Lecture 2 - Bootloader

- **ARM architecture overview**
- **Bootloader overview**
- **Case study – UBoot**
  - PART1: Startup
  - PART2: Monitoring mode
  - PART3: OS Boot
- **Available bootloaders for Linux**

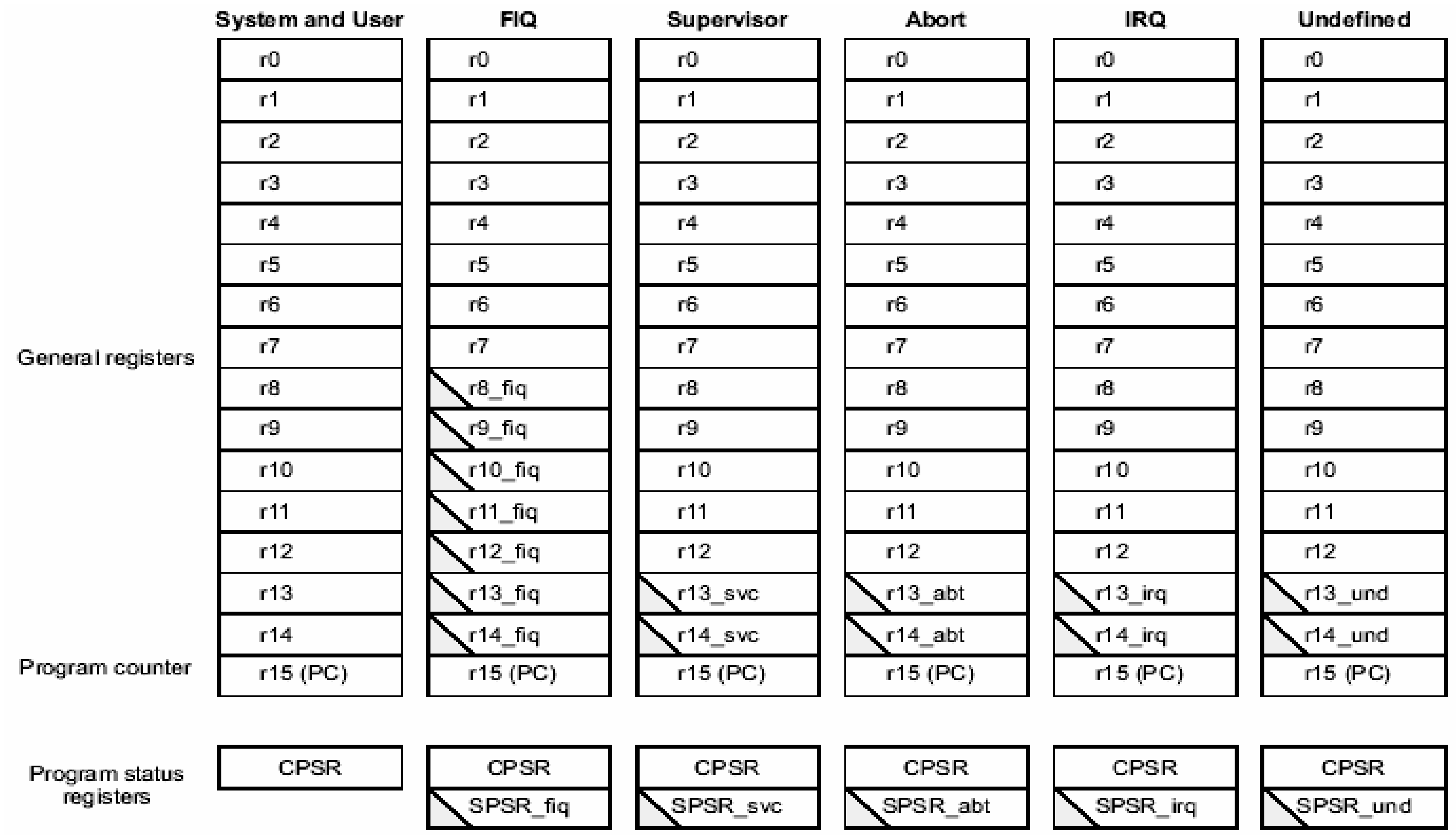
ARM Architecture Overview

- **ARM Operating Modes**
  - **User mode:** *a normal program execution state*
  - **FIQ – Fast Interrupt:** *for fast interrupt handling*
  - **IRQ – Normal Interrupt:** *for general purpose interrupt handling*
  - **Supervisor mode (SVC):** *a protected mode for operating system*
  - **Abort mode:** *when a data or instruction pre-fetch is aborted*
  - **Undefined mode:** *when an undefined instruction is executed*
  - **System mode:** *a privileged user mode for the operating system*

- **ARM Processor States**
  - **Arm state:** all instructions are 32bits long, word-aligned.
  - **Thumb state:** all instructions are 16 bits wide, half-word aligned.
  - **Jazelle state:** all instructions are 8 bits wide for Java Bytecode (for v5TEJ only)

ARM Architecture Overview

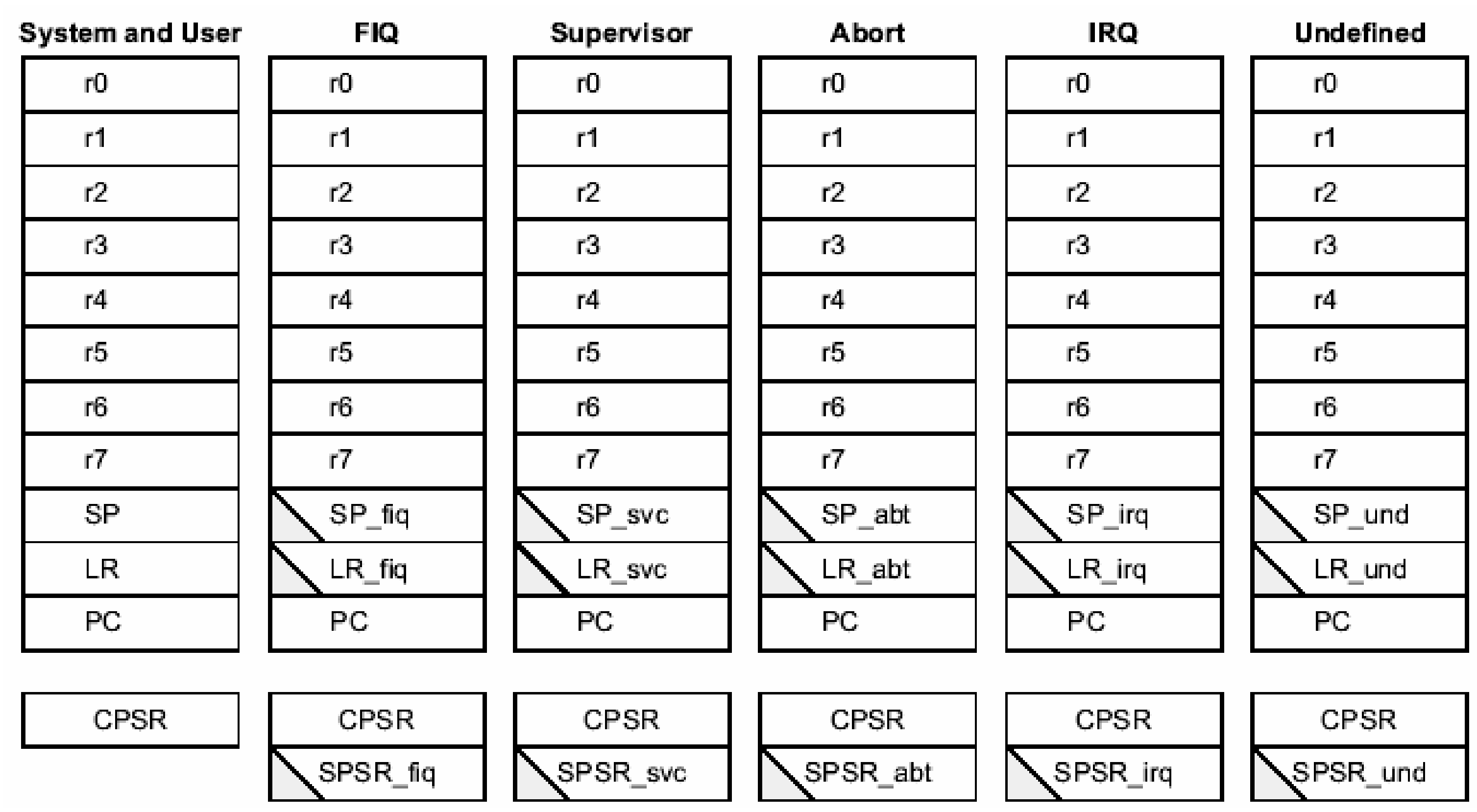
- **ARM-State Registers**



	System and User	FIQ	Supervisor	Abort	IRQ	Undefined
	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
General registers	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
Program counter	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)
Program status registers	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

