μARM Informal Specifications

Marco Melletti - melletti.marco@gmail.com
http://mellotanica.github.io/uARM/

Contents

Chapter 1

Introduction

μARM 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 μMPS2 architecture, which in turn takes inspiration from commonly known architectures.

In a more schematic fashion, μARM 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 Dustsheet* and the *ARM7TDMI Technical Reference Manual*.

Notational conventions:

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

- Fields are *italicized.*
- Instructions are `monospaced` and assembly instructions are `CAPITALIZED AND MONOSPACED`.
- 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 μARM 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 (**CPRS**), 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.

ARM State General Registers

User / System	*FIQ*	*Svc*	*Abort*	*IRQ*	*Undef*
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	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

▨ = banked register

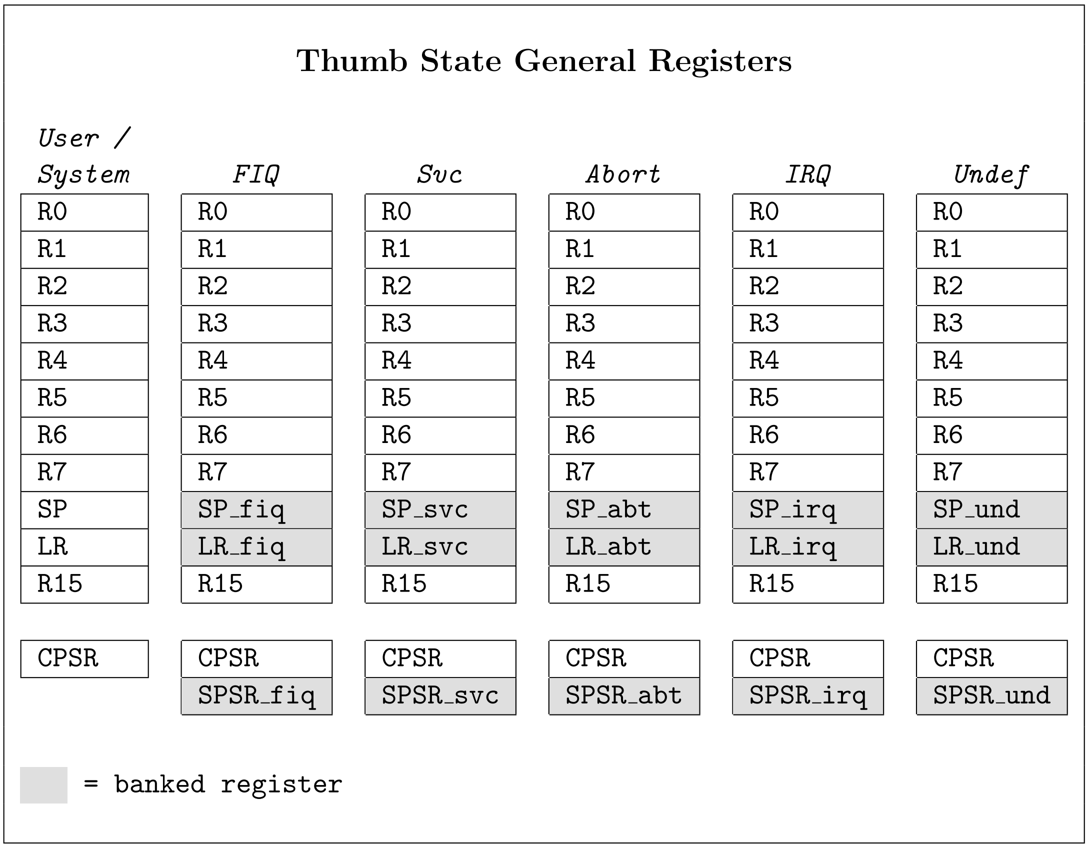
Even if the global 16 registers are defined as general purpose registers, there are some conventions adopted by the compiler in their use. The following list shows the full set of processor register visible in each mode with their conventional abbreviations and extended meaning:

- **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_*mode*** - 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.



Thumb State General Registers

User / System	*FIQ*	*Svc*	*Abort*	*IRQ*	*Undef*
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

▨ = banked register

2.2 System Coprocessor

CP15 gives access to a total of three 64-bit and four 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.

Coprocessors Access Register

31 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
SBZ	cp13	cp12	cp11	cp10	cp9	cp8	cp7	cp6	cp5	cp4	cp3	cp2	cp1	cp0

```
SBZ :     Should Be Zero
cp* :  00 Access Denied.
          (Accessing this coprocessor generates Undefined Exception)
       01 Privileged Access Only.
          (Accessing this coprocessor in user mode generates Undefined Exception)
       10 RESERVED.
          (Unpredictable)
       11 Full Access.
```

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.

The lower 32 bits of this register are addressed as **CP15.R2.EntryLow** or just **EntryLow** and the higher 32 bits are addressed as **CP15.R2.EntryHi** or simply **EntryHi**.

Register 6 - Faulting Address

R6 is a read-only register that is automatically loaded each time a Page Fault Exception is raised with the memory address that generated the exception.

Register 8 - TLB Random

R8 is a special read-only register used to index the TLB randomly (see sec. 4.2.3). This register is also addressed as **Random** or **TLBR**.

Register 10 - TLB Index

R10 is a special register used to index the TLB programmatically (see sec. 4.2.3). This register is also addressed as **Index** or **TLBI**.

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.

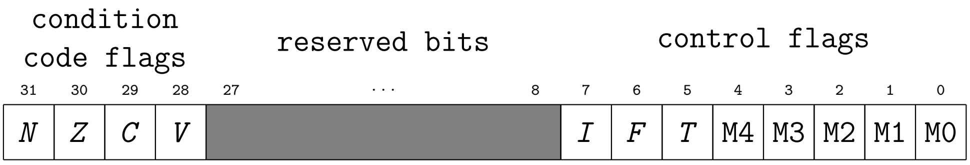
2.3 Execution Control

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:

Program Status Register



N	:	Negative / less then	*I*	:	IRQ disabled
Z	:	Zero	*F*	:	FIQ disabled
C	:	Carry / Borrow / Extend	*T*	:	State bit (Thumb enabled)
V	:	Overflow	*M*	:	Mode bits

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

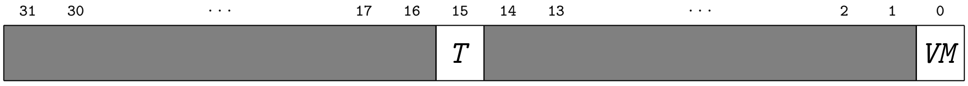
User Mode is the only unprivileged mode, this means that if the processor is running in User Mode, it cannot access reserved memory regions (see System Bus chapter) and it cannot modify CPSR control bits.

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):

System Control Register



VM : if set, the virtual address translation is enabled
T : if set, changes to CPSR.*T* are ignored

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
AND	logical AND
BIC	bit clear
BX	branch and exchange
CMN	compare negative
EOR	logical exclusive OR
LDM	load multiple registers from memory
LDRH	load halfword from memory
LDRSH	load signed halfword from memory
MLA	multiply accumulative
MOV	move register or constant
MRS	move PRS status/flags to register
MUL	multiply
MVN	move negative register
RSB	reverse subtract
SBC	subtract with carry
STM	store multiple
STRH	store halfword
SWI	software interrupt
TEQ	test bitwiser equality
UND	undefined instruction

ADD	add
B	brach
BL	branch with link
CDP	coprocessor data processing
CMP	compare
LDC	load coprocessor register from memory
LDR	load register from memory
LDRSB	load signed byte from memory
MCR	move cpu register to coprocessor register
MLAL	multiply accumulative long
MRC	move coprocessor register to cpu register
MSR	move register to PRS status/flags
MULL	multiply long
ORR	logical OR
RSC	reverse subtract with carry
STC	store coprocessor register to memory
STR	store register to memory
SUB	subtract
SWP	swap register with memory
TST	test bits

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
AND	logical AND
B	unconditonal branch
BIC	bit clear
BX	branch and exchange
CMP	compare
LDMIA	load multiple (increment after)
LDRB	load byte to register
LDRSB	load signed byte to register
LSL	logical shift left
MOV	move from register to register
MVN	move negative register
ORR	logical OR
PUSH	push to stack
SBC	subtract with carry
STR	store register to memory
STRH	store halfword to memory
SWI	software interrupt

ADD	add
ASR	arithmetical shift right
B[cond]	conditioned branch
BL	branch with link
CMN	compare negative
EOR	logical exclusive OR
LDR	load word to register
LDRH	load halfword to register
LDRSH	load signed halfword to register
LSR	logical shift right
MUL	multiply
NEG	negate word
POP	pop from stack
ROR	rotate right
STMIA	store multiple (increment after)
STRB	store byte to memory
SUB	subtract
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 `0x0000.0000`.

The first bus word, 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 `0x0000.0300`), 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 `0x0000.0000` to `0xFFFF.FFFF`), 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. 3.1.7). While storing the processor state, all ROM Level Exception Handlers move the value of the **lr** register to the **pc** register, making easier the return to the regular execution.

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.

2.5.3 Notes About Exception Handlers

When writing Exception Handlers code, it is well advised to pay attention to the Program Counter value stored in the Old Area. As described above, each exception leaves a different value in Link Return register and this value is automatically moved to **Old Area.pc** by the ROM 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.

Chapter 3

System Bus

The system bus connects each component in the system and lets processing units access physical memory and devices, as well as some special purpose registers.

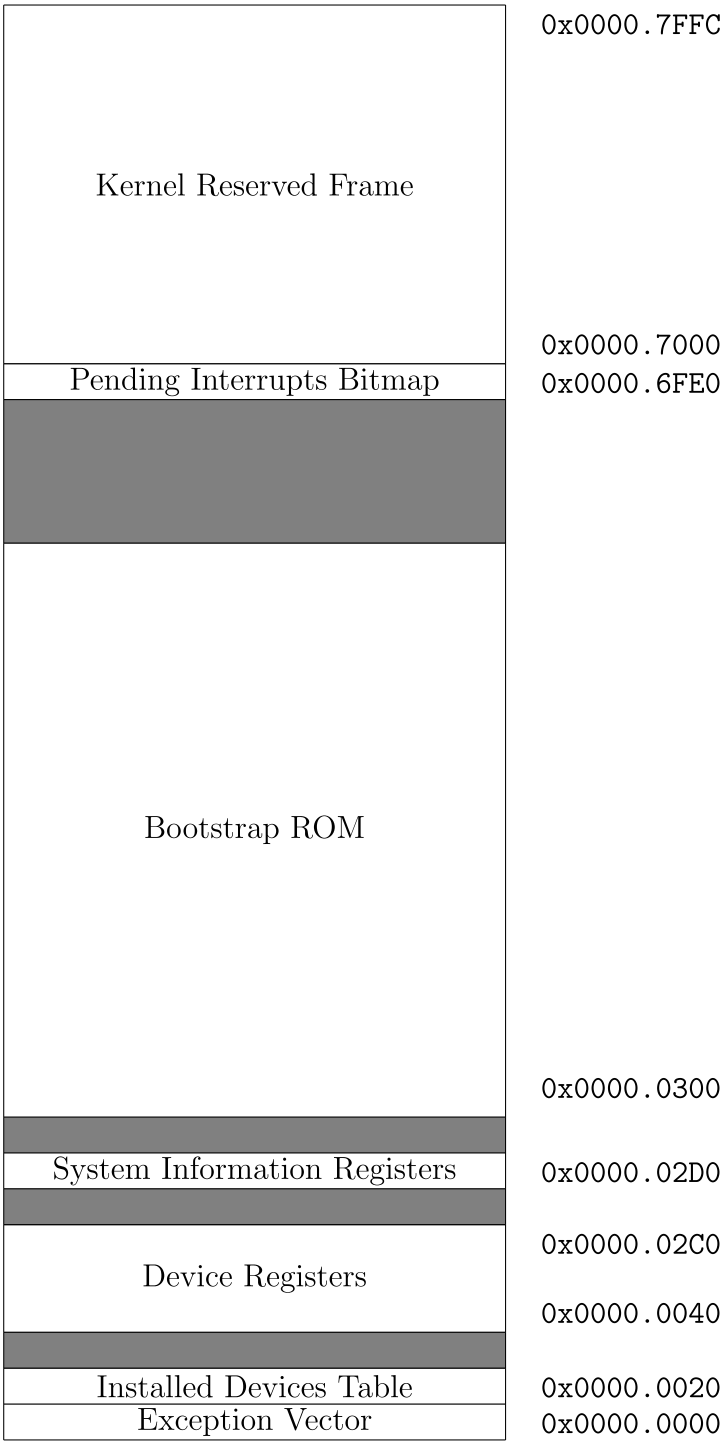
The CPU and the System Coprocessor (CP15) can directly access the bus reading or writing values from or to specific addresses.

