

CE5045

Embedded System Design

Embedded Software Architecture

<https://github.com/tychen-NCU/EMBS-NCU>

Instructor: Dr. Chen, Tseng-Yi

Computer Science & Information Engineering

Outline

- How to Boot an Embedded System (ES)
 - ✓ Grub for x86 Architecture
 - ✓ U-boot for ARM Architecture
 - ✓ How to Implement a Bootloader in an Embedded System.
- Microkernel for Embedded System
 - ✓ What is Process?
 - ✓ The Process Concept in an Embedded System
 - ✓ The Practical Knowledge of Process Management

Kernel space
(Basic IPC, process
scheduler, etc)

Bootloader
(U-Boot, LILO, etc)

Hardware

Outline

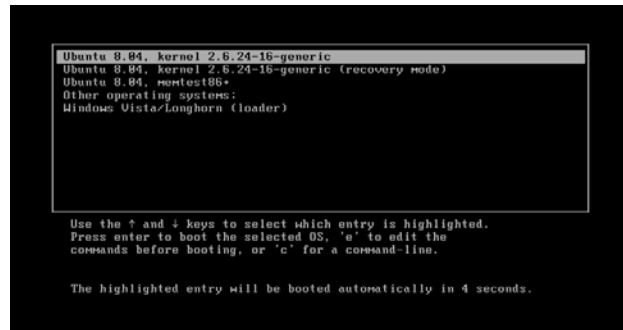
- How to Boot an Embedded System (ES)
 - ✓ Grub for x86 Architecture
 - ✓ U-boot for ARM Architecture
 - ✓ How to Implement a Bootloader in an Embedded System.
- Microkernel for Embedded System
 - ✓ What is Process?
 - ✓ The Process Concept in an Embedded System
 - ✓ The Practical Knowledge of Process Management

What is GRUB?

➤ Before Grub

✓ What is boot sector?

- A boot sector is generally the first sector of the hard drive that is accessed when the computer is turned on.
- Master boot record (MBR) or GUID partition table (GPT)



➤ So Grub ...

- ✓ Is a piece of software that exists in the MBR or GPT
- ✓ Allows a user to opt between multiple Operating systems that are installed on one or more drives existing on the computer

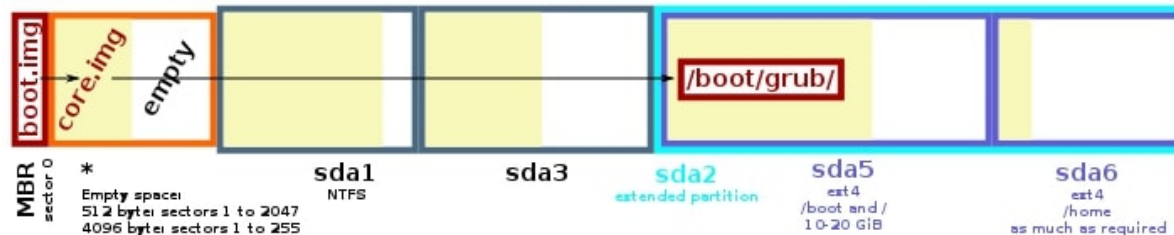
How to Boot with Grub

- Depending upon the boot sector that is either the master boot record or the GUID partition table the physical allocation of the GRUB change.

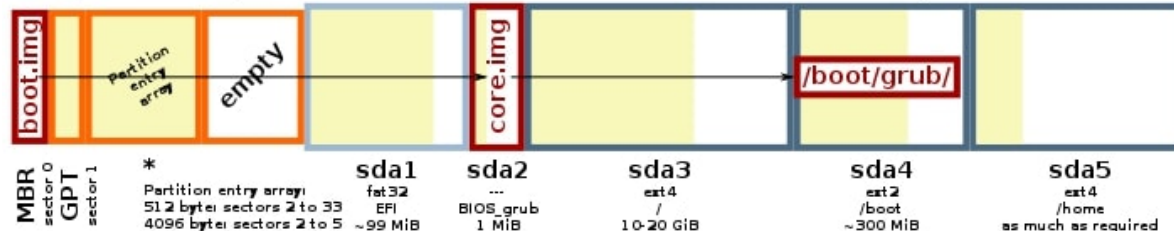
GNU GRUB 2

Locations of *boot.img*, *core.img* and the */boot/grub/* directory

Example 1: An MBR-partitioned hard disk with sector size of 512 or 4096 bytes



Example 2: A GPT-partitioned hard disk with sector size of 512 or 4096 bytes



The Image Files for Grub

➤ boot.img

- ✓ Its size is 446 bytes
- ✓ Is written to the MBR (sector 0)
- ✓ Contains utilities, Operating system, kernel files, diagnostics and various other drivers for the hardware to initiate

➤ core.img

- ✓ Is written to the empty sectors between the MBR and the first partition
- ✓ Default RAW disk image loaded on the hard disk by the manufacturer

➤ File system

- ✓ Takes care of the addressing of the data onto a physical sector. Ex: NTFS, FAT, Ex-FAT, HFS etc.

The Functionality of GRUB (I/II)

➤ Stage 1

- ✓ Find the boot.img in Master Boot Record (MBR) or GUID partition table (GPT)
- ✓ At this stage GRUB directs the next stage using an address of the kernel files of various Operating systems

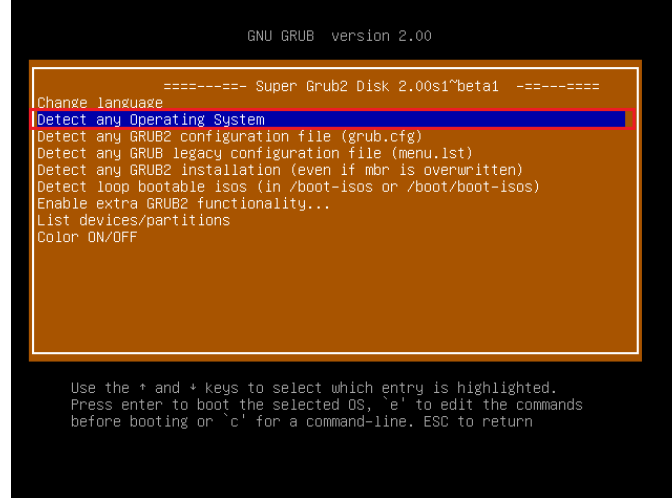
➤ Stage 1.5

- ✓ The core.img will load the file needed for configuration with various other modules like file system drivers acquired from boot.img

