interface**.
  - It contains the only firmware (software) that is implemented
  - The software is dependent on how the hardware supports two main components of the IEEE802.3 Ethernet protocol:
    - Media access management
    - Data encapsulation.

# Ethernet initialization routines

- In an Ethernet LAN, all devices connected via Ethernet cables can be set up as a bus or star topology

**Basic Ethernet Frame**

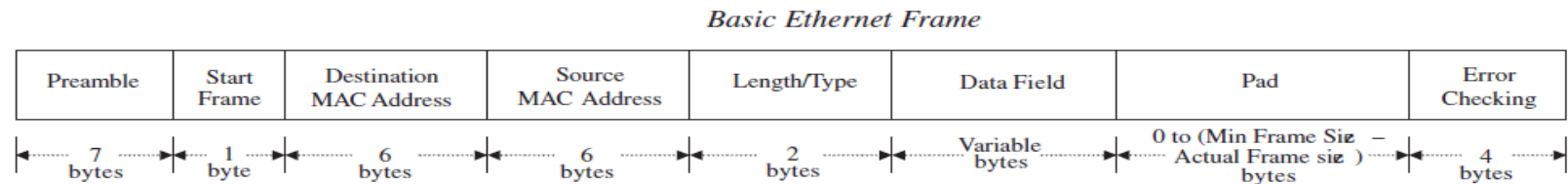| Preamble | Start Frame | Destination MAC Address | Source MAC Address | Length/Type | Data Field | Pad | Error Checking |
|---|---|---|---|---|---|---|---|
| 7 bytes | 1 byte | 6 bytes | 6 bytes | 2 bytes | Variable bytes | 0 to (Min Frame Size − Actual Frame size ) bytes | 4 bytes |

- **Ethernet frame**
  - Contains the data as well as the information needed to communicate to each device which device the data is actually intended for
  - The preamble bytes tell devices on the LAN that a signal is being sent.
  - They are followed by "10101011" to indicate the start of a frame.
  - The media access control (MAC) addresses in the Ethernet frame are physical addresses unique to each Ethernet interface in a device, so every device has one.

# Ethernet frame(cntd..)

- The **data field** can vary in size.
  - If the data field is less than or equal to 1500 then the **Length/Type** field indicates the number of bytes in the data field.
  - If the data field is greater than 1500, then the type of MAC protocol used in the device that sent the frame is defined in **Length/Type.**
- **Pad field**
  - While the data field size can vary, the MAC Addresses, the Length/Type, the Data, Pad, and Error checking fields must add up to be at least 64 bytes long.
  - If not, the pad field is used to bring up the frame to its minimum required length.
- The **error-checking field** is created using the MAC Addresses, Length/Type, Data Field, and Pad fields.
  - A 4-byte CRC (cyclical redundancy check) value is calculated from these fields and stored at the end of the frame before transmission.
  - At the receiving device, the value is recalculated, and if it doesn't match the frame is discarded.

*Basic Ethernet Frame*

| Preamble | Start Frame | Destination MAC Address | Source MAC Address | Length/Type | Data Field | Pad | Error Checking |
|---|---|---|---|---|---|---|---|
| 7 bytes | 1 byte | 6 bytes | 6 bytes | 2 bytes | Variable bytes | 0 to (Min Frame Size − Actual Frame size) bytes | 4 bytes |

# Serial communication controllers (SCCs)

- Configuring the MPC823 to implement Ethernet is done through SCCs

- The MPC823 has two serial communication controllers (SCC2 and SCC3) that can be configured independently to implement different protocols.

- They can be used to implement bridging functions, routers, gateways, and interfaces with a wide variety of standard WANs, LANs, and proprietary networks etc.

