Boot Process: ARM Classic & i386

Overview

The way that instructions execute when a computer system powers on is collectively referred to as its *boot process*. In some systems, this will begin by executing instructions from ROM storage (which may itself be hardwired or firmware). In other systems, data may be executed directly from an addressable storage device (such as flash storage). Generally, after the system is initialized according to the specification standard (if any), a **bootloader** is loaded and executed. The bootloader usually is used to load and start the operating system. In systems with subsequence bootloaders, the different subsections may be referred to as the ***first stage bootloader***, second stage, and so on.

Boot Process for i386

The x86 processor architecture depends on a Basic Input-Output System (BIOS) that is loaded from ROM and initializes the basic system. This includes loading drivers for and initializing basic input and output devices (e.g., keyboards, screens, and storage). Each x86 system also has some designated non-volatile RAM (NVRAM) where system settings are stored. Once devices have been initialized, the BIOS will use the NVRAM to identify the startup device (which is usually set in the ***BIOS settings***). For typical storage devices, the BIOS will then load the first data block – typically referred to as the device's ***boot block*** (first 512 bytes) – into RAM, then begin executing its instructions. While the NVRAM storage locations for settings can vary greatly between devices, the existence of the BIOS – along with its functions and how it loads the boot block – is completely defined and standardized. Additionally, the mapping of memory and ports is also standardized.

A bootloader can be written that will run on the vast majority of x86 systems. To signify that a device is bootable, its boot block should end in the bytes **`0x55`** and **`0xAA`** (little-endian **`0xAA55`**). The x86 BIOS system not only performs necessary device initialization, it also loads the boot block directly into RAM; in other words, first stage bootloaders for x86 processors need not worry about initialization or loading the bootloader itself into RAM.

The 32-bit x86 line starts in ***Real Mode***, a mode in which programs have complete control over hardware. However, Real Mode has some limitations; principally, it can only access 1 MiB of memory, and some of that memory is designated for the BIOS and dedicated devices; As a result, programs are generally limited to the Conventional Memory sections of RAM.

The x86 Memory Map			
Range	***Size***	***Description***	
0x000000 – 0x0003FF	1 KiB	Real Mode Interrupt Vector Table	640 KiB (Plenty!)
0x000400 – 0x0004FF	256 B	BIOS data area (partially standardized)	
0x000500 – 0x007BFF	29.75 KiB	Conventional (**Usable**) Memory	
0x007C00 – 0x007DFF	512 B	Boot Sector (Loaded by BIOS)	
0x007E00 – 0x07FFFF	480.5 KiB	Conventional (**Usable**) Memory	
0x080000 – 0x09FFFF	128 KiB	Extended BIOS Data Area	
0x0A0000 – 0x0BFFFF	128 KiB	Video Display Memory (hardware mapped)	HW Mem. Mapped
0x0C0000 – 0x0C7FFF	32 KiB	Video BIOS	
0x0C8000 – 0x0EFFFF	160 KiB	BIOS Expansions	
0x0F0000 – 0x0FFFFF	64 KiB	Motherboard BIOS	

510.25 KiB should be enough for everyone! Right?!

Boot Process for ARM Classic

While x86 platforms are standardized, ARM is not; ARM processors do not have a standardized firmware boot sequence, nor standardized address mapping for devices. As a result at least its first stage of each bootloader must be developed explicitly for the given hardware configuration. For example, one processor may map a flash storage device to memory address 0x0, while another may map a ROM or firmware to 0x0; and yet another may map a device register to the same address. However, there are a few standards to which ARM Classic (through ARMv6) processors adhere:

1) The processor trap table is at either address **`0x0`** or **`0xffff0000`** (hardware configured); and
2) Execution begins at the start of the trap table (which begins with reset entry).

Trap Table

The trap table contains a list of instructions that will be executed whenever a particular type of trap occurs. As a result, it typically holds a set of "jump" or "loader program counter" instructions (to jump to a handler routine). Here, you can see the trap table entries, along with an example:

Offset	Trap
0x00	System Reset
0x04	Undef. Instruction
0x08	Supervisor Call
0x0c	Prefetch Abort
0x10	Data Abort
0x14	(Reserved)
0x18	Interrupt
0x1c	Fast Interrupt

Interrupt Table from Default "Akita" ROM

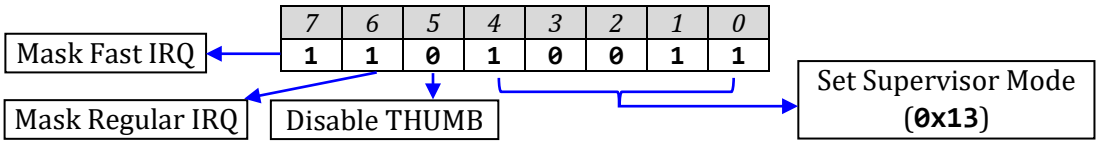
```
0x00:  b    0x34                 # Jump to RESET (0x34)
0x04:  ldr pc, [pc, #156]; 0xa8 # Load addr from 0xa8
0x08:  ldr pc, [pc, #156]; 0xac # Loads addr from 0xac
0x0c:  ldr pc, [pc, #156]; 0xb0 # Loads addr from 0xb0
0x10:  ldr pc, [pc, #156]; 0xb4 # Loads addr from 0xb0
0x14:  nop (mov r0, r0)         # No operation
0x18:  ldr pc, [pc, #152]; 0xb8 # Loads addr from 0xb4
0x1c:  ldr pc, [pc, #152]; 0xbc # Loads addr from 0xb8
```

Initializing the ARM Processor

Typically, on boot, an ARM bootloader will need to…

1) Set supervisor mode (**`0x13`**)
2) Turn off THUMB instructions (as this in not the standard mode)
3) Mask (disable) regular and "fast" IRQs (so that the initial boot is not interrupted)

Some or all of these may be set at boot, but there is no guarantee; as such, to ensure proper boot, the bootloader will need to set the Current Program Status Register (**CPSR**) to hold these values in the lowest byte / lowest 8 bits:



To initialize the CPU, we can read the CSPR, clear the lower bits, set the bits, and then rewrite it:

```
mrs r0, CSPR       // "Move to Register from System Coproc." (MRS): CSPR → r0
bic r0, #0xff      // "Bit clear" (BIC): Clear lower byte (8 bits)
orr r0, r0, #0xd3  // "Or, Register" (ORR): Add b11010011 (0xd3) to r0
mrs CSPR_c, r0     // "Move to System Coproc. from Register" (MSR): r0 → CSPR_c
```

When writing, we can elect to write only the control byte (lower byte) using the ***CSPR_c*** symbol.