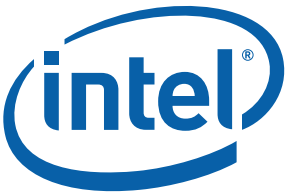


Intel® 81348 I/O Processor Initialization

Application Note

September 2006

Order Number: 315047-001US

Contents

Figures

Tables

Revision Histroy

Date	Revision	Description
September 2006	001	Initial Release.

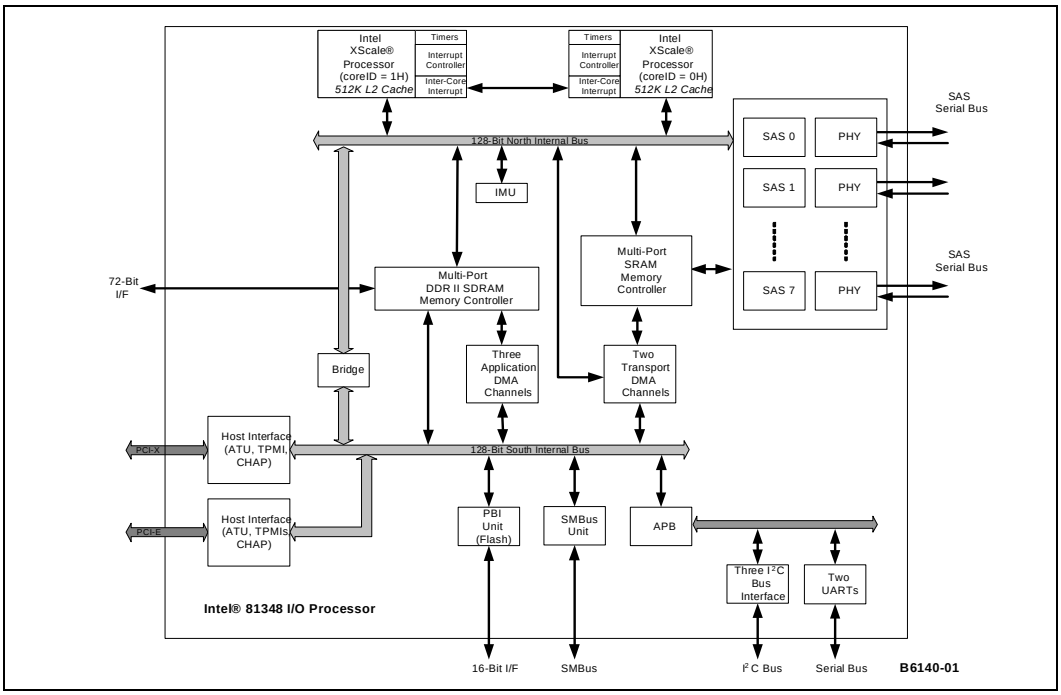
1.0 Introduction

This document outlines the Intel® 81348 I/O processor (81348) initialization based on Intel XScale® microarchitecture[1]. The initialization procedure described in this document can be implemented by an application, to bootstrap the 81348 Application Core (Core1). The procedure and sample code provided is used by RedBoot* running on the Application Core, to bootstrap the *Host Bus Adapter Customer Reference Board Manual for Intel® 8134x I/O Processors* (CRB). The intended audience for this document is software and hardware engineers developing a product based on the 81348, that require information on its initialization.

The 81348 is a dual-Intel XScale® processor, dual-interface architecture. One cores is dedicated to running the SAS transport firmware (FW) and is called the Transport Core (Core0). The other core is the "Application Core", where customer BSPs and applications execute. Intel provides a 2 MB binary, residing at 0x0 in Flash and provides for both initial boot code (for both cores, called "common boot"), as well as FW that provides programming interface into the transport core. The common boot code model assures synchronized start-up of the cores, while maintaining maximum flexibility for the application core developer. Source code for the common boot code is not provided. However, full particulars on the state of the application core, upon first customer instruction fetch, is covered later in this document.

The dual-interface references the fact that there is no transparent bridge inside the 81348 architecture. Instead, two Address Translation Units (ATUs) have been implemented. One is the ATUe, which connects to a PCI Express Bus and the other is the ATU-X, which connects to a PCI-X bus. For the CRB, the host-connection is on the PCIe/ATUe unit and the ATU-X is the Central Resource for the PCI-X bus.

Figure 1. Intel® 81348 I/O Processor Initialization 8-Port Functional Block Diagram



1. ARM* architecture compliant.

1.1 Related Documentation

Below is a list of related documents:

- The *Intel® 81348 I/O Processor Developer's Manual*, covers the same kind of information as previous generations of I/O processors (IOPs); operation of the part and its various units, register interfaces, etc.
- *Intel XScale® Core Developer's Manual* (http://www.intel.com/design/intelxscale/273473.htm)
- Intel XScale® Technology Web site (http://www.intel.com/design/intelxscale/)
- *Intel®81348 I/O Processor Specification Update*, contains detailed ***errata information***.
- *ARM Architecture Reference Manual*
- RedBoot Homepage: http://ecos.sourceware.org/redboot/

Documentation for the customer reference board, which was used to develop the example code, can be found in:

- *Host Bus Adapter Customer Reference Board Manual for Intel® 8134x I/O Processors* (CRB).

Note: Contact your local Intel Field Representative for the latest SSAS, SAS/SATA and SCDL documentation.

2.0 Terms and Definitions

Table 1. Terms and Definitions (Sheet 1 of 2)

Term	Definition
ADMA	Application DMA
API	Application Programming Interface
Application Core	Core 1 Processor, based on Intel XScale® microarchitecture. This is the processor on which the initialization code runs to bootstrap the board.
ARM	Refers to both the microprocessor architecture and the company that licenses it.
ATU	Address Translation Unit
ATUe	Address Translation Unit for PCI Express Bus
ATU-X	Address Translation Unit for PCI-X Bus
Common Boot Code	The common instructions executed by both cores upon power-up. This code is provided in binary format only as part of the SAS Transport Firmware Image.
CRB	Customer Reference Board
DCache	Data Cache
DDR	Double Data Rate
DIMM	Dual Inline Memory Module
DMA	Direct Memory Access
eCos	Embedded Configurable operating system. An open-source, royalty-free, highly configurable, application-specific operating system ideal for embedded systems development originally developed by Cygnus Solutions*; then bought by Red Hat; now publicly supported through an open-source development process.
FACx	Flash Access for Core x semaphore, where x can be 0 or 1. These two semaphores coordinate the usage of Flash, as it is shared between the Application and Protocol core.
FW	Firmware
ICache	Instruction Cache
I^2C	Inter-Integrated Circuit
IBL	Intel Business Link
I/O	In/Out
IOP	I/O processor
MB	MegaBytes
MHz	Mega Hertz
MMR	Memory Mapped Register
MMU	Memory Management Unit
PCI	Peripheral Component Interface
PCIe	PCI Express bus
PCI-X	PCI-X bus
PCSR	PCI Configuration and Status Register
PHY	Physical Layer
Protocol Core	Core 0 Processor, based on Intel XScale® microarchitecture. This is the processor on which the SAS Transport Firmware runs to control the SAS Ports.
RedBoot	Red Hat Embedded Debug and Bootstrap firmware is a complete bootstrap environment for embedded systems. Based on the eCos RTOS Hardware Abstraction Layer.
SAS	Serial Attached 'Small Computer Systems Interface (SCSI)'

Table 1. Terms and Definitions (Sheet 2 of 2)

Term	Definition
SDRAM	Synchronous Dynamic Random Access Memory
SPD	Serial Presence Detect
SRAM	Synchronous Random Access Memory
Transport Core	Another term used for the Protocol Core
Transport Firmware	The firmware loaded in the first 2MB of Flash which contains the code to run the Protocol Core and the SAS engines.
TSR	Test and Set Register
UART	Universal Asynchronous Receiver-Transmitter

3.0 Memory Mapped Registers (MMR)

Because writes are posted when an MMR value setting needs to be set (before the next operation occurs), then a read from the MMR must be issued after the write, to guarantee that the data has made it to the MMR. The MMR base address defaults to 0xFFD8.0000 and each register in the relevant specification is an offset from the MMR base address. Do not change this base address since it controls MMR access for the transport core as well. When register locations need to be remapped from the application core perspective, use the MMU to do this.

Note: In previous parts, many MMRs had dual-access, either memory mapped or coprocessor. In the 81348 however, registers are either one or the other. For example, the ICU registers in the 81348 are only accessible via CP access, whereas in the Intel® 80331 I/O processor (80331)/Intel® 80332 I/O processor (80332) they can be accessed via either method.