ARM Architecture Overview

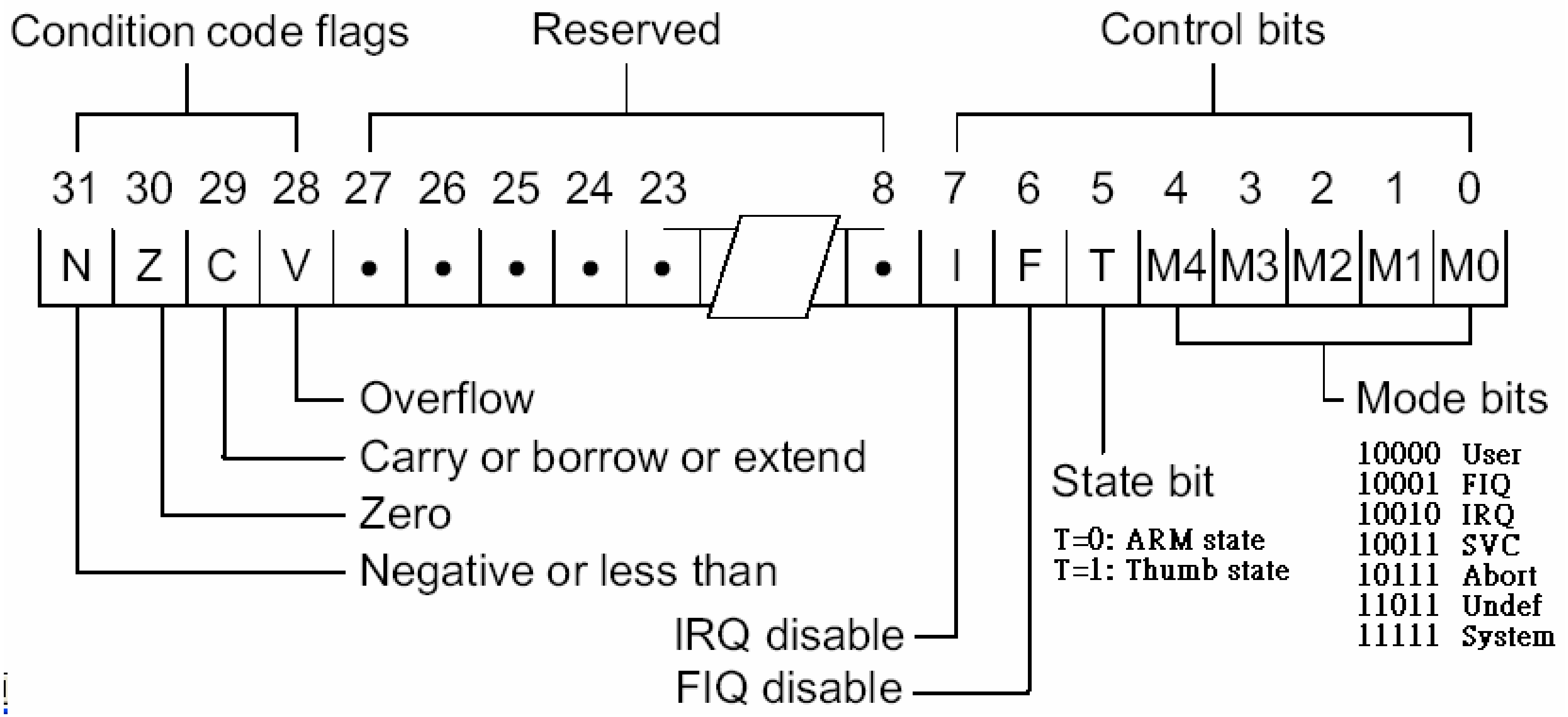
- **Thumb-State Registers**



System and User	FIQ	Supervisor	Abort	IRQ	Undefined
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
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

ARM Architecture Overview

- **Program Status Register (PSR)**
  - **CPSR**: Current Program Status Register
  - **SPSR**: Saved Program Status Register



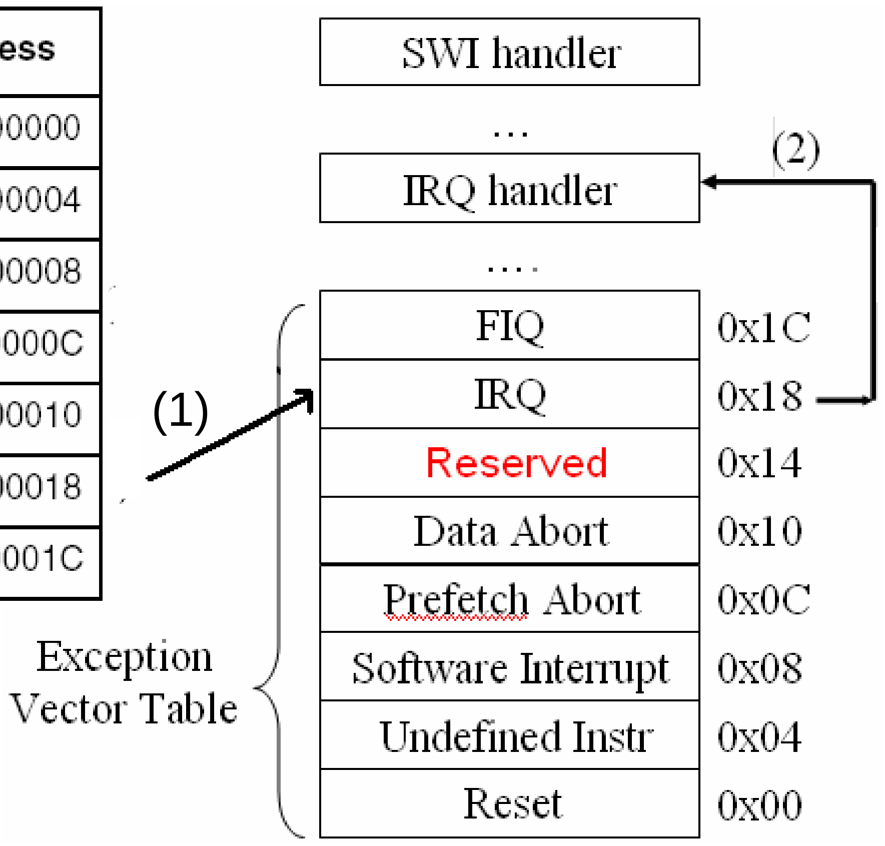
ARM Architecture Overview

- **ARM Exception Types**
  - **Reset**
    - *Hardware reset: when the processor reset pin is asserted*
    - *Software reset: by branching to the reset vector (0x0000)*
  - **Undefined instruction**
    - *the processor cannot recognize the currently execution instruction*
  - **Software Interrupt (SWI)**
    - *By s/w instruction, to allow a program running in User mode to request privileged operations that are in Supervisor mode*
  - **Prefetch Abort**
    - *Fetch an instruction from an illegal address*
  - **Data Abort**
    - *A data transfer instr. try to load or store data at an illegal address*
  - **IRQ:** ***The processor IRQ pin is asserted and the I bit in CPSR is clear***
  - **FIQ:** ***The processor FIQ pin is asserted and the F bit in CPSR is clear***

ARM Architecture Overview

- **ARM Exception Vectors**

Exception Type	Mode	Address
Reset	Supervisor	0x00000000
Undefined Instructions	Undefined	0x00000004
Software Interrupts (SWI)	Supervisor	0x00000008
Prefetch Abort	Abort	0x0000000C
Data Abort	Abort	0x00000010
IRQ (Normal Interrupt)	IRQ	0x00000018
FIQ (Fast interrupt)	FIQ	0x0000001C



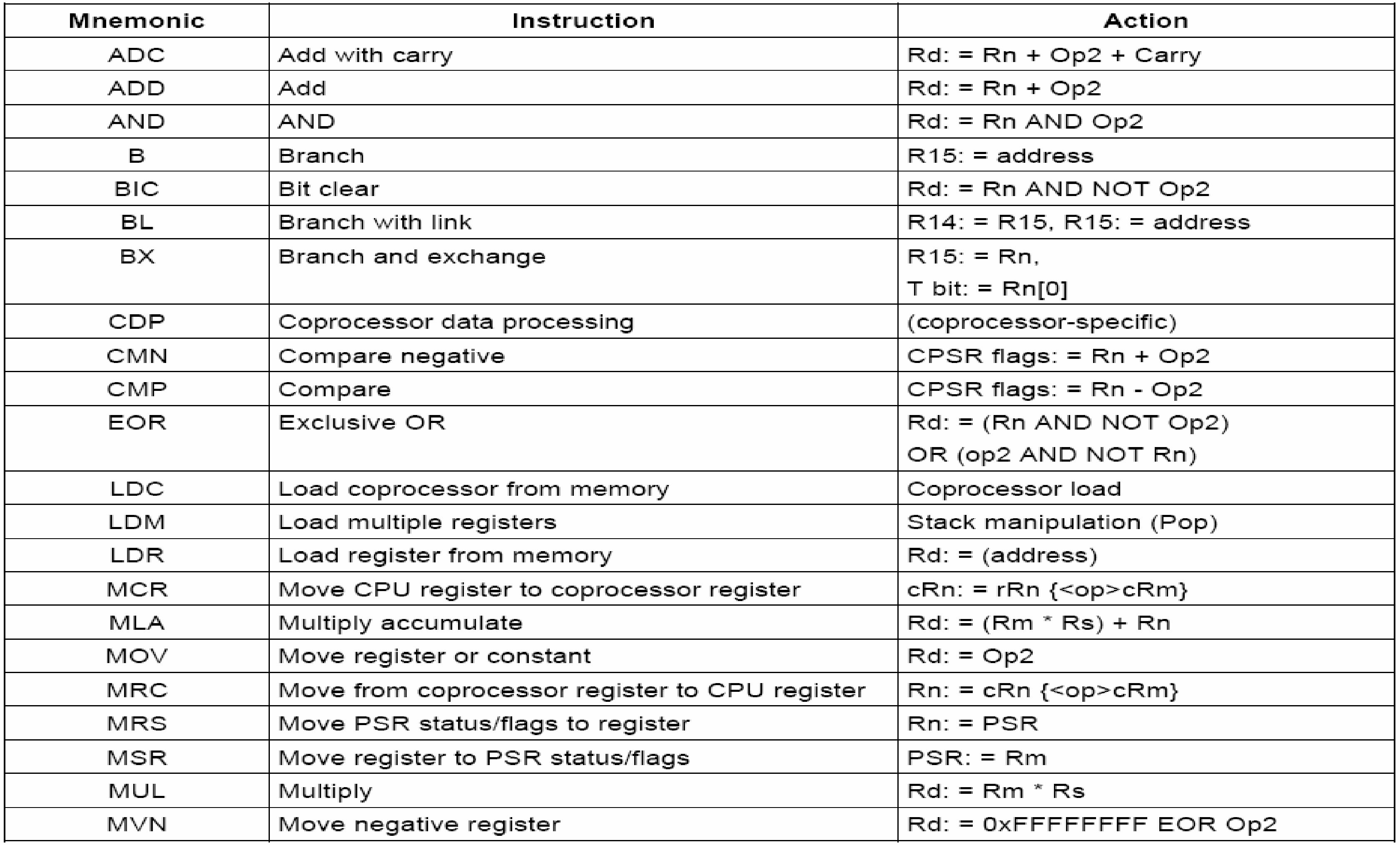
Memory

ARM Architecture Overview

- **ARM Instruction Set**
  - **Data processing instruction (x24)**
    - Arithmetic instruction (x8)
    - Bit-wise Logic instruction (x4)
    - Movement instruction (x2)
    - Comparison instruction (x4)
    - Shift and rotator instruction (x6)
  - **Load and store instructions (x4)**
    - Single-register load/store instructions (x2)
    - Multiple-register load/store instructions (x2)
  - **Branch instruction (x2)**
  - **Status Register Transfer instruction (x2)**
  - **Exception Generating instructions (x2)**

