Company Name
North West Building,
1000th Floor,
Binary Estate, Guindy,
TPI - 611110, India

Super Simple Document

Mohamed Thalib H

Version 1.0, 10-02-2013

customer/partner
Beautiful

Table of Contents

Table 1. Revision History

No.	Date	Author	Description
1.0	01/10/2016	Very Good	Simple Test
2.0	05/10/2016	Very Good	Simple Test

1. Finding out the Entry point

Listing 1. Vector Table

```
.section ".text.boot"
.globl _start
_start:
    b       reset           ①
    b       arm_undefined       ②
    b       arm_syscall     ③
    b       arm_prefetch_abort  ④
    b       arm_data_abort      ⑤
    b       arm_reserved        ⑥
    b       arm_irq         ⑦
    b       arm_fiq ⑧
```

① Reset Vector

② Undefined instruction vector

③ svc vector

④ arm data abort

⑤ not used

⑥ interrupt

⑦ fast interrupt

2. System Control Arm cortex A53

The system registers control and provide status information for the functions implemented in the processor. The main functions of the system registers are:

- Overall system control and configuration.
- Memory Management Unit (MMU) configuration and management.
- Cache configuration and management.
- System performance monitoring.
- GIC configuration and management.

2.1. AARCH64 and AARCH32

Historically, the ARM instruction set has included a space for «coprocessors». Originally, these were external blocks of logic which were connected to the core via a dedicated coprocessor interface. More recently, this support for external coprocessors has been dropped and the instruction set space is used

for extension instructions. One specific use of it has been to provide for system configuration and control operations via the notional «coprocessor 15». You won't find anything like this in In AARCH32 the system control is done by Coprocessor p15 instruction , But in AARCH64 the coprocessor is removed

2.1.1. Why is Coprocessor support removed in AARCH64?

Even in AArch32 the coprocessors are only conceptual, with the opcode space being used to access system registers (CP15), debug registers (CP14), and FP / NEON (CP10 & CP11). This is contrast to older versions of the architecture where there needed to be a way to access actual coprocessors in the system.

In AArch64, the conceptual coprocessors have been removed and we now only access system registers using MSR/MRS instructions with the target register's name.

Take for example CPACR_EL1, which is architecturally mapped to the AArch32 register CPACR. To access CPACR in AArch32 we need to use coprocessor instructions:

```
MRC p15,0,<Rt>,c1,c0,2 ; Read CPACR into Rt
MCR p15,0,<Rt>,c1,c0,2 ; Write Rt to CPACR
```

Whereas to access CPACR_EL1 in AArch64 we just use the register name:

```
MRS <Xt>, CPACR_EL1 ; Read CPACR_EL1 into Xt
MSR CPACR_EL1, <Xt> ; Write Xt to CPACR_EL1
```

Hardly anyone (including those of us in ARM) knows what MRC p15,0,<Rt>,c1,c0,2 does, without relying on comments in the code or checking the docs. The new System Register mnemonics just make life easier, because you can guess what they do. However "under the hood" they are still basically co-processor ops.

2.1.2. AARCH32

```
It does not refer to other processors in a multi-processor cluster. You can find the
actual definitions of the CP15 registers in the core-specific Technical Reference Manual
(TRM) .
```

MRC{cond} coproc, opcode1, Rd, CRn, CRm{, opcode2}
MRC Instruction

Move to ARM register from coprocessor. Depending on the coprocessor, you might be able to specify various operations in addition.
Syntax

```
MRC{cond} coproc, opcode1, Rd, CRn, CRm{, opcode2}
MRC2 coproc, opcode1, Rd, CRn, CRm{, opcode2}
```

cond	is an optional condition code (see Conditional execution).
coproc	is the name of the coprocessor the instruction isfor. The standard name is pn, where n is an integer in the range 0-15.
opcode1	is a coprocessor-specific opcode.
Rd	is the ARM destination register. If Rd is r15, only the flags field is affected.
CRn, CRm	are coprocessor registers.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.

opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.
opcode2	is an optional coprocessor-specific opcode.

2.1.3. Entry point in c code

```
.L__copy_loop:
    cmp     r1, r2
    ldrlt   r3, [r0], #4
    strlt   r3, [r1], #4
    blt     .L__copy_loop

.L__do_bss:
    /* clear out the bss */
    ldr     r0, =__bss_start
    ldr     r1, =_end
    mov     r2, #0
.L__bss_loop:
    cmp     r0, r1
    strlt   r2, [r0], #4
    blt     .L__bss_loop

#ifdef ARM_CPU_CORTEX_A8
    DSB
    ISB
#endif

    bl      kmain
    b       .
```

```
/* called from crt0.S */
void kmain(void) __NO_RETURN __EXTERNALLY_VISIBLE;
void kmain(void)
{
    // get us into some sort of thread context
    thread_init_early();

    // early arch stuff
    arch_early_init();

    // do any super early platform initialization
    platform_early_init();

    // do any super early target initialization
    target_early_init();

    dprintf(INFO, "welcome to lk\n\n");
    bs_set_timestamp(BS_BL_START);

    // deal with any static constructors
    dprintf(SPEW, "calling constructors\n");
    call_constructors();

    // bring up the kernel heap
    dprintf(SPEW, "initializing heap\n");
    heap_init();

    __stack_chk_guard_setup();

    // initialize the threading system
    dprintf(SPEW, "initializing threads\n");
    thread_init();

    // initialize the dpc system
    dprintf(SPEW, "initializing dpc\n");
    dpc_init();

    // initialize kernel timers
    dprintf(SPEW, "initializing timers\n");
    timer_init();

#if (!ENABLE_NANDWRITE)
    // create a thread to complete system initialization
    dprintf(SPEW, "creating bootstrap completion thread\n");
    thread_resume(thread_create("bootstrap2", &bootstrap2, NULL, DEFAULT_PRIORITY,
DEFAULT_STACK_SIZE));
```

```
    // enable interrupts
    exit_critical_section();

    // become the idle thread
    thread_become_idle();
#else
        bootstrap_nandwrite();
#endif
}
```