The Functionality of GRUB (II/II)

➤ Stage 2

- ✓ In this final stage a Text-based User Interface is displayed which will enable the user to select between the Operating systems.
- ✓ You can specify a default Operating system to load after a user define timeout.



```
GNU GRUB version 2.00

----- Super Grub2 Disk 2.00s1~beta1 -----
Change language
Detect any Operating System
Detect any GRUB2 configuration file (grub.cfg)
Detect any GRUB2 legacy configuration file (menu.lst)
Detect any GRUB2 installation (even if mbr is overwritten)
Detect loop bootable isos (in /boot-isos or /boot/boot-isos)
Enable extra GRUB2 functionality...
List devices/partitions
Color ON/OFF

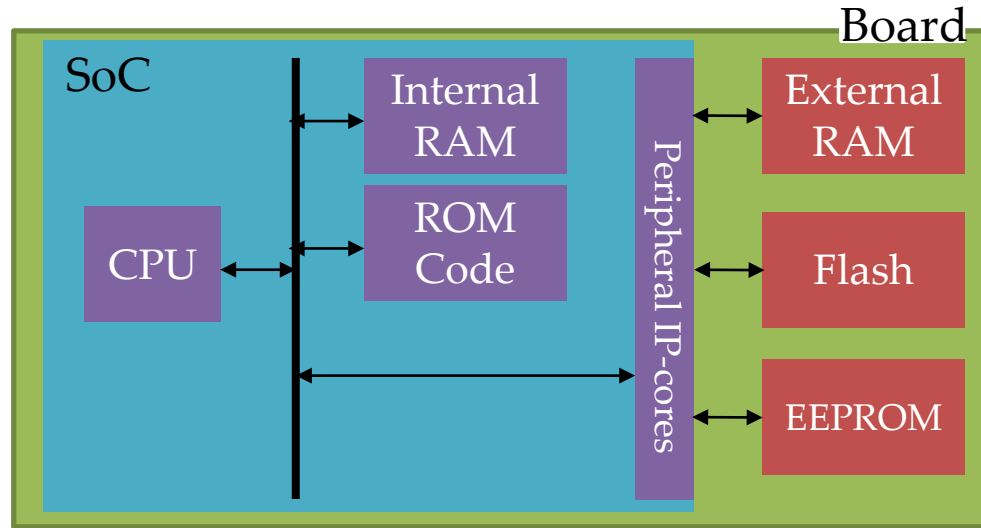
Use the + and - keys to select which entry is highlighted.
Press enter to boot the selected OS, `e` to edit the commands
before booting or `c` for a command-line. ESC to return
```


The Differences Between PC and ES

- The bootloader of embedded system is more complicated than that of PC
 - ✓ Embedded systems do not have a BIOS to perform the initial system configuration.
- Bootloader in x86 architecture consists of two parts
 - ✓ BIOS (Basic Input/Output System)
 - ✓ OS loader (located in MBR of hard disk)
 - e.g., Linux LOader(LILO) and GRUB

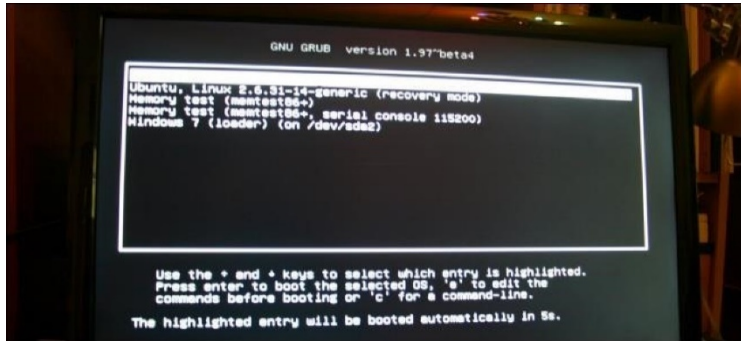


Embedded Board



What is Bootloader?

- Somebody will say ...
 - ✓ The first section of code to be executed after the embedded system is powered on or reset on any platform.
 - ✓ A program that starts whenever a device is powered on to activate the right operating system.



The Purpose of Bootloader

➤ Bootloader in (embedded) computing systems

✓ ROM code has limitations:

- Doesn't know about RAM address
- Doesn't know about board name
- Not flexible enough

ROM Code



Kernel

The Purpose of Bootloader

➤ Bootloader in (embedded) computing systems

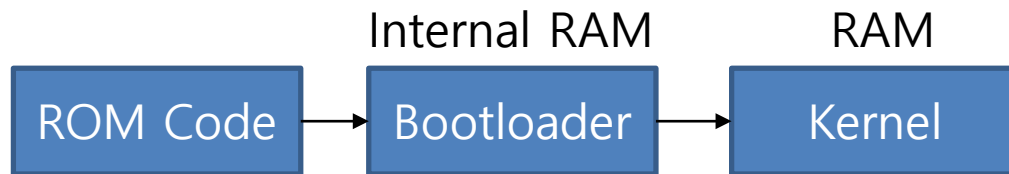
✓ ROM code has limitations:

- Doesn't know about RAM address
- Doesn't know about board name
- Not flexible enough