4.0 Initialization

The sequence of events that occur when the *Host Bus Adapter Customer Reference Board Manual for Intel® 8134x I/O Processors* is powered on, is as follows:

1. Reset:
   Both Application and Protocol Cores fetch the first instruction at Physical Address 0x0, which is located in Flash. This code is provided as a binary to developers as part of the SAS Transport Firmware and is called "Common Boot Code."
2. Common Boot Code Execution:
   Common boot code (including exception handler stubs) is locked into ICache and Flash and, relocated to physical address 0xF000_0000. Application and Protocol Cores execute Common Boot Code in parallel, until the Application core is instructed to jump to the 2 MB offset in Flash, which is where the Application core initialization code (for example, RedBoot) is stored.
3. Application Firmware Entry (at 2 MB offset in Flash):
   Interrupts are disabled and coprocessor access is enabled.
4. PBAR1 Initialization:
   Is setup so onboard peripherals (LEDs and CPLD registers) can be accessed.
5. Enable MMU:
   Memory Management unit is enabled with the Translation Table Base (TTB) register referencing the Page Table Entries in Flash.
6. Setup L1 Cache RAM:
   Create a temporary RAM by locking L1 DCache in order to create a stack so standard C routines can be called.
7. DDR-II Setup:
   SDRAM SPD is scanned for operating parameters, memory is scrubbed, and ECC is enabled.
8. ATU Initialization:
   The Address Translation Unit (ATU) is setup so that Configuration Retry can be released, which allows a host BIOS to continue booting.
9. ROM-to-RAM:
   RedBoot Initialization code is copied to SDRAM in order to continue initialization by executing out of SDRAM. The FAC1 semaphore is released after execution continues out of RAM.
10. ATU Outbound Memory Window Translation.
11. UART Initialization.

Each of these events is described in more detail in the following sections.

4.1 Reset

The 81348 must have its exception vectors located at physical address 0x0 when powered on. When the 81348 comes out of reset, the first instruction fetched and executed is the reset exception vector, which is located at address 0x0. Both of the cores (Application and Protocol Cores) inside the 81348, each send individual requests on the internal bus, for a 4-byte read from address 0x0. The default setting for the 81348, is to enable Peripheral Bus Interface 0 (PBI0) to claim internal bus transactions, starting at address 0x0. The *Host Bus Adapter Customer Reference Board Manual for Intel® 8134x I/O Processors* uses an 8 MB Intel StrataFlash® device on PBI0, so the reset vector is read from the Flash and returned to the requesting core. The Application Core initialization code must be stored at a 2 MB, offset from the base of Flash, because the first 2 MB of Flash is reserved for Common Boot code and Transport Firmware. The source code for these binaries cannot be released.

4.2 Common Boot Code

The "Common Boot Code" is executed by both cores at power-up. The Application firmware must be located at the 0x20_0000 (2M) offset from the base of Flash and is jumped into by the Common Boot code. Because the common boot code remaps the Flash to 0xF000_0000, application firmware begins executing at address 0xF020_0000. The Application firmware has its typical ARM exception vectors located at the 2M offset in Flash. The Application firmware initializes SDRAM as soon as possible, jumps to execute out of SDRAM, and then releases the FAC1 semaphore.

When Common Boot Code jumps to the 2 M offset, the state of the Application Core is:

- ATUe outbound enable bit in the ATUCR is set.
- PBI0 is set to base address 0xF000_0000 and PBI0 control is 0x3DD.
- ICache lines are locked with the Common Boot code inside of it.
  - The Application Core unlocks and invalidates all of the ICache.
- FAC1 is taken by the Application Core.
  - The Application Core releases FAC1 as soon as it is running entirely out of SDRAM and does not need to read from Flash anymore.
- Exception vectors in the pre-alpha common boot code simply branch to themselves. The application core is aware of this and reload/lock their own vectors in ICache, remap via MMU, or make use of Vector Relocation Mode when desired.
- Exception vectors in the beta common boot code determine which core had the exception and, in the event of the application core, jump to relative off set from 0xF020_0000 of the exception that occurred. Further details on this scheme are provided when implemented.

4.2.1 Flash Semaphores (FAC0 and FAC1)

The Transport Core (Core0) and the Application Core (Core1) share Flash. In order to control Read and Write access to the Flash, two semaphores are defined, FAC0 and FAC1. The FAC semaphores need to be accessed byte-wise and are located in the Test and Set Registers (TSR) block at 0xFFD8_0B00.

Note: Hardware only provides for the hardware semaphores, but does *not* enforce the use of the Flash semaphores. Proper adherence to the Flash access methodology is critical to assure the integrity of Flash and correct operation of the system.

The two FAC semaphores are located at:

- FAC0: 0xFFD8.0B00
- FAC1: 0xFFD8.0B01

FAC Usage:

To Read from Flash, a core must claim its FAC semaphore (FAC0 for Core0 and FAC1 for Core1). To Write to Flash, a core must claim both semaphores.

To claim a semaphore, a core does a byte-wise read to the semaphore address. When it receives back a "0" or its core ID+1, then it has claimed the semaphore. When it receives back the other core ID+1, then it has not claimed the semaphore and must keep trying.

To release a semaphore, a core does a byte-wise write of 0x0 to the semaphore address. As soon as the Application Core is up and running completely out of SDRAM, it releases FAC1. The Common Boot code claims FAC1 for the Application Core before branching to the 0x20_0000 offset.

When the Application Core wants to read from Flash, it must obtain FAC1 by doing a byte read from FAC1 (0xFFD8_0B01). When the Application Core reads a 0x0 or 0x2, then it has obtained the semaphore. When it reads a 0x1, Core0 currently has the semaphore and the Application Core must retry the read until it gets back a 0x0 or 0x2.

4.2.2 Exception Vectors

The Application Firmware must have its exception vector table located at the 2 MB offset in Flash. The exception vector offsets are located Table 2. The exception vector address must contain an instruction to be executed. Typically, this instruction is an indirect load of the program counter with the address of an exception handling routine. See Example 1 for RedBoot implementation of the exception vector table.

When an exception occurs before the MMU has been initialized, the Application Core branches to the Common Boot exception handlers, which are locked in Instruction Cache. The Common Boot exception handlers determine that the exception occurred from the Application core and jumps to the appropriate offset in the Application Core exception table, which must be at 0xF020_0000. For example, when a data abort occurs during firmware execution, before the MMU has been setup, the core immediately branches to 0x10, which is the location of the data abort handler in the Common Boot Code, that is still locked in ICache. The data abort handler determines, via the Core ID, that the exception is from the Application core and it immediately branches to 0xF020_0010.

- Exception vectors in the pre-alpha common boot code simply branch to themselves. The application core should be aware of this and reload/lock their own vectors in icache, remap via MMU, or make use of Vector Relocation Mode when desired.
- Exception vectors in the beta release of the common boot code determines which core had the exception and, in the event of the application core, jump to a relative off set from 0xF020.0000 of the exception that occurred. Further details on this scheme are provided when implemented.

Table 2. Exception Vector Addresses

Exception Type	Address
Reset	0x0
Undefined Instruction	0x4
Software Interrupt (SWI)	0x8
Prefetch Abort (instruction fetch memory abort)	0xC
Data Abort	0x10
IRQ (Interrupt Request)	0x18
FIQ (Fast Interrupt Request)	0x1C

Example 1. Exception Vector Table in RedBoot

```
ldr     pc, .reset_vector                   // 0x00

ldr     pc, .undefined_instruction          // 0x04

ldr     pc, .software_interrupt             // 0x08

ldr     pc, .abort_prefetch                 // 0x0C

ldr     pc, .abort_data                     // 0x10

            .word 0                         // unused

ldr     pc, .IRQ                            // 0x18

ldr     pc, .FIQ                            // 0x1C
```

4.3 Application Firmware Entry

Using RedBoot, the first 81348 instructions executed, write to the current program status register (CPSR), to keep interrupts disabled, maintain supervisor mode, and enable coprocessor access. The I and F bits in the CPSR are the Interrupt Disable bits and when they are set, the 81348 cannot be interrupted by an IRQ or FIQ interrupt. See Table 3 for the layout of the CPSR.