ARM Architecture Overview

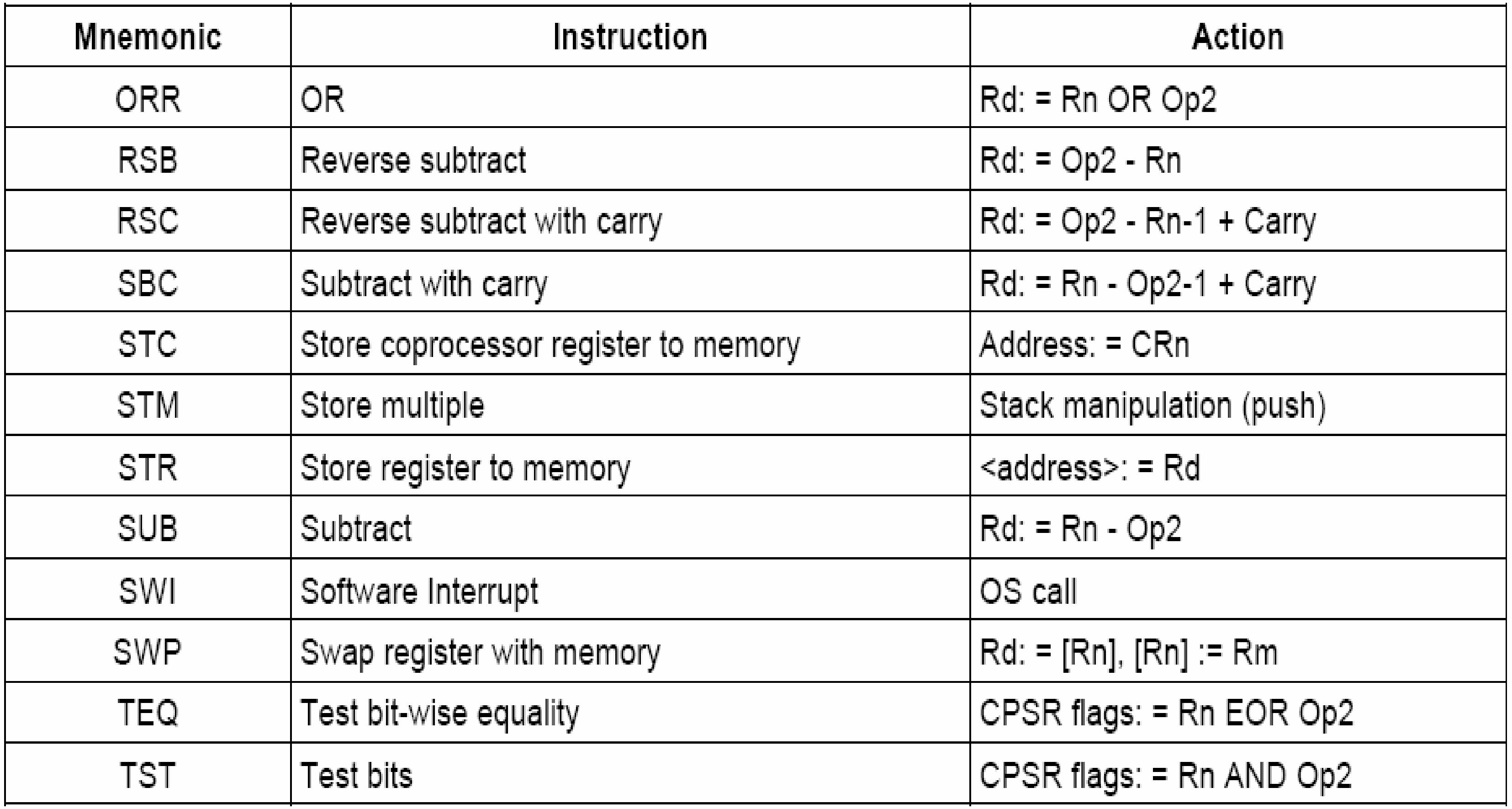
- **ARM Instruction Summary**



Mnemonic	Instruction	Action
ADC	Add with carry	Rd: = Rn + Op2 + Carry
ADD	Add	Rd: = Rn + Op2
AND	AND	Rd: = Rn AND Op2
B	Branch	R15: = address
BIC	Bit clear	Rd: = Rn AND NOT Op2
BL	Branch with link	R14: = R15, R15: = address
BX	Branch and exchange	R15: = Rn, T bit: = Rn[0]
CDP	Coprocessor data processing	(coprocessor-specific)
CMN	Compare negative	CPSR flags: = Rn + Op2
CMP	Compare	CPSR flags: = Rn - Op2
EOR	Exclusive OR	Rd: = (Rn AND NOT Op2) OR (op2 AND NOT Rn)
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd: = (address)
MCR	Move CPU register to coprocessor register	cRn: = rRn {<op>cRm}
MLA	Multiply accumulate	Rd: = (Rm * Rs) + Rn
MOV	Move register or constant	Rd: = Op2
MRC	Move from coprocessor register to CPU register	Rn: = cRn {<op>cRm}
MRS	Move PSR status/flags to register	Rn: = PSR
MSR	Move register to PSR status/flags	PSR: = Rm
MUL	Multiply	Rd: = Rm * Rs
MVN	Move negative register	Rd: = 0xFFFFFFFF EOR Op2

ARM Architecture Overview

- **ARM Instruction Summary**

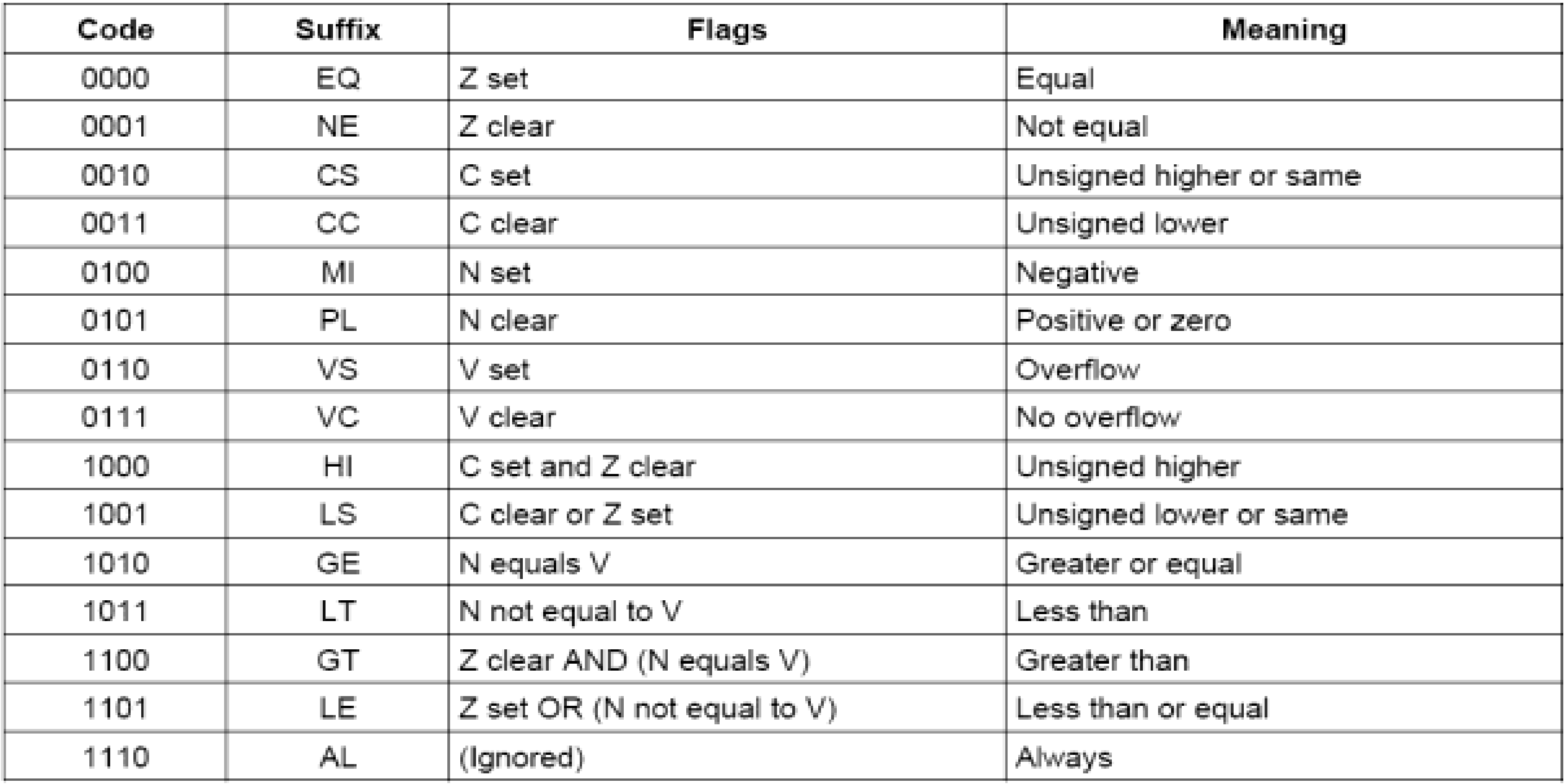


Mnemonic	Instruction	Action
ORR	OR	Rd: = Rn OR Op2
RSB	Reverse subtract	Rd: = Op2 - Rn
RSC	Reverse subtract with carry	Rd: = Op2 - Rn-1 + Carry
SBC	Subtract with carry	Rd: = Rn - Op2-1 + Carry
STC	Store coprocessor register to memory	Address: = CRn
STM	Store multiple	Stack manipulation (push)
STR	Store register to memory	<address>: = Rd
SUB	Subtract	Rd: = Rn - Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd: = [Rn], [Rn] := Rm
TEQ	Test bit-wise equality	CPSR flags: = Rn EOR Op2
TST	Test bits	CPSR flags: = Rn AND Op2

ARM Architecture Overview

- **ARM Instruction Condition Code**

  In ARM, all instructions can be conditionally executed according to the state of the CPSR condition codes



Code	Suffix	Flags	Meaning
0000	EQ	Z set	Equal
0001	NE	Z clear	Not equal
0010	CS	C set	Unsigned higher or same
0011	CC	C clear	Unsigned lower
0100	MI	N set	Negative
0101	PL	N clear	Positive or zero
0110	VS	V set	Overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N equals V	Greater or equal
1011	LT	N not equal to V	Less than
1100	GT	Z clear AND (N equals V)	Greater than
1101	LE	Z set OR (N not equal to V)	Less than or equal
1110	AL	(Ignored)	Always

Bootloader Overview

- **Bootstrapping**
  - When system is powered on, the first piece of code (called boot-strapping) initializes basic hardware and then bring up OS kernel.
  - It can be included in a bootloader, or in OS directly.