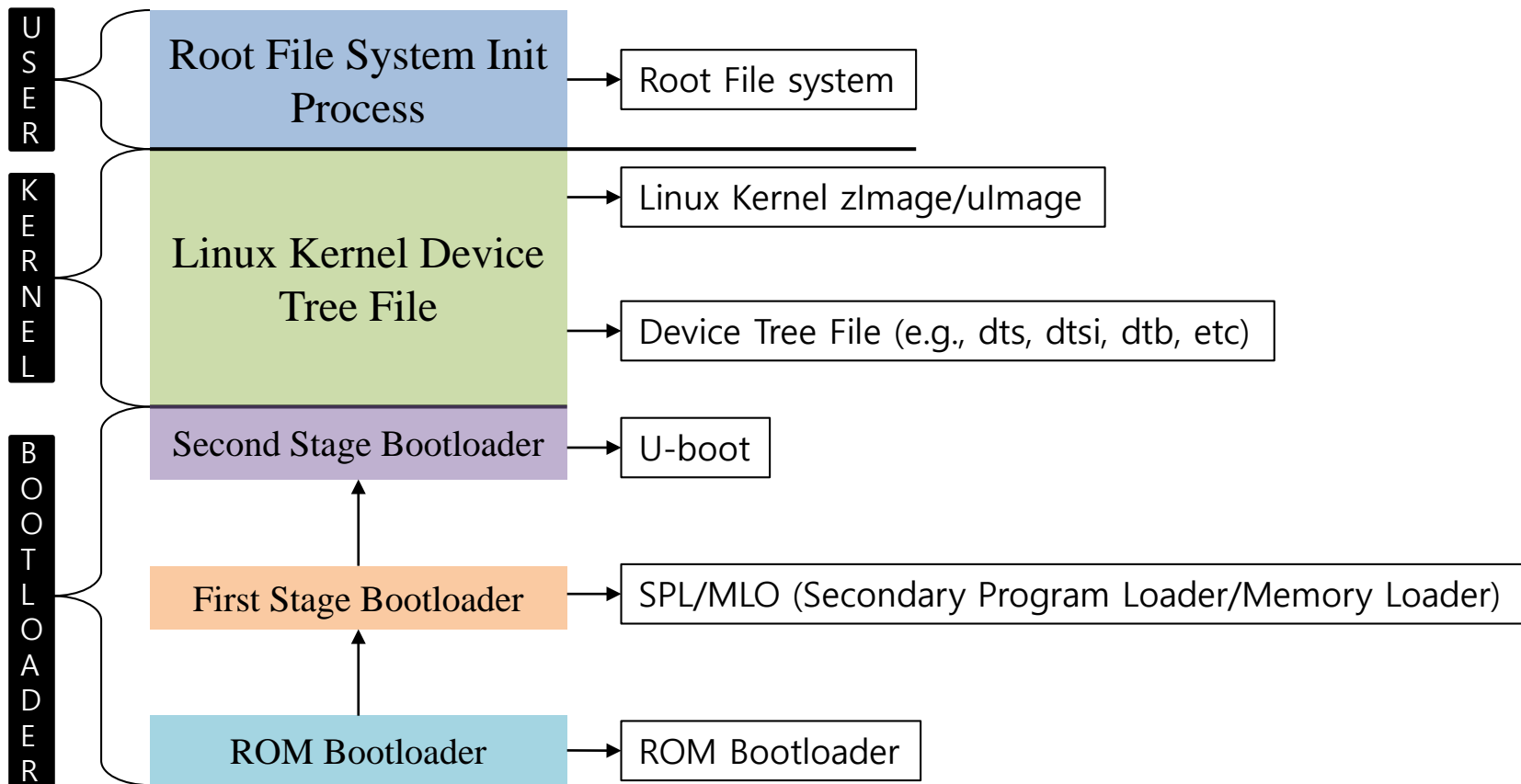


✓ Bootloader to the rescue:

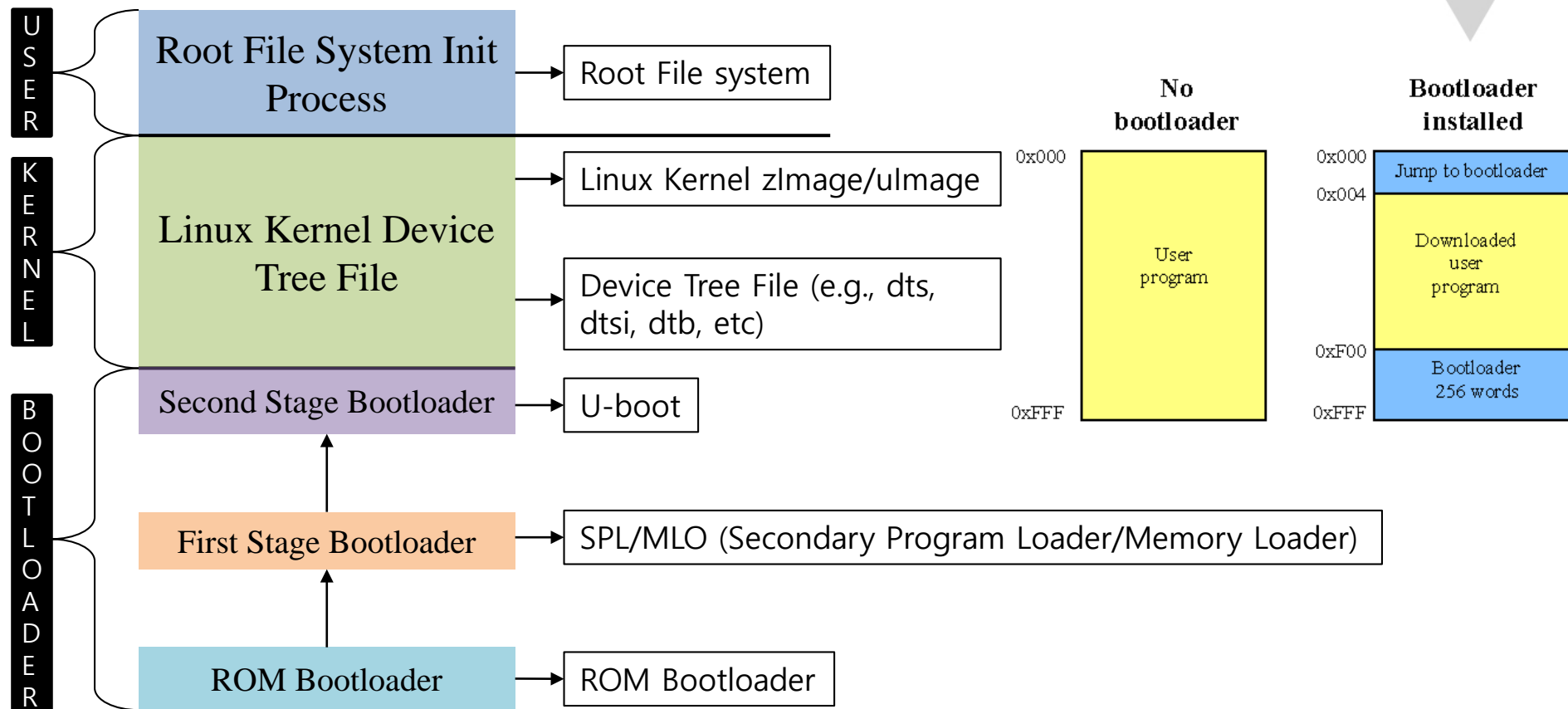
- Reside in flash. But why?
- Able to configure RAM
- Knows boot procedure
- Convenient features



How Does Bootloader Work?



