

# $\mu$ ARM Informal Specifications

Marco Melletti - [melletti.marco@gmail.com](mailto:melletti.marco@gmail.com)  
<http://mellotanica.github.io/uARM/>

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Processor</b>	<b>5</b>
2.1	Operating modes and Processor registers . . . . .	5
2.2	System Coprocessor . . . . .	8
2.3	Execution Control . . . . .	10
2.3.1	Program Status Register . . . . .	10
2.3.2	System Control Register . . . . .	11
2.4	Processor States . . . . .	11
2.4.1	ARM ISA . . . . .	11
2.4.2	Thumb ISA . . . . .	12
2.5	Exception Handling . . . . .	13
2.5.1	Hardware Level Exception Handling . . . . .	13
2.5.2	ROM Level Exception Handling . . . . .	15
<b>3</b>	<b>System Bus</b>	<b>17</b>
3.1	Reserved address space . . . . .	17
3.1.1	Exception Vector . . . . .	17
3.1.2	System Information Registers . . . . .	18
3.1.3	Bootstrap ROM . . . . .	18
3.1.4	Kernel Reserved Frame . . . . .	18
3.2	Memory address space . . . . .	20
<b>4</b>	<b>Memory Interface</b>	<b>21</b>
4.1	Physical addressing mode . . . . .	21
4.2	Virtual addressing mode . . . . .	22
<b>5</b>	<b>External Devices</b>	<b>26</b>
5.0.1	Installed Device Table . . . . .	26
5.0.2	Device Registers . . . . .	26
5.0.3	Pending Interrupt Bitmap . . . . .	26

<b>6</b>	<b>BIOS &amp; System Library</b>	<b>28</b>
6.1	BIOS . . . . .	28
6.1.1	Bootstrap Function . . . . .	28
6.1.2	Low Level Handlers . . . . .	28
6.1.3	Low Level Services . . . . .	29
6.2	System Library . . . . .	30
<b>7</b>	<b>Graphical User Interface</b>	<b>33</b>
<b>8</b>	<b>Notes</b>	<b>34</b>
8.1	Compilers and compiling . . . . .	34
8.1.1	Compilers . . . . .	34
8.1.2	Compiling . . . . .	34
8.2	Binary Formats . . . . .	35
8.3	Hints . . . . .	36
8.4	Known bugs . . . . .	36

# Chapter 1

## Introduction

$\mu$ ARM is an emulator program that implements a complete system with an emulated version of the ARM7TDMI processor as its core component. The processor's specifications have been respected, so instruction set, exception handling and processor structure are the same as any real processor of the same family. Since ARM7TDMI architecture does not detail a device interface and the memory management scheme is a bit too complex, these two components are based on  $\mu$ MPS2 architecture, which in turn takes inspiration from commonly known architectures.

In a more schematic fashion,  $\mu$ ARM is composed of:

- An ARM7TDMI processor.
- A system coprocessor, *CP15*, incorporated into the processor.
- Bootstrap and execution ROM.
- RAM memory little-endian subsystem with optional virtual address translation mechanisms based on Translation Lookaside Buffer.
- Peripheral devices: up to eight instances for each of five device classes. The five device classes are disks, tape devices, printers, terminals, and network interface devices.
- A system bus connecting all the system components.

This document will describe the main aspects of the emulated system, taking into exam each of its components and describing their interactions, further details regarding the processor can be found in the *ARM7TDMI Datasheet* and the *ARM7TDMI Technical Reference Manual*.

Notational conventions:

- Registers and storage units are **bold**-marked.

- Fields are *italicized*.
- Instructions are CAPITALIZED.
- Field  $F$  of register  $\mathbf{R}$  is denoted  $\mathbf{R}.F$ .
- Bits of storage units are numbered right-to-left, starting with 0.
- The  $i$ -th bit of a storage unit named  $N$  is denoted  $N[i]$ .
- Memory addresses and operation codes are given in hexadecimal and displayed in big-endian format.
- All diagrams illustrating memory are going from low addresses to high addresses using a left to right, bottom to top orientation.

# Chapter 2

## Processor

The  $\mu$ ARM machine runs on an emulated ARM7TDMI processor, which implements both ARM and Thumb instruction sets, is able to perform each operation listed in *ARM7TDMI Data Sheet* (a brief summary is shown below) and to accept painlessly binary programs compiled with the Gnu C Compiler for ARM7 architecture.

### 2.1 Operating modes and Processor registers

The processor can work in seven different modes:

- User mode (usr) - regular user process execution
- System mode (sys) - typical privileged mode execution (e.g. kernel code execution)
- Supervisor (svc) - protected mode kernel execution
- Fast Interrupt (fiq) - protected mode for fast interrupt handling
- Interrupt (irq) - protected mode for regular interrupt handling
- Abort (abt) - protected mode for data/instruction abort exception handling
- Undefined (und) - protected mode for undefined instruction exception handling

In each mode the processor can access a limited portion of all its registers, varying from 16 registers in User/System modes to 17 registers in each protected mode in ARM state, plus the Current Program Status Register (**CPSR**), which is shared by all modes. Only protected modes have the 17th register, which automatically stores the previous value of the **CPSR** when raising an exception.

The first 8 registers, in addition to the Program Counter (**R15**) are common to each "window" of visible registers, each protected mode has its dedicated Stack Pointer

and Link Return registers and Fast Interrupt mode has all the last 7 registers uniquely banked, to allow for a fast context switch, while System and User mode share the full set of 16 general purpose registers.