Coprocessors are additional processing units within the 81348 core, these perform a specific hardware task. Each coprocessor contains a set of registers that control the coprocessor. These registers are accessed using MRC and MCR instructions. The 81348 contains four coprocessors, of which two must be explicitly enabled. Coprocessors 15 and 14 (CP15 and CP14) do not have to be explicitly enabled, but they can only be accessed in a privileged mode, such as supervisor.

- CP15: system control coprocessor used to control the MMU, caches, and other system attributes.
- CP14: performance monitoring unit and also contains trace buffer controls.

Coprocessors 7 and 6 need to be enabled within CP15 before they can be accessed.

- CP6: coprocessor controlling the Interrupt Controller Unit and two internal timers.
- CP7: contains error logging registers for the L2 cache and the Bus Interface Unit.

The code to enable CP7 and CP6 access is shown in Example 2.

Table 3. Current Program Status Register (CPSR)

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	N	Z	C	V	DNM/RAZ																				I	F	T	M4	M3	M2	M1	M0

Example 2. Enabling Coprocessor Access

```
#define CPSR_IRQ_DISABLE   0x80// IRQ disabled when =1

#define CPSR_FIQ_DISABLE   0x40// FIQ disabled when =1

#define CPSR_SUPERVISOR_MODE0x13


    .reset_vector

     ldr   r0,=(CPSR_IRQ_DISABLE|CPSR_FIQ_DISABLE|CPSR_SUPERVISOR_MODE)

     msr   cpsr, r0


      // enable coprocessor access

        // CP07 - Error logging for Ext. Bus and L2 Cache Parity errors

        // CP06 - Interrupts/Timers

     ldr   r0, =0x20c1              // CP7,CP6

     mcr   p15, 0, r0, c15, c1, 0
```

Example 3. Enable MMU

```
// This section of code is position independent
// It has been linked at 0x0 (RAM address)

// Get the physical address of the Memory Descriptor Table
ldr     r11, =(mmu_table + CYGMEM_REGION_rom)

#ifdef CYGSEM_HAL_ENABLE_L2_CACHE
// Allow table walks to load descriptors into L2
ldr     r0, =OUTER_CACHEABLE_TABLE_WALKS
add     r11, r0, r11
#endif

// Enable permission checks in all domains
ldr     r0, =0x55555555
mcr     p15, 0, r0, c3, c0, 0

// Value for the ARM Control register to Enable the MMU
mrc     p15, 0, r12, c1, c0, 0
orr     r12, r12, #(MMU_Control_M)
orr     r12, r12, #(MMU_Control_R)
#ifdef CYGSEM_HAL_ENABLE_L2_CACHE
orr     r12, r12, #(MMU_Control_L2)
#endif

// Branch to ICache boundary so code is executing from ICache
// when the MMU is enabled. This is not critical in the ROMRAM
// version because Common Boot has already remapped Flash and
// we will not be remapping Flash.
// For other RedBoot versions that remap Flash, this is required

b       icache_boundary

        .p2align 5
icache_boundary:

mcr     p15, 0, r11, c2, c0, 0  // Set the TTB
mcr     p15, 0, r12, c1, c0, 0  // Enable the MMU
```

4.4 PBAR1 Initialization

The next instructions executed by RedBoot sets-up Peripheral Bus Interface #1 (PBAR1), which has many of the peripherals on the board, including:

- CPLD
- Hex LEDs
- Rotary Switch
- Discrete LEDs

PBAR1 is placed at an offset of 32 MB above PBAR0, which equates to 0xF200_0000. Enabling PBAR1 allows RedBoot to output diagnostic codes to the LEDs, so a user can visually see when the boot sequence is completed. The memory map, including the location of the peripherals on PBAR1 is located in Appendix A, "Host Bus Adapter Customer Reference Board Manual for Intel® 8134x I/O Processors RedBoot Address Map".

4.5 Enable MMU

RedBoot enables the MMU using Page Table Entries (PTE), that are initially located in Flash. The majority of the PTE defined by RedBoot are 1 M section descriptors. An example macro, which sets up a 1 M section descriptor is shown in Example 4. An example call to this macro, which sets up a single 1 M section descriptor for the MMRs, is shown in Example 5. The only descriptors that RedBoot creates that are not 1 M section descriptors, are used for the ATU Outbound memory windows. The ATU Outbound Memory windows are located in 36-bit physical internal bus physical address space. To access the ATU Outbound memory windows requires a Super Section PTE descriptor to be created. Each Super Section PT entry references 16 MB of memory and the entry must be repeated in 16 consecutive locations in the MMU table. An example macro of setting up a Super Section descriptor is shown in Example 6. For more information on Super Section descriptor usage, refer to the *Intel XScale® Core Developer's Manual* (http://www.intel.com/design/intelxscale/273473.htm.)

Enabling the MMU allows RedBoot to use the 32 K Data Cache and the 512 K L2 cache. The MMU is enabled by first setting the Translation Table Base (TTB) register to point to the PT descriptors and then setting the MMU-enable and L2-enable bit in the ARM Control register in CP15.

The RedBoot code is eventually copied into RAM and executed, so RedBoot is linked at address 0x0. This means that RedBoot must calculate the proper offset into Flash for the MMU Page tables, before they are copied into RAM. After the Page Table Entries are copied to RAM, the TTB is updated with the address of the PTEs in RAM. The sample code for this is shown in Example 3 on page 17.

4.5.1 L2 Cache Considerations

When the 512 K unified Level Two (L2) cache is going to be used, the enable bit in the ARM Control register must be set when the MMU is enabled. Once the L2 bit is enabled, it cannot be disabled via the ARM Control register. The only way to disable the L2 cache is to change all page table entries to be Non Outer-Cacheable. When the page table entries are stored in the L2, the Outer Cacheable (OC) field in the TTB must be set to allow table walks to occur in L2. When the PTEs are going to be changed to disable L2 caching, then flush the L2 cache first and clear the OC bits in the TTB first. The L2 cache control commands, such as invalidate and flush can be found in the *Intel XScale® Core Developer's Manual*.

Example 4. Section Descriptor Macro

```
// form a first-level section entry

.macro FL_SECTION_ENTRY base,x,ap,p,d,c,b

.word (\base << 20) | (\x << 12) | (\ap << 10) | (\p << 9) |\

      (\d << 5) | (\c << 3) | (\b << 2) | 2

.endm
```

Example 5. Creating PTE for MMRs

```
// 0x0.FFD0.0000 - 0x0.FFE0.0000 (1MB)

    // 1MB PMMR

    // X=0, C=0, B=0

    .set__base,0xFFD

    .rept1

    FL_SECTION_ENTRY __base,0,3,0,0,0,0

    .set__base,__base+1

    .endr
```

Example 6. SuperSection Descriptor Macro

```
// form a first-level supersection entry

.macro FL_SUPERSECTION_ENTRY base,upper_base,x,ap,p,d,c,b

.word (\base << 24) | (\upper_base << 20) | (1 << 18) | (\x << 12) | (\ap << 10) |
(\p << 9) |\

      (\d << 5) | (\c << 3) | (\b << 2) | 2

.endm
```

4.6 Setup L1 Cache RAM

In order to call standard C functions, a stack must be present so that function prologs and epilogs can be executed properly. Since SDRAM has not been setup at this point, RedBoot locks L1 Data Cache as a temporary RAM, to create a stack, so that C functions can be called. The 81348 is capable of locking 24 K bytes of L1 Cache as RAM. This provides an adequate amount of stack space for initialization until SDRAM has been setup. The code to lock L1 Data Cache as RAM is shown in Example 7.