How Does Bootloader Work?



What is U-Boot

➤ Das U-Boot

- ✓ A GPL'ed cross-platform boot loader
- ✓ Created by Wolfgang Denk

➤ U-Boot provides out-of-the-box support for hundreds of embedded boards and a wide variety of CPUs including PowerPC, ARM, XScale, MIPS, Coldfire, NIOS, Microblaze, and x86

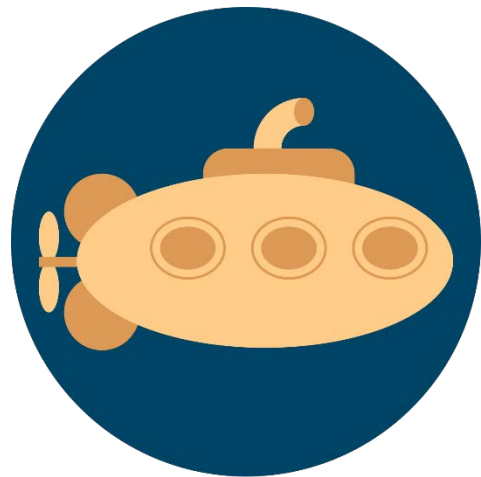
➤ Official website

- ✓ <https://www.denx.de/wiki/U-Boot/WebHome>



Why U-Boot?

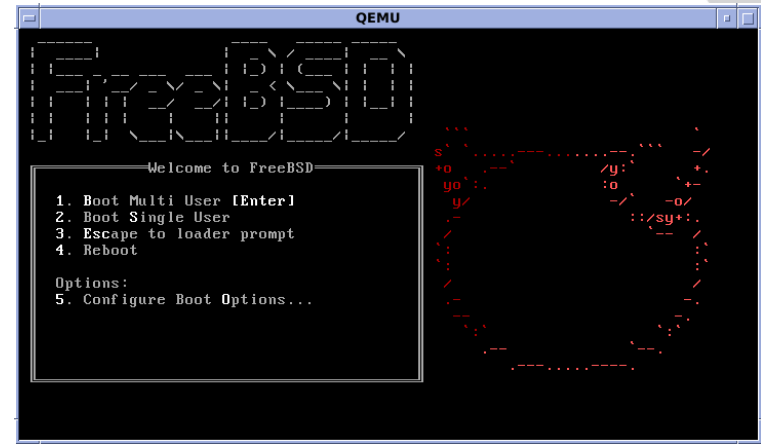
- Bootloader for embedded boards
 - ✓ Popular for Android device
 - ✓ Adoption in automotive
- 13 architectures (including ARM, x86, MIPS, etc.)
- ~300 boards
- Device drivers and lib routines
- Resembles Linux kernel a lot
- Scripting, extensive command set



U-Boot

U-Boot Features

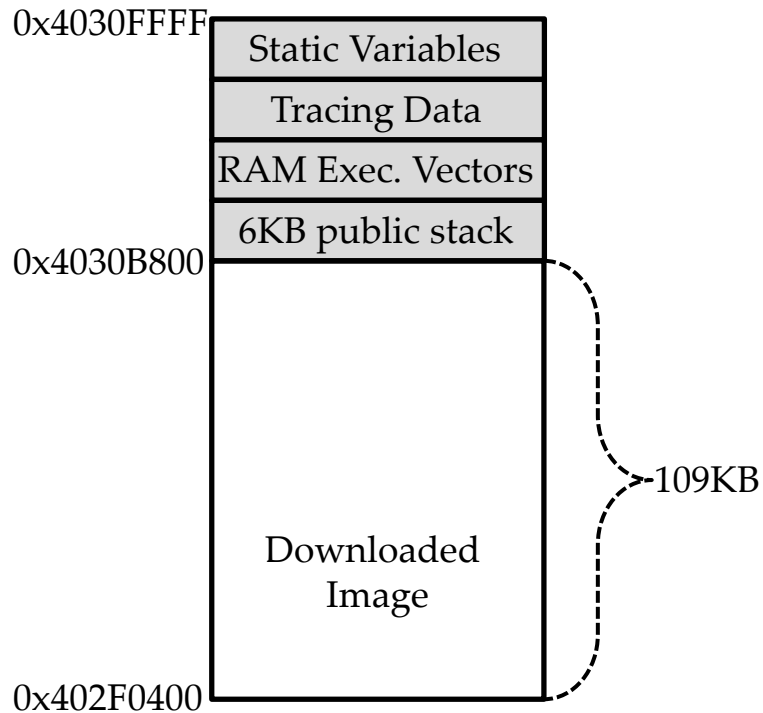
- Boots from various source
 - ✓ Network source
 - ✓ External storage
- Boots various O.S.s
 - ✓ OpenBSD, NetBSD, FreeBSD, 4.4BSD, Linux, SVR4, Esix, Solaris, Irix, SCO, Dell, NCR, VxWorks, LynxOS, pSOS, QNX, RTEMS, ARTOS Device drivers and lib routines
- 2 Stage boot (SPL + U-Boot)
- Falcon mode (SPL only)



ORACLE®
SOLARIS



Why Two-stage Boot



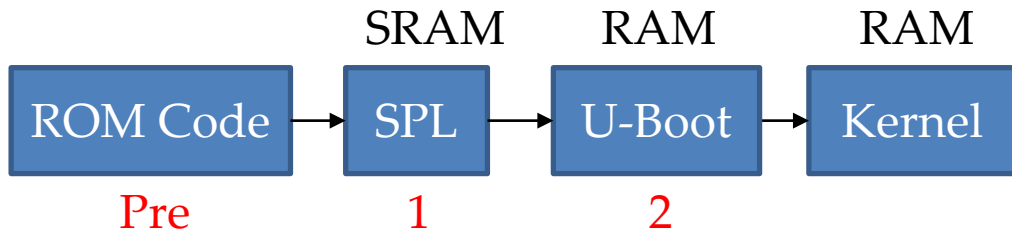
SRAM layout (From AM335x TRM)

But bootloader is so big!