The lower addresses (below `0x0000.8000`) are reserved for special uses and are accessible under certain conditions.

3.1 Reserved address space

The address region between `0x0000.0000` and `0x0000.8000` holds the fast exception vector, device access registers, system information registers, bootstrap ROM and the kernel reserved frame (i.e. the first RAM frame). Any access to this memory area in User mode is (should be) prohibited and treated by the system bus as errors.

Reserved Address Space



3.1.1 Exception Vector

The first bus addresses (`0x0000.0000` → `0x0000.001C`) are occupied by the fast exception vectors. Whenever an exception is risen, the processor automatically changes the **PC** register to point to one of these addresses. This way, if the system was correctly set up, a branch instruction will lead execution to the correct exception handler. It is the bootstrap ROM which typically writes a set of branch instructions to exception handlers in these fields.

The exception vector is organized as follows:

Exception Vector

Fast Interrupt Request	`0x0000.001C`
Interrupt Request	`0x0000.0018`
reserved/unused	`0x0000.0014`
Data Abort	`0x0000.0010`
Prefetch Abort	`0x0000.000C`
Software Interrupt	`0x0000.0008`
Undefined Instruction	`0x0000.0004`
Reset	`0x0000.0000`

3.1.2 Installed Devices Table

Five words, from `0x0000.0020` to `0x0000.0030`, show the status of active devices. Each word represents a device class, for each word, if a specific device i is enabled, i^{th} bit in relative word has value 1.

Installed Devices Table

Terminals	`0x0000.0030`
Printers	`0x0000.002C`
Network	`0x0000.0028`
Tapes	`0x0000.0024`
Disks	`0x0000.0020`

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
Device Class Bitmap	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

3.1.3 Device Registers

Addresses from `0x0000.0040` to `0x0000.02C0` hold device registers. Each device type has its own communication protocol, as described in chapter 5.

3.1.4 System Information Registers

Six registers, from address `0x0000.02D0` to `0x0000.02E4`, show system specific information:

System Information Registers

Interval Timer	`0x0000.02E4`
Time of day (Low)	`0x0000.02E0`
Time of day (High)	`0x0000.02DC`
Device registers base addr.	`0x0000.02D8`
RAM top address	`0x0000.02D4`
RAM base address	`0x0000.02D0`

These registers are all read-only, except for the interval timer which is a special device register (see sec. 5.2.1), and are aimed to provide useful information to the operating system.

3.1.5 Bootstrap ROM

The bootstrap ROM is loaded starting from address `0x0000.0300`, its maximum size is 109 KB. The content of the ROM is actually flashed at each reboot of the emulator copying each byte of the input image starting from the ROM base address, so the BIOS image does not need any special offset set by the linker.

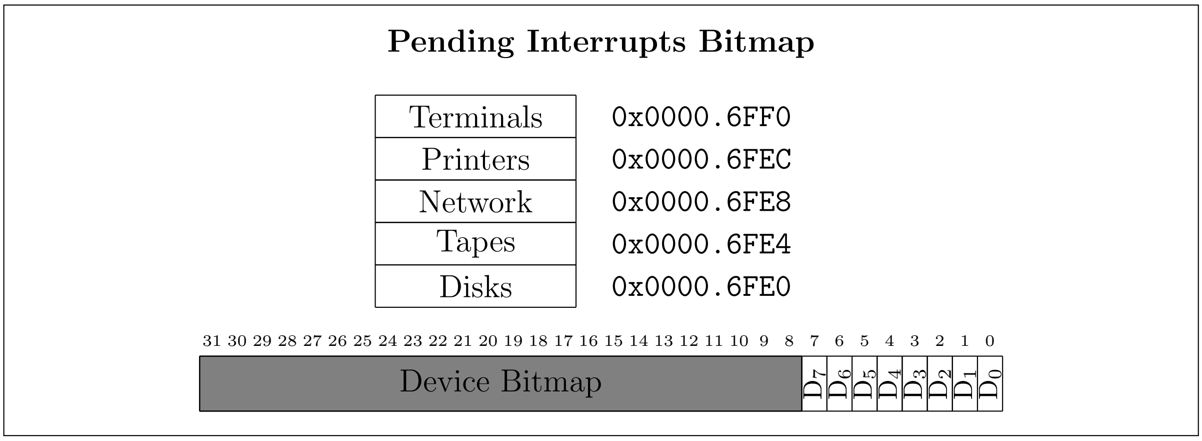
See section 6.1 for further details regarding provided BIOS implementation.

3.1.6 Pending Interrupts Bitmap

All the devices of one class are connected to the same interrupt line, when a device needs to notify some activity to the processor it sends a message through its interrupt line. To identify which specific device is requesting an interruption, there are five registers from address `0x0000.6FE0` to `0x0000.6FF0` that hold a bitmap of interrupting devices per interrupt line.

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

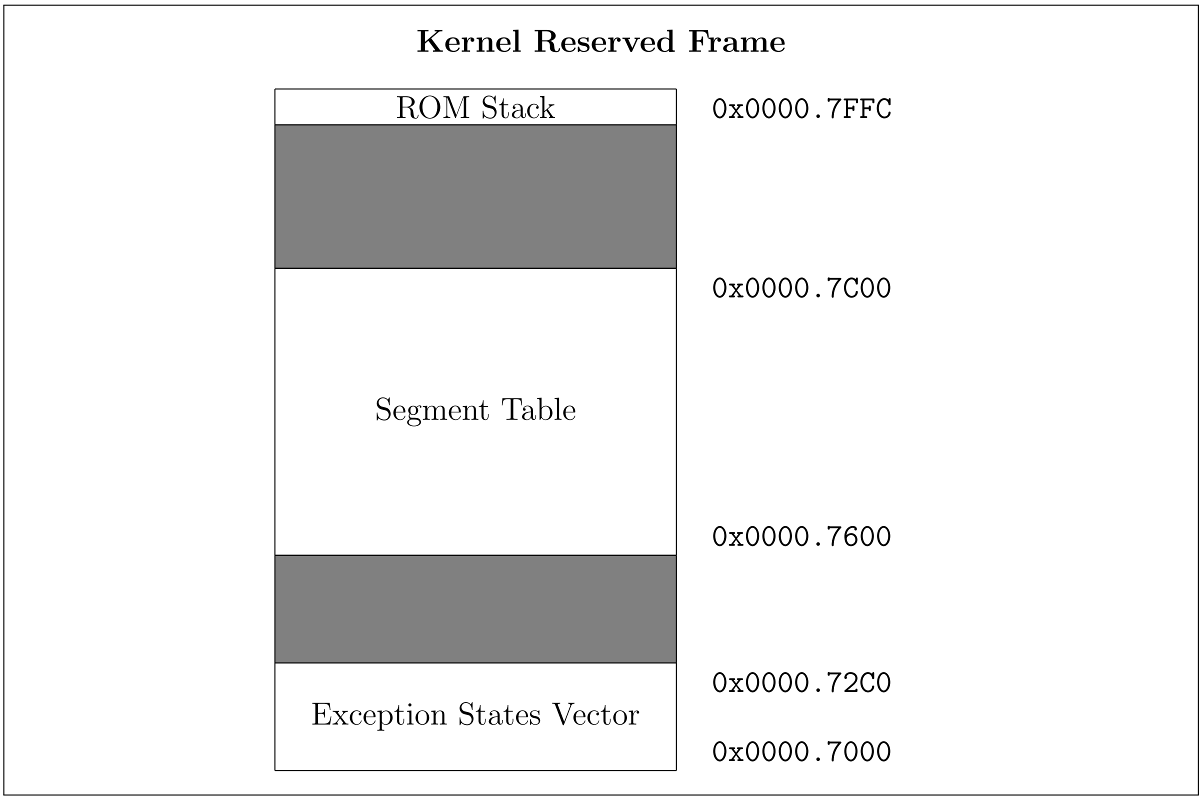
This region is organized exactly as the Installed Device Table:



Pending Interrupts Bitmap

3.1.7 Kernel Reserved Frame

The first memory frame (0x0000.7000 → 0x0000.7FFC) is reserved for kernel use:



Kernel Reserved Frame

The Exception states vector is a memory area to which processor states are saved and loaded from when entering into/exiting from exception handlers code. The Segment table holds 128 elements describing the virtual address space, for each ASID (corresponding to one entry in the segment table) there are three pointers to ASID's page tables (see sec. 4.2.2). When invoking ROM functions, the four words wide ROM stack (the last four words in the frame) is used as a memory stack and to pass parameters by the low level routines.

Stored Processor States

Processor states are defined by library data structure *state_t*, this structure is composed of 22 unsigned 32-bit integers representing processor registers' values and coprocessor's system control registers' values as well as saving time.