Example 7. Lock DCache as RAM

```
// This macro locks the CACHE as SRAM at address \sram_base_addr

.macro _enable_and_lock_cache_as_sram sram_base_addr

mcr     p15, 0,  r0, c7, c10, 4        // drain the write & fill buffers

mrc     p15, 0,  r0, c1, c0, 0         // Enable the Dcache

orr     r0,  r0, #MMU_Control_C

mcr     p15, 0,  r0, c1, c0, 0

mcr     p15, 0,  r0, c7, c10, 4        // drain the write & fill buffers

ldr     r0,  =1                        // enable the data cache lock mode

mcr     p15, 0,  r0, c9, c6, 0         // New DCache Lock register write command

CPWAIT r0

// What address to lock into cache

ldr     r0,  =\sram_base_addr

// How many lines to lock

ldr     r1,  =(HAL_DCACHE_LOCKABLE_SIZE/HAL_DCACHE_LINE_SIZE)

ldr     r2,  =0                        // Used

ldr     r3,  =0                        // to

ldr     r4,  =0                        // zero

ldr     r5,  =0                        // initialize

ldr     r6,  =0                        // the

ldr     r7,  =0                        // cache

ldr     r8,  =0                        // region

ldr     r9,  =0                        //

889:

mcr     p15, 0,  r0, c7, c2, 5         // allocate DCache line at r0

mcr     15,  0,  r0, c7, c10, 4        // drain the write & fill buffers

stmia   r0!, {r2-r9}                   // Zero initialize

subs    r1,  r1, #1                    // Check next line

bne     889b

mcr     p15, 0,  r0, c7, c10, 4        // drain the write & fill buffers

ldr     r0,  =0

mcr     p15, 0,  r0, c9, c6, 0         // disable the data cache lock mode

.endm
```

4.7 DDR-II Setup

The 81348 has a high-performance Memory Controller unit, which supports double data rate 2 (DDR-II) SDRAM DIMMs at frequencies of 400 MHz and 533 MHz. The Application Core has access to two I^2C units, one of which is used by RedBoot to read the serial presence detect (SPD) EEPROM, on the DIMM that is installed on the 8134x HBA Customer Reference Board. The SPD device on a DIMM contains information about:

- Refresh rate
- CAS latency
- Density
- Many other operating parameters for the DIMM

RedBoot reads each of these parameters, inserts the parameters into the DDR equations from the *Intel® 81348 I/O Processor Developer's Manual*, and then programs the MCU registers based on the outcome of the equations. Once the operating parameters have been programmed into the Memory Controller Block, the JEDEC initialization sequence must be sent to the DIMM, using the SDIR register in the MCU block. The example code for initializing the MCU is included in Appendix B, "MCU Initialization Code".

The SDRAM memory is placed at physical address 0x0 by writing to the SDRAM Base Register (SDBR). The SDRAM is then enabled by writing to the refresh register (RFR). Once the memory is enabled, it is cleared by writing zeroes to every location. One of the ADMA channels of the 81348 is used to write up to 15 MB sized blocks of zeroes into the SDRAM, which is faster than using the Intel XScale® microarchitecture to write zeroes to every location. Once the memory has been zeroed out, ECC is enabled to provide protection against single and multi-bit errors. Single bit errors can be corrected by the MCU, but multi-bit ECC errors are uncorrectable and causes a data abort to occur.

4.8 ATU Initialization

4.8.1 Inbound Window Initialization

Once memory has been setup, the Address Translation Units (ATU-e and ATU-X) inbound windows are initialized. When enabled, the ATUs claim inbound PCI transactions and use the translate value register to translate the PCI address to an internal bus address. It is important to initialize the ATU that is connected to the host, so that the retry bit can be cleared and the host BIOS can continue scanning the PCI bus. On 8134x HBA, RedBoot initializes the ATUe Inbound Limit (IALR) and Translate value register (IATVR) to translate inbound PCI addresses to SDRAM. The other BARs are not enabled by RedBoot. Once the limit and translate value registers have been setup, the Retry bit (bit2) is cleared in the PCI Configuration and Status Register (PCSR).

For the 256 MB DIMM that is installed on the 8134x HBA CRB, the IALR and IATVR are programmed with the following values:

- IALR0 : 0xF000_0000 (Setup for a 256 MB Limit)
- IATVR0 : 0x0000_0000 (Translate inbound accesses that hit BAR0 to 0x0 on the internal bus - which is claimed by the MCU)

4.8.2 Release PCI-X Bus Reset

The 8134x HBA CRB operates as a Central Resource on the PCI-X bus, meaning that the 81348 has to release Reset on the PCI-X bus before scanning and configuring any devices. To deassert Reset on the PCI-X bus, clear bit 21 in the PCSR of the ATU-X. Once bit21 is cleared, the FW must wait 2^25 PCI Clocks before doing any cycles on the bus.

Note: 2^25 PCI clocks is approximately 1.02 seconds at 33 MHz.

4.9 ROM-to-RAM

In order to release the FAC1 semaphore, RedBoot must be running entirely out of SDRAM. To accomplish this, RedBoot is linked at 0x0 and copies all of its initialization code into SDRAM. The macro to copy the entire RedBoot initialization sequence into RAM and continue execution out of SDRAM is shown in Example 8.

Example 8. Copy ROM to RAM Macro

```
// Flush the Prefetch Buffer for the PC

.macro PREFETCH_FLUSH reg0
   mov\reg0, #0
   mcr p15, 0, \reg0, c7, c5, 4
   .endm


#define COPY_ROM_TO_RAM                                                 ;\
    ldr     r0,=(CYGMEM_REGION_rom)                                     ;\
    ldr     r1,=SDRAM_PHYS_BASE                                         ;\
    ldr     r2,=(__bss_end)                                             ;\
1:  ldr     r3,[r0],#4                                                  ;\
    str     r3,[r1],#4                                                  ;\
    cmp     r1,r2                                                       ;\
    bne     1b                                                          ;\
    ldr     r0,=2f                                                      ;\
    mcr     p15,0,r1,c7,c5,0          /* clear instruction cache */     ;\
    mcr     p15,0,r1,c8,c5,0          /* flush I TLB only */            ;\
                                      /* cpuwait */                     ;\
    mrc     p15,0,r1,c2,c0,0          /* arbitrary read   */            ;\
    mov     r1,r1                                                       ;\
    sub     pc,pc,#4                                                    ;\
    PREFETCH_FLUSH r1                                                   ;\
    mov     pc,r0                                                       ;\
2:
#endif
```

Once RedBoot has completed the COPY_ROM_TO_RAM macro, the TTB register needs to be updated to point to the page table entries in SDRAM. Once the TTB is updated with the SDRAM address of the page table entries, RedBoot is executing solely out of SDRAM and can release the FAC1 semaphore. The code to release the FAC1 semaphore is shown in Example 9.

Example 9. Release FAC1

```
#define IMU_TSR_FAC1 0xffd80b01
// We have copied all our code to SDRAM and updated the TTB.
     //  We are executing solely out of SDRAM and can release our FAC1 semaphore
        ldr     r0, =IMU_TSR_FAC1
        mov     r1, #0
        strb    r1, [r0]
```

4.10 ATU Outbound Window Initialization

There are four outbound windows for each ATU and they are located above the 4 GB memory region. The outbound memory windows are a way for the Intel XScale® processor to directly perform PCI Memory transactions (Read/Write) on a PCI Bus (PCI-X or PCIe) by reading or writing to the Outbound Memory Window. Each of the four outbound windows is 4 GB in size. In order to access the outbound window for either ATU, the Intel XScale® processor must create SuperSection Page Table Descriptors for the MMU. These SuperSection descriptors map a 32-bit virtual address into a 36-bit physical address. Each SuperSection descriptor describes a memory range of 16 MB and the same SuperSection descriptor must be repeated in 16 consecutive memory locations in the MMU Table (as each entry in the table is supposed to define 1 MB - so a 16 MB descriptor must be repeated 16x).

The Outbound Memory Windows only have a translate value register for the Upper 32-bits of the 64-bit address. The lower 32-bits of the Outbound Memory transaction pass through untranslated (in other words, there is no outbound translation value register for the lower 32 bits). The key point is, that the SuperSection page table entries must be modified each time that the Outbound Memory lower 32-bit window needs to move. An example function of how to modify the SuperSection PT Entries to slide the Outbound Memory Window is shown in Appendix C, "Modifying PT Entries to Slide Outbound Window".

The Outbound Enable bits in each ATUs Configuration Register (offset 0x70) must be set to '1' before any outbound transactions occurs. Be careful when setting the ATUe ATUCR register to just "OR" the register with a '2' so you do not clear bit 6 (which can prevent hosts from booting).

4.10.1 Conflicts

The Outbound Memory Windows (OUMBARs) for the ATUe and ATU-X are enabled and overlap by default when the 81348 is powered on. This causes internal bus conflicts when writing to an Outbound Window as each ATU tries and claim that address. The simplest fix is, to only enable OUMBAR0 for ATU-X, and OUMBAR1 for ATUe. The enable bit in the other OUMBARs (Bit31) must be clear to disable them. When not planning on utilizing the PCI-X bus, disabling all of the OUMBARs of the ATU-X prevents the conflict.

4.11 UART Initialization

There is one UART available for Application Core output on the 8134x HBA CRB, its register base is located at 0xFFD8_2340. The UART supports all functions of a 16550 UART. RedBoot sets up the UART to operate at a baud rate of 115200, word length of 8 bits, no parity bits, and enables the Receive and Transmit FIFOs. Code for enabling the UART is shown in Example 10.