For BeagleBone Black, u-boot.img is 391KB

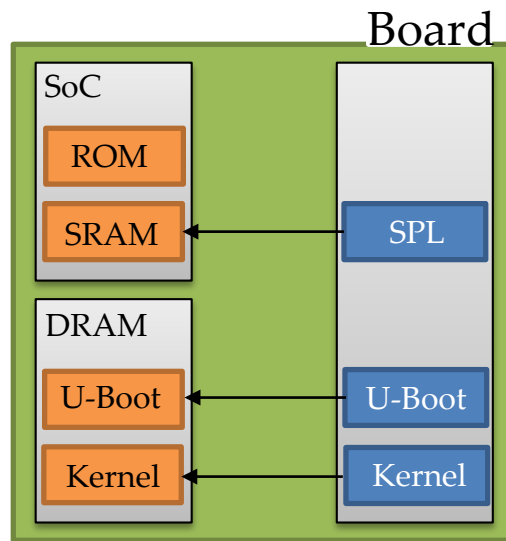
What is Two-stage Boot

An intermediate stage (SPL):



Example: BeagleBone Black

Stage	Size
SPL	75 KB
U-Boot	391 KB



Start-up an Embedded System (I/II)

- Step 1: Load the first instruction from a base address (ROM Boot).
 - ✓ 0x00000000 for ARM
 - ✓ 0xBFC00000 for MIPS
 - ✓ Two main functions
 - Configuration of the device and initialization of primary peripherals
 - Ready device for next bootloader

- Step 2: Secondary program loader (SPL) also referred as to MLO
 - ✓ The first stage of U-boot
 - ✓ To set-up the boot process for the next bootloader stage

Start-up an Embedded System (II/II)

➤ Step 3: U-Boot

- ✓ Powerful command-based control over the kernel boot environment via a serial terminal
- ✓ Environment variables in the uEnv.txt file
- ✓ These environment variables can be viewed, modified, and saved using the `printenv`, `setenv`, and `saveenv` commands, respectively.

```
CCCCCCCC
U-Boot SPL 2011.09 (Jul 26 2012 - 17:18:20)
Texas Instruments Revision detection unimplemented
Found a daughter card connected
OMAP SD/MMC: 0
reading u-boot.img
reading u-boot.img

U-Boot 2011.09 (Jul 26 2012 - 17:13:38)

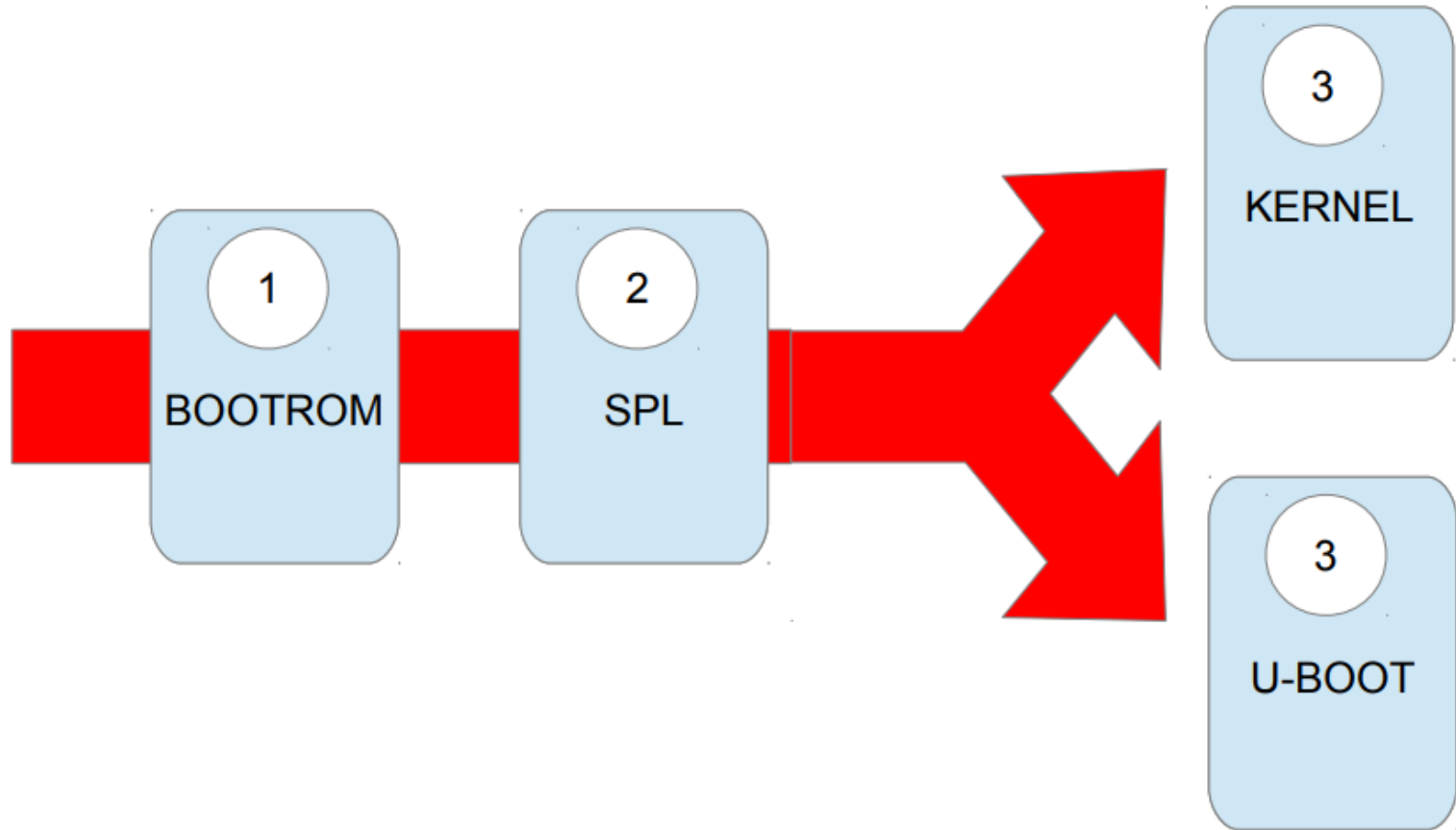
I2C:   ready
DRAM:  256 MiB
WARNING: Caches not enabled
Found a daughter card connected
NAND:  HW ECC Hamming Code selected
256 MiB
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
*** Warning - bad CRC, using default environment

Net:   cpsw
Hit any key to stop autoboot:  0
U-Boot#
```

➤ Step 4: Kernel Image

- ✓ uImage is the kernel image wrapped with header info that describes the kernel.