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 sl;    //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;
   unsigned int TOD_Hi;
   unsigned int TOD_Low;
} state_t;
```

These structures take 88 bytes each. Given this value, the BIOS code will look for the Old/New entries at the following addresses:

Exception States Vector

Syscall New	0x0000.7268
Syscall Old	0x0000.7210
PGMT New	0x0000.71B8
PGMT Old	0x0000.7160
TLB New	0x0000.7108
TLB Old	0x0000.70B0
Interrupt New	0x0000.7058
Interrupt Old	0x0000.7000

Each time an exception is risen, the BIOS handlers will store the processor state before the exception into the proper Old area, perform other tasks where required (see sec. 2.5.2) and eventually load the processor state stored in the corresponding New area.

The New areas must be filled with valid processor states pointing to kernel level exception handlers by kernel initialization stage.

3.2 Memory address space

The remaining addresses (`0x0000.8000` → `RAMTOP`) are mapped to memory subsystem, this bus region can be directly accessed by the processor and the coprocessor, it is used to store the kernel execution code and data as well as any other program that has to be executed.

The memory is accessible in physical addressing mode or virtual addressing mode, access modes and memory structure are described in detail in chapter 4.

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.

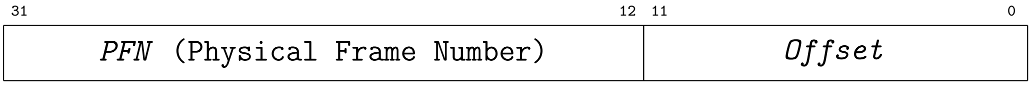
As stated in section 3.1, addresses below `0x0000.8000` are reserved for hardware/protected functions and belong to the reserved address space independently from the active addressing mode.

4.1 Physical addressing mode

This is the basic memory access scheme, when enabled, any memory access is directed to the physical address specified in the request, without any conversions. The machine begins execution with this mode active.

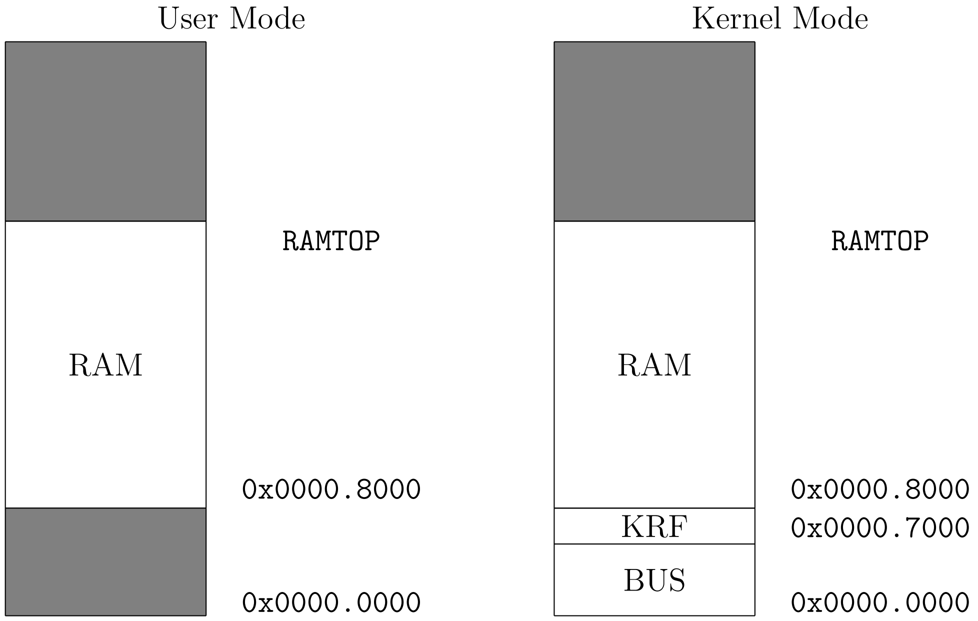
The physical address space is divided into equal sized frames of 4KB each. Hence a physical address has two components; a 20-bit Physical Frame Number or *PFN*, and a 12-bit *Offset* into the frame. Physical addresses have the following format:

Physical Address Format



All the available memory is directly accessible in Privileged mode and any address over `0x0000.8000` is directly accessible in User mode.

Physical Memory View



Trying to access an address below `0x0000.8000` while in User Mode will raise an `Address Error` exception.

The installed physical RAM starts at `0x0000.7000` and continues up to `RAMTOP`, this area will hold:

- The operating system code (.text), global variables/structures (.data), and stack(s).
- The user processes .text, .data and stacks.
- The Kernel Reserved Frame. As detailed in section 3.1.7, the BIOS code needs some writable storage to interact with the Kernel. The first 4KB (i.e. the first frame) of physical RAM are reserved for this purpose.

The first 7KB of the physical address space are reserved for Bus functions, as described in section 3.1. Any attempt to access an unidentified memory area will generate a `BUSERROR` exception.

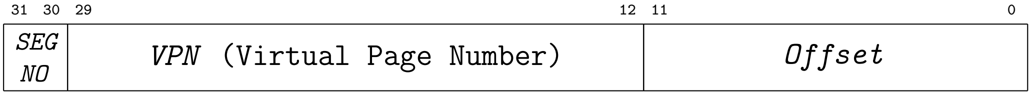
4.2 Virtual addressing mode

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 reserved address region (see sec. 3.1).

By setting *M* flag in System Coprocessor's register 1 (**CP15.R1**.*M*), you enable memory address translation.

The first two bits of a virtual address are the Segment Number (*SEGNO*). Virtual pages are the same size as physical frames, the final 12-bits indicate an *Offset* into a memory frame. The remaining 18-bits indicate the Virtual Page Number or *VPN*. Virtual addresses have the following format:

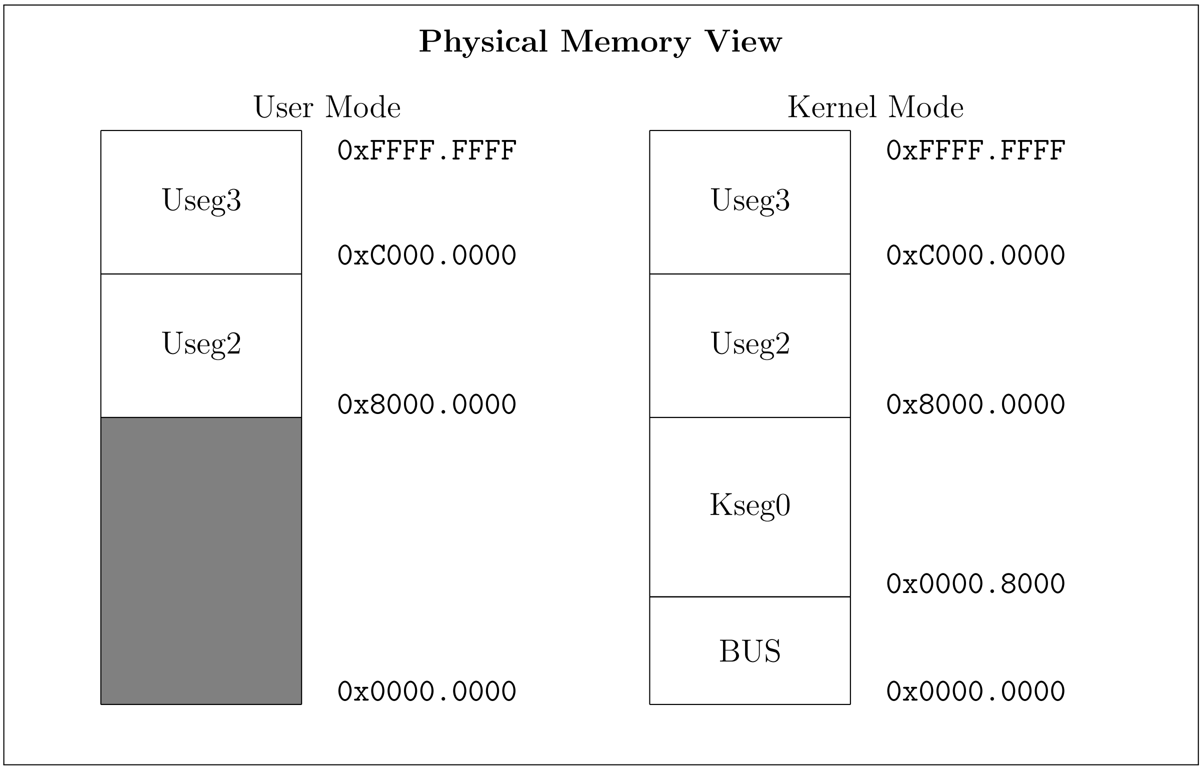
Virtual Address Format



The segment number is composed of two bits, the most important one differentiates kernel and user segments, the least important one is meaningful only in user segment and identifies private segment and global segment:

- Kseg0 (*SEGNO* 0 and 1) is the 2GB segment for the OS .text, .data, stacks, as well as the ROM code and device registers that sit at the beginning of this segment.
- Useg2 (*SEGNO* 2) is the 1GB virtual address space for the use of User mode processes as private memory region.
- Useg3 (*SEGNO* 3) is the 1GB virtual address space for the use of User mode processes as shared/global memory region.

In Privileged mode all logical memory is accessible, while in User mode only User Segments are accessible and any access to Kseg0 segment will generate an `Address Error` exception.



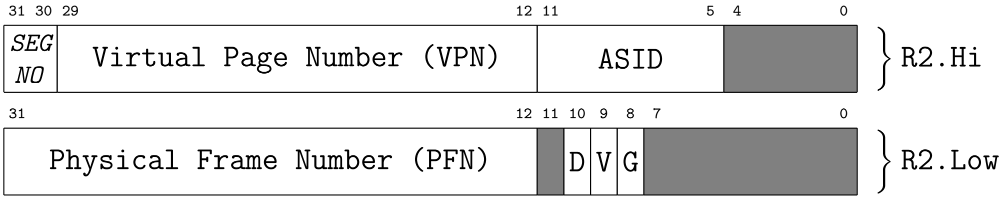
As part of its VM implementation, μARM assigns to each process a 7-bit identifier; hence μARM natively allows up to $2^7 = 128$ concurrent processes. To reflect the fact that each of these processes will run in its own virtual address space, this identifier is called the Address Space Identifier (*ASID*). The current *ASID* is part of the processor state and is stored in **EntryHi**.*ASID* (**CP15.R2.EntryHi**.*ASID*).

When the MMU is enabled the user process *ASID* is stored in the **EntryHi** register along with the Virtual Page number (i.e. the 20 most significant bits of the logical address). The **EntryLow** register is filled with the Physical Frame Number from the relevant page table and is kept up to date after each modification of **CP15.R2.EntryHi** value.

4.2.1 Page Table

The **CP15.R2** register is organized as a Page Table Entry (PTE):

Page Table Entry

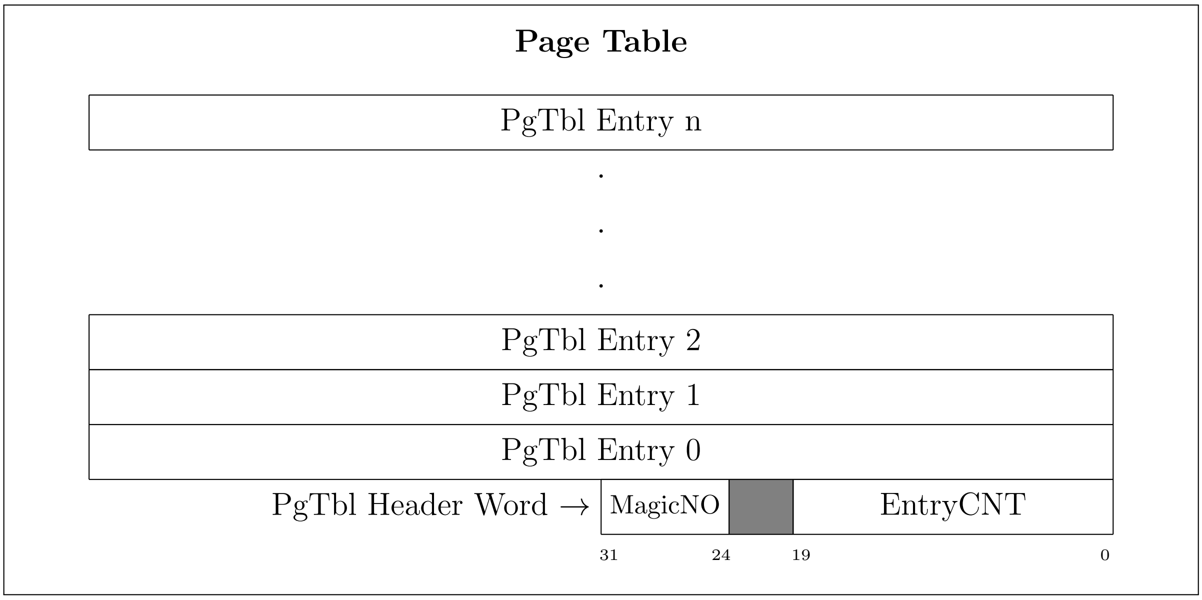


V : Valid bit   D : Dirty bit
G : Global bit

The Hi half of each entry identifies the logical frame to which the entry refers and the *ASID* of the owning process. The Low half of each entry specifies the physical corresponding frame (if any) and contains 3 flags used for memory protection schemes:

- **D**irty bit: if the flag is clear, any write access to the physical frame locations will rise a `TLB-Modification` exception.
- **V**alid bit: if the flag is set the Page Table Entry is considered valid, otherwise a `TLB-Invalid` exception is raised. This flag should be set only when the *PFN* points to the actual memory frame.
- **G**lobal bit: if the flag 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, as shown below.



Page Table

PgTbl Entry n

.

.

.

PgTbl Entry 2

PgTbl Entry 1

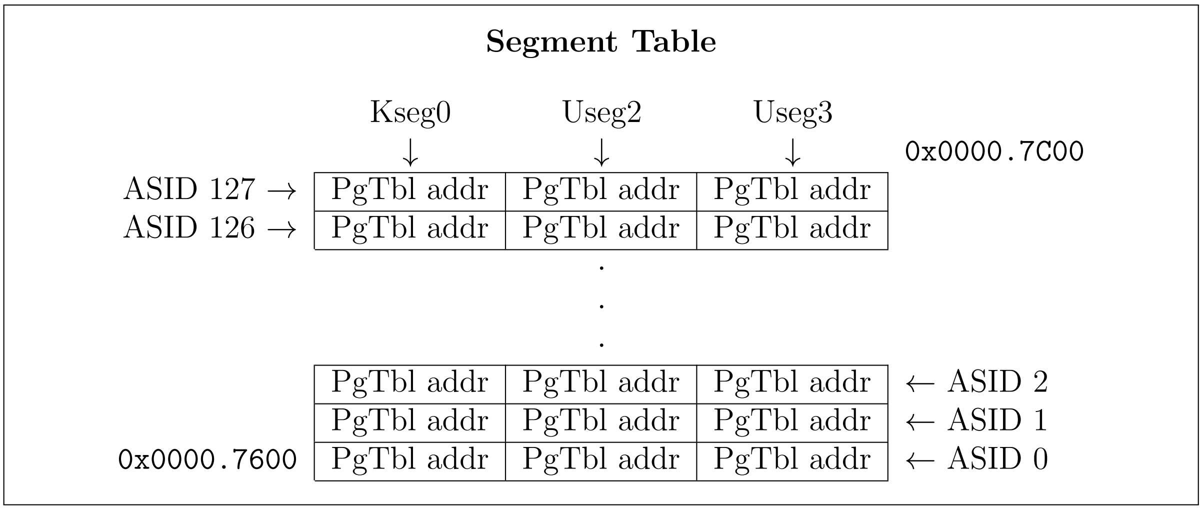
PgTbl Entry 0

PgTbl Header Word →

MagicNO		EntryCNT
31	24	19 … 0

4.2.2 Segment Table

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



Segment Table

	Kseg0 ↓	Useg2 ↓	Useg3 ↓	`0x0000.7C00`
ASID 127 →	PgTbl addr	PgTbl addr	PgTbl addr	
ASID 126 →	PgTbl addr	PgTbl addr	PgTbl addr	
	. . .			
	PgTbl addr	PgTbl addr	PgTbl addr	← ASID 2
	PgTbl addr	PgTbl addr	PgTbl addr	← ASID 1
`0x0000.7600`	PgTbl addr	PgTbl addr	PgTbl addr	← ASID 0

The segment table is automatically accessed from BIOS code when the Page Table Entry describing a needed memory frame is not present inside the TLB and needs to be retrieved (see sec. 4.2.3).

4.2.3 Translation Lookaside Buffer

μARM implements a Translation Lookaside Buffer (TLB) to translate virtual addresses to physical addresses, the buffer contains a specific amount of recently used PTEs and can use a variety of algorithms to select which entry to replace with a newly retrieved PTE (the BIOS handler implements a simple random selection). The number of available TLB slots is variable between 4 and 64 elements, it is configurable through the settings window of the emulator 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 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 is composed of 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).

   If one of these checks fails a `Bad-PgTbl` exception is raised.

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.

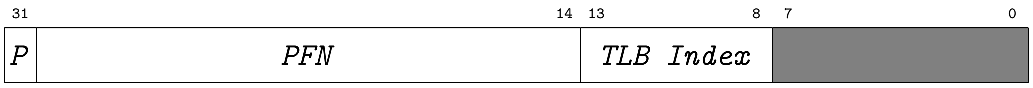
At this point, either there is a matching PTE, or an exception has been raised (either an `Address Error`, `Bad-PgTbl`, or `PTE-Miss` exception). If there is a matching TLB entry then the *V* and *D* control bits of the matching PTE are checked respectively. If no `TLB-Invalid` or `TLB-Modification` exception is raised, the physical address is constructed by concatenating the *Offset* from the virtual address to be translated to the *PFN* from the matching PTE.

CP15 registers used in address translation

CP15 implements four registers used to support virtual address translation. The first two have already been described, they are **CP15.R1** and **CP15.R2**, used respectively to turn on or off address translation and to cache the active Page Table Entry.

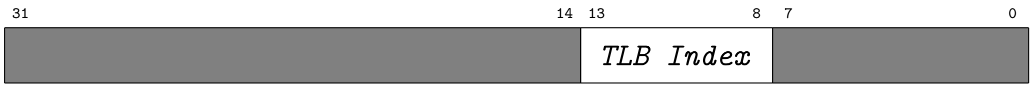
The contents of the TLB can be modified by writing values into the **EntryHi** and **EntryLo** registers and issuing either the TLB-Write-Index (TLBWI) or TLB-Write-Random (TLBWR) CP15 instruction. Which slot in the TLB the entry is written into is determined by which instruction is used and the contents of either the **CP15.R8** (**Random**) or **CP15.R10** (**Index**) register. Both the **Random** and the **Index** registers have a 6-bit *TLB-Index* field which addresses one of the `TLBSIZE` slots in the TLB.

CP15.R8 - TLB Index register



The **Index** register is a read/writable register. When a TLBWI instruction is executed, the contents of the **CP15.R2** register are written into the slot indicated by **Index**.*TLB-Index*.

CP15.R8 - TLB Random register



The **Random** register is a read-only register used to index the TLB randomly; allowing for more effective TLB-refiling schemes. **Random**.*TLB-Index* is initialized to `TLBSIZE`-1 and is automatically decremented by one every processor cycle until it reaches 1 at which point it starts back again at `TLBSIZE`-1. This leaves one TLB safe entry (entry 0) which cannot be indexed by **Random**. When a TLBWR instruction is executed, the contents of the **CP15.R2** register are written into the slot indicated by **Random**.*TLB-Index*. (μARMs TLB Refill algorithm uses TLBWR to populate the TLB.)

Three other useful CP15 instructions associated with the TLB are the TLB-Read (TLBR), TLB-Probe (TLBP), and the TLB-Clear (TLBCLR) commands.

- The TLBR command places the TLB entry indexed by **Index**.*TLB-Index* into the **CP15.R2** register. Note, that this instruction has the potentially dangerous affect of altering the value of **EntryHi**.*ASID*.

- The TLBP command initiates a TLB search for a matching entry in the TLB that matches the current values in the **EntryHi** register. If a matching entry is found in the TLB the corresponding index value is loaded into **Index**.*TLB-Index* and the Probe bit (**Index**.*P*) is set to 0. If no match is found, **Index**.*P* is set to 1.

- The TLBCLR command zeros out the unsafe TLB entries; entries 1 through `TLBSIZE`-1. This command effectively invalidates the current contents of the TLB cache.

See Sections 6.2 for more details on the TLBWI, TLBWR, TLBR, TLBP, TLBCLR CP15 instructions and how to access the **CP15.R2** and **Index** registers.

Chapter 5

External Devices

This Chapter is an adapted revision of μMPS2 - Principles of Operation, Chapter 5.

μARM supports five different classes of external devices: disk, tape, network card, printer and terminal. Furthermore, μARM can support up to eight instances of each device type. Each single device is operated by a controller. Controllers exchange information with the processor via device registers; special memory locations.

A device register is a consecutive 4-word block of memory. By writing and reading specific fields in a given device register, the processor may both issue commands and test device status and responses. μARM implements the full-handshake interrupt-driven protocol. Specifically:

1. Communication with device i is initiated by the writing of a command code into device is device register.

2. Device is controller responds by both starting the indicated operation and setting a status field in is device register.

3. When the indicated operation completes, device is controller will again set some fields in is device register; including the status field. Furthermore, device is controller will generate an interrupt exception by asserting the appropriate interrupt line. The generated interrupt exception informs the processor that the requested operation has concluded and that the device requires its attention.

4. The interrupt is acknowledged by writing the acknowledge command code in device is device register.

5. Device is controller will de-assert the interrupt line and the protocol can restart. For performance purposes, writing a new command after the interrupt is generated will both acknowledge the interrupt and start a new operation immediately.

The device registers are located in low-memory starting at `0x0000.0040`. As explained in section 4.2, regardless of **CP15.R1**.*M*, all addresses between `0x0000.0200` and `DEVTOP` are interpreted as physical addresses. Furthermore, the device registers can only be accessed when the processor is executing in privileged mode.

The following table details the correspondence between device class/type and interrupt line.

Interrupt Line #	Device Class
0	Inter-processor interrupts
1	Processor Local Timer
2	Bus (Interval Timer)
3	Disk Devices
4	Tape Devices
5	Network (Ethernet) Devices
6	Printer Devices
7	Terminal Devices

Some important issues relating to device management:

- Since there are multiple interrupt lines, and multiple devices attached to the same interrupt line, at any point in time there may be multiple interrupts pending simultaneously; both across interrupt lines and on the same interrupt line.