<i>ARM State General Registers</i>					
<i>User / System</i>	<i>FIQ</i>	<i>Svc</i>	<i>Abort</i>	<i>IRQ</i>	<i>Undef</i>
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (pc)	R15 (pc)	R15 (pc)	R15 (pc)	R15 (pc)	R15 (pc)
CPSR	CPSR SPSR_fiq	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_irq	CPSR SPSR_und
<div></div> = banked register					

Even if the global 16 registers are defined as general purpose registers, there are some conventions adopted by the compiler in their use. The following list shows the full set of processor register visible in each mode with their conventional abbreviations and extended meaning:

- **R0 (a1)** - first function argument / integer result
- **R1 (a2)** - second function argument
- **R2 (a3)** - third function argument
- **R3 (a4)** - fourth function argument

- **R4 (v1)** - register variable
- **R5 (v2)** - register variable
- **R6 (v3)** - register variable
- **R7 (v4)** - register variable
- **R8 (v5)** - register variable
- **R9 (v6/rfp)** - register variable / real frame pointer
- **R10 (sl)** - stack limit
- **R11 (fp)** - frame pointer / argument pointer
- **R12 (ip)** - instruction pointer / temporary workspace
- **R13 (sp)** - stack pointer
- **R14 (lr)** - link register
- **R15 (pc)** - program counter
- **CPSR** - current program status register
- **SPSR<sub>mode</sub>** - saved program status register

When the processor is in Thumb state the register window is reduced, showing 12 registers in User/System mode and 13 registers in protected modes, in addition to the Current Program Status Register, which is common to all modes.

Only the first 8 registers (**R0** → **R7**) are general purpose, the higher 3 are specialized registers that act as Stack Pointer, Link Return and Program Counter. Each protected mode has its own banked instance of Stack Pointer and Link Return in addition to Saved Program Status Register to allow for faster exception handling.



<i>Thumb State General Registers</i>					
<i>User / System</i>	<i>FIQ</i>	<i>Svc</i>	<i>Abort</i>	<i>IRQ</i>	<i>Undef</i>
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
SP	SP_fiq	SP_svc	SP_abt	SP_irq	SP_und
LR	LR_fiq	LR_svc	LR_abt	LR_irq	LR_und
R15	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und
<div></div> = banked register					

## 2.2 System Coprocessor

CP15 gives access to a total of three 64-bit and one 32-bit registers which provide additional information and functionalities to regular processor operations:

### Register 0 (IDC) - ID Codes

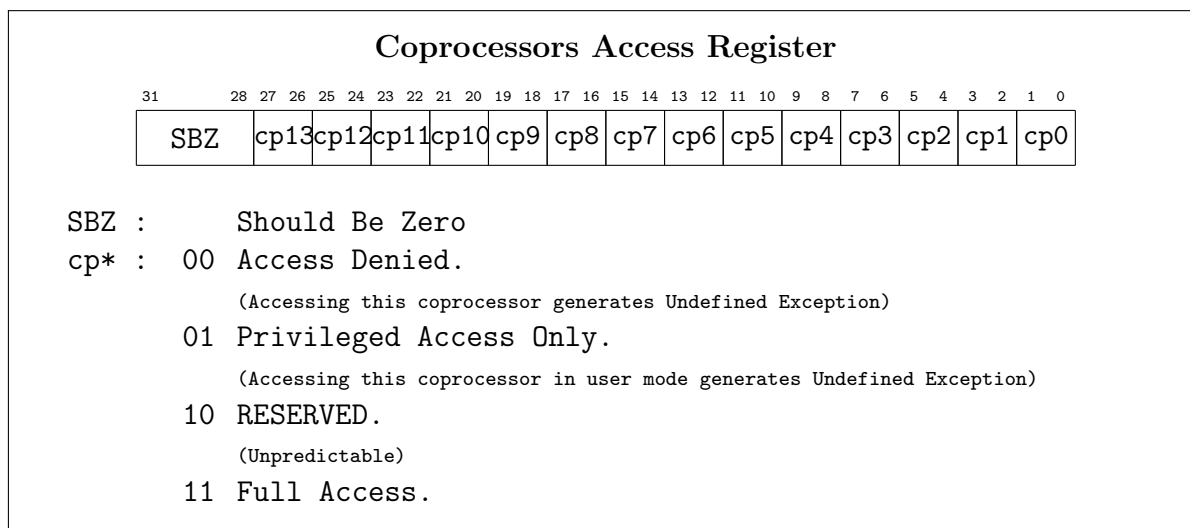
**R0** is a read-only 64-bit register that contains system implementation information such as *Processor ID* and *TLB type*, as required by ARM specifications.

### Register 1 (SCB) - System Control Bits

**R1.SCB** is the System Control Register, this register holds system-wide settings flags. See sec. 2.3.2 for further details.

## Register 1 (CCB) - Coprocessors Access Register

**R1.CCB** shows which coprocessors are available. Values can be written to this register to enable/disable available coprocessors a part from CP15.



## Register 2 - Page Table Entry

**R2** is a 64-bit register which contains the active Page Table Entry when MMU is enabled. Its structure is the same as the Page Table Entry described in sec. 4.2.

## Register 15 (*Cause*) - Exception Cause

**R15.Cause** contains the last raised exception cause, it can be read or written by the processor.

A scheme of Memory Access exception cause codes is shown below:

Memory Error	= 1
Bus Error	= 2
Address Error	= 3
Segment Error	= 4
Page Error	= 5

## Register 15 (*IPC*) - Interrupt Cause

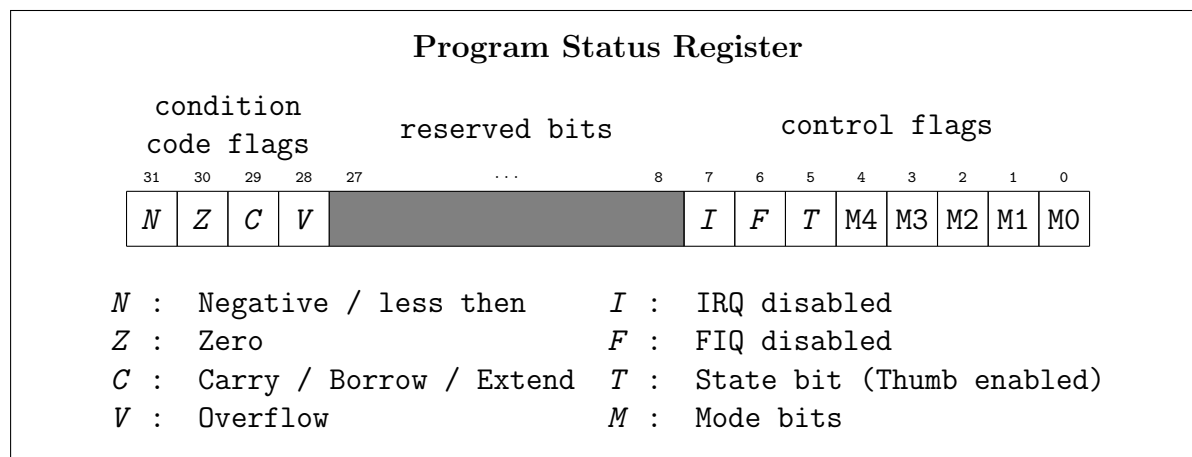
**R15.IPC** shows on which lines interrupts are pending, when interrupts have been handled (i.e. the interrupt request has been acknowledged, see sec. 5) the value of this register is updated.