- **Reference documents that you need**
  - The data sheet and user's manual of the target platform
  - Processor's Architectural Reference Manual
  - Processor's developer's guide
  - The user guides for Processor's Assembly, Compiler and Linker.
  - Boot requirement for the target OS

Bootloader Overview

- **Bootloader Selection**
  - **The Processor architectures and platforms that it supports**
  - **The OS that it supports**
  - **Debugging abilities, Stability and Portability**
  - **Functionalities:**
    - simple bootup, or with monitoring capability and lots of drivers support.
  - **The medium they are installed in**
    - usually ROM, EEPROM or Flash
  - **The storage device from which the kernel image is downloaded**
    - host PC, local Flash, disk, CF, etc.
  - **The device and protocol by which the kernel image is transmitted**
    - Serial port, USB, ethernet, IDE, etc.
  - **The 'host-side' software used to transmit the kernel image**
    - if it is coming from an external device/machine.
  - **'sleep' reset support**
    - Not to do a normal reboot if this was a sleep wakeup.

Case Study - UBoot

- **Das UBoot features**
  - **UBoot stands for "Universal Bootloader"**
  - **Open source bootloader, under GPL license**
    - Source code available at http://sourceforge.net/projects/u-boot
  - **Based on PPCBoot and ARMBoot projects**
  - **Support many CPU architectures and platform types**
    - ARM, PPC, MIPS, x86, and their variance
  - **Support many OS**
    - Linux, NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS
  - **Support booting image from TFTP, IDE/SCSI HDD, Flash, …**
  - **Support lots of device drivers and monitor commands**
  - **Supports user configuration using Environment Variables**
    - which can be made persistent by saving to Flash memory.
  - **Support tool to compress (by gzip, bzip2) and pack images**

Make U-Boot

- **Check the target boards supported by U-Boot**

vi /u-boot-1.1.2/makefile

```
                :
smdk2410_config  :          unconfig
        @./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
                :
```

- **Config your target board** – *smdk2410 as an example*

```
[root@test u-boot-1.1.2]# make smdk2410_config
Configuring for smdk2410 board ….
```

- **Make u-boot**

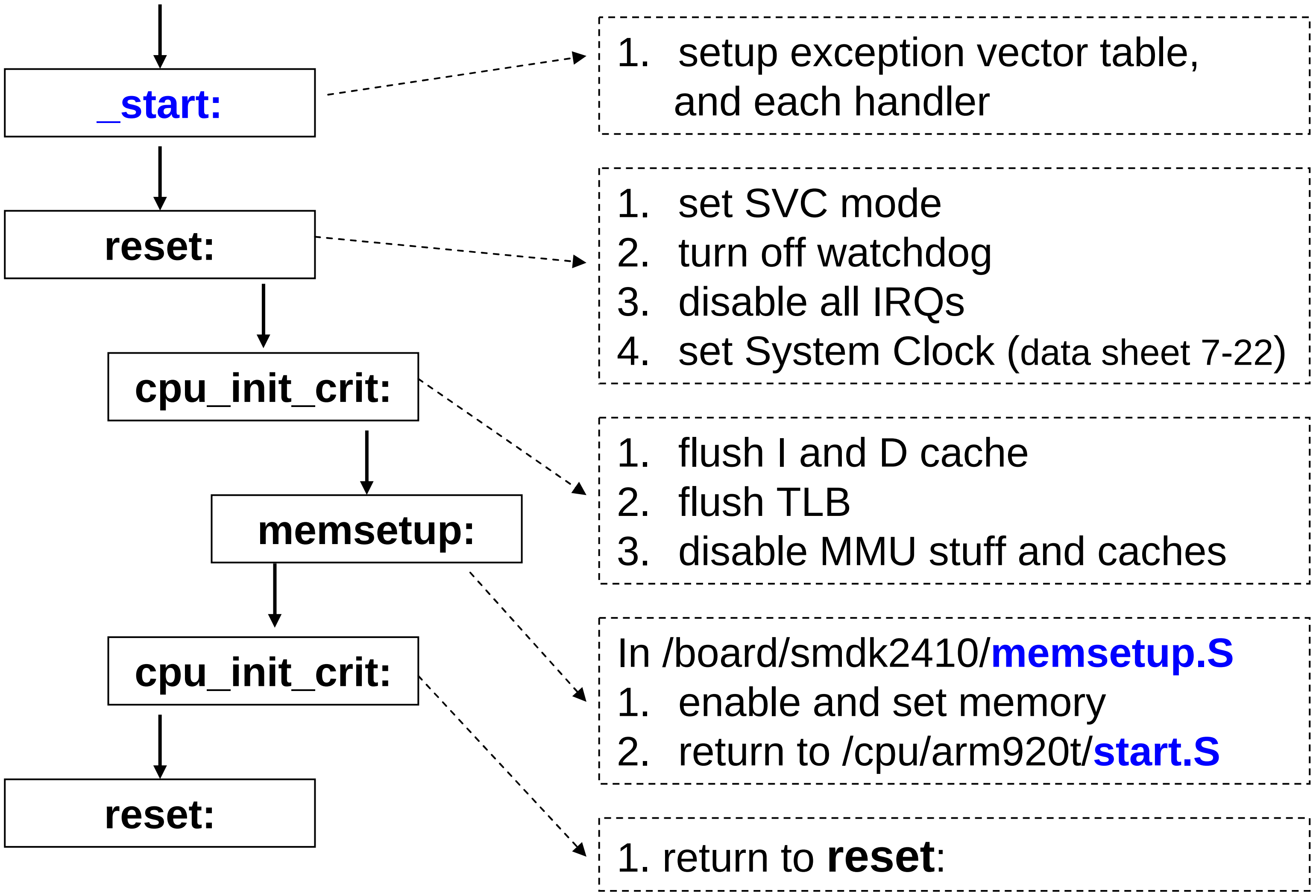
```
[root@test u-boot-1.1.2]# make
                              :
              -Map u-boot.map –o u-boot
arm-linux-objcopy --gap-fill=0xff –O srec u-boot u-boot.srec
arm-linux-objcopy --gap-fill=0xff –O binary u-boot u-boot.bin
```

ELF format

Motorola S-Record

plain binary

UBoot Startup



1. setup exception vector table, and each handler

1. set SVC mode
2. turn off watchdog
3. disable all IRQs
4. set System Clock (data sheet 7-22)

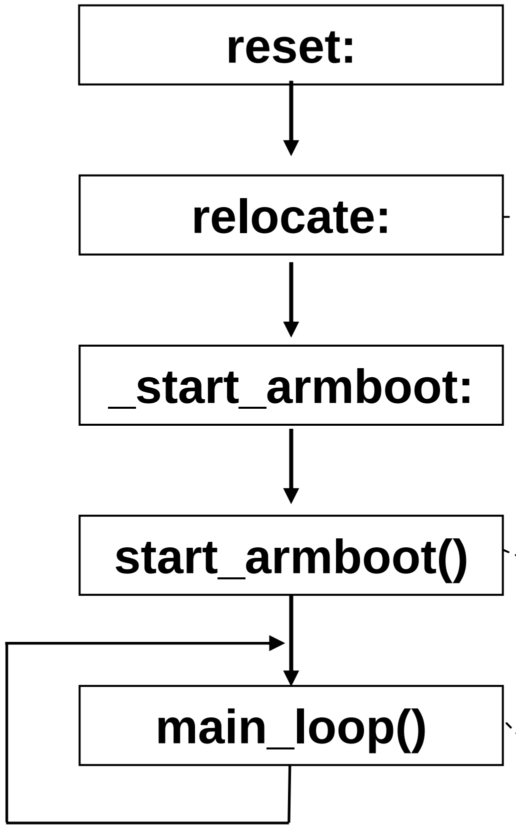
1. flush I and D cache
2. flush TLB
3. disable MMU stuff and caches

In /board/smdk2410/**memsetup.S**
1. enable and set memory
2. return to /cpu/arm920t/**start.S**

1. return to **reset**:

Part- I:

UBoot Startup



1. Setup C environment and run on RAM
  - copy U-Boot code/ initialized data to RAM
    from: _start: 0x0, Flash base
    to: _TEXT_BASE, 0x33F80000, Entry on RAM
    size: _armboot_start ~ bss_start
        (0x33F80000) (0x33F9-8558)
  - setup stack pointer
    sp = _TEXT_BASE – glb data – heap
        – irq_stack – fiq_stack – abrt_stack
  - clear BSS (_bss_start ~ _bss_end)

In lib_arm/**board.c**

1. Flash initialization
2. Init environment
3. Set IP addr
4. Set MAC addr
5. Devices initialize
6. Console initialize
7. Enable interrupts
8. Enable Ethernet

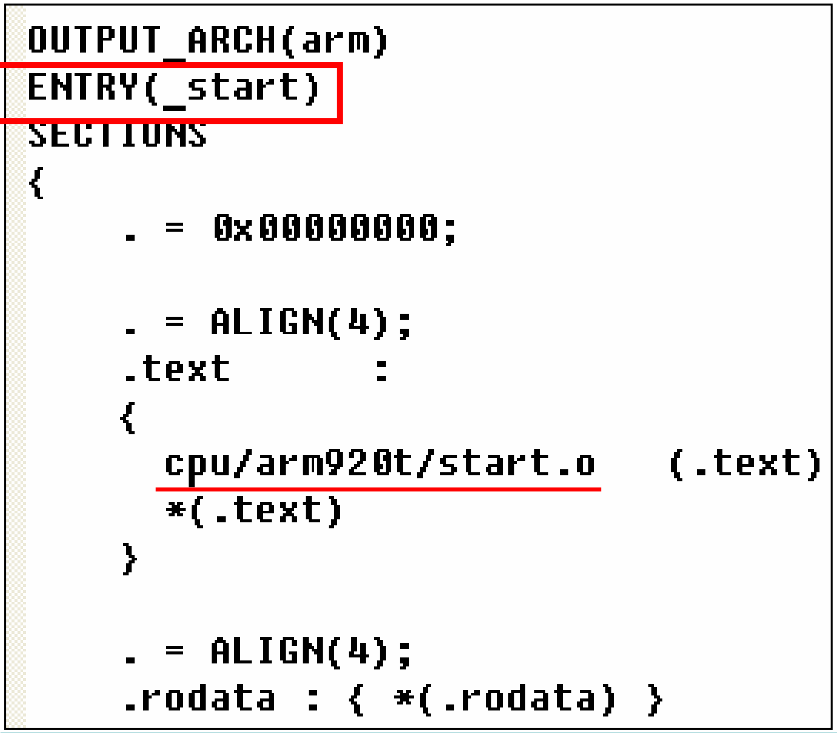
In C code

In common/**main.c**

Part- I:

Linker Descriptor

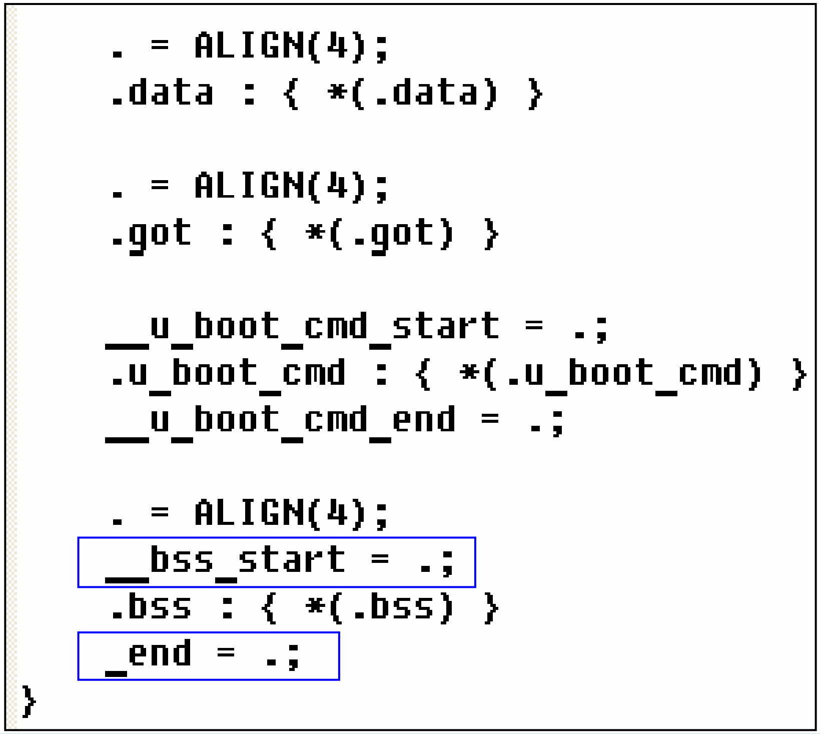
U-boot-1.1.2
/board/smdk2410/**u-boot.lds**



```
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
        . = 0x00000000;

        . = ALIGN(4);
        .text      :
        {
          cpu/arm920t/start.o    (.text)
          *(.text)
        }

        . = ALIGN(4);
        .rodata : { *(.rodata) }
```



```
        . = ALIGN(4);
        .data : { *(.data) }

        . = ALIGN(4);
        .got : { *(.got) }

        __u_boot_cmd_start = .;
        .u_boot_cmd : { *(.u_boot_cmd) }
        __u_boot_cmd_end = .;

        . = ALIGN(4);
        __bss_start = .;
        .bss : { *(.bss) }
        _end = .;
}
```

```
[root@test u-boot-1.1.2]# make
          :
arm-linux-ld –Bstatic –Tuboot-1.1.2/board/smdk2410/u-boot.lds –Ttext 0x33F80000 …
          :
```

U-boot-1.1.2/board/smdk2410/**config.mk**

TEXT_BASE = 0x33F8000 ← **UBoot entry point address**

Part- I:

UBoot trace- Startup (1/7)

U-boot-1.1.2
/cpu/arm920t/**start.S**

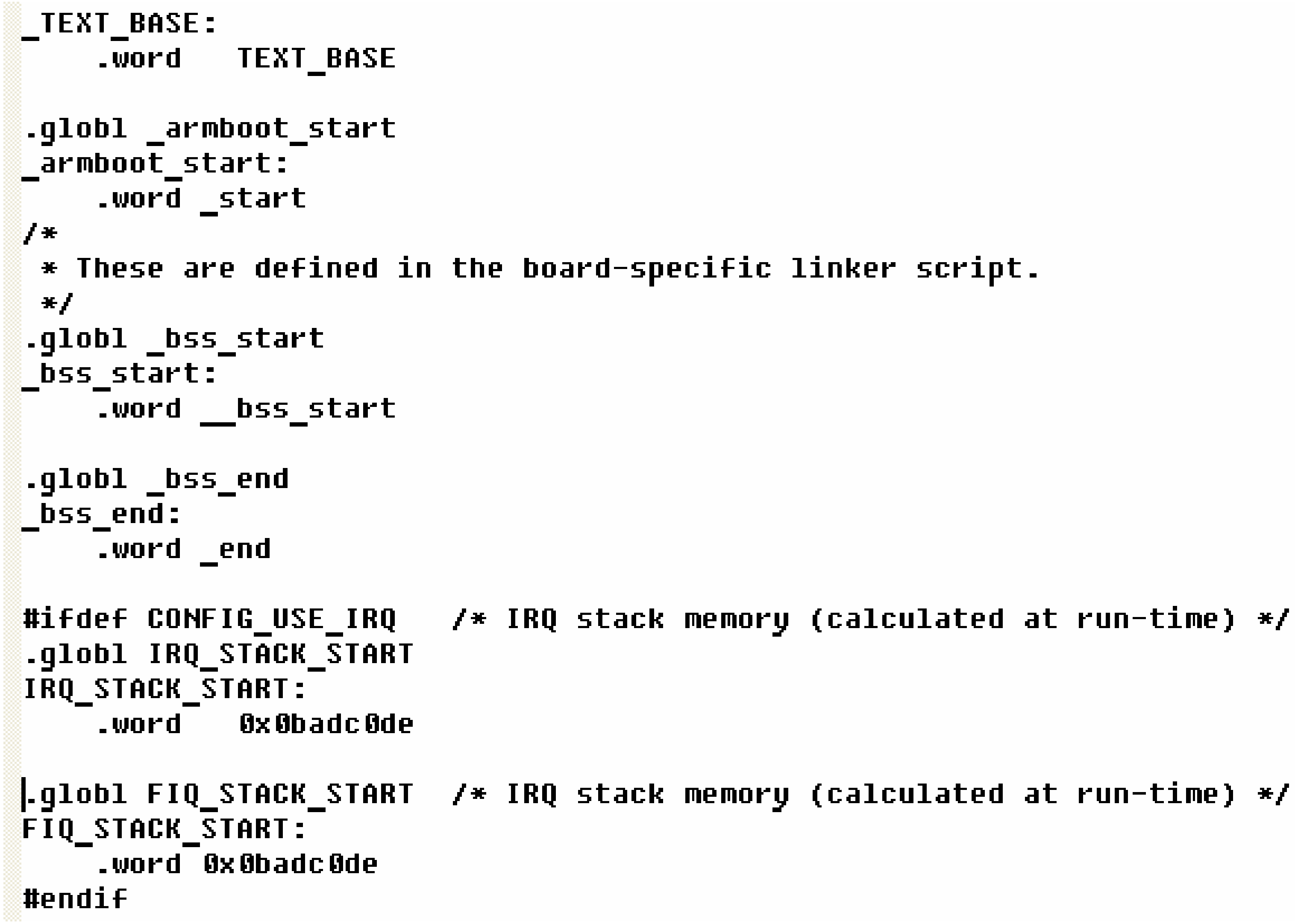
```
/*
 *************************************************************
 *
 * Jump vector table as in table 3.1 in [1]
 *
 *************************************************************
 */
.globl _start
_start: b       reset
    ldr pc, _undefined_instruction
    ldr pc, _software_interrupt
    ldr pc, _prefetch_abort
    ldr pc, _data_abort
    ldr pc, _not_used
    ldr pc, _irq
    ldr pc, _fiq

_undefined_instruction: .word undefined_instruction
_software_interrupt:    .word software_interrupt
_prefetch_abort:        .word prefetch_abort
_data_abort:            .word data_abort
_not_used:              .word not_used
_irq:                   .word irq
_fiq:                   .word fiq
```

Part- I:

UBoot trace- Startup (2/7)

U-boot-1.1.2
/cpu/arm920t/start.S



```
_TEXT_BASE:
    .word   TEXT_BASE

.globl _armboot_start
_armboot_start:
    .word _start
/*
 * These are defined in the board-specific linker script.
 */
.globl _bss_start
_bss_start:
    .word __bss_start

.globl _bss_end
_bss_end:
    .word _end

#ifdef CONFIG_USE_IRQ   /* IRQ stack memory (calculated at run-time) */
.globl IRQ_STACK_START
IRQ_STACK_START:
    .word   0x0badc0de

.globl FIQ_STACK_START  /* IRQ stack memory (calculated at run-time) */
FIQ_STACK_START:
    .word 0x0badc0de
#endif
```

Part- I:
UBoot trace- Startup (3/7)

U-boot-1.1.2
/cpu/arm920t/**start.S**

```
reset:
        /* set the cpu to SVC32 mode */
        mrs r0,cpsr
        bic r0,r0,#0x1f
        orr r0,r0,#0xd3
        msr cpsr,r0

/* turn off the watchdog */
# define pWTCON         0x53000000
# define INTMSK         0x4A000008   /* Interupt-Controller base addr */
# define INTSUBMSK      0x4A00001C
# define CLKDIVN        0x4C000014   /* clock divisor register */

#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)
        ldr     r0, =pWTCON
        mov     r1, #0x0
        str     r1, [r0]
        /*
         * mask all IRQs by setting all bits in the INTMR - default
         */
        mov r1, #0xffffffff
        ldr r0, =INTMSK
        str r1, [r0]
# if defined(CONFIG_S3C2410)
        ldr r1, =0x3ff
        ldr r0, =INTSUBMSK
        str r1, [r0]
# endif
```