Example 10. Initialize UART - (ROB) NEEDS SOME FORMATTING HELP!

```
#define CYG_DEV_IER 0x4
#define SIO_IER_UUE 0x40
#define CYG_DEV_LCR 0xC
#define SIO_LCR_WLS0 0x1
#define SIO_LCR_WLS1 0x2
#define SIO_LCR_DLAB is 0x80

// Baud Rate Table for 81348
struct baud_config baud_conf[] = {
    {9600,   0, 217},
    {19200,  0, 109},
    {38400,  0,  54},
    {57600,  0,  36},
    {115200, 0,  18}};

typedef struct {
    cyg_uint32* base;
    cyg_int32 msec_timeout;
    int isr_vector;
    cyg_int32 baud_rate;
    int irq_state;
    channel_data_t;

static void
cyg_hal_plf_serial_init_channel(void* __ch_data)
{

    cyg_uint32* base = ((channel_data_t*)__ch_data)->base;
    channel_data_t* chan = (channel_data_t*)__ch_data;

// Enable the UART function
HAL_WRITE_UINT8(base+CYG_DEV_IER, SIO_IER_UUE);

// 8-1-no parity.
HAL_WRITE_UINT8(base+CYG_DEV_LCR, SIO_LCR_WLS0 | SIO_LCR_WLS1);
    set_baud( chan );
    HAL_WRITE_UINT8(base+CYG_DEV_FCR, 0x07);  // Enable & clear FIFO
}

static int
set_baud( channel_data_t *chan )
{
    cyg_uint32* base = chan->base;
    cyg_uint8 i;

for (i=0; i<(sizeof(baud_conf)/sizeof(baud_conf[0])); i++)
    {
        if (chan->baud_rate == baud_conf[i].baud_rate) {
            cyg_uint8 lcr;
            HAL_READ_UINT8(base+CYG_DEV_LCR, lcr);
            HAL_WRITE_UINT8(base+CYG_DEV_LCR, lcr|SIO_LCR_DLAB);
            HAL_WRITE_UINT8(base+CYG_DEV_DLL, baud_conf[i].lsb);
            HAL_WRITE_UINT8(base+CYG_DEV_DLM, baud_conf[i].msb);
            HAL_WRITE_UINT8(base+CYG_DEV_LCR, lcr);
            return 1;
        }
    }
    return -1;
}
```

5.0 Accessing Flash After Initialization

Once the board has been initialized, RedBoot requires write access to Flash when the FIS commands are executed. In order to do so, it must claim FAC1 and FAC0. Once it has written to the Flash it can release FAC0 and FAC1. The code to obtain and release the FAC semaphores is located in Example 11.

Example 11. FAC Macros

```
// IMU_TSR_FAC0 is 0xFFD8_0B00

// IMU_TSR_FAC1 is 0xFFD8_0B01

#define GET_FAC_SEMAPHORES()                                    \

    volatile cyg_uint8  *fac0_p = IMU_TSR_FAC0;                 \

    volatile cyg_uint8  *fac1_p = IMU_TSR_FAC1;                 \

    volatile cyg_uint8  fac0, fac1;                             \

    /* Try to get semaphores - could add timeout here */        \

    do {                                                        \

        fac0 = *fac0_p;                                         \

        fac1 = *fac1_p;                                         \

    } while ((fac0 == 0x1) || (fac1 == 0x1));


#define RELEASE_FAC_SEMAPHORES()                                \

    volatile cyg_uint8  *fac0_p = IMU_TSR_FAC0;                 \

    volatile cyg_uint8  *fac1_p = IMU_TSR_FAC1;                 \

    *fac0_p = 0;                                                \

    *fac1_p = 0;
```

6.0 Summary

This document outlined the initialization sequence of the 81348. The procedure described in this document can be utilized by any application to bootstrap the 81348. Code examples inside this document come directly from RedBoot initialization code which is running on the 8134x HBA CRB. The 8134x HBA.

Refer to the references section for more documentation on the Intel® 81348 I/O processor, *Host Bus Adapter Customer Reference Board Manual for Intel® 8134x I/O Processors* CRB, *Intel XScale® Core Developer's Manual*, and the interaction with the Transport Core. For more information on RedBoot, visit the RedBoot homepage: http://ecos.sourceware.org/redboot/.

Also contact your Intel Field Representative for the RedBoot source code which supports 81348.

Appendix A *Host Bus Adapter Customer Reference Board Manual for Intel® 8134x I/O Processors* RedBoot Address Map

```
//  Virtual Address    Physical Address   Size (MB)  Description
//  ---------------    ----------------   ---------  -----------
//  0x00000000         0x0.0000.0000      2048       SDRAM - 64-bit ECC
//  0x80000000         0x1.8000.0000       128       ATU-X Outbound Mem Window
//  0x88000000         0x2.8800.0000       128       ATUe Outbound Mem Window
//  0x90000000         0x0.9000.0000         1       ATU-X Outbound IO Window
//  0x90100000         0x0.9010.0000         1       ATUe Outbound IO Window
//  0x90200000         ----------          254       Unused
//  0xA0000000         0x0.0000.0000       512       SDRAM - 64-bit Uncached Alias
//  0xC0000000         ----------          256       Unused
//  0xD0000000         ----------          512       Unused (32-bit ECC unused)
//  0xF0000000         0x0.F000.0000        32       Flash (PCE0#)
//  0xF2000000         0x0.F200.0000         1       PCE1#
//  0xF2100000         0x0.F210.0000        15       Unused
//  0xF3000000         0x0.F300.0000         1       Cache Flush, DCache Lock
//  0xF3100000         ----------          204       Unused
//  0xFFD00000         0x0.FFD0.0000         1       MMR
//  0xFFE00000         0x0.FFE0.0000         2       Unused


PCE1 (PBAR1) Mapping of peripherals
//  0xF2000000 : Product Code Register
//  0xF2010000 : Board Stepping Register
//  0xF2020000 : CPLD Firmware Revision Register
//  0xF2030000 : Discrete LEDs Register
//  0xF2040000 : 7-seg LED (Left)
//  0xF2050000 : 7-seg LED (Right)
//  0xF2060000 : Buzzer Control Register
//  0xF2070000 : 32kB 8-bit access NVRAM
//  0xF20D0000 : Rotary switch Regsiter
```