- It does not include the physical interface, but it is the logic that formats and manipulates the data obtained from the physical interface

**MPC823 Ethernet Driver Pseudocode**

```
// disabling SCC2
    // Clear GSMR_L[ENR] to disable the receiver
    GSMR_L = GSMR_L & 0x00000020
    // Issue Init Stop TX Command for the SCC
    Execute Command (GRACEFUL_STOP_TX)
    // clear GSLM_L[ENT] to indicate that transmission has stopped
    GSMR_L = GSMR_L & 0x00000010


-=-=-=-=


 //  Configure port A to enable TXD1 and RXD1 – step 1 from user's manual
PADIR =  PADIR &  0xFFF3      // Set PAPAR[12,13]
PAPAR = PAPAR  | 0x000C                  // clear PADIR[12,13]
PAODR = PAODR & 0xFFF7      // clear PAODR[12]


//  Configure port C to enable CLSN and RENA – step 2 from user's manual
PCDIR = PCDIR  & 0xFF3F                  // clear PCDIR[8,9]
PCPAR = PCPAR & 0xFF3F                  // Clear PCPAR[8,9]
PCSO =   PCSO  | 0x00C0                  // set PCSO[8,9]


//step 3 – do nothing now


// configure port A to enable the CLK2 and CLK4 pins.- step 4 from user's manual
PAPAR = PAPAR  | 0x0A00                 //  set PAPAR[6] (CLK2) and PAPAR[4] (CLK4).
PADIR = PADIR & 0xF5FF                  // Clear PADIR[4] and PADIR[6]. (All 16-bit)


//  Initializing the SI Clock Route Register (SICR) for SCC2.
//  Set SICR[R2CS] to 111 and Set SICR[T2CS] to 101,  Connect SCC2 to NMSI  and  Clear
SICR[SC2] – steps 5 & 6 from user's manual
SICR = SICR & 0xFFFFBFFF
SICR = SICR | 0x00003800
SICR = (SICR & 0xFFFFF8FF) | 0x00000500


//  Initializing the SDMA configuration register – step 7
 SDCR = 0x01                            // Set SDCR to 0x1 (SDCR is 32-bit) – step 7 from user's manual


// Write RBASE in the SCC1 parameter RAM to point to the RxBD table and the TxBD table in the
// dual-port RAM and specify the
// size of the respective buffer descriptor pools.  - step 8 user's manual
RBase = 0x00 (for example)
```

# Cntd..

```
RxSize =  1500 bytes (for example)
TBase = 0x02 (for example)
TxSize = 1500 bytes (for example)
Index = 0
While (index < RxSize) do
{
//Set up one receive buffer descriptor that tells the communication processor that the next packet is
//ready to be received – similar to  step 25
//Set up one transmit buffer descriptor that tells the communication processor that the next packet is
//ready to be transmitted – similar step 26
index = index+1 }

//Program the CPCR to execute the INIT_RX_AND_TX_PARAMS – deviation from step 9 in user's
//guide
execute Command(INIT_RX_AND_TX_PARAMS)

//write RFCR and TFCR with 0x10 for normal operation (All 8-bits) or 0x18 for normal operation
//and Motorola/Freescale byte ordering – step 10 from user's manual
RFCR = 0x10
TFCR = 0x10

//Write MRBLR with the maximum number of bytes per receive buffer and assume 16 bytes – step
//11 user's manual
MRBLR = 1520

// write C_PRES with 0xFFFFFFFF to comply with the 32 bit CRC-CCITT – step 12 user's manual
C_PRES = 0xFFFFFFFF

// write C_MASK with 0xDEBB20E3 to comply with the 16 bit CRC-CCITT – step 13 user's
// manual
C_MASK = 0xDEBB20E3

// Clear CRCEC, ALEC, and DISFC for clarity – step 14 user's manual
CRCEC = 0x0
ALEC = 0x0
DISFC = 0x0

// Write PAD with 0x8888 for the PAD value – step 15 user's manual
PAD = 0x8888

//Write RET_LIM to specify how many retries (with 0x000F for example) – step 16
RET_LIM = 0x000F

//Write MFLR with 0x05EE to make the maximum frame size 1518 bytes – step 17
MFLR = 0x05EE
```

# Thank You