Part- I:

UBoot trace- Startup (4/7)

U-boot-1.1.2
/cpu/arm920t/start.S

```
    /* FCLK:HCLK:PCLK = 1:2:4     default FCLK is 120 MHz ! */
    ldr r0, =CLKDIVN
    mov r1, #3
    str r1, [r0]
#endif  /* CONFIG_S3C2400 || CONFIG_S3C2410 */
    /*
     * We do sys-critical inits only at reboot, not when booting from ram!
     */
#ifdef CONFIG_INIT_CRITICAL
    bl  cpu_init_crit
#endif
relocate:                    /* relocate U-Boot to RAM          */
    adr r0, _start           /* r0 <- current position of code  */
    ldr r1, _TEXT_BASE       /* test if we run from flash or RAM */
    cmp     r0, r1           /* don't reloc during debug         */
    beq     stack_setup

    ldr r2, _armboot_start
    ldr r3, _bss_start
    sub r2, r3, r2              /* r2 <- size of armboot         */
    add r2, r0, r2              /* r2 <- source end address      */
copy_loop:
    ldmia   r0!, {r3-r10}       /* copy from source address [r0] */
    stmia   r1!, {r3-r10}       /* copy to   target address [r1] */
    cmp r0, r2                  /* until source end addreee [r2] */
    ble copy_loop
```

Part- I:

UBoot trace- Startup (5/7)

U-boot-1.1.2
/cpu/arm920t/**start.S**

```
    /* Set up the stack */
stack_setup:
    ldr r0, _TEXT_BASE    /* upper 128 KiB: relocated uboot */
    sub r0, r0, #CFG_MALLOC_LEN                  /* malloc area */
    sub r0, r0, #CFG_GBL_DATA_SIZE               /* bdinfo      */
#ifdef CONFIG_USE_IRQ
    sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
    sub sp, r0, #12       /* leave 3 words for abort-stack  */

clear_bss:
    ldr r0, _bss_start        /* find start of bss segment  */
    ldr r1, _bss_end          /* stop here                  */
    mov     r2, #0x00000000   /* clear                      */

clbss_l:str r2, [r0]          /* clear loop...              */
    add r0, r0, #4
    cmp r0, r1
    bne clbss_l

    ldr pc, _start_armboot

_start_armboot: .word start_armboot
```

$\longrightarrow$ /lib_arm/**board.c**

Part- I:

UBoot trace- Startup (6/7)

U-boot-1.1.2
/cpu/arm920t/**start.S**

```
cpu_init_crit:
    /*
     * flush v4 I/D caches
     */
    mov r0, #0
    mcr p15, 0, r0, c7, c7, 0   /* flush v3/v4 cache */
    mcr p15, 0, r0, c8, c7, 0   /* flush v4 TLB */
    /*
     * disable MMU stuff and caches
     */
    mrc p15, 0, r0, c1, c0, 0
    bic r0, r0, #0x00002300 @ clear bits 13, 9:8 (--V- --RS)
    bic r0, r0, #0x00000087 @ clear bits 7, 2:0 (B--- -CAM)
    orr r0, r0, #0x00000002 @ set bit 2 (A) Align
    orr r0, r0, #0x00001000 @ set bit 12 (I) I-Cache
    mcr p15, 0, r0, c1, c0, 0
    /*
     * before relocating, we have to setup RAM timing
     * because memory timing is board-dependend, you will
     * find a memsetup.S in your board directory.
     */
    mov ip, lr
    bl  memsetup
    mov lr, ip

    mov pc, lr
```

$\longrightarrow$ /board/smdk2410/**memsetup.S**

Part- I:
UBoot trace- Startup (7/7)

U-boot-1.1.2
/board/smdk2410/**memsetup.S**

```
.globl memsetup
memsetup:
    /* memory control configuration */
    /* make r0 relative the current location so that it */
    /* reads SMRDATA out of FLASH rather than memory ! */
    ldr     r0, =SMRDATA
    ldr r1, _TEXT_BASE
    sub r0, r0, r1
    ldr r1, =BWSCON
    add     r2, r0, #13*4
0:
    ldr     r3, [r0], #4
    str     r3, [r1], #4
    cmp     r2, r0
    bne     0b
    /* everything is fine now */
    mov pc, lr

    .ltorg
/* the literal pools origin */
SMRDATA:
    .word (0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BW …
    .word ((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+ …
    .word ((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+ …
     ......
```

r0 as src (SMRDARA addr on Flash)

r1 as dest (BWSCON reg addr)

r2 as copy length (13 words)

13 words

```
#define BWSCON  0x48000000

/* BWSCON */
#define DW8          (0x0)
#define DW16         (0x1)
#define DW32         (0x2)
#define WAIT         (0x1<<2)
#define UBLB         (0x1<<3)

#define B1_BWSCON   (DW32)
#define B2_BWSCON   (DW16)

.....
.....
```

Part- II: Monitoring mode

UBoot Commands

➢ help / ?	- print online help	$\longrightarrow$ list all UBoot commands
…		
➢ boot	- boot default, i.e., run ‘bootcmd’	
➢ bootelf	- boot from an ELF image in memory	
➢ bootm	- boot application image from memory	
➢ bootp	- boot image via network using BootP/TFTP	Boot
➢ tftpboot	- boot image via network using TFTP protocol	
➢ go	- start application at address ‘addr’	
…		
➢ cp	- memory copy	
➢ md	- memory display	Memory
➢ mm	- memory modify	
➢ mtest	- simple RAM test	
…		
➢ erase	- erase FLASH memory	
➢ finfo	- print FLASH memory information	Flash
➢ protect	- enable or disable FLASH write protection	
➢ imls	- list all images found in flash	
…		
➢ setenv	- set environment variables	
➢ saveenv	- save environment variables (on flash)	Environment variables
➢ printenv	- print environment variables	

Part- II:

UBoot Environment Variables

- **ethaddr, ipaddr, gatewayip, netmask:** target board information
- **serverip:** tftp server IP address
- **bootfile:** filepath/name on the tftp server
- **fileaddr:** addr on the target to put the file

$\left.\right\}$ TFTP download info

- **bootargs:** kernel commands
- **bootdelay:** auto-boot delay time
- **bootcmd:** auto-boot startup command

$\left.\right\}$ auto-boot Kernel info

EX.1 **boot from network**

```
  ...
# setenv bootfile /tftpboot/multi-2.4.18.img
# setenv fileaddr 33000000
# setenv bootargs root=/dev/ram0 console=ttyS0,115200 init=linuxrc
# setenv boot tftp\; bootm 33000000 ⬅
# setenv bootcmd run boot
# saveenv
```

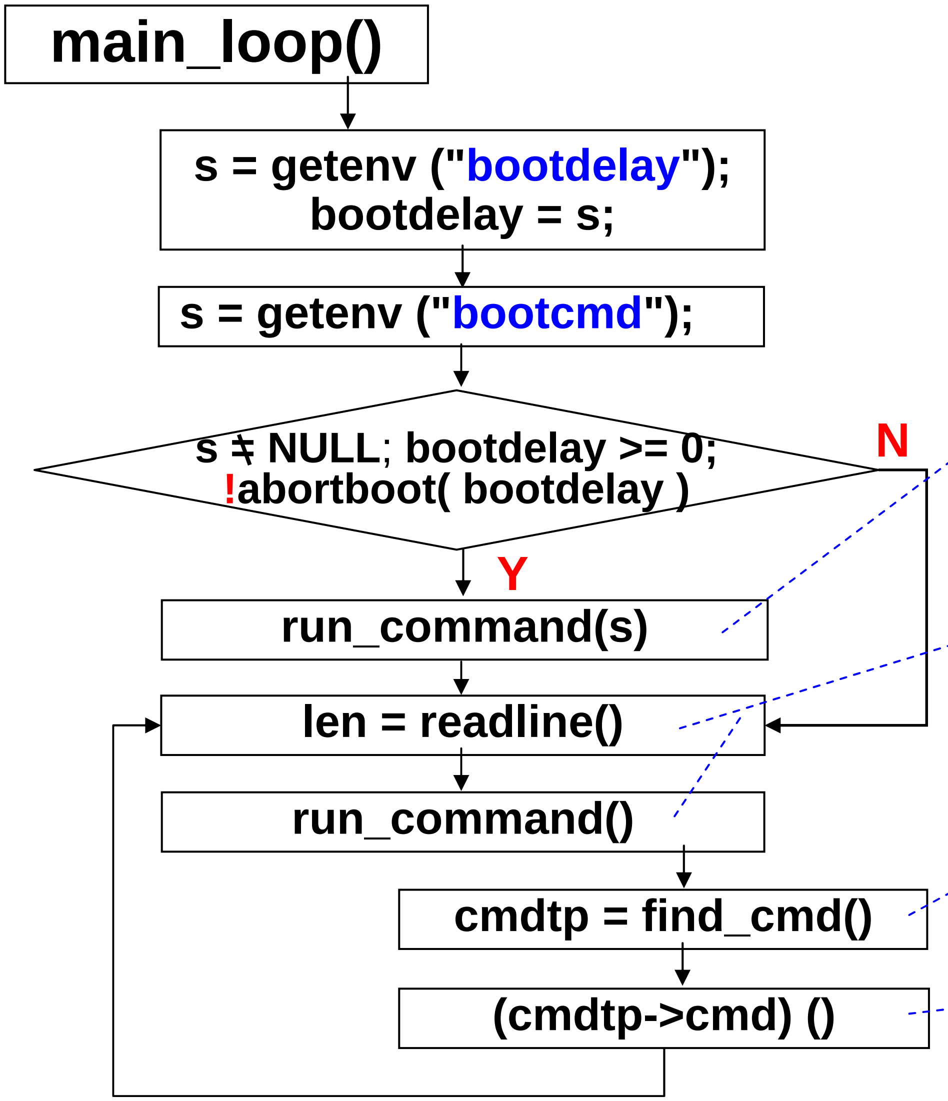
EX.2 **boot from flash**

```
# tftp                                }
# erase 20000 1effff                  } download image and put on flash
# cp.b 33000000 20000 $(filesize)     }
# setenv boot cp.b 20000 33000000 1059a4 \; bootm 33000000 ⬅
# saveenv
```

Part- II:

UBoot trace- Commands

U-boot-1.1.2
/common/**main.c**



If booting is NOT aborted by the user during the initial 'bootdelay' seconds, it will run the commands pre-set in 'bootcmd' automatically.