Appendix B MCU Initialization Code

```
// Definitions for the MACROS used below are available in the
// RedBoot source code. Contact your field sales representative
// for access to the 81348 RedBoot source.


//
// Perform MCU Initialization
//
void mcu_initialization(cyg_uint32 *sdram_size,
                        cyg_bool *sdram_installed,
                        cyg_bool *ecc_cap)
{
    cyg_uint32 ddr_type, num_banks, num_rows, num_cols, temp;
    cyg_uint32 refresh = 0, sdram_width = 0, tCAS = 0;
    cyg_uint32 ps_per_cycle, tRP, tRCD, tRAS, tRC, tWTR, tRFC, tREG, tWR;
    cyg_uint32 tRTP, tEDP;
    cyg_uint32 bank_size_Mbytes = 0, sbsr_val, emrs = 0, mrs = 0;
    cyg_uint32 sdcr0_val = 0, sdcr1_val = 0, sdbr_val = 0;
    cyg_uint32 t;
    cyg_uint8 subfield_a, subfield_b, subfield_c, i2c_bytes[SPD_NUM_BYTES],
        cksum = 0;
    int i;
    char buf;


    i2c_init();


    // Write 0 to the SDRAM SPD to reset the EEPROM pointer
    buf = 0;
    if (i2c_write_bytes(SDRAM_DEVID, &buf, 1) == -1) {
        // Timeout Assume no device in system
        MEM_PARAM_ERROR(0, 7);
    }


    // Read the bytes
```


```
if (i2c_read_bytes(SDRAM_DEVID, i2c_bytes, SPD_NUM_BYTES) == -1) {
    // Timeout Assume no device in system
    // hit the leds if an error occurred
    MEM_PARAM_ERROR(0, 8);
}

// Verify the checksum
for (i = 0; i < SPD_NUM_BYTES; i++)
    if (i != SPD_CHECKSUM)
        cksum += i2c_bytes[i];
    if (cksum != i2c_bytes[SPD_CHECKSUM])
        MEM_PARAM_ERROR(0, 9);

// Verify/Determine the DDR type
// Only support DDR-II for IQ8134x
ddr_type = i2c_bytes[SPD_MEMTYPE];
if (ddr_type != SPD_MEMTYPE_SDRAM_DDRII)
    MEM_PARAM_ERROR(0, A);

// Verify the number of row addresses
num_rows = i2c_bytes[SPD_NUM_ROWS];
if ((num_rows < SBR_ROW_ADDRESS_MIN) ||
    (num_rows > SBR_ROW_ADDRESS_MAX))
    MEM_PARAM_ERROR(0, B);

// Verify the number of column addresses
num_cols = i2c_bytes[SPD_NUM_COLS];
if ((num_cols < SBR_COL_ADDRESS_MIN) ||
    (num_cols > SBR_COL_ADDRESS_MAX))
    MEM_PARAM_ERROR(0, C);

// Verify/Determine the number of banks
// DDR-II
num_banks = (i2c_bytes[SPD_BANKCNT] & SPD_DDRII_BANKCNT_MASK) + 1;
```

```
    if (num_banks > 2)
        MEM_PARAM_ERROR(0, D);

    // Verify/Determine the data bus width
    if (i2c_bytes[SPD_MOD_WIDTH1] == 32)
        // 32-bit data
        sdcr0_val |= SDCR_BUS_WIDTH_32_BITS;
    else if (i2c_bytes[SPD_MOD_WIDTH1] == 40)
        // 32-bit data + 8-bit ECC
        sdcr0_val |= SDCR_BUS_WIDTH_32_BITS;
    else if (i2c_bytes[SPD_MOD_WIDTH1] == 64)
        // 64-bit data
        sdcr0_val |= SDCR_BUS_WIDTH_64_BITS;
    else if (i2c_bytes[SPD_MOD_WIDTH1] == 72)
        // 64-bit data + 8-bit ECC
        sdcr0_val |= SDCR_BUS_WIDTH_64_BITS;
    else
        MEM_PARAM_ERROR(0, E);

    if (i2c_bytes[SPD_MOD_WIDTH2] != 0)
        MEM_PARAM_ERROR(0, F);

    // Verify/Determine ECC capabilities
#ifdef CYGSEM_HAL_ARM_IOP34X_ENABLE_ECC
    if (i2c_bytes[SPD_CONFIG] == SPD_CONFIG_NONE)
        // No error correction
        *ecc_cap = 0;
    else if (i2c_bytes[SPD_CONFIG] == SPD_CONFIG_ECC)
        // ECC error correction
        *ecc_cap = 1;
    else
        MEM_PARAM_ERROR(1, 0);
#else
    *ecc_cap = 0;
```

```
#endif

    // Verify/Determine DDR Cycle Time.  266MHz = DDR2-533. 200MHz=DDDR2-400
    // Also make sure we're strapped for that setting
    if ((*PBIU_ESSTSR0 & MEM_FREQ_MASK) == MEM_FREQ_533) {
        ps_per_cycle = 3750;
        if (i2c_bytes[SPD_CYCLE_TIME] == SPD_CYCLE_200MHZ)
            MEM_PARAM_ERROR(1, 1);   // DIMM doesn't support 533MHz operation
    }
    else if ((*PBIU_ESSTSR0 & MEM_FREQ_MASK) == MEM_FREQ_400)
        ps_per_cycle = 5000;
    else
        MEM_PARAM_ERROR(1, 1);

    // Verify/Determine Refresh rate based on Refresh setting and
    // DDR cycle time.  Also set ps_per_cycle for later calculations.
    if ((i2c_bytes[SPD_REFRESH] & SPD_RFR_MASK) == SPD_RFR_15_625us) {
        if (ps_per_cycle == 3750)
            refresh = RFR_533_MHZ_15_6us;
        else
            refresh = RFR_400_MHZ_15_6us;
    }
    else if ((i2c_bytes[SPD_REFRESH] & SPD_RFR_MASK) == SPD_RFR_7_8us) {
        if (ps_per_cycle == 3750)
            refresh = RFR_533_MHZ_7_8us;
        else
            refresh = RFR_400_MHZ_7_8us;
    }
    else
        MEM_PARAM_ERROR(1, 2);

    // Verify/Determine SDRAM Width
    sdram_width = i2c_bytes[SPD_SDRAM_WIDTH];
    if ((sdram_width != 16) && (sdram_width != 8))
```

```
            MEM_PARAM_ERROR(1, 3);

        // Determine CAS and WDL fields.  IQ8134x supports tCAS of 4, 5
        // tCAS comes from SPD device.
        // CAS field is set from tCAS: CAS = tCAS - 1  (Equation 8)
        // WDL field is set from tCAS: WDL = tCAS - 2  (Equation 7)

    // IQ8134x only supports tCAS of 4 if ODT is enabled
    //  if ((i2c_bytes[SPD_TCAS] & SPD_DDRII_tCAS_LAT_3) && (ps_per_cycle == 5000))
        if (i2c_bytes[SPD_TCAS] & SPD_DDRII_tCAS_LAT_4)
            tCAS = 4;
        else if (i2c_bytes[SPD_TCAS] & SPD_DDRII_tCAS_LAT_5)
            tCAS = 5;
        else
            MEM_PARAM_ERROR(1, 4);
        sdcr0_val |= (SDCR_CAS_MCLK(tCAS) | SDCR_WDL_MCLK(tCAS));

        // Verify/Determine Memory Module Attributes
        // SPD byte 20, bit0 = Registered, bit1 = Unbuffered
        // However, the SDCR0 on 8134x must ALWAYS BE SET for Registered
        // and then the DLLRCVER is set to Unbuffered or Registered (bit 16).
        sdcr0_val |= SDCR_DIMM_TYPE_REGISTERED;
        if (i2c_bytes[SPD_DIMM_TYPE] & SPD_REGISTERED)
            tREG = 0;  // DLLRCVER uses 0 to mean registered
        else if (i2c_bytes[SPD_DIMM_TYPE] & SPD_UNBUFFERED)
            tREG = 1;  // and 1 to mean unbuffered
        else
             MEM_PARAM_ERROR(1, 5);

        // SPD_tRP_to_ps/SPD_tRCD_to_ps macro strips off bits 1:0 which are
        // <1nS increments and multiplies by 1000 to get psec because 533MHz
        // DDR2 has a 3.75ns cycle time and I don't want to mess w/ decimals
        // Verify/Determine tRP
        tRP = ps_to_cycles(SPD_tRP_to_ps(i2c_bytes[SPD_TRP]), ps_per_cycle);
```

```
if (tRP <= SDCR_tRP_MAX)
    sdcr0_val |= SDCR_tRP_MCLK(tRP);
else
    MEM_PARAM_ERROR(1, 6);

// Verify/Determine tRCD
tRCD = ps_to_cycles(SPD_tRCD_to_ps(i2c_bytes[SPD_TRCD]), ps_per_cycle);
if (tRCD <= SDCR_tRCD_MAX)
    sdcr0_val |= SDCR_tRCD_MCLK(tRCD);
else
    MEM_PARAM_ERROR(1, 7);

// Verify/Determine tRAS (tRAS reported in integer nS - no fractions)
tRAS = ps_to_cycles(SPD_tRAS_to_ps(i2c_bytes[SPD_TRAS]), ps_per_cycle);
if (tRAS <= SDCR_tRAS_MAX)
    sdcr0_val |= SDCR_tRAS_MCLK(tRAS);
else
    MEM_PARAM_ERROR(1, 8);

// Verify/Determine Bank sizes
// DDR-II - SPD supports bank sizes up to 16GB but iq8134x
// only support a max of 1024MB banks for DDR-II
if (i2c_bytes[SPD_BANKSZ] == SPD_DDRII_BANKSZ_1GB)
    bank_size_Mbytes = 1024;
else if (i2c_bytes[SPD_BANKSZ] == SPD_DDRII_BANKSZ_512MB)
    bank_size_Mbytes = 512;
else if (i2c_bytes[SPD_BANKSZ] == SPD_DDRII_BANKSZ_256MB)
    bank_size_Mbytes = 256;
else if (i2c_bytes[SPD_BANKSZ] == SPD_DDRII_BANKSZ_128MB)
    bank_size_Mbytes = 128;
else
    MEM_PARAM_ERROR(1, 9);

*sdram_size = (((bank_size_Mbytes * num_banks)
```

```
                - CYGSEM_HAL_ARM_IOP34X_32BIT_ECC_REGION_SIZE*2) << 20);


// Verify/Determine tRC
// DDR-II decoding based on JEDEC Spec dated 1/13/2003
t = (cyg_uint16)SPD_tRC_to_ns(i2c_bytes[SPD_TRC]);
subfield_c = (i2c_bytes[SPD_TRFC_TRC_EXT] & 0xF0) >> 4;
if (subfield_c == 0) {
    // No additional fractional ns part to tRC.
    ;
}
else if ((subfield_c >= 1) && (subfield_c <= 5)) {
    // There is an additional fractional ns part to tRFC.
    t += 1;
}
else {
    MEM_PARAM_ERROR(1, A);
}

// Get cycles based on psec value of tRC (t is in nS)
tRC = ps_to_cycles(t*1000, ps_per_cycle);
if (tRC <= SDCR_tRC_MAX)
    sdcr1_val |= SDCR_tRC_MCLK(tRC);
else
    MEM_PARAM_ERROR(1, B);

// Verify/Determine tWTR
// DDR-II -- read tWTR from the SPD
tWTR = ps_to_cycles(SPD_tWTR_to_ps(i2c_bytes[SPD_TWTR]), ps_per_cycle);

// Verify/Determine tRFC
t = (cyg_uint16)SPD_tRFC_to_ns(i2c_bytes[SPD_TRFC]);

// Change nS to pSec to help addition of subfield parameters
t *= 1000;
```


```
// DDR-II decoding based on JEDEC Spec dated 1/13/2003
subfield_a = ((i2c_bytes[SPD_TRFC_TRC_EXT] & 0x01) |
    (i2c_bytes[SPD_TRFC_TRC_EXT] & 0x80) >> 6);
subfield_b = (((i2c_bytes[SPD_TRFC_TRC_EXT] & 0x0E) >> 1) |
    (i2c_bytes[SPD_TRFC_TRC_EXT] & 0x80) >> 6);
if (subfield_a == 1) {
    t += 256*1000;
}

if (subfield_b == 0) {
    // No additional fractional ns part to tRC.
    ;
}
else if ((subfield_b >= 1) && (subfield_b <= 5)) {
    // There is an additional fractional ns part to tRFC.