## 2.3 Execution Control

The processor behavior can be set up by modifying the contents of two special registers: the Current Program Status Register (**CPSR**) and the System Control Register (**CP15.SCB**). Each of the two has a special structure and changes the way the system operates.

### 2.3.1 Program Status Register

The **CPSR** (as well as the **SPSR**, if the active mode has one) is always accessible in ARM state via the special instructions MSR (move register to PRS) and MRS (move PRS to register). This register shows arithmetical instructions' additional results (condition code flags), allows to toggle interrupts and switch states/modes. Its structure is shown below:



The first 5 bits of **CPSR** are used to set processor execution mode, the possible values are:

0x10	User Mode
0x11	Fast Interrupt Mode
0x12	Interrupt Mode
0x13	Supervisor Mode
0x17	Abort Mode
0x1B	Undefined Mode
0x1F	System Mode

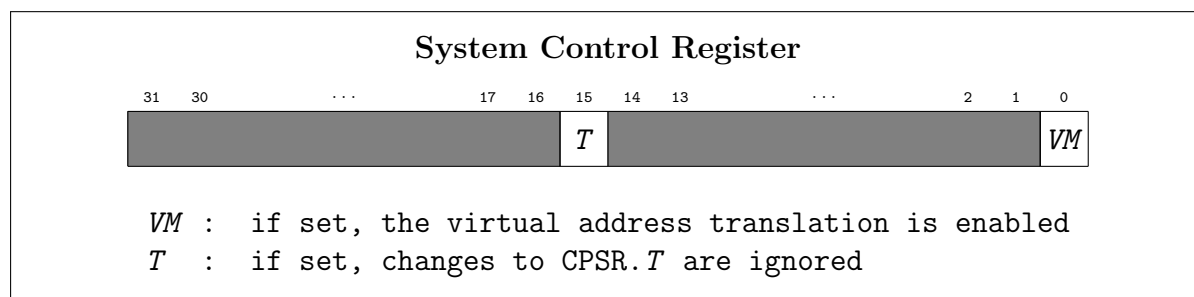
User Mode is the only unprivileged mode, this means that if the processor is running in User Mode, it cannot access reserved memory regions (see System Bus chapter) and it cannot modify CPSR control bits.

System Mode is the execution mode reserved for regular Kernel code execution, all

other modes are automatically activated when exceptions are raised (see sec. 2.5).

## 2.3.2 System Control Register

System Coprocessor (CP15) holds the System control register (**CP15.R1**), which controls Virtual Memory and Thumb availability (plus some other hardware specific settings that are not implemented in current release):



## 2.4 Processor States

The *T* flag of the Program Status Register shows the state of the processor, when the bit is clear the processor operates in ARM state, otherwise it works in Thumb state. To switch between the two states a Branch and Exchange (BX) instruction is required.

The first difference between the two states is the register set that is accessible (see sec. 2.1), the other main difference is the Instruction Set the processor is able to decode.

### 2.4.1 ARM ISA

The main Instruction Set is the ARM ISA, the processor starts execution in this state and is forced to ARM state when an exception is raised.

ARM instructions are 32 bits long and must be word-aligned. The table below shows a brief summary of the instruction set. (For a much detailed description of each instruction refer to *ARM7TMI Data Sheet* and *ARM7TMI Technical Reference Manual*)

ADC	add with carry	ADD	add
AND	logical AND	B	branch
BIC	bit clear	BL	branch with link
BX	branch and exchange	CDP	coprocessor data processing
CMN	compare negative	CMP	compare
EOR	logical exclusive OR	LDC	load coprocessor register from memory
LDM	load multiple registers from memory	LDR	load register from memory
LDRH	load halfword from memory	LDRSB	load signed byte from memory
LDRSH	load signed halfword from memory	MCR	move cpu register to coprocessor register
MLA	multiply accumulative	MLAL	multiply accumulative long
MOV	move register or constant	MRC	move coprocessor register to cpu register
MRS	move PRS status/flags to register	MSR	move register to PRS status/flags
MUL	multiply	MULL	multiply long
MVN	move negative register	ORR	logical OR
RSB	reverse subtract	RSC	reverse subtract with carry
SBC	subtract with carry	STC	store coprocessor register to memory
STM	store multiple	STR	store register to memory
STRH	store halfword	SUB	subtract
SWI	software interrupt	SWP	swap register with memory
TEQ	test bitwiser equality	TST	test bits
UND	undefined instruction		

## 2.4.2 Thumb ISA

Thumb instruction set is a simpler (smaller) instruction set composed of 16-bit, halfword aligned instructions, which offer less refined functionalities but less memory usage.

Thumb instructions can be seen as "shortcuts" to execute ARM code, as the performed operations are the same but this ISA offers less options for each instruction.