Otherwise it will run the commands input by user in the run-time.

In common/**command.c**
Look up commands in the command table

call function to do the command

Boot Requirements for Linux (1/6)

- To boot Linux, bootloader has to load two things into memory
  - the Linux kernel
    - also possible to configure the kernel to run on FLASH/ ROM
  - a root filesystem (RFS)
    - which usually takes the form of an initial RAMdisk (or 'initrd') which is a gzip-compress normal ext2 filesystem image
    - it can also be a ramfs, cramfs or jffs2 image.
- Most boot-loaders are capable of two operation modes
  - Bootloading
    - autonomously load its Kernel and RFS from an internal device - i.e. the normal booting procedure.
  - Downloading
    - downloads the Kernel and RFS from an external device, often under the control of that device.
    - This is the mode that is used to first install the Kernel and RFS, and for subsequent updates.

Boot Requirements for Linux (2/6)

- **What does the loader do before giving control to Linux kernel?**

1. Initialize base hardware

- **CPU speed,**
- **memory timings,**
- **detect location and size of RAM (passed to Kernel by *tagged list*)**
- **detect machine type (passed to Kernel by *r1*)**

2. Initialize devices

- **any device that needs for reading Kernel and RFS images**
- **Init UART to be Kernel console (passed to Kernel by *tagged list*.)**

3. Copy Kernel and RFS images to RAM

- **arrange a block of contiguous physical memory for the Kernel, and another contiguous one for RFS (no need to be adjacent)**
- **Copy (or download remotely) and decompress (if needed) the Kernel and RFS image to their contiguous areas**

Note. Linux Kernel kernel uses 16K of RAM below it to store *Page Tables.* The recommended placement is 32KB into RAM.

Boot Requirements for Linux (3/6)

4. Setup the Kernel Tagged List (see next slide)

5. Calling the Linux Kernel with following settings

CPU register settings

- r0 = 0,
- r1 = machine architecture number (the MACH_TYPE_XXX in kernel)
  Must match one of define in linux/arch/arm/tools/mach-types.
- r2 = physical address of tagged list in system RAM

CPU mode

- All forms of interrupts must be disabled (IRQs and FIQs)
- CPU must be in SVC mode

Cache, MMU

- MMU must be off
- D-cache must be off and no any stale data (I-Cache can be on/off)

Device

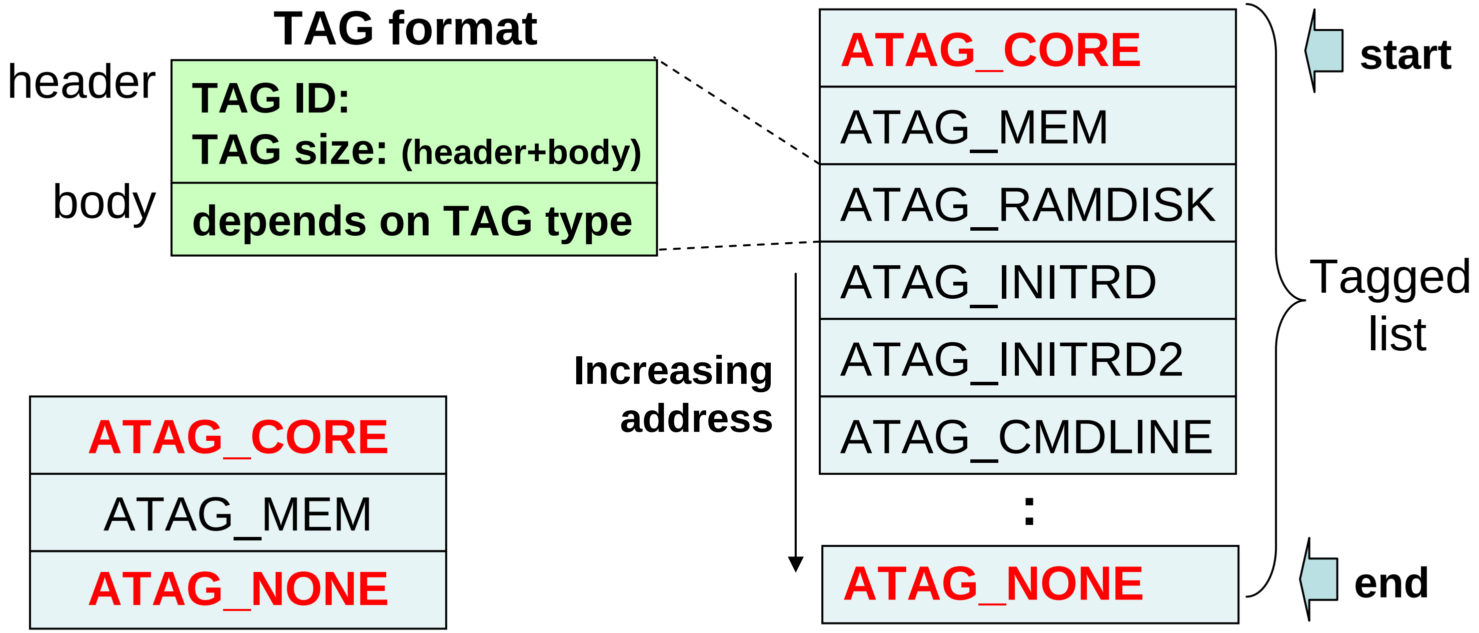
- DMA to/from devices should be quiesced.

Boot Requirements for Linux (4/6)

- How does boot-loader pass data to the Kernel?
  - CPU registers (r0, r1, r2)
  - Kernel Tagged List (located in memory)
    - A tag includes a header and a body
      - Tag header has two fields: tag ID and tag size (header+body)
      - Tag body: the structure depends on tag ID.
    - A tagged list starts with ATAG_CORE and ends with ATAG_NONE
      - ATAG_CORE may or may not be empty.
      - An empty ATAG_CORE has size = 2, ATAG_NONE has size = 0
    - The loader must pass at a minimum the size and location of the system memory (specified in ATAG_MEM).
      - A minimum Tagged list: ATAG_CORE, ATAG_MEM, ATAG_NONE.
    - There are also some other TAGs for use
      - *ATAG_RAMDISK*: how the ramdisk will be used in kernel
      - *ATAG_INITRD*: virtual addr of the compressed ramdisk image
      - *ATAG_INITRD2*: physical addr of the compressed ramdisk image
      - *ATAG_CMDLINE*

Part- III:

Boot Requirements for Linux (5/6)



A minimum tagged list **An example of a tagged list**

Note: *For some embedded systems, the tagged list is hard-coded in the Kernel. In such case, the bootloader does not need to setup it.*

Boot Requirements for Linux (6/6)

- **ARM Linux Kernel Boot Requirements**
  - **1. Setup and initialise RAM**
    - Existing boot loaders: MANDATORY
      New boot loaders: MANDATORY
  - **2. Initialise one serial port**
    - Existing boot loaders: OPTIONAL, RECOMMENDED
      New boot loaders: OPTIONAL, RECOMMENDED
  - **3. Detect the machine type**
    - Existing boot loaders: OPTIONAL
      New boot loaders: MANDATORY
  - **4. Setup the kernel tagged list**
    - Existing boot loaders: OPTIONAL, HIGHLY RECOMMENDED
      New boot loaders: MANDATORY
  - **5. Calling the kernel image (with special settings)**
    - Existing boot loaders: MANDATORY
      New boot loaders: MANDATORY

Reference **http://www.arm.linux.org.uk/developer/booting.php**

Part- III:

UBoot Tool - mkimage

- **Build mkimage and intall**

```
[root@test u-boot-1.1.2]# make tools
[root@test u-boot-1.1.2]# cp tools/mkimage  /usr/bin
```

- **Usage:**

```
mkimage –A arch –O os –T type –C compress –a loadaddr –e entrypoint  \
            –n name –d data_file[:data_file]  outputimage
```

Example 1: Kernel

```
[root@test tftpboot]# gzip -9 < Image > Image.gz
[root@test tftpboot]# mkimage –n “Kernel 2.4.18” –A arm –O linux \
                      > –T kernel –C gzip –a 30008000 –e 30008000 \
                      > -d Image.gz  vmlinux-2.4.18.img
```

Example 2: Kernel+RFS

```
[root@test tftpboot]# mkimage –n “Kernel+initrd 2.4.18” –A arm –O linux \
                      > –T multi –C gzip –a 30008000 –e 30008000 \
                      > -d Image.gz:initrd.gz  multi-2.4.18.img
```

Part- III:

UBoot trace- Linux boot (1/3)

U-boot-1.1.2
/include/**image.h**

```
#define IH_TYPE_INVALID      0    /* Invalid Image          */
#define IH_TYPE_STANDALONE   1    /* Standalone Program     */
#define IH_TYPE_KERNEL       2    /* OS Kernel Image        */
#define IH_TYPE_RAMDISK      3    /* RAMDisk Image          */
#define IH_TYPE_MULTI        4    /* Multi-File Image       */
#define IH_TYPE_FIRMWARE     5    /* Firmware Image         */
#define IH_TYPE_SCRIPT       6    /* Script file            */
#define IH_TYPE_FILESYSTEM   7    /* Filesystem Image       */

#define IH_COMP_NONE         0    /*  No   Compression      */
#define IH_COMP_GZIP         1    /* gzip  Compression      */
#define IH_COMP_BZIP2        2    /* bzip2 Compression      */

#define IH_MAGIC     0x27051956   /* Image Magic Number     */
#define IH_NMLEN             32   /* Image Name Length      */

typedef struct image_header {
    uint32_t    ih_magic;    /* Image Header Magic Number*/
    uint32_t    ih_hcrc;     /* Image Header CRC Checksum*/
    uint32_t    ih_time;     /* Image Creation Timestamp */
    uint32_t    ih_size;     /* Image Data Size          */
    uint32_t    ih_load;     /* Data  Load  Address      */
    uint32_t    ih_ep;       /* Entry Point Address      */
    uint32_t    ih_dcrc;     /* Image Data CRC Checksum  */
    uint8_t     ih_os;       /* Operating System         */
    uint8_t     ih_arch;     /* CPU architecture         */
    uint8_t     ih_type;     /* Image Type               */
    uint8_t     ih_comp;     /* Compression Type         */
    uint8_t     ih_name[IH_NMLEN];  /* Image Name        */
} image_header_t;
```