- The lower the interrupt line number, the higher the priority of the interrupt. Note how fast/critical devices (e.g. disk devices) are attached to a high priority interrupt line while slow devices are attached to the low priority interrupt lines.

- Interrupt lines 3-7 are used for external devices. Interrupt lines 0-2 are for internally generated interrupts. Lines 0-1 are present for future multiprocessor support, but currently unused.

- Disk and tape devices support Direct Memory Access (DMA); that is through cooperation with the bus, these devices are able to transfer whole blocks of data to/from memory from/to the device. Data blocks must be both wordaligned and of multiple-word in size. μARM supports any number of concurrent DMA operations; each on a different device. Care must be taken to prevent simultaneous DMA operations on the same chunk of memory.

- After an operation has begun on a device, its device register freezes - becomes read-only - and will not accept any other commands until the operation completes.

- Any device register for an uninstalled device is frozen - set to zero - and subsequent writes to the device register have no effect.

- Device registers use only physical addresses; this includes addresses used in DMA operations.

- Each external device in μARM is identified by the interrupt line it is attached to and its device number; an integer in [0..7]. μARM limits the number of devices per interrupt line to eight.

- For performance reasons, devices in the same class are, by default, attached to the same interrupt line.

5.1 Device Registers

All external devices share the same device register structure.

While each device class has a specific use and format for these fields, all device classes, except terminal devices, use:

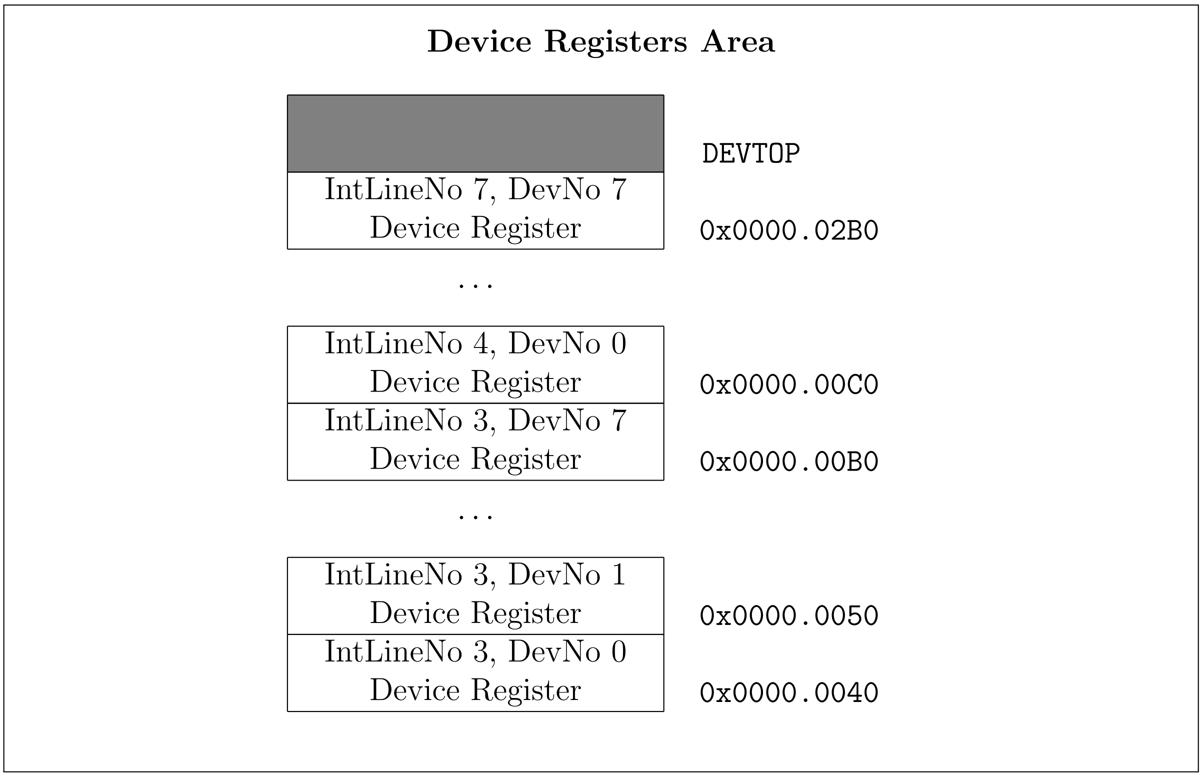
- **COMMAND** to allow commands to be issued to the device controller.

- **STATUS** for the device controller to communicate the device status to the processor.

- **DATA0** & **DATA1** to pass additional parameters to the device controller or the passing of data from the device controller.

Field #	Address	Field Name
0	(base) + `0x0`	**STATUS**
1	(base) + `0x4`	**COMMAND**
2	(base) + `0x8`	**DATA0**
3	(base) + `0xC`	**DATA1**

All 40 device registers in μARM are located in low memory starting at `0x0000.0040`.



Given an interrupt line (IntLineNo) and a device number (DevNo) one can compute the starting address of the devices device register:

devAddrBase = `0x0000.0040` + ((IntlineNo - 3) * `0x80`) + (DevNo * `0x10`)

5.2 The Bus Device and Interval Timer

The bus acts as the interface between the processor and the RAM, ROM, and all the external devices. In particular the bus performs the following tasks:

1. Management of the time of Day (TOD) clock and Interval Timer.
2. Arbitration among the interrupt lines, the devices attached to each interrupt line and the device registers.
3. Repository of basic system information.

5.2.1 Interval Timer

A read/writable unsigned word that is decremented by one every processor cycle and is set by μARM circuitry to `0xFFFF.FFFF` at system boot/reset time. The Interval Timer will generate an interrupt on interrupt line 2 whenever it makes the `0x0000.0000` → `0xFFFF.FFFF` transition. This is the only device attached to interrupt line 2, hence any interrupt on this line may be assumed to be associated with the Interval Timer. Interval Timer interrupts are acknowledged by writing a new value into the Interval Timer register.

The Interval Timer device register is located at `0x0000.02E4` (see sec. 3.1.4).

5.3 Disk Devices

μARM supports up to eight DMA supporting read/writable hard disk drive devices. All μARM disk drives have a blocksize equal to the μARM framesize of 4KB. Each installed disk drives device register **DATA1** field is read-only and describes the physical characteristics of the devices geometry.

Disk Device DATA1 Field

31 16	15 8	7 0
MAXCYL	*MAXHEAD*	*MAXSECT*

μARM disk drives can have up to 65536 cylinders/track, addressed [0..(*MAXCYL*-1)]; 256 heads (or track surfaces), addressed [0..(*MAXHEAD*-1)]; and 256 sectors/track, addressed [0..(*MAXSECT*-1)]. Each 4KB physical disk block (or sector) can be addressed by specifying its coordinates: (cyl, head, sect).

A disk drives device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Seek Error	Illegal parameter/hardware failure
5	Read Error	Illegal parameter/hardware failure
6	Write Error	Illegal parameter/hardware failure
7	DMA Transfer Error	Illegal physical address/hardware failure

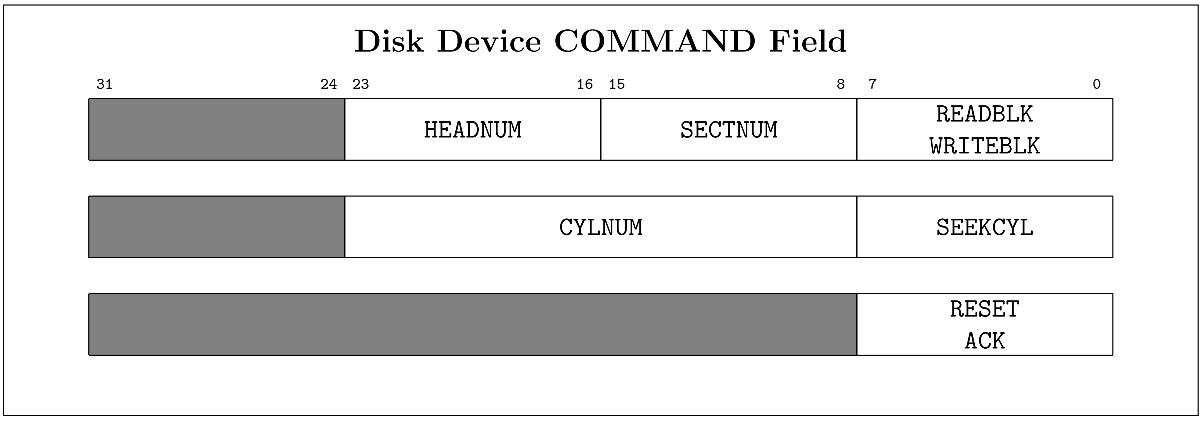
Status codes 1, 2, and 4-7 are completion codes. An illegal parameter may be an out of bounds value (e.g. a cylinder number outside of [0..(*MAXCYL*-1)]), or a non-existent physical address for DMA transfers.

A disk drives device register **DATA0** field is read/writable and is used to specify the starting physical address for a read or write DMA operation. Since memory is addressed from low addresses to high, this address is the lowest word-aligned physical address of the 4KB block about to be transferred.

A disk drives device register **COMMAND** field is read/writable and is used to issue commands to the disk drive.

Code	Command	Operation
0	RESET	Reset the device and move the boom to cylinder 0
1	ACK	Acknowledge a pending interrupt
2	SEEKCYL	Seek to the specified *CYLNUM*
3	READBLK	Read the block located at (*HEADNUM*, *SECTNUM*) in the current cylinder and copy it into RAM starting at the address in **DATA0**
4	WRITEBLK	Copy the 4KB of RAM starting at the address in **DATA0** into the block located at (*HEADNUM*, *SECTNUM*) in the current cylinder

The format of the **COMMAND** field, as illustrated in Figure 5.4, differs depending on which command is to be issued:



Disk Device COMMAND Field

A disk operation is started by loading the appropriate value into the **COMMAND** field. For the duration of the operation the devices status is Device Busy. Upon completion of the operation an interrupt is raised and an appropriate status code is set; Device Ready for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an ACK or RESET command.

Disk device performance, because both read and write operations are DMA-based, strongly depends on the system clock speed. While read/write throughput may reach MBs/sec in magnitude, the disk hardware operations remain in the millisecond range.

5.4 Tape Devices

μARM supports up to eight tape-removable, DMA supporting, read-only tape devices. All μARM tape devices support a blocksize of 4KB. Each installed tape devices register **DATA1** field is read-only and describes the current marker under the tape head when the device is idle.

Code	Marker	Meaning
0	EOT	End of Tape
1	EOF	End of File
2	EOB	End of Block
3	TS	Tape Start

A tape starts with a TS marker and ends with an EOT marker. It may be viewed as a collection of blocks, delimited by EOB markers, which are divided into files, delimited by EOF markers. An EOF marker acts as the EOB marker for the last block of the file and the EOT marker act as the EOF (and therefore also an EOB) marker for the last file on the tape.

When there is no tape cartridge loaded into the tape device, the **DATA1** field will contain the EOT marker, and the **STATUS** field will contain the Device Ready code. Since there is no tape cartridge present, the **COMMAND** field, though, will not accept any commands. Only when a tape is loaded does the device wake up and begin accepting commands. When a tape cartridge is loaded, the tape device rewinds the cartridge back to the TS marker.

A tape drives device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Skip Error	Illegal command/hardware failure
5	Read Error	Illegal command/hardware failure
6	Back 1 Block Error	Illegal command/hardware failure
7	DMA Transfer Error	Illegal physical address/hardware failure

Status codes 1, 2, and 4-7 are completion codes. An illegal parameter may be an attempt to read beyond the EOT marker or a non-existent physical address for DMA transfers.

A tape drives device register **DATA0** field is read/writable and is used to specify the starting physical address for a DMA read operation. Since memory is addressed from low addresses to high, this address is the lowest word-aligned physical address of the 4 KB block about to be transferred.

A tape drives device register **COMMAND** field is read/writable and is used to issue commands to the tape drive.

Code	Command	Operation
0	RESET	Reset the device and rewind the tape to TS marker
1	ACK	Acknowledge a pending interrupt
2	SKIPBLK	Forward the tape up to the next EOB/EOT
3	READBLK	Read the current block up to the next EOB/EOT marker and copy it into RAM starting at the address in **DATA0**
4	BACKBLK	Rewind the tape to the previous EOB/EOT marker

A tape operation is started by loading the appropriate value into the **COMMAND** field. For the duration of the operation the devices status is Device Busy. Upon completion of the operation an interrupt is raised and an appropriate status code is set; Device Ready for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an ACK or RESET command.

Tape device performance, because read operations are DMA-based, strongly depends on the system clock speed. Tape read throughput can range from 2 MB/sec when the processor clock is set at 1 MHz, to over 4 MB/sec when the processor clock is bumped up to 99 MHz.

5.5 Network (Ethernet) Adapters

μARM supports up to eight DMA supporting network (i.e. Ethernet) adapters. Though these devices are DMA-based, they are not block devices. Network adapters operate at the byte level and transfer into/out of memory only the amount of data called for. Since packets on a network typically follow standard MTU sizes, this data should never exceed (by much) 1500 bytes.

Network adapters share some characteristics with terminal devices; they are simultaneously both an input device and an output device. As an output device, network adapters behave like other peripherals: a write command is issued and when the write (i.e. transmit) is completed, an interrupt is generated.

For packet receipt, there are two modes of operation:

- Interrupt Enabled: Whenever a packet arrives, an interrupt is generated - this interrupt is not the result of an earlier command. After ACKing this interrupt one issues a READNET command to read the packet. When the read is completed, another interrupt is generated, which itself must also be ACKed. In Interrupt

Enabled mode, each incoming packet, when successfully read, is a two-interrupt sequence.

- Interrupt Disabled: When packets arrive, no interrupt is generated. The network adapter must be polled to determine if a packet is available. The READNET command is non-blocking, and returns 0 if there is no packet to be read. The READNET command will still generate an interrupt, which must be ACKed, upon its conclusion.

A network adapters device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Readya	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
5	Read Error	Error reading packet from device
6	Write Error	Error attempt to send packet
7	DMA Transfer Error	Illegal physical address/hardware failure
128	Read Pending	Interrupts Enabled and packet present

Status codes 1, 2, and 5-7 are completion codes. An illegal address may be an out of bounds value or a non-existent physical address for DMA transfers.

Status code 128 is not a distinct status code, it is used in a logical OR fashion with the other status codes. Hence there are actually thirteen status values: 0, (1 & 129), (2 & 130),. . . , (7 & 135). For example, a status code value of 130 indicates that both an illegal operation was requested AND there is a packet pending for reading. The Read Pending status codes are only used when the network adapter is operating Interrupt Enable mode.

Code	Command	Operation
0	RESET	Reset the device and reset all configuration data to defaults
1	ACK	Acknowledge a pending interrupt
2	READCONF	Read configuration data into **DATA0** & **DATA1**
3	READNET	Read the next packet from the adapter and copy it into RAM starting at the address in **DATA0**
4	WRITENET	Send a packet of data starting at the RAM address in **DATA0**, whose length is in **DATA1**
5	CONFIG	Update adapter configuration data from values in **DATA0** & **DATA1**

Network Adapter DATA0 Field

31 24	23 16	15	14 11	10	9	8	7 0
2nd MAC Octet	1st MAC Octet	*SM*		*PQ*	*IE*	*ND*	

Network Adapter DATA1 Field

31 24	23 16	15 8	7 0
6th MAC Octet	5th MAC Octet	4th MAC Octet	3rd MAC Octet

The **DATA0** fields, during configuration operations (READCONF & CONFIG), are defined as follows:

- *ND* (NAMED, bit 8): When **DATA0**.*ND*=1, the network adapter will automatically fill all outgoing packets source MAC address field with the network adapters MAC address.

- *IE* (Interrupt Enable, bit 9): If **DATA0**.*IE*=1, whenever a packet is pending on the device (i.e. waiting to be read), it will immediately generate an interrupt. After ACKing this interrupt, one issues a READNET command to facilitate the reading of the packet. The READNET command must then also be ACKed.

- *PQ* (PROMISQ, bit 10): If **DATA0**.*PQ*=1 the network adapter will capture and save all packets its receives. When **DATA0**.*PQ*=0, the device will ignore/drop any packets not intended for its MAC address. Broadcast packets will still be received even when **DATA0**.*PQ*=0.

- *SM* (SetMAC, bit 15): When **DATA0**.*SM*=1 and a CONFIG command is issued, the MAC address of the adapter is updated to the values in **DATA0** & **DATA1**. When **DATA0**.*SM*=0 and a CONFIG command is issued, the adapters MAC address remains unchanged.

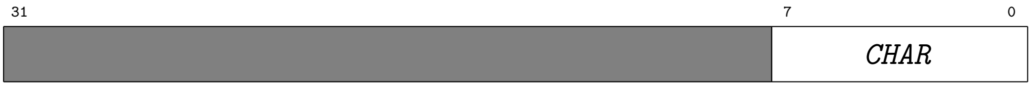