The following table summarizes Thumb instructions. (For a much detailed description of each instruction refer to *ARM7TMI Data Sheet* and *ARM7TMI Technical Reference Manual*)

ADC	add with carry	ADD	add
AND	logical AND	ASR	arithmetical shift right
B	unconditonal branch	B[cond]	conditioned branch
BIC	bit clear	BL	branch with link
BX	branch and exchange	CMN	compare negative
CMP	compare	EOR	logical exclusive OR
LDMIA	load multiple (increment after)	LDR	load word to register
LDRB	load byte to register	LDRH	load halfword to register
LDRSB	load signed byte to register	LDRSH	load signed halfword to register
LSL	logical shift left	LSR	logical shift right
MOV	move from register to register	MUL	multiply
MVN	move negative register	NEG	negate word
ORR	logical OR	POP	pop from stack
PUSH	push to stack	ROR	rotate right
SBC	subtract with carry	STMIA	store multiple (increment after)
STR	store register to memory	STRB	store byte to memory
STRH	store halfword to memory	SUB	subtract
SWI	software interrupt	TST	test bits

## 2.5 Exception Handling

When an exception is raised (e.g. a read instruction is performed on a forbidden bus address), the processor automatically begins a special routine to solve the problem. In addition to low level automatic exception handling facilities, the BIOS code implements a wrapper to simplify OS level handlers setup and functioning.

### 2.5.1 Hardware Level Exception Handling

There are seven different exceptions handled by the processor, each of those has a specific bus address to which the execution jumps on exception raising (see sec. 3.1.1).

When an exception is raised, the processor state is forced to ARM mode, the execution mode and the interrupt flags are set accordingly to the exception type, the **LR** register is filled with a return address and the **PC** is loaded with the correct address of the bus area Exception Vector. If the Exception Vector has been correctly initialized, the instruction pointed to by the **PC** is a Branch instruction leading to some handler code.

Follows a brief description of the exceptions and the different return addresses.

## **Reset Exception**

This exception is automatically raised each time the machine is started.

This exception is handled in Supervisor mode with all interrupts disabled, Link Return and SPSR registers have unpredictable values and execution starts from bus address 0x00000000.

The first bus world, when the execution starts, is always filled with a fixed Branch instruction that makes the processor jump to the beginning of the bus-mapped ROM (address 0x00000300), where the BIOS is stored.

## **Undefined Instruction Exception**

If a Coprocessor instruction cannot be executed from any Coprocessor or if an UNDEFINED instruction is executed, this exception is raised.

Processor mode is set to Undefined, normal interrupts are disabled and Link Return register points to the instruction right after the one that caused the Undefined Exception.

## **Software Interrupt Exception**

This exception is caused by a SWI instruction and is meant to provide a neat way to implement System Calls.

When handling Software Interrupt Exceptions, the processor switches to Supervisor mode with normal interrupts disabled and the Link Return register points to the instruction after the SWI that caused the exception.

## **Data Abort Exception**

If the processor tries to access a memory address that is not valid or available, this exception is raised.

When handling Data Aborts, the processor switches to Abort mode with normal interrupts disabled and Link Return register is set to the address of the instruction that caused the Abort plus 8.

## **Prefetch Abort Exception**

If the processor tries to execute an instruction that generated a data abort while being fetched, this exception is raised.

When handling Prefetch Aborts, the processor enters Abort mode with normal interrupts disabled and Link Return register points to the address of the instruction after the one that caused the exception.

## **Interrupt Request Exception**

When a connected device requires the processor's attention, it fires an Interrupt Request.

When handling Interrupt Requests, the processor enters Interrupt mode with normal interrupts disabled and Link Return register is set to the address of the instruction that was not executed plus 4.

## **Fast Interrupt Request Exception**

Fast Interrupts have higher priority than normal Interrupts, system Interval Timer is connected to this line of interrupts.

The Interval Timer's value is decreased at each execution cycle, when an underflow occurs (i.e. its value changes from 0x00000000 to 0xFFFFFFFF), a Fast Interrupt is requested.

When handling Fast Interrupt Requests, the processor enters Fast Interrupt mode with all interrupts disabled and Link Return register points to the address of the instruction that was not executed plus 4.

## **2.5.2 ROM Level Exception Handling**

During the bootstrap process, six of the seven Exception Vector registers must be initialized (the reset exception only occurs at system startup and the relative register has a fixed value). The BIOS code fills these registers with jump instruction opcodes pointing to its internal handler procedures.

The BIOS exception handlers provide a safe and automatic way to enter kernel level handlers by storing the processor state as it was before the exception was raised and loading the kernel handler's processor state from a known memory location inside the Kernel Reserved Frame (see sec. 3.1.4).

In addition to this general behavior, some handlers provide other functionalities as described below.

### **Undefined Instruction Handler**

An Undefined Instruction Exception needs no special treatment, its handler stores the old processor state into the PGMT Old Area and loads the processor state stored into the PGMT New Area.

### **Software Interrupt Handler**

Software interruptions recognized by the BIOS handler can be of two types: System Calls or Breakpoints, the foremost being interpreted as a request to the kernel, while the latter can also be a BIOS service request.



This handler is capable of recognizing BIOS service requests and serving them directly, if a System Call or an unrecognized Breakpoint is requested, the exception is handled with the default behavior, the old processor state is stored into the Syscall Old Area and the processor state stored into the Syscall New Area is loaded.

### **Data Abort and Prefetch Abort Handler**

If Virtual Memory is enabled (see sec. 4.2), both Data and Prefetch Aborts can be raised while accessing a memory frame whose VPN is not loaded into the TLB, event signaled by the memory subsystem through these two exceptions. If this is the case, the BIOS will automatically perform a TLB-Refill cycle, searching the active Page Tables for the required entry; otherwise the exception is treated as a generic exception, storing the old processor state into TLB Old Area and then loading the processor state stored into the TLB New Area.

Since the two different exception types have different return address offsets (see sec. 2.5.1), this handler modifies the return address stored within the old processor state in order to be the correct address to jump to. This behavior is necessary, because discerning the exception type from kernel level handler could be quite difficult, or even impossible at times, without the knowledge of the hardware level exception, that is given from processor mode and **CPSR.R15.Cause** and could be lost during the pass-up.

### **Interrupt and Fast Interrupt Handlers**

Both these exceptions are treated as generic exceptions and the BIOS handlers will adopt the default behavior, storing the old processor state into the Interrupt Old Area and then loading the processor state stored into the Interrupt New Area.

# Chapter 3

## System Bus

The system bus connects each component of the system and lets processing units access physical memory and devices.

The CPU and the System Coprocessor (CP15) can directly access the bus reading or writing values from or to specific addresses.

The lower addresses (below 0x8000) are reserved for special uses and are accessible under certain conditions.