Falcon Boot



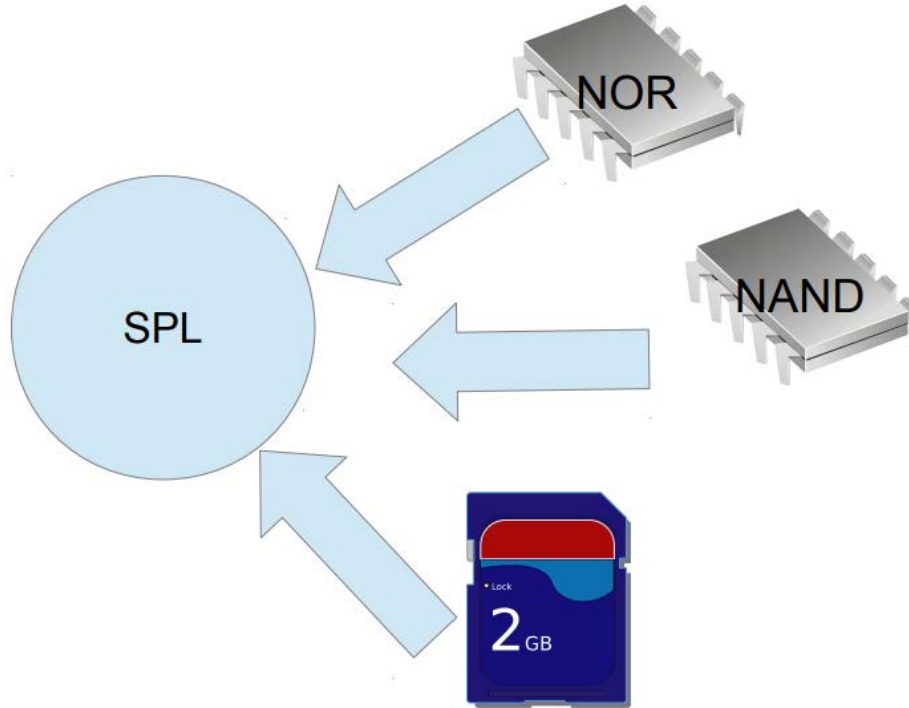
Why Falcon

- Saves time to load U-BOOT
- Saves U-BOOT execution time
- Save time to prepare Boot Parameter Area (legacy kernel)

Supported Boards

- A3m071 (PowerPC MPC 5200)
- Lwmon5 (PowerPC 440 EPX)
- Ipam390 (TI davinci)
- TI OMAP5 boards (dra7xx, uevm) NAND only
- Twister, devkit8000 (TI AM3517)
- Am335_evm (TI AM335x)

Supported Storage



Brief Summary: U-Boot

- The first execution is the fixed ROM Code

Start address = 0x00000000



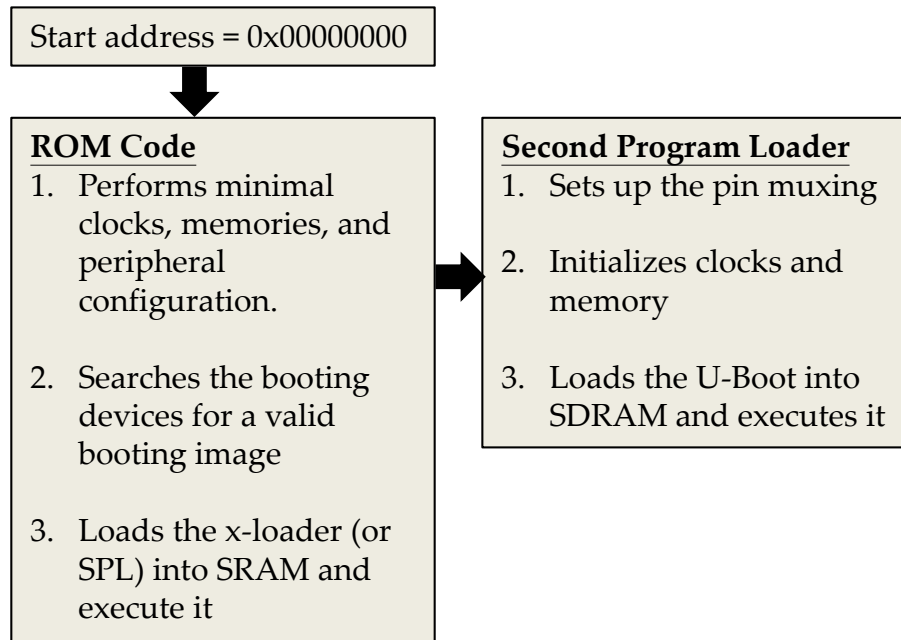
ROM Code

1. Performs minimal clocks and peripheral configuration.
2. Searches the booting devices for a valid booting image
3. Load the x-loader (or SPL) into SRAM and execute it