    switch (subfield_b) {
        case 1:
            t += 250;  // Add .25nS
            break;
        case 2:
            t+= 330;  // Add .33nS
            break;
        case 3:
            t+= 500;  // Add .5nS
            break;
        case 4:
            t+= 660;  // Add .66nS
            break;
        case 5:
            t+= 750;  // Add .75nS
            break;
    }
}
```

```
else
    MEM_PARAM_ERROR(1, C);
// Convert temp var, t into psec and get cycles back
tRFC = ps_to_cycles(t, ps_per_cycle);
if (tRFC <= SDCR_tRFC_MAX)
    sdcr1_val |= SDCR_tRFC_MCLK(tRFC);
else
    MEM_PARAM_ERROR(1, D);

// Set ODT Field to 75 Ohms - DDR-II
sdcr0_val |= SDCR_ODT_TERMINATION_75_OHM;

// Normal Operation for FIFO
sdcr0_val |= SDCR_NORMAL_FIFO;

// Determine tWR from SPD (currently should always report 15nS)
tWR = ps_to_cycles(SPD_tWR_to_ps(i2c_bytes[SPD_TWR]), ps_per_cycle);
if (tWR <= SDCR_tWR_MAX)
    sdcr1_val |= SDCR_tWR_MCLK(tWR);

// DQS# Disable Field
// DDR-II
// Use differential operation for DDR-II
sdcr1_val &= ~SDCR_DQS_ENABLE_MASK;

// Determine WTRD - Equation 15 (tCAS - 1 + (BL/2) + tWTR)
// (BL == 4)
//  = tCAS + 1 +tWTR
sdcr1_val |= SDCR_tWTRD_MCLK(tCAS + tWTR + 1);

// Determine tEDP - Equation TBD - just use default from C-spec
tEDP = tEDP_FROM_DESIGN_8_INCH_TRACE;
sdcr0_val |= SDCR_tEDP_MCLK(tEDP);
```


```
// Determine tRTCMD - Equation 9
// From C-spec: Equation 9: (RTCMD = tRTP = X)
// Where X == 2 for 400&533 MHz DDR-II
// Grab tRTP from SPD, but this field should always be 2
tRTP = ps_to_cycles(SPD_tRTP_to_ps(i2c_bytes[SPD_TRTP]), ps_per_cycle);
if (tRTP <= SDCR_tRTP_MAX)
    sdcr1_val |= SDCR_tRTCMD_MCLK(tRTP);
else if (tRTP != 2) {
    MEM_PARAM_ERROR(1, E);
}
else
    MEM_PARAM_ERROR(1, F);

// Determine RTW  - Equation 11
// From C-spec: Equation 11: RTW = (BL/2) + 2
// (BL == 4)
sdcr1_val |= SDCR_tRTW_MCLK(4);

// Determine WTCMD - Equation 10
// From C-spec: Equation 10: WTCMD = tCAS - 1 + (BL/2) + tWR
// (BL == 4): WTCMD = tCAS + 1 + tWR
sdcr1_val |= SDCR_tWTCMD_MCLK(tCAS + 1 + tWR);

// Set up sbsr_val
// Assumption: All banks are the same size.
if (bank_size_Mbytes == 128)
    sbsr_val = SBR_128MEG;
else if (bank_size_Mbytes == 256)
    sbsr_val = SBR_256MEG;
else if (bank_size_Mbytes == 512)
    sbsr_val = SBR_512MEG;
else if (bank_size_Mbytes == 1024)
    sbsr_val = SBR_1GIG;
else
```

```
        MEM_PARAM_ERROR(2, 0);


    // Detemine RMW : Equation 16
    // From C-spec: RMW = tCAS + (BL/2) + tEDP
    // BL ==4; tEDP is TBD - using 1 for now (default)
    sbsr_val |= SDCR_tRMW_MCLK(tCAS + 2 + tEDP);


    if (num_banks == 1)
        sbsr_val |= SBSR_ONE_BANK;
    else if (num_banks == 2)
        sbsr_val |= SBSR_TWO_BANKS;


    // SDBR is the Physical Base
    sdbr_val = (SDRAM_PHYS_BASE);


    // Set the MCU MMRs
    HAL_WRITE_UINT32(MCU_SDBR, sdbr_val);
     HAL_WRITE_UINT32(MCU_SDUBR, 0);


    HAL_WRITE_UINT32(MCU_SDCR0, sdcr0_val);  // Write SDCR0
    HAL_READ_UINT32 (MCU_SDCR0, sdcr0_val);  // ReadBack to make sure it's written


    HAL_WRITE_UINT32(MCU_SDCR1, sdcr1_val);
     HAL_WRITE_UINT32(MCU_SBSR, sbsr_val);

HAL_WRITE_UINT32(MCU_S32SR,S32SR_MEG(CYGSEM_HAL_ARM_IOP34X_32BIT_ECC_REGION_SIZE))
;


    // Program DLL receive enable delay register with values from design
    // and the Unbuffered/Registered DIMM setting
    if (ps_per_cycle == 5000)
        HAL_WRITE_UINT32(MCU_DLLRCVER, (0x00011 | (tREG << 16)));
        // 8" trace/60 Ohms - 400MHz - From Joe 4/2/2005
    else
        HAL_WRITE_UINT32(MCU_DLLRCVER, (0x00017 | (tREG << 16)));
        // 8" trace/60 Ohms - 533MHz - scaled from 400
```


```
// Read-back certain MCU registers after writing, but before
// accessing memory per C-Spec
{
    volatile cyg_uint32 ign;
    HAL_READ_UINT32(MCU_SDCR0, ign);
    HAL_READ_UINT32(MCU_SDCR1, ign);
    HAL_READ_UINT32(MCU_SDBR, ign);
    HAL_READ_UINT32(MCU_SDUBR, ign);
    HAL_READ_UINT32(MCU_SBSR, ign);
    HAL_READ_UINT32(MCU_S32SR, ign);
    HAL_READ_UINT32(MCU_DLLRCVER, ign);
}

HEX_DISPLAY_QUICK(2, 1);

// Clear any pending MCU interrupts
HAL_WRITE_UINT32(MCU_DMCISR, 0x30F);

// SDRAM Initialization Sequence as specified by the JEDEC Spec
// Steps are numbered based on JESD79-2 (DDR2 Spec)
// STEPs 1-2 of JEDEC are done in Hardware up to this point
HAL_WRITE_UINT32(MCU_RFR, 0);     // Disable the refresh counter

// Call delay_mclks with either MClk delay desired or (delay time desired / 5nS
// Step 3 : Issue NOP cycle
delay_mclks(40000);               // 200us for clocks to stabilize
HAL_WRITE_UINT32(MCU_SDIR, SDIR_CMD_NOP);  // One NOP cycle
HAL_READ_UINT32(MCU_SDIR, temp);

// Step 4 : Issue Precharge all
delay_mclks(80);                           // Delay for 400nS minimum
HAL_WRITE_UINT32(MCU_SDIR, SDIR_CMD_PRECHARGE_ALL); // Precharge-All command
HAL_READ_UINT32(MCU_SDIR, temp);
```

```
    delay_mclks(tRP);                                   // Delay for tRP clocks


    // Step 5 : EMRS(2) programming
    HAL_WRITE_UINT32(MCU_SDIR, SDIR_CMD_EMRS2_PROGRAM_TO_0); // Program EMRS(2) to
0
    HAL_READ_UINT32(MCU_SDIR, temp);
    delay_mclks(10);                                    // Delay for tMRD (Min of 2)


    // Step 6 : EMRS(3) programming
    HAL_WRITE_UINT32(MCU_SDIR, SDIR_CMD_EMRS3_PROGRAM_TO_0); // Program EMRS(3) to
0
    HAL_READ_UINT32(MCU_SDIR, temp);
    delay_mclks(10);                                  // Delay for tMRD (Min of 2)


    // Step 7 : EMRS to enable DLL - make sure RTT field is written as well
    //          DQS enabled, RDQS disabled, Additive latency = 0, OutBuff enabled
    switch (sdcr0_val & SDCR_ODT_TERMINATION_MASK) {
        case SDCR_ODT_TERMINATION_DISABLED:
            emrs = SDIR_CMD_EMRS_DLL_ENABLED_NO_RTT;
            break;
        case SDCR_ODT_TERMINATION_75_OHM:
            emrs = SDIR_CMD_EMRS_DLL_ENABLED_75_OHM_RTT;
            break;
        case SDCR_ODT_TERMINATION_150_OHM:
            emrs = SDIR_CMD_EMRS_DLL_ENABLED_150_OHM_RTT;
            break;
    }
    HAL_WRITE_UINT32(MCU_SDIR, emrs);                 // EMRS to enable the DLL
    HAL_READ_UINT32(MCU_SDIR, temp);
    delay_mclks(10);                                  // Delay for tMRD (Min of 2)


    // Step 8 : MRS to reset DLL -> Setting operating parameters too
    if (tCAS == 3) {
        if (tWR == 3)
            mrs = SDIR_CMD_MRS_DLL_IS_RESET_CAS3_WR3;
```

```
        else
            mrs = SDIR_CMD_MRS_DLL_IS_RESET_CAS3_WR4;
    }
    else if (tCAS == 4) {
        if (tWR == 3)
            mrs = SDIR_CMD_MRS_DLL_IS_RESET_CAS4_WR3;
        else
            mrs = SDIR_CMD_MRS_DLL_IS_RESET_CAS4_WR4;
    }
    else if (tCAS == 5) {
        if (tWR == 3)
            mrs = SDIR_CMD_MRS_DLL_IS_RESET_CAS5_WR3;
        else
            mrs = SDIR_CMD_MRS_DLL_IS_RESET_CAS5_WR4;
    }
    HAL_WRITE_UINT32(MCU_SDIR, mrs);// MRS to program the SDRAM parameters and DLL
reset
    HAL_READ_UINT32(MCU_SDIR, temp);
    delay_mclks(10);                // Delay for tMRD (Min of 2)

    // Step 9 : Precharge All Command
    HAL_WRITE_UINT32(MCU_SDIR, SDIR_CMD_PRECHARGE_ALL);// Precharge-All command
    HAL_READ_UINT32(MCU_SDIR, temp);
    delay_mclks(SDCR_tRP_MAX + 100);      // Delay for Trp - no max spec'd

    // Step 10 : Minimum of two auto refresh cycles
    HAL_WRITE_UINT32(MCU_SDIR, SDIR_CMD_AUTO_REFRESH);  // 1st of 2 required Auto
Refresh Cycles
    HAL_READ_UINT32(MCU_SDIR, temp);
    delay_mclks(100);          // Delay at least tRFC cycles
    HAL_WRITE_UINT32(MCU_SDIR, SDIR_CMD_AUTO_REFRESH);// 2nd of 2 required Auto
Refresh Cycles
    HAL_READ_UINT32(MCU_SDIR, temp);
    delay_mclks(100);               // Delay at least tRFC cycles
```

```
    // Step 11 : MRS with DLL Not reset and operating parameters
    mrs &= ~(MRS_DLL_RESET);// Clear DLL Reset Bit from Mode Register
    HAL_WRITE_UINT32(MCU_SDIR, mrs); // MRS to program the SDRAM parameters and no
DLL reset
    HAL_READ_UINT32(MCU_SDIR, temp);
    delay_mclks(300);                  // Delay for Tmrd cycles

    // Must be 200 clocks after step 8 (DLL Reset)
    // Step 12 : OCD Calibration - Set Default and then Exit command
    delay_mclks(300);
    emrs |= SDIR_CMD_EMRS_OCD_DEFAULT;
    HAL_WRITE_UINT32(MCU_SDIR, emrs);    // EMRS OCD Default Command
    HAL_READ_UINT32(MCU_SDIR, temp);

    delay_mclks(300);
    emrs &= ~(SDIR_CMD_EMRS_OCD_DEFAULT);
    HAL_WRITE_UINT32(MCU_SDIR, emrs);       // EMRS OCD Exit Command
    HAL_READ_UINT32(MCU_SDIR, temp);
    delay_mclks(300);

    // Step 13 : DDR2 is now ready for normal operation
    HAL_WRITE_UINT32(MCU_RFR, refresh);   // Re-enable the refresh counter.

    HEX_DISPLAY_QUICK(2, 2);

    *sdram_installed = true;
    return;
}
```