### 3.1 Reserved address space

The address region between 0x0 and 0x8000 holds the fast exception vector, device registers, system information, bootstrap ROM and the kernel reserved frame. Any access to this memory area in User mode are (should be) prohibited and treated by the system bus as errors.

#### 3.1.1 Exception Vector

The first locations (0x0 → 0x1C) are occupied by exception vector, the processor jumps automatically to these addresses if an exception is raised and the bootstrap ROM typically writes a set of branch instructions to exception handlers in these fields.

Exception vector is organized as follows:

0x0	Reset
0x4	Undefined Instruction
0x8	Software Interrupt
0xC	Prefetch Abort
0x10	Data Abort
0x14	<i>reserved</i>
0x18	Interrupt Request
0x1C	Fast Interrupt Request

### 3.1.2 System Information Registers

Six registers, from address 0x2D0 to 0x2E8, show system specific information:

0x2D0	ram base address
0x2D4	ram top address
0x2D8	device registers base address
0x2DC	time of day (Hi)
0x2E0	time of day (Low)
0x2E4	interval timer
0x2E8	timer scale (fixed to 1 MHz)

#### Interval Timer

Interval timer is decremented at each CPU cycle, when its value becomes 0 a software interrupt is thrown. It can be set to a desired value by writing its address in any privileged mode.

### 3.1.3 Bootstrap ROM

The bootstrap ROM is loaded starting from address 0x300, its maximum size is 109 KB. See BIOS chapter for details.

### 3.1.4 Kernel Reserved Frame

The last memory frame (0x7000 → 0x7FFC) is reserved for kernel use:

- 0x7000 → 0x7500: Exception states vector - memory area in which processor states are saved and loaded when entering into/exiting from exception handlers code.
- 0x7600 → 0x7C00: Segment table - here is stored the 128 elements segment table describing the virtual address space, for each ASID (entries in the segment table) there are two pointers to ASID's private and shared page tables.

- 0x7FF0 → 0x7FFC: Rom stack - when invoking ROM functions this stack is used to pass parameters to the hardware routines.

## Stored Processor States

Processor states are defined by library data structure *state\_t*, this structure is composed of 20 unsigned 32-bit integers representing processor registers' values and coprocessor's system control registers' values.

Its structure is shown below:

```
typedef struct{
    unsigned int a1;    //r0
    unsigned int a2;    //r1
    unsigned int a3;    //r2
    unsigned int a4;    //r3
    unsigned int v1;    //r4
    unsigned int v2;    //r5
    unsigned int v3;    //r6
    unsigned int v4;    //r7
    unsigned int v5;    //r8
    unsigned int v6;    //r9
    unsigned int s1;    //r10
    unsigned int fp;    //r11
    unsigned int ip;    //r12
    unsigned int sp;    //r13
    unsigned int lr;    //r14
    unsigned int pc;    //r15
    unsigned int cpsr;
    unsigned int CP15_Control;
    unsigned int CP15_EntryHi;
    unsigned int CP15_Cause;
}state_t;
```

These structures take 80 bytes each. Given this value, the BIOS code will look for the Old/New entries at the following addresses:

0x7000	Interrupt Old
0x7080	Interrupt New
0x7100	TLB Old
0x7180	TLB New
0x7200	PGMT Old
0x7280	PGMT New
0x7300	Syscall Old
0x7380	Syscall New

### ASID 0

The first ASID is reserved for kernel address space and is automatically enabled in any Privileged mode, this way if a User mode program performs a System Call, the kernel routine has access to its address space and to the program's address space through saved processor state.

## 3.2 Memory address space

The remaining addresses are mapped to memory subsystem and can be accessed through a number of instructions, see Memory Interface chapter for details.

# Chapter 4

## Memory Interface

Memory system is controlled by Program Status Register (CPSR) and System Coprocessor's registers 1 and 2 (CP15.R1 & CPSR.R2). It supports two operating modes:

- physical addressing mode,
- virtual addressing mode.

In addition to address translation modes, the portion of accessible memory is dictated by processor operating mode:

- User mode → User Space
- Privileged mode → All Memory

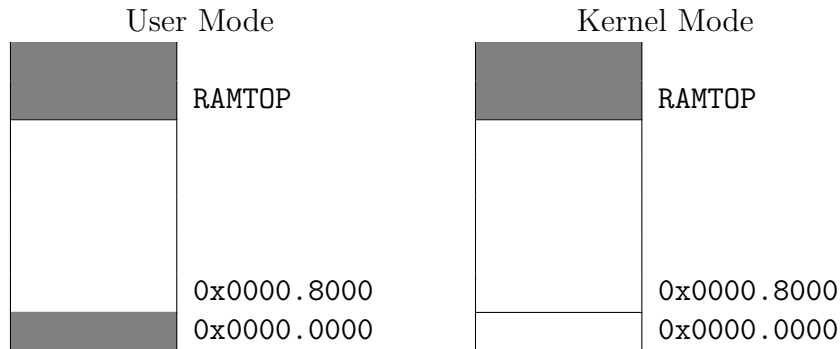
In each addressing mode these portions have a specific definition.

As described in System Bus chapter, addresses below 0x00008000 are reserved for hardware/protected functions and belong to the *reserved address space*.

### 4.1 Physical addressing mode

The machine starts execution in this mode, each address is used directly without conversions.

All the available memory is directly accessible in Privileged mode and any address over 0x00008000 is directly accessible in User mode.

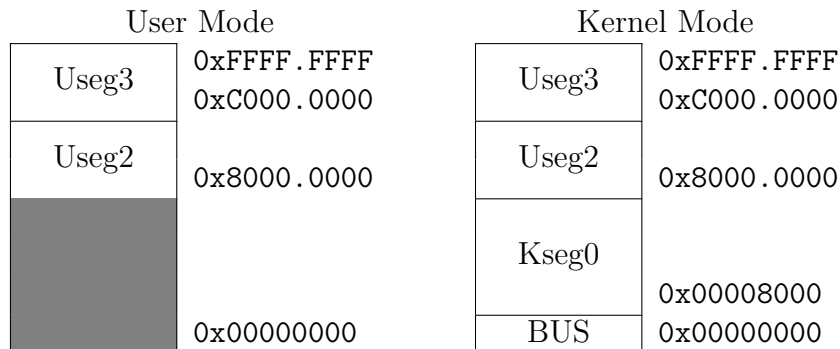


## 4.2 Virtual addressing mode

By setting M flag in System Coprocessor's register 1 (CP15.R1.M, that is least significant bit), you enable memory address translation.