- A two-stage boot loader is employed

Brief Summary: U-Boot

➤ The first execution is the fixed ROM Code

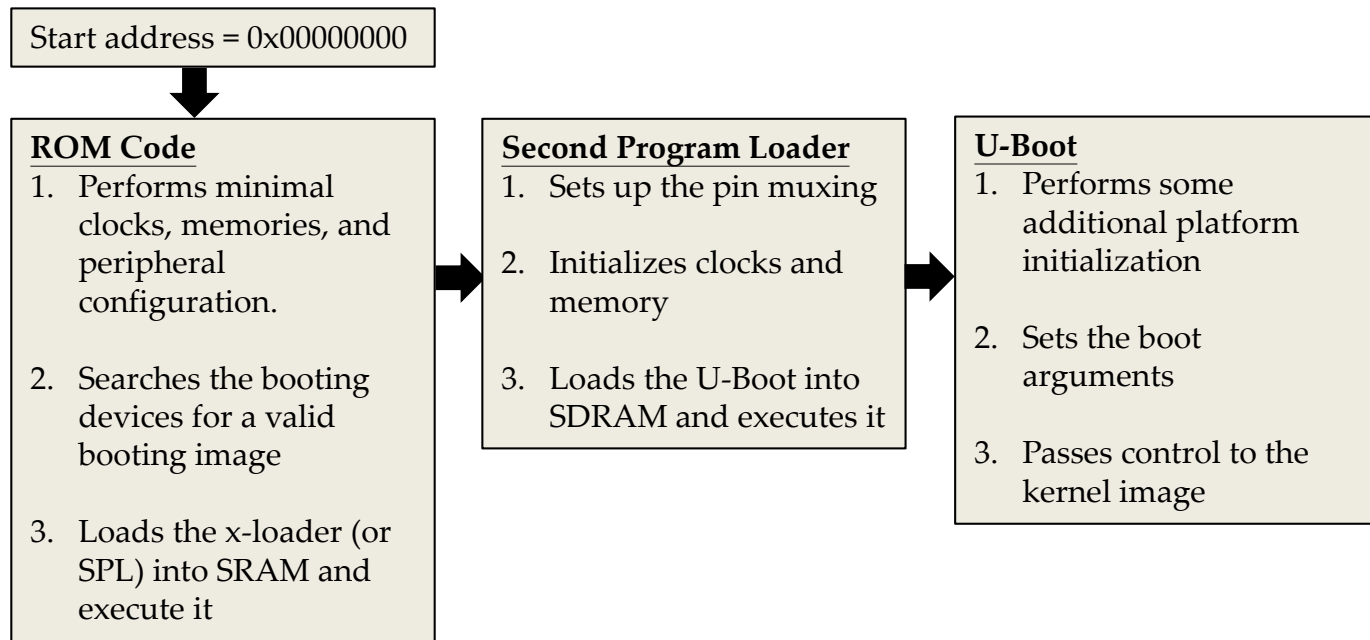


➤ In Stage 1

- ✓ Codes are implemented in assembly language. (For efficiency)
- ✓ Detect the machine type
- ✓ Mask all interrupt request
- ✓ Initialize the indicators (i.e., LED)
- ✓ Disable data and instruction Cache
- ✓ Load preparing
 - Prepare RAM space for Stage 2
 - Setup stack
 - Jump to the entry-point of Stage 2