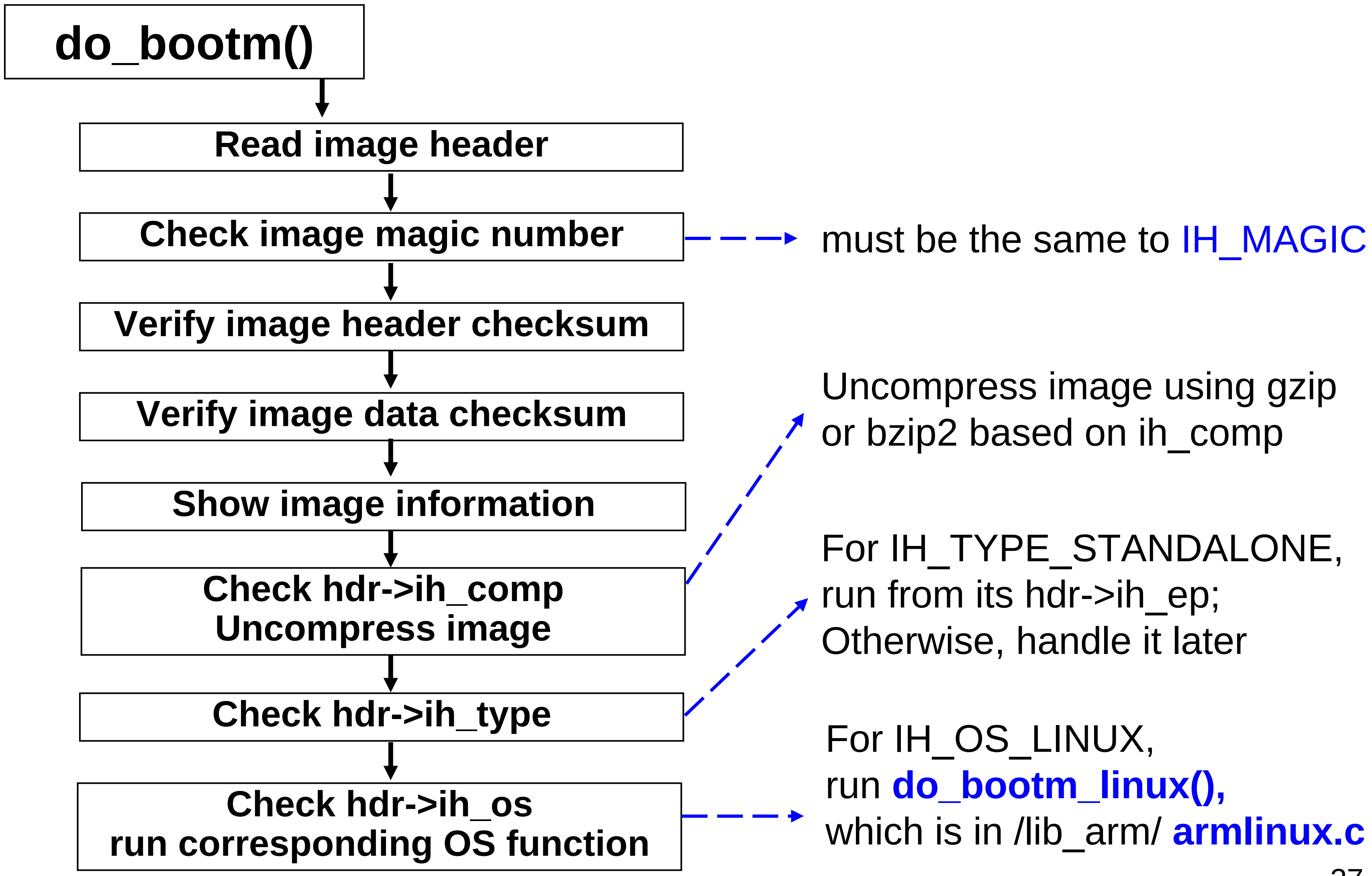
```
/*
 * Operating System Codes
 */
#define IH_OS_INVALID    0
#define IH_OS_OPENBSD    1
#define IH_OS_NETBSD     2
#define IH_OS_FREEBSD    3
#define IH_OS_4_4BSD     4
#define IH_OS_LINUX      5
#define IH_OS_SVR4       6
#define IH_OS_ESIX       7
#define IH_OS_SOLARIS    8
#define IH_OS_IRIX       9
#define IH_OS_SCO        10
#define IH_OS_DELL       11
#define IH_OS_NCR        12
#define IH_OS_LYNXOS     13
#define IH_OS_VXWORKS    14
#define IH_OS_PSOS       15
#define IH_OS_QNX        16
#define IH_OS_U_BOOT     17
#define IH_OS_RTEMS      18
#define IH_OS_ARTOS      19
#define IH_OS_UNITY      20
```

Image header

Part- III:

UBoot trace- Linux boot (2/3)

U-boot-1.1.2
/common/**cmd_bootm.c**


do_bootm()
Read image header
Check image magic number
must be the same to IH_MAGIC
Verify image header checksum
Verify image data checksum
Uncompress image using gzip or bzip2 based on ih_comp
Show image information
Check hdr->ih_comp Uncompress image
For IH_TYPE_STANDALONE, run from its hdr->ih_ep; Otherwise, handle it later
Check hdr->ih_type
For IH_OS_LINUX, run do_bootm_linux(), which is in /lib_arm/ armlinux.c
Check hdr->ih_os run corresponding OS function

Part- III:

UBoot trace- Linux boot (3/3)

U-boot-1.1.2
/lib_arm/**armlinux.c**

do_bootm_linux()

↓

***commandline = getenv ("bootargs");** → see P.xx for what's bootargs

↓

theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);

↓

Check whether initrd image exists → If initrd exists, load it as well

↓

Setup Tagged list
setup_start_tag(…)
setup_memory_tags(…)
setup_initrd_tag(…)
setup_commandline_tag(commandline)
setup_end_tag()

↓

cleanup_before_linux (); → disable interrupts
turn off I/D-cache
flush I/D-cache

↓

theKernel (0, arch_number, boot_params); → ARM uses r0~r3 for function parameters. So, this will make
r0 = 0
r1 = arch_number
r2 = boot_params
before startup Kernel

Available Bootloaders for Linux (1/3)

- Bootloaders for **Multiple** Architectures
  - **U-Boot (**http://sourceforge.net/projects/u-boot/ **)**
    - Universal Boot loader supports PowerPC, ARM, MIPS, …
  - **RedBoot (**http://www.redhat.com/embedded/technologies/redboot/)
    - By RedHat. capable of flash and network booting of Linux kernel.
    - Supports ARM, MIPS, PowerPC, Hitachi SHx, and x86.
  - **Smart Firmware** (http://www.codegen.com/SmartFirmware/)
    - By CodeGen. It is designed to be very easy and fast to port.
    - Supports M68K, PowerPC, x86, MIPS, Alpha, Sparc, ARM, …
  - **MicroMonitor** (http://www.linuxdevices.com/links/LK3381469889.html)
    - A open source embedded system boot platform. The majority of the code is CPU and platform independent and has been used with a variety of different embedded operating systems.
    - x86, Mips, PowerPC, SH, ARM, and M68K.

Available Bootloaders for Linux (2/3)

- Bootloaders for **X86** Architectures
  - **Lilo** (http://www.acm.uiuc.edu/workshops/linux_install/lilo.html)
  - **GRUB** (http://www.gnu.org/software/grub/)
  - **Etherboot** (http://etherboot.sourceforge.net/)
    - The bootROM on NIC can download code over Ethernet to run on x86.
  - **LinuxBIOS** (http://www.acl.lanl.gov/linuxbios/index.html)
    - replacing the normal BIOS with fast boot from a cold start.
  - **ROLO** (ftp://www.elinos.com/pub/elinos/rolo/)
    - By SYSGO. Capable of booting directly from ROM without BIOS.
  - **SYSLINUX** (http://syslinux.zytor.com/)
    - It is a lightweight bootloaders for floppy media.
  - **NILO** (http://nilo.sourceforge.net/)
    - it is the Network Interface Loader. NILO will boot Linux, FreeBSD, Windows 95/98/NT4 and support the Intel PXE standard.

Available Bootloaders for Linux (3/3)

- Bootloaders for **ARM**$^{(TM)}$ Architectures
  - **Blob** (http://www.lart.tudelft.nl/lartware/blob/)
    - Boot Loader OBject for SA1100's
  - **IPAQ** (http://www.handhelds.org/Compaq/iPAQH3600/OSloader.html)
    - A bootloader, which can boot Linux from Windows CE.
- Bootloaders for **PPC**$^{(TM)}$ Architectures
  - **PPCBoot** (http://ppcboot.sourceforge.net/)
    - the PPCBoot project has been superceded by U-Boot.
  - **Yaboot** (http://penguinppc.org/projects/yaboot/)
    - It works on "New" class PowerMacs (iMac and later) only.
- Bootloaders for **MIPS**$^{(TM)}$ Architectures
  - **PMON** (http://pmon.groupbsd.org)
- Bootloaders for **Sun**$^{(TM)}$ Architectures
  - **Sun3 Bootloader** (ftp://sammy.net/pub/m68k/sun3/boot/)
- Bootloaders for **Super-H**$^{(TM)}$ Architectures
  - **sh-ipl+g** (ftp://ftp.m17n.org/pub/super-h)
    - sh-ipl+g stands for “SH Initial Program Loader with gdb-stub”.

References

- ARM7TDMI Technical Reference Manual
  - http://www.eecs.umich.edu/~tnm/power/ARM7TDMIvE.pdf
- Samsung S3C2410X 32-Bit RISC Microprocessor user's manual
  - http://www.samsung.com/Products/Semiconductor/SystemLSI/MobileSolutions/MobileASSP/MobileComputing/S3C2410X/um_s3c2410s_rev12_030428.pdf
- U-Boot source code
  - http://sourceforge.net/projects/u-boot
  - Related documents:
  - PPCBoot: http://ppcboot.sourceforge.net
  - ARMBoot: http://armboot.sourceforge.net
- ARM Linux Kernel Boot Requirement
  - http://www.arm.linux.org.uk/developer/booting.php