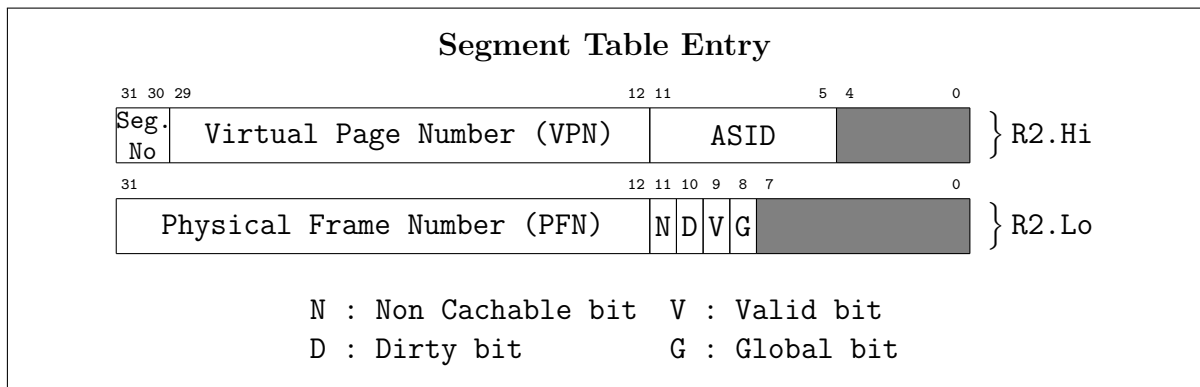
When virtual memory is active, each address above 0x0000.8000 is treated as a logical address and translated to the corresponding physical address from the memory subsystem, addresses below 0x0000.8000 are always treated as physical addresses, as they refer to a memory region reserved for bus access.

In Privileged mode all logical memory is accessible, when exacting in User mode, only User Segments are accessible instead: the first one (Useg2) starts from address 0x8000.0000 and extends over the next GB, the second one (Useg3) starts from address 0xC000.0000 and terminates at the top of the logical memory, address 0xFFFF.FFFF.



When the MMU is enabled the user process ASID is stored in the EntryHy field of CP15's 64bit register R2 along with the Virtual Page number (e.g. the 20 most significant bits of the logical address). The EntryLow field is filled with the Physical Page Frame address and is kept up to date after each modification of CP15.R2.EntryHy value.

The CP15 register 2 is organized as a Page Table Entry (PTE):

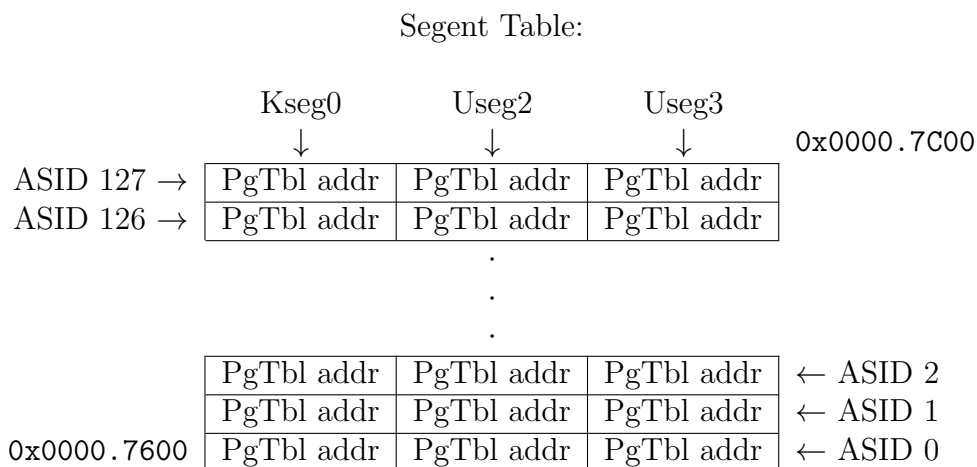


The Low half of each entry contains 4 flags used for memory protection:

- **Non Chachable bit:** not used by  $\mu$ ARM implementation.
- **Dirty bit:** if the bit is clear, any write access to the physical frame locations will rise a TLB-Modification exception.
- **Valid bit:** if bit is set the Page Table Entry is considered valid, otherwise a TLB-Invalid exception is raised.
- **Global bit:** if the bit is set, the Page Table Entry will match the corresponding VPN regardless of the ASID.

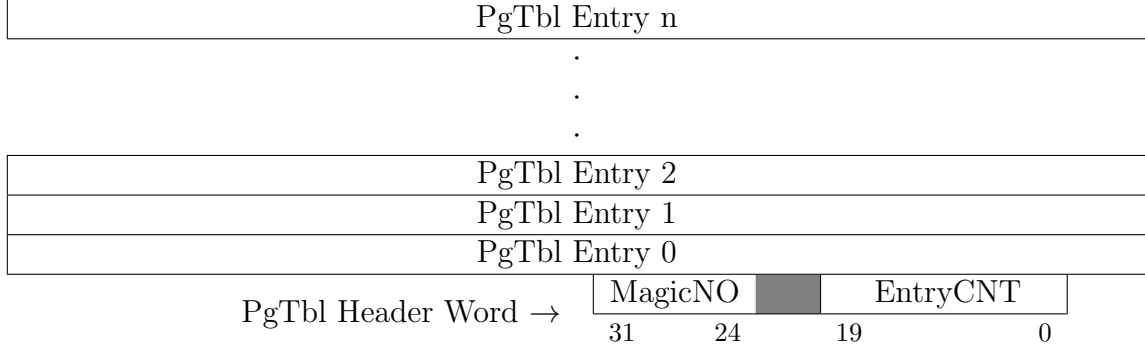
Page Table Entries are grouped together in Page Tables, each Page Table begins with a special word (PgTbl-Header) composed of the PgTbl Magic Number **0x2A** stored in the most significant 8 bits and the number of page table entries in the least significant 20 bits.