As described above, the **DATA0** & **DATA1** fields are overloaded; either containing device status values or DMA addresses and lengths. One uses the CONFIG to set network adapter configuration values. Similarly, after a READNET or WRITENET operation, one can use a READCONF operation to reset the **DATA0** & **DATA1** registers to reflect the current adapter configuration values.

5.6 Printer Devices

μARM supports up to eight parallel printer interfaces, each one with a single 8-bit character transmission capability.

The **DATA0** field for printer devices is read/writable and is used to set the character to be transmitted to the printer. The character is placed in the low-order byte of the **DATA0** field. The **DATA1** field, for printer devices is not used.

Printer Device DATA0 Field



A printers device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Print Error	Error during character transmission

Status codes 1, 2, and 4 are completion codes.

A printers device register **COMMAND** field is read/writable and is used to issue commands to the printer interface.

Code	Command	Operation
0	RESET	Reset the device interface
1	ACK	Acknowledge a pending interrupt
2	PRINTCHR	Transmit the character in **DATA0** over the line

A printer operation is started by loading the appropriate value into the **COMMAND** field. For the duration of the operation the devices status is Device Busy. Upon completion of the operation an interrupt is raised and an appropriate status code is set; Device Ready for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an ACK or RESET command.

The printer interfaces maximum throughput is 125 KB/sec.

5.7 Terminal Devices

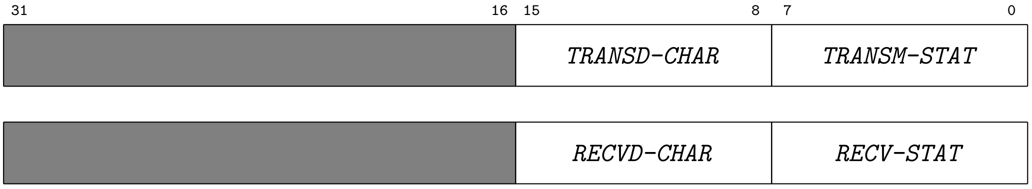
μARM supports up to eight serial terminal device interfaces, each one with a single 8-bit character transmission and receipt capability.

Each terminal interface contains two sub-devices; a transmitter and a receiver. These two sub-devices operate independently and concurrently. To support the two-subdevices a terminal interfaces device register is redefined as follows:

Field #	Address	Field Name
0	(base) + 0x0	**RECV-STATUS**
1	(base) + 0x4	**RECV-COMMAND**
2	(base) + 0x8	**TRANSM-STATUS**
3	(base) + 0xc	**TRANSM-COMMAND**

The **TRANSM_STATUS** and **RECV_STATUS** fields (device register fields 0 & 2) are read-only and have the following format.

Terminal Device TRANSM_STATUS and RECV_STATUS Fields



The *Status* byte has the following meaning:

Code	*RECV-STATUS*	*TRANSM-STATUS*
0	Device Not Installed	Device not installed
1	Device Ready	Device Ready
2	Illegal Operation Code Error	Illegal Operation Code Error
3	Device Busy	Device Busy
4	Receive Error	Transmission Error
5	Character Received	Character Transmitted

The meaning of status codes 0-4 are the same as with other device types. Furthermore:

- The Character Received code (5) is set when a character is correctly received from the terminal and is placed in **RECV_STATUS**.*RECVD-CHAR*.

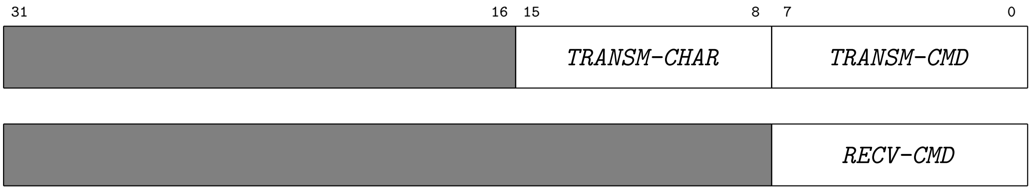
- The Character Transmitted code (5) is set when a character is correctlytransmitted to the terminal and is placed in **TRANSM_STATUS**.*TRANSD-CHAR*.
- The Device Ready code (1) is set as a response to an ACK or RESET command.

A terminals **TRANSM_COMMAND** and **RECV_COMMAND** fields are read/ writable and are used to issue commands to the terminals interface.

Code	*TRANSM-CMD*	*RECV-CMD*	Operation
0	RESET	RESET	Reset the transmitter or receiver interface
1	ACK	ACK	Ack a pending interrupt
2	TRANSMITCHAR	RECEIVECHAR	Transmit or Receive the character over the line

The **TRANSM_COMMAND** and **RECV_COMMAND** fields have the following format:

Terminal Device TRANSM_COMMAND and RECV_COMMAND Fields



RECV_COMMAND.*RECV-CMD* is simply the command.
The **TRANSM_COMMAND** field has two parts; the command itself (**TRANSM_COMMAND**.*TRANSM-CMD*) and the character to be transmitted (**TRANSM_COMMAND**.*TRANSM-CHAR*).

A character is received, and placed in **RECV_STATUS**.*RECVD-CHAR* only after a RECEIVECHAR command has been issued to the receiver.

The operation of a terminal device is more complicated than other devices because it is two sub-devices sharing the same device register interface. When a terminal device generates an interrupt, the (operating systems) terminal device interrupt handler, after determining which terminal generated the interrupt, must furthermore determine if the interrupt is for receiving a character, for transmitting a character, or both; i.e. two interrupts pending simultaneously.

If there are two interrupts pending simultaneously, both must be acknowledged in order to have the appropriate interrupt pending bit in the Interrupt Line 7 Interrupting Devices Bit Map turned off.

To make it possible to determine which sub-device has a pending interrupt there are two sub-device ready conditions; Device Ready and Character Received/Transmitted. While other device types can use a Device Ready code to signal a successful completion, this is insufficient for terminal devices. For terminal devices it is necessary to distinguish between a state of successful completion though the interrupt is not yet acknowledged, Character Received/Transmitted, and a command whose completion has been acknowledged, Device Ready.

A terminal operation is started by loading the appropriate value(s) into the **TRANSM_COMMAND** or **RECV_COMMAND** field. For the duration of the operation the sub-devices status is Device Busy. Upon completion of the operation an interrupt is raised and an appropriate status code is set in **TRANSM_STATUS** or **RECV_STATUS** respectively; Character Transmitted/Received for successful completion or one of the error codes. The interrupt is acknowledged by issuing an ACK or RESET command to which the sub-device responds by setting the Device Ready code in the respective status field.

The terminal interfaces maximum throughput is 12.5 KB/sec for both character transmission and receipt.

Chapter 6

BIOS & System Library

6.1 BIOS

6.1.1 Bootstrap Function

The bootstrap program bundled with μARM installation initializes hardware facilities and starts execution. The code for the bootstrap routine can be found in the file `BIOS.s`.

The BIOS code performs the following operations at boot time:

1. Populate of the Exception Vector with Branch instructions to ROM Level Exception Handlers;

2. Set default Exception States Vector entries with `Branch to PANIC` instructions;

3. Retrieve Entry Point from core binary code loaded in RAM;

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.

`BIOS.s` source file also contains the code for ROM Level Exception Handlers (see Sec. 2.5.2) and ROM Level Services; the default installation directory of this and the other support files is `/usr/include/uarm/`.

6.1.2 ROM Level Services

ROM Level Services are requested by issuing a `SWI` instructions with certain parameters and are served by the BIOS code:

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` instructions will begin the loading of the processor state stored at the address loaded into **a1** register to actual processor's registers, checking destination mode and setting only the right register window.

Wait

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

System Calls / Breakpoints

If a `SWI #8` or `SWI #9` instructions 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 set of methods to access low level functionalities from *C* language:

`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()`

Request BIOS Halt service.

`PANIC()`

Request BIOS Panic service.

`WAIT()`

Request BIOS Wait service.

`LDST(void *addr)`

Request BIOS LDST service setting **a1** to `addr` value.

`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. **a1** is loaded with `sysNum`, **a2** is loaded with `arg1`, **a3** is loaded with `arg2` and **a4** is loaded with `arg3`

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

Generates a software exception leading to kernel defined Breakpoint handler. **a1** is loaded with `arg0`, **a2** is loaded with `arg1`, **a3** is loaded with `arg2` and **a4** is loaded with `arg3`

`getSTATUS() / setSTATUS()`

Manipulate Current Program Status Register (**CPSR**).

`getCAUSE() / setCAUSE()`

Manipulate Exception/Interrupt Cause register (**CP15.R15**).

`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 (**CP15.SCB**).

`getTLB_Index() / setTLB_Index(unsigned int index)`

Manipulate TLB Index register.

`getTLB_Random()`

Returns TLB Random register.

`getEntryHi() / setEntryHi(unsigned int hi) / getEntryLo() / setEntryLo(unsigned int lo)`

Manipulate Page Table Entry Hi and Entry Low registers (**CP15.R2.EntryHi** and **CP15.R2.EntryLow**).

`getBadVAddr()`

Returns Faulting Address register (**CP15.R6**).

`TLBWR()`

Write the contents of **CP15.R2** to the TLB slot indicated by TLB Random register value.

`TLBWI()`

Write the contents of **CP15.R2** to the TLB slot indicated by TLB Index register value.

`TLBR()`

Read the contents of TLB slot indicated by TLB Index register value to **CP15.R2** register.

`TLBP()`

Scan the TLB searching for a pair that matches *VPN* in and *ASID* in **CP15.R2.EntryHi** or that has *G* flag set in **CP15.R2.EntryLo** and is *V*alid, if a match is found, its index in the TLB cache is stored as TLB Index register value, otherwise that register will have the most significant bit set to 1.

TLBCLR()

Set all TLB contents to 0.

6.2.1 Additional Libraries

Two more simple libraries are provided in addition to the system library:

`ulibuarm`

A simple subset of `libuarm` to be used by user mode programs, only exposes a mean to request system calls through the same `SYSCALL` instruction; see Section 6.2.

`libdiv`

This library implements integer division and module operations that are not provided by the processor instruction set, it must be linked together with any program that uses divisions.

Chapter 7

Emulator Usage

The following sections will cover the usage of the emulator itself, along with the necessary tools required for compiling the programs to be run, debugging functionalities provided by μARM and the support tool `uarm-mkdev` used to create device files needed for advanced usage.

7.1 Compiling and Running the Machine

μARM is an ARM7tdmi-based system emulator; in order to build a program for the correct ARM architecture, an ARM compiler is needed. Once a valid executable file is ready, the machine must be configured to load the proper core file (and optionally BIOS file). At last the machine is run and the output is read from terminal screens or printed files.

7.1.1 C Language Development for μARM

Run time C-library support utilities are obviously not available. This includes I/O statements (e.g. `printf` from `stdio.h`), storage allocation calls (e.g. `malloc`) and file manipulation methods. In general any C-library method that interfaces with the operating system is not supported; μARM does not have an OS to support these calls - unless you write one to do so. The `libuarm` library, described in Section 6.2, is the only support library available.

The μARM linker requires a small function, named `__start()`. This function is to be the entry point to the program being linked. Typically `__start()` is provided from system library and will initialize some registers and then call `main()`. After `main()` concludes, control is returned to `start()` which should perform some appropriate termination service. Two such functions, written in assembler, are provided:

- `crtso.o`: This file is to be used when linking together the files for the kernel/OS. The version of `__start()` in this file simply runs the `main()` function of the program (i.e. kernel), assuming it is loaded in RAM beginning at `0x0000.8000`.

- `crti.o`: This file is to be used when linking together the files for individual U-procs. The version of `__start()` in this file assumes that the programs (i.e. U-procs) header has 0x8000.0000 as its starting (virtual) address. Some registers are stored and restored before and after the `main()` call. `__start()` assumes that the kernel will initialize $SP. The last instruction if the `__start()` routine is the peaceful termination of the program using the dedicated Syscall. As the Syscall numbering is implementation dependent, the value at the beginning of this file needs to be modified accordingly from the OS developer.

7.1.2 Compiling

`arm-none-eabi` toolchain will be taken into example to explain the compiling process, but any ARM cross-toolchain, or any toolchain run on an ARM system should be able to generate proper code for μARM execution.

To be sure the compiler will not include the host system's libraries (see Sec. 7.1.1), the `-c` option is necessary while compiling each source file, as well as the `-mcpu=arm7tdmi` to ensure maximum compatibility with the system.

Once each source file has been compiled into an object file, everything has to be linked together using the provided start files (`crtso.o` and `crti.o`) and the `-T` option to select the right memory map:

- `elf32ltsarm.h.uarmcore.x` is the memory map used for kernel binaries, which are meant to be executed with virtual memory turned off;

- `elf32ltsarm.h.uarmaout.x` is the memory map used for Uproc's binaries, which are meant to be executed with virtual memory enabled.

Compiling an Operating System

Take as an example a program composed of the following modules:

- `core.c`, `core.h`: core module, uses `libuarm` library;

- `service.c`, `service.h`: library implementing service functions;

- `test.c`: test module to check program behavior.

To build such a program using `arm-none-eabi` toolchain one should execute the following commands:

```
arm-none-eabi-gcc -c -mcpu=arm7tdmi core.c -o core.o
arm-none-eabi-gcc -c -mcpu=arm7tdmi service.c -o service.o
arm-none-eabi-gcc -c -mcpu=arm7tdmi test.c -o test.o

arm-none-eabi-ld -T \\
/usr/include/uarm/ldscripts/elf32ltsarm.h.uarmcore.x \\
/usr/include/uarm/crtso.o /usr/include/uarm/libuarm.o \\
core.o service.o test.o -o kernel
```

`/usr/include/uarm/` is the default installation directory for support files, in this example, and in the ones that will follow, this location will be considered as valid and filled with the provided support files. The order of the object files in the linking (last) command is important: specifically, the first two support files must be in their respective positions.

Compiling a Uproc file

Take as an example a user mode program that one wishes to run on an already existing μARM operating system: `uproc.c`. This program will use system calls as well as integer divisions, respectively provided by `ulibuarm` and `libdiv` libraries.

To build such a program using `arm-none-eabi` toolchain one should execute the following commands:

```
arm-none-eabi-gcc -c -mcpu=arm7tdmi uproc.c -o uproc.o

arm-none-eabi-ld -T
/usr/include/uarm/ldscripts/elf32ltsarm.h.uarmaout.x \\
/usr/include/uarm/crti.o /usr/include/uarm/ulibuarm.o \\
/usr/include/uarm/libdiv.o uproc.o -o uproc
```

Finally, this executable file can be (optionally) loaded onto a tape cartridge with the following command:

```
uarm-mkdev -t uproc.uarm uproc
```

which produces the preloaded tape cartridge file: `uproc.uarm` (for further details see Sec. 7.2).

Compiling BIOS ROM

ROM code development must be done in ARM assembler. Consider the case where one has a new version of the execution time ROM routines: `testrom.s`.

One should assemble the source file using the command:

```
arm-none-eabi-gcc -mcpu=arm7tdmi -c -fPIC testrom.s -o testrom
```

Note the use of the `-fPIC` option to generate position independent code. (i.e. No relocations)

7.1.3 Settings Breakdown

All the possible configurations for the simulation execution are accessible through the main settings window inside μARM. To get show the settings window simply click on the *Machine Config Button* (the first button in the main bar with the Screwdriver and Wrench icon).

The configurations are split into three categories: *General*, *Devices* and *Accessibility*.

General Settings

The first tab provides configurations for the main simulator features and is divided into three categories:

- *Hardware* - These settings modify core hardware structure
  - Default Clock Rate (MHz): This value represents the simulated clock speed of the processor, it does not influence the simulation speed but the external devices will take this value into account to establish the cycles count needed to complete a task.
  - RAM Size (Frames): This value represents the actual size of installed RAM expressed in ram Frames, a label next to the input field will show the corresponding value in Bytes.
  - TLB Size (Entries): This value represents the number of Entries (i.e. the size) of the Translation Lookaside Buffer (see Sec. 4.2.3).
- *Firmware and Software* - These settings specify options regarding the Firmware binaries and the Core/Bootstrap binaries.
  - Execution ROM: The ARM ELF file specified by this setting will be loaded into the Execution ROM and will be executed at boot time, the default value is `/usr/include/uarm/BIOS`.
  - Core file: The ARM ELF file specified by this setting will be loaded into RAM starting from address `0x0000.8000` and will be automatically loaded from default BIOS after the startup routines (see Sec. 6.1).
- *Debugging Support* - These settings modify the behavior of some debugging tools included into the simulator

  - Enable constant refresh: If enabled, the contents of the debugger components will be automatically updated during the machine running time, otherwise the contents will be updated each time the execution is paused.
  - GUI Refresh Rate: If constant refresh is enabled, this value represents the number of processor cycles that will elapse between each GUI update.
  - Pause execution on Exception: If enabled, the execution will pause each time an exception is raised, i.e. each time the **pc** value is below `0x0000.0020`.
  - Symbol Table ASID: Sets the default ASID for associated with loaded Symbol Table in the Breakpoint Window.
  - External Symbol Table: If enabled, the ARM ELF file specified will be the one from which the Symbol Table loaded in Breakpoint and Data Structures windows is generated.

Devices Settings

Settings for external devices are accessible from this tab. The main drop-down menu is used to select the type of device to configure, then the main contents will change accordingly, showing the settings for each one of the eight devices of the selected type.

Each single device has a relative “Enable” checkbox that will affect its presence into the simulation.

Device specific configurations are:

- *Tapes*
  - Device File: A tape file created with `uarm-mkdev` tool to be loaded into the tape drive
- *Disks*
  - Device File: A hard disk file created with `uarm-mkdev` tool to be attached to the system
- *Terminals*
  - Device File: A text file onto which the simulator will dump all the terminal contents
- *Printers*
  - Device File: A text file onto which the printer will write its output
- *Network Interfaces*

- Device File: The path of the vde_switch that will be created and will be used from the network interface
- Fixed MAC address: If enabled, the network interface MAC address will be fixed to a user defined value.
- MAC address: If fixed MAC address is enabled, this will be the MAC address of the network interface.

Accessibility Settings

Setting related to accessibility features are accessible through this tab. At the time of writing the only available option is *Enable Increased Accessibility*. When enabled and after a program restart, this setting will change the graphical interface in favor of a set of much accessible widgets.

As an example, the main processor viewer window, that is implemented with a matrix of text labels, will be replaced by an array of text fields, each one displaying a full "window" of registers (see Sec. 2.1).

Global Settings

Global interface settings are stored in `/etc/default/uarm` in Unix-based systems. This file is used to choose the preferred font face and size to be used by μARM for displaying registers and Bus contents in its graphical user interface.

The default `Monospace` font face should be available in some form in any system, but these settings could be some times necessary to fix display problems. The default size leads to a correct sizing of the emulator windows most of the times, but if there are sizing issues (windows/fonts too big/small), this is the value that needs to be adjusted.

Configuration File Fields

All configurations are stored in a *Json* file, the default location for this file is `$HOME/.config/uarm/machine.uarm.cfg`, where `$HOME` is the current user's home directory. Using the `-c` command line option (see Sec. 7.1.5) a different configuration file can be specified. If the configuration file is missing (both the default file or a user selected one), a new file with default settings values is created in its place.

A configuration file can be directly modified outside of μARM with any text editor and will be recognized by the machine if all fields are valid. Any missing field will be initialized by the program with its default value.

The structure of a simple configuration file is shown below:

`machine.uarm.cfg` File Format

```
{
    "accessible-mode": false,
    "clock-rate": 1,
    "core-file": "kernel",
    "devices": {
        "disk0": {
            "enabled": false,
            "file": "disks/disk0.uarm"
        },
        "eth0": {
            "address": "ba:98:76:54:32:10",
            "enabled": false,
            "file": "tap0"
        },
        "terminal0": {
            "enabled": true,
            "file": "term0.uarm"
        }
    },
    "execution-rom": "/usr/include/uarm/BIOS",
    "num-ram-frames": 10240,
    "pause-on-exc": false,
    "pause-on-tlb": false,
    "refresh-on-pause": false,
    "refresh-rate": 600,
    "symbol-table": {
        "asid": 0,
        "external-stab": false,
        "file": ""
    },
    "tlb-size": 16
}
```

Each field of the configuration file refers to a setting that can be reached through the Main Settings Window:

Field	Type	Settings' Tab	Setting Name
`accessible-mode`	`bool`	Accessibility	Enable Increased Accessibility
`clock-rate`	`int`	General	Default Clock Rate
`core-file`	`string`	General	Core file

`devices:`
`{dev}[0-7]:`
`enabled`
	`bool`	Devices	Enable

`devices:`
`{dev}[0-7]:`
`file`
	`string`	Devices	Device File

`devices:`
`eth[0-7]:`
`address`
	`string`	Devices	Fixed MAC address & MAC address

`execution-rom`	`string`	General	Execution ROM
`num-ram-frames`	`int`	General	RAM Size
`pause-on-exc`	`bool`	General	Pause execution on Exception
`pause-on-tlb`	`bool`	Breakpoints[1]	Stop on TLB change
`refresh-on-pause`	`bool`	General	Enable constant refresh
`refresh-rate`	`int`	General	GUI Refresh Rate

`symbol-table:`
`asid`
	`int`	General	Symbol Table ASID

`symbol-table:`
`external-stab`
	`bool`	General	External Symbol Table (checkbox)

`symbol-table:`
`file`
	`string`	General	External Symbol Table (line edit)

`tlb-size`	`int`	General	TLB Size

Where {`dev`} is one of the following:

{`dev`}	Device
`disk`	Disks
`eth`	Network
`printer`	Printers
`tape`	Tapes
`terminal`	Terminals

7.1.4 Terminal Windows

Once the machine is powered on, a dedicated window is accessible for each enabled terminal through the *Terminals* sub menu.

[1] This setting can be found in the Breakpoints window, see Section 7.3.2.

Each Terminal Window shows the terminal contents, both input and output, and has a control bar at the bottom. Through the control bar the user can simulate a hardware failure to test special system features and see a hidden status bar through the *Show Status* button.

The terminal status shows the value of the terminal transmitter (`TX`) and receiver (`RX`) sub-devices status words along with the **TOD** value relative to the last update of the status registers.

7.1.5 Command Line Options

A small set of additional options are available by running the program from a command line. The full launch command synopsis is:

```
uarm [-c <config.conf>] [-e [-x]] [--dumpExec <dumpfile>]
```

Where:

- `-c <config.cfg>` is used to load/create a user defined configuration file (see Sec. 7.1.3).
- `-e` enables *Autorun*: the machine is powered on and started with the program start.
- `-x` enables *Run and Exit*: if Autorun is enabled, exit the program when the machine reaches the `HALT` instruction (see Sec. 6.2).
- `--dumpExec <dumpfile>` enables *Execution Dump*: the file `<dumpfile>` is filled with every binary instruction executed by the machine at run time and the relative decompiled assembly code. The file `<dumpfile>` is overwritten if already existent.

7.1.6 Binary Formats

The cross-compiler and cross-linker generate code in the *Executable and Linking Format* (ELF). While the ELF format allows for efficient compilation and execution by an OS it is also quite complex. Using the ELF format would therefore un-necessarily complicate the student OS development process since there are no program loaders or support libraries available until one writes them.

Hence μARM converts on the fly the executable core files in a simpler format, based on the predecessor to the ELF format: *a.out*. User mode programs are converted in a.out format as well as they get loaded on tape by the tool `uarm-mkdev`.

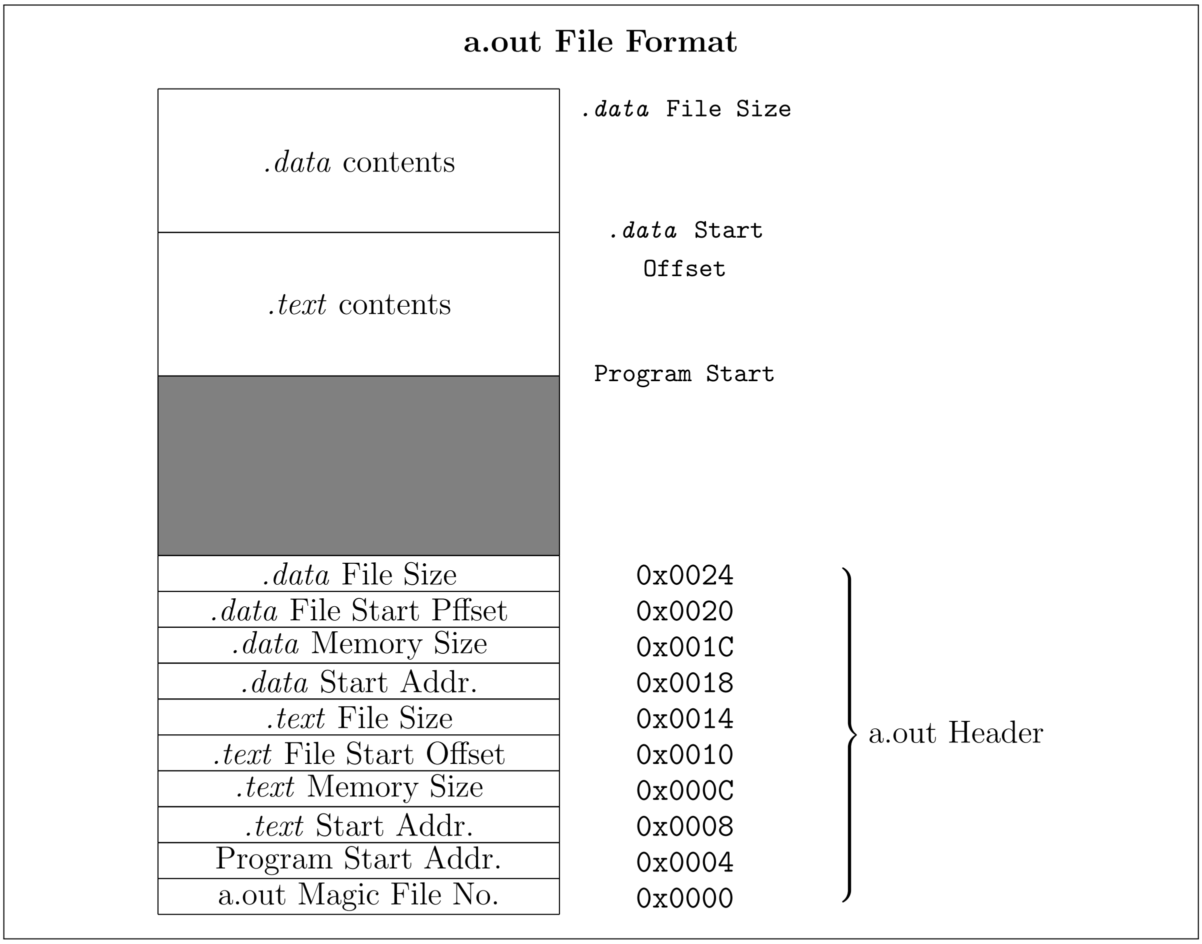
The a.out Format

A program, once compiled and linked may be logically split into two areas or sections. The primary areas are:

- *.text*: This area contains all the compiled code for the executable program. All of the programs functions are placed contiguously one after another in the order the functions are presented to the linker.
- *.data*: This area contains all the global and static variables and data structures. It in turn is logically divided into two sub-sections:
  - *.data*: Those global and static variables and data structures that have a defined (i.e. initialized) value at program start time.
  - *.bss*: Those global and static variables and data structures that do NOT have a defined (i.e. initialized) value at program start time.

Local, i.e. automatic, variables are allocated/deallocated on/from the programs stack, while dynamic variables are allocated from the programs *heap*. A heap, like a stack, is an OS allocated segment of a programs (virtual) address space. Unlike stack management, which is dealt with automatically by the code produced by the compiler, heap management is performed by the OS. The compiler can produce stack management code since the number and size of each functions local variables are known at compile time. Since the number and size of dynamic variables cannot be known until run-time, heap management falls to the OS. Heap management can safely be ignored by OS authors who are not supporting dynamic variables, i.e. there are no `malloc`-type SYSCALLs.

a.out Header		
Field Name	**File Offset**	**Field Description**
a.out Magic File No.	`0x0000`	Special identifier used for file type recognition.
Program Start Addr.	`0x0004`	Address (virtual) from which program execution should begin. Typically this is `0x0000.8074` fore kernel and `0x8000.0094` for user mode programs.
.text Start Addr.	`0x0008`	Address (virtual) for the start of the .text area. It is fixed to `0x0000.0000` for core and `0x8000.0000` for user mode programs.
.text Memory Size	`0x000C`	Size of the memory space occupied by the .text section.
.text File Start Offset	`0x0010`	Offset into a.out file where *.text* begins. Since the header is part of *.text*, this is always `0x0000.0000`
.text File Size	`0x0014`	Size of *.text* area in the a.out file. Larger than *.text* Mem. Size since its padded to the nearest 4KB block boundary.
.data Start Addr.	`0x0018`	Address (virtual) for the start of the *.data* area. The *.data* area is placed immediately after the *.text* area at the start of a 4KB block, i.e. *.text* Start Addr. + *.text* File Size.
.data Memory Size	`0x001C`	Size of the memory space occupied by the full *.data* area, including the *.bss* area.
.data File Start Offset	`0x0020`	Offset into the a.out file where *.data* begins. This should be the same as the *.text* File Size.
.data File Size	`0x0024`	Size of *.data* area in the a.out file. Different from the *.data* Memory Size since it doesnt include the *.bss* area but is padded to the nearest 4KB block boundary.



a.out File Format

Important Point: The *.data* area is given an address space immediately after the *.text* address space, aligned to the next 4KB block, insuring that *.text* and *.data* areas are completely separated. The *.bss* area immediately follows the *.data* area and is NOT aligned to a separate 4KB block.

.text and *.data* Memory Sizes are provided for sophisticated memory allocation purposes:

- The size of each U-procs PgTbl can be determined dynamically, instead of one size fits all approach.
- PTEs that represent the *.text* area can be marked as read-only, while entries that represent the *.data* area can be marked as writable.

The program loader which reads in the contents of a U-procs a.out file, needs to be aware that the *.text* and *.data* areas are contiguous and have a starting virtual address of `0x8000.0000`. The *.bss* area, while not explicitly described in the a.out file, will occupy the virtual address space immediately after the *.data* area. Zeroing out the *.bss* area will insure that all uninitialized global and static variables and data structures begin with an initial value of zero. Finally, the loader loads the **pc** with the Program Start Addr.; i.e. the contents of the second word of the a.out program header.

a.out files have padded *.text* and *.data* sections to facilitate file reading/loading. Each section is padded to a multiple of the frame size or disk and tape block size. This allows the kernel/OS to easily load the program and insure that the programs *.text* and *.data* occupy disjoint frame sets.

7.2 `uarm-mkdev` tool

While the log files for holding terminal and printer output are standard text files, and which if not present for any active printer or terminal, will be automatically created by μARM at startup time, the disk and tape cartridge files must be explicitly created beforehand. One uses the `uarm-mkdev` device creation utility to create the files that represent these persistent memory devices.

7.2.1 Creating Disk Devices

Disks in μARM are read/write sealed devices with specific performance figures. The `uarm-mkdev` utility allows one to create an empty disk only; this way an OS developer may elect any desired disk data organization.

The created disk file represents the entire disk contents, even when empty. Hence this file may be very large. It is recommended to create small disks which can be used to represent a little portion of an otherwise very large disk unit.

Disks are created via:

```
uarm-mkdev -d <diskfile.uarm> [cyl [head [sect [rpm [seekt [datas]]]]]]
```

where:

- `-d` instructs the utility to build a disk file image.
- `<diskfile.uarm>` is the name of the disk file image to be created.
- The following six optional parameters allow one to set the drives geometry: number of cylinders, heads/surfaces, and sectors, and the drives performance statistics: the disk rotation speed in rotations per minute, the average cylinder-to-cylinder seek time, and the sector data occupancy percentage.

As with real disks, differing performance statistics result in differing simulated drive performance. e.g. A faster rotation speed results in less latency delay and a smaller sector data occupancy percentage results in shorter read/write times.

The default values for all these parameters are shown when entering the `uarm-mkdev` command alone without any parameters.

7.2.2 Creating Tape Cartridges

Tape devices in μARM are read-only devices which are typically used for the fast loading of large quantities of data into the simulation without having to resort to typing the data directly into a terminal. Tapes are typically used to load user programs (Uprocs).

A tape cartridge file image will contain a properly-formatted copy of the file(s) the user wishes loaded onto it.

Tape cartridge image files are created via:

```
uarm-mkdev -t <tapefile.uarm> <file> [<file>] ... [<file>]
```

where:

- `-t` instructs the utility to build a tape cartridge file image.
- `<tapefile.uarm>` is the name of the tape cartridge file image to be created.
- The concluding space-separated list of `<file>` names are the files that will be included on the tape cartridge file image. These files, of which there must be at least one, executable ARM ELF files. Each file will be zero-padded to a multiple of the 4KB block size and sliced up using the EOB and EOF block markers. The tapes end will be marked with a EOT marker.

7.3 Debugging

Some more tools can be accessed through the main interface bar. These tools are meant to be used during run time to debug the program running on μARM.

All the contents shown by viewers gets updated each time the execution is paused or when the emulation speed is limited to low (i.e. non maximum) values. In addition to these updates, the viewers' contents can also be updated programmatically by toggling the *Enable constant refresh* setting (see Sec. 7.1.3).

7.3.1 Registers Contents View

The main window shows the contents of all processor registers, along with the relevant coprocessor registers and some memory mapped system information, see Sections 2.1, 2.2 and 3.1.4 for further explanations of each register.

On top of the processor registers matrix is shown the Execution Pipeline and the assembly translation of the currently executing operation.

This viewer gives some useful information about the current status of the machine, along with the first four arguments of each function call that the compiler usually stores in **a1**-**a4** registers.

7.3.2 Breakpoints

The *Breakpoint* button in the main bar shows the relative window. Through the breakpoint window the user can set any number of breakpoints by specifying the memory address that the machine will have to look for to pause execution.

Each time the value of the **pc** register equals any of the set breapoints, and the *Stop on Breakpoint* option is enabled, the execution is paused and all the matching breakpoints are highlighted.

Breakpoints can be automatically set at the beginning of a function by selecting the function name from the top list and clicking the *Add* button, or they can be set at any custom address by specifying the desired value by hand in the *Address* field. Breakpoints can be individually disabled by toggling the checkbox near their identifier.

ASID field is used when virtual memory is enabled, in this case the breakpoints will trigger only if the desired virtual address is reached with the specific ASID set in **CPSR.R2.EntryHy**.*ASID*.

Stop on TLB change will enable automatic execution pause each time the content of the TLB (see Sec. 4.2.3) changes.

One common strategy for debugging with breakpoints is to add one or more debugging functions with empty bodies to be placed into the suspect function code and set breakpoints on the debugging functions to check the execution flow. A more complex approach, that requires a bit of ARM assembly understanding, is to decompile the target binary using the cross toolchain (an example command is shown below) and manually setting breakpoints at interesting points inside function body.

```
arm-none-eabi-objdump -d kernel | less
```

7.3.3 Bus Inspector

The *Memory* menu gives access to the *Bus Inspector* tool. The inspector permits to investigate the contents of the System Bus, beginning with Exception Vector, up to the actual RAM memory (see Sec. 3.1 and 4).

To view the contents of a specific bus region, the bus area can be specified in two ways:

- inserting start and end addresses, or
- inserting start address and area size.

Both input methods require the beginning address of the bus region; to specify a size, the *+* button must be enabled. Addresses and sizes must be inserted in hexadecimal format.

Once the area has been selected, clicking the *Display Portion* button will extend the inspector window showing the bus contents. The bus region maximum size allowed by the Bus Inspector is 10KB, if a larger area is requested, an alert message will inform the user that it is not possible to show a bus portion bigger than 10KB.

A text field below the address bar is left for the user to note down a friendly name for the shown bus region. If the name and the region match the ones of a known object found in the symbol table, the address will be updated across machine resets (i.e. if the program is modified and built again between two runs and a global data structure changes its position in memory or size, the inspector will change the addresses accordingly).

The last component of the Bus Inspector tool window is the bus area contents viewer: the memory addresses are displayed in the leftmost column, the central column shows the raw hexadecimal contents and on the right is shown the decompiled ARM code.[2]

Unlike the other debugging tools, the number of Bus Inspector windows is not limited to one. Given the nature of this tool, it can be useful for the programmer to keep multiple inspectors open to check different bus regions simultaneously.

7.3.4 Structures Viewer

The second entry in *Memory* menu is *Structures Viewer.* This tool shows a list of all the global data structures[3] present in the loaded symbol table along with their contents.

Once the desired object has been chosen from the top table, its memory contents are shown in the lower section of the window in the same fashion as Bus Inspector (see Sec. 7.3.3).

The *Show in Bus Inspector* button opens a new Bus Inspector window preloaded with address, size and name of the active object, this way the Bus Inspector will keep the address and size updated across machine resets (see Sec. 7.3.3).

7.3.5 TLB Viewer

The *TLB* button in the main bar leads to the TLB Viewer tool window. This window shows the contents of the Translation Lookaside Buffer (see Sec. 4.2.3).

Each entry in the TLB is displayed in one row of the central table, split in two 32-bit words: EntryHi and EntryLo. The right panel shows the breakdown of both words into the composing fields as described in Section 4.2.1.

When the execution is paused, the contents of the TLB can be modified through the central table. This feature can be useful to debug Paging Algorithms and TLB Exception Handlers.

[2] The Bus Inspector tries to decompile each memory word, without knowing if it is code or data, it is up to the programmer to decide if the decompiled assembly is meaningful or garbage.

[3] All global variables are present in the symbol table, thus not only variables of some `struct` type are listed, but all variables, including base type ones, that aren't local to some function.