Appendix C Modifying PT Entries to Slide Outbound Window

```
#define _Read_Translation_Table_Base \
({ unsigned _val_; \
   asm volatile("mrc\tp15, 0, %0, c2, c0, 0" : "=r" (_val_)); \
 val_; \
})


// Modify_omw_pt : Modifies the Outbound Memory Page tables so the
//                 lower 32-bits of the outbound address line up with
//                 where we are allocating PCI memory addresses.


void modify_omw_pt(cyg_uint64 hal_pci_alloc_base_memory, cyg_uint32
outbound_mem_window) {
    cyg_uint32 ttb_base_va, ttb_base_pa, omw_offset, counter, count_max, new_entry;
    cyg_uint32 *omw_pt_entry;
    ttb_base_pa = (_Read_Translation_Table_Base & 0xFFFFC000);
    ttb_base_va = cygarc_virtual_address (ttb_base_pa);
    omw_offset = (outbound_mem_window >> 18);
        // Outbound mem base and 4bytes per entry
    omw_pt_entry = (cyg_uint32*)(ttb_base_va + omw_offset);
        // Pointer to the PT entries81348


#ifdef DEBUG_ATU
    diag_printf ("OMW is: 0x%08x \n", outbound_mem_window);
    diag_printf ("TTB_Base_PA is: 0x%08x\n", ttb_base_pa);
    diag_printf ("TTB_Base_VA is: 0x%08x\n", ttb_base_va);
    diag_printf ("omw_offset is: 0x%08x and 1st omw_pt_entry is: 0x%08x\n", \
        omw_offset, *omw_pt_entry);
#endif


    // Calculate # of PT entries to modify
    count_max = (PCI_OUTBOUND_MEM_WINDOW_0_TOP - PCI_OUTBOUND_MEM_WINDOW_0) >> 20;
    new_entry = (hal_pci_alloc_base_memory & SUPERSECTION_BASE_ADDRESS);
    for (counter = 1; counter <= count_max; counter++ ) {
        *omw_pt_entry &= ~(SUPERSECTION_BASE_ADDRESS);
        *omw_pt_entry |= new_entry;
#ifdef DEBUG_ATU
        diag_printf ("Modifed PT entry being written: 0x%08x\n", *omw_pt_entry);
#endif
        omw_pt_entry++;
        if ((counter % 16) == 0) {  // Every Super Entry takes 16 descriptors
            new_entry += 0x1000000;
        }
    }
    HAL_DCACHE_SYNC();  // Sync cache w/ Memory
    HAL_DCACHE_INVALIDATE_ALL();  // Also clears TLBs
    HAL_L2_CACHE_SYNC();

}
```