The Segment Table specifies the physical addresses of the Page Tables of the three Segments for each ASID, the general structure is shown below:





Page Table:



$\mu$ ARM implements a Translation Lookaside Buffer (TLB) to translate logical addresses to physical ones, the buffer contains a specific amount of recently used PTEs and uses a random algorithm to select which entry to replace with a newly retrieved PTE. The number of available TLB slots is variable between 4 and 64 elements, it is configurable through the settings window and needs a reset of the machine to effectively change.

Each time a memory access is requested, the memory subsystem checks if the requested virtual Page has a corresponding PTE in the TLB for the current ASID or with the **G** flag set, if this is the case it concatenates the physical frame address with the actual requested offset to access the right address.

If the necessary PTE is not present in the TLB, a TLB-Miss exception is raised and the BIOS reacts with a TLB Refill event, which executes the next steps:

1. Retrieve the PgTbl address from the Segment Table for the current ASID and required Segment.
2. Access the PgTbl and check if it is well-formed and well-located:
  - Address must be greater than 0x0000.8000,
  - Address must be word aligned,
  - PgTbl-Header must be valid (magic number is 0x2A),
  - PgTbl must not extend outside physical memory (e.g. [PgTbl addr + PgTbl size] < RAMTOP).
3. Linearly search the PgTbl for matching Virtual Page with correct ASID or **G** flag set.
4. If a matching PTE has been found, write it back in a random slot of the TLB and resume execution from the same instruction that raised the TLB-Miss exception, else raise a PTE-Miss exception.

The random algorithm uses all the TLB slots besides the first one (e.g. item 0) to ensure a safe entry is always available.

# Chapter 5

## External Devices

### 5.0.1 Installed Device Table

Five words, from 0x20 to 0x30, show the status of active devices. Each word represent a device line:

0x20	Disks
0x24	Tapes
0x28	Network
0x2C	Printers
0x30	Terminals

For each device line, if a specific device  $i$  is enable,  $i^{\text{th}}$  bit in representing word has value 1.

### 5.0.2 Device Registers

Addresses 0x40 to 0x2C0 hold device registers, the behavior of this memory region is the same as  $\mu$ MPS machine's.

### 5.0.3 Pending Interrupt Bitmap

Most of the interrupt lines are shared through all the devices of the same class, to identify which device is requesting for interrupt there are five registers from address 0x6FE0 to 0x6FF0 that hold a bitmap of interrupting devices per interrupt line.

This region is organized exactly as the Installed Device Table:

0x6FE0	Disks
0x6FE4	Tapes
0x6FE8	Network
0x6FEC	Printers
0x6FF0	Terminals

For each word,  $i$  bit is set if  $i^{\text{th}}$  device on that line is requesting for interrupt.

# Chapter 6

## BIOS & System Library

### 6.1 BIOS

#### 6.1.1 Bootstrap Function

The bootstrap program bundled with  $\mu$ ARM installation has initializes hardware facilities and starts execution.

The operation it performs are:

1. populate Exception Vector with Branch instructions to basic exception handling routines
2. set default Exception States Vector entries with **Branch to PANIC** instructions
3. retrieve entry point from kernel binary file
4. set execution mode to System mode with ARM ISA and all interrupts enabled
5. set exit point and ramtop value
6. clear all used scratch registers
7. jump to entry point

#### 6.1.2 Low Level Handlers

In addition to bootstrapping the machine, BIOS code provides low level exception handlers:

## Undefined Instruction Exception Handler

When an Undefined Instruction Exception is raised, the processor stores its internal state at the moment of the exception into the *Old Area* of the exception states vector relative to Undefined Exceptions and loads the content of the *New Area*.

## Interrupts Exception Handlers

For both Fast and regular Interrupt Exceptions the processor behaves in a way similar to Undefined Exceptions, it stores its state at the time of the interrupt and loads the *New Area* from Interrupts Exception States Vector's slot.

## Software Interrupt Handler

In addition to the regular operations (the same as the Undef and Interrupts handlers), in case of an SWI instruction the BIOS responds for known Low Level Services and sets the Cause register to the right value for the requested interruption (Syscall or Breakpoint).

## Data and Prefetch Exception Handlers

These two handlers share most of their code and perform special operations in case Virtual Memory is turned on:

- if VM is disabled, Abort Exceptions are treated as Undefined Exceptions,
- else low level TLB management is required.

In the latter case, the handler corrects the return address, setting it to the address of the faulting instruction and stores the processor state in the *TLB Old Area* of the Exception States Vector. It then checks if the Abort was labeled as a TLB-Miss and, if it was the case, it performs the TLB Refill procedure described in Virtual Memory section.

### 6.1.3 Low Level Services

Low level services are requested by issuing a SWI instruction with the right parameter:

#### Halt

By executing SWI #1, the BIOS will print "SYSTEM HALTED." on Terminal 0 and shut down the virtual machine.

## **Panic**

By executing **SWI #2**, the BIOS will print "KERNEL PANIC." on Terminal 0 and enter an infinite loop.

## **LDST**

A **SWI #3** instruction will begin the loading of the processor state stored at the address specified by *a1* register to actual processor's registers, checking destination mode and setting only the right processor's registers window.

## **Wait**

By executing **SWI #4**, the BIOS will put the machine in IDLE state waiting for an interrupt to wake the system up.

## **System Calls / Breakpoints**

If a **SWI #8** or **SWI #9** instruction is executed, the syscall handler passes up the call, setting the right cause in CP15 Cause register.

# **6.2 System Library**

System library is provided by *libuarm*, it offers a small but increasing set of methods to access low level functionalities.

## **tprint(char \*s)**

Print a '\0' terminated array of chars to Terminal 0.

This function uses busy waiting to wait for the device to be ready.

## **HALT()**

Run BIOS Halt function.

## **PANIC()**

Run BIOS Panic function.

## **WAIT()**

Run BIOS Wait function.

**LDST(void \*addr)**

Calls the BIOS function that loads the entire processor state from state\_t stored at *addr* address.

**STST(void \*addr)**

Stores the actual processor state in state\_t structure pointed by *addr*.

**SYSCALL(unsigned int sysNum, unsigned int arg1, unsigned int arg2, unsigned int arg3)**

Generates a software exception leading to kernel defined Syscall handler.

**BREAK(unsigned int arg0, unsigned int arg1, unsigned int arg2, unsigned int arg3)**

Generates a software exception leading to kernel defined Breakpoint handler.

**getStatus() / setStatus()**

Manipulate Current Program Status Register.

**getCause() / setCause()**

Manipulate Exception/Interrupt Cause register.

**getTimer() / setTimer()**

Manipulate Interval Timer.

**getTodHi() / getTodLo()**

Returns the upper/lower part of Time of Day 64-bit register.

**getControl() / setControl()**

Manipulate System Control Register.

**getTlbIndex() / setTlbIndex(unsigned int index)**

Manipulate TLB Index register.



### **getTLB\_Random()**

Returns TLB Random register.

### **getEntryHi() / setEntryHi(unsigned int hi) / getEntryLo() / setEntryLo(unsigned int lo)**

Manipulate TLB Entry Hi and Entry Low registers.

### **getBadVAddr()**

Returns BadVAddr register, which contains the faulting address in case of Page Fault Exceptions.

### **TLBWR()**

Write the contents of EntryHi and EntryLo to the TLB slot indicated by TLB Random register value.

### **TLBWI()**

Write the contents of EntryHi and EntryLo to the TLB slot indicated by TLB Index register value.

### **TLBR()**

Read the contents of TLB slot indicated by TLB Index register value in EntryHi and EntryLo registers.

### **TLBP()**

Scan the TLB searching for a pair that matches VPN in EntryHi and ASID in EntryHi or that has G flag set in EntryLo and is Valid, if a match is found, its index in the TLB cache is stored as TLB Index register value, otherwise that register will have 31st bit set to 1.

### **TLBCLR()**

Set all TLB contents to 0.

## Chapter 7

# Graphical User Interface

# Chapter 8

## Notes

### 8.1 Compilers and compiling

#### 8.1.1 Compilers

To compile a program to be run in  $\mu$ ARM you need an ARM compiler in order to generate code that the CPU is able to understand.

If you happen to be running a machine which is not ARM-based, you will need a cross-compiler (or better a cross-toolchain).

`arm-linux-gnueabi` and `arm-none-eabi` are the cross-toolchains with which the program is tested so they are likely the most compatible, but any toolchain able to compile code for ARM7TDMI processor will work.

#### 8.1.2 Compiling

When it comes the time to actually compile your code, be aware that you need to manually compile and then link each executable, because if `gcc` does all the work, it will include Linux system libraries as well. These libraries will try to "prepare" your program to be executed under Linux OS, adding initializing functions that will "break" execution in a bare metal system like  $\mu$ ARM.

So remember to add `-c` option while compiling each source file as well as `-mcpu=arm7tdmi` to ensure maximum compatibility with the system.

When you have all the required object files, you have to link them together with the following command:

```
arm-some-abi-gcc -nostartfiles -T \\  
/usr/include/uarm/ldscripts/elf32ltsarm.h.uarmcore.x \\  
/usr/include/uarm/crtso.o1
```

---

<sup>1</sup>optionally you may want to add `/usr/include/uarm/libuarm.o` if you included system library in your code, you can also use `elf32ltsarm.h.uarmaout.x` script to link user mode programs.

The last tool needed for preparing code for execution is `elf2uarm`, it converts elf binary files into  $\mu$ ARM elf files. The tool has three operating modes and two options:

- `-k` : create kernel core file (extension `*.core.uarm`) with symbol table map (extension `*.uarm.stab`)
- `-b` : create bootstrap BIOS file (extension `*.rom.uarm`)
- `-a` : create a.out user program file (extension `*.aout.uarm`)
- `-m` : force creation of symbol table map
- `-v` : verbose mode

## 8.2 Binary Formats

Output file formats used for core and user program files (`*.core.uarm` and `*.uarm` files) are essentially based on ELF standard. They have the same structure:

- header section (16 header fields described below)
- `.text` segment
- optional `.data` segment

Header section has the following format:

0	<code>AOUT_HE_TAG:</code>	Header TAG (aout.h defines different executable files magic numbers)
1	<code>AOUT_HE_ENTRY:</code>	Program Entry Point
2	<code>AOUT_HE_TEXT_VADDR:</code>	<code>.text</code> segment beginning virtual address
3	<code>AOUT_HE_TEXT_MEMSZ:</code>	<code>.text</code> segment actual Byte size
4	<code>AOUT_HE_TEXT_OFFSET:</code>	<code>.text</code> segment offset from Header section end
5	<code>AOUT_HE_TEXT_FILESZ:</code>	<code>.text</code> segment size (rounded to the next 4KB block)
6	<code>AOUT_HE_DATA_VADDR:</code>	<code>.data</code> segment beginning virtual address
7	<code>AOUT_HE_DATA_MEMSZ:</code>	<code>.data</code> segment actual Byte size
8	<code>AOUT_HE_DATA_OFFSET:</code>	<code>.data</code> segment offset from Header section end
9	<code>AOUT_HE_DATA_FILESZ:</code>	<code>.data</code> segment size (rounded to the next 4KB block)

the last 6 header fields are unused.

## 8.3 Hints

When writing Exception Handlers code, it is well advised to pay attention to the Program Counter value stored in the Old Area. As described in Exception Handling section, each exception leave a different value in Link Return register and this value is automatically moved to **Old Area.pc** from low level exception handlers, so, for example, when handling an interrupt, the **Old Area.pc** has to be decreased by 4 to point to the right return instruction.

You may also want look closely to the Low Level Handlers section to better understand what actions are performed by the BIOS code.

## 8.4 Known bugs

- Thumb ISA is misbehaving with some Branch instructions
- Terminals shortcuts are still not working
- If Symbol Table file is changed during execution the machine can misbehave