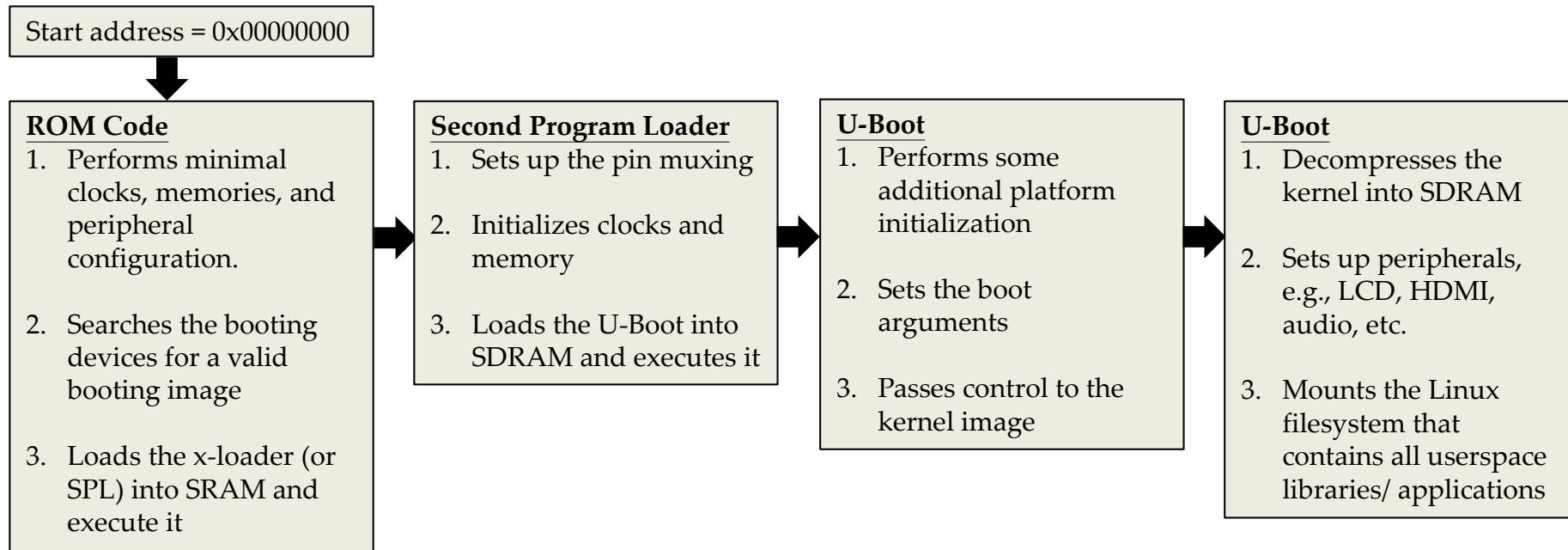
Brief Summary: U-Boot

- The first execution is the fixed ROM Code

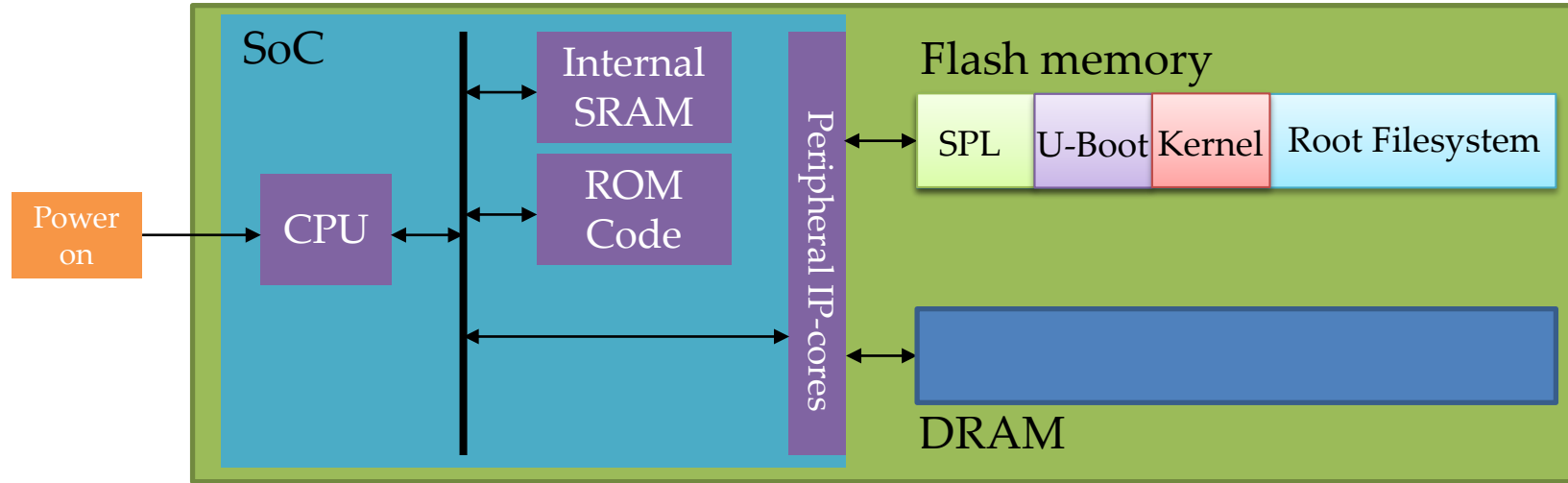


Brief Summary: U-Boot

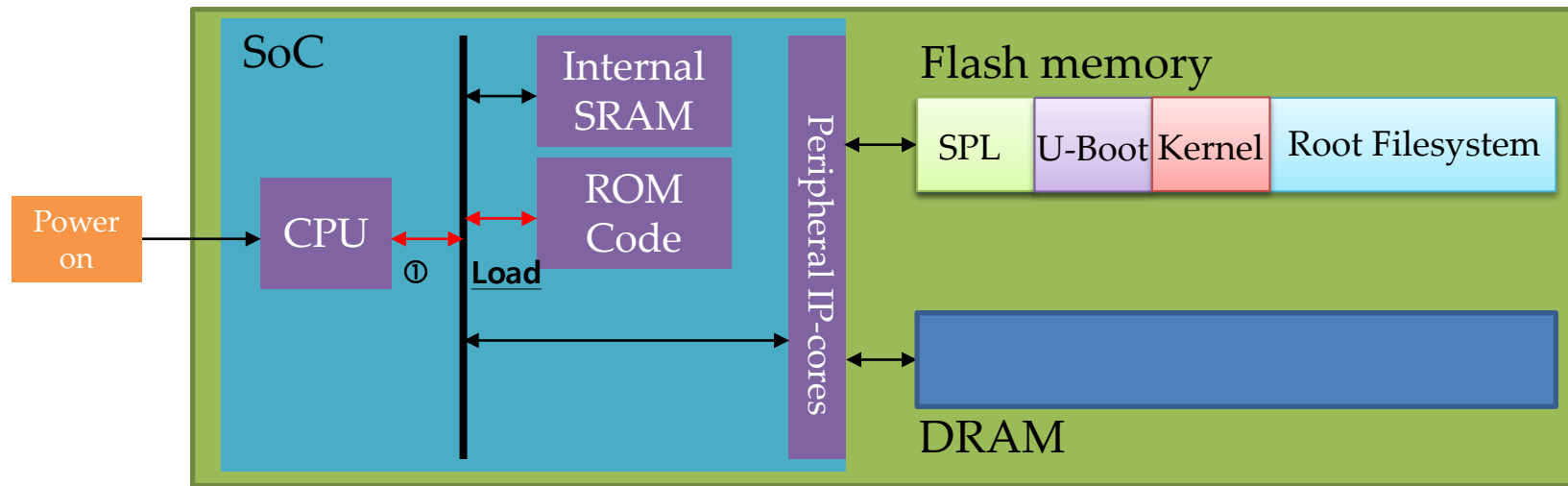
- The first execution is the fixed ROM Code



Flow for Boot Process



Flow for Boot Process



- Load bootrom code from 0x00000000

ROM Code for Execution

- You can refer to U-Boot linker script (e.g., `\board\samsung\smdk6400\u-boot-nand.ld`)

```
SECTIONS
{
    . = 0x00000000; /* Start address 0x0 */
    . = ALIGN(4);
    .text      :    /* start.o first */
    {
        arch/arm/cpu/arm1176/start.o      (.text)
        *(.text)
    }
}
```

ROM Code for Execution (cont'd)

~/u-boot_orig/nat/u-boot

```
<+> mx5
<+> omap-common
<+> omap3
<+> omap4
<+> s5p-common
<+> s5pc1xx
<+> s5pc2xx
<+> tegra2
<+> u8500
[+] Makefile
[+] cache_v7.c
[+] config.mk
[+] cpu.c
[?] start.S
[+] syslib.c
[?] u-boot.lds
```

```
26 LIB = $(obj)lib$(CPU).o
27
28 START := start.o
29
30 ifndef CONFIG_SPL_BUILD
31 COBJS += cache_v7.o
32 COBJS += cpu.o
33 endif
34
35 COBJS += syslib.o
36
37 SRCS := $(START:.o=.S) $(COBJS:.o=.c)
38 OBJS := $(addprefix $(obj),$(COBJS))
39 START := $(addprefix $(obj),$(START))
40
41 all: $(obj).depend $(START) $(LIB)
42
43 $(LIB): $(OBJS)
44     $(call cmd_link_o_target, $(OBJS))
45
46 #####
```

ROM Code for Execution (cont'd)

➤ Board initialization in assembly

```
[+] config.mk
[+] cpu.c
[?] start.S
[+] syslib.c
[?] u-boot.lds

167 #endif /* NAND Boot */
168 #endif
169 /* the mask ROM code should have PLL and others stable */
170 #ifndef CONFIG_SKIP_LOWLEVEL_INIT
171     bl cpu_init_crit
172 #endif
173
174 /* Set stackpointer in internal RAM to call board_init_f */
175 call_board_init_f:
176     ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
177     bic sp, sp, #7 /* 8-byte alignment for ABI compliance */
178     ldr r0,=0x00000000
179     bl board_init_f
180
2 c:--- start.S          34% L179  (Assembler yas Anzu wb Undo-Tree VHL HelmGtags Helm Projectile
pattern:
1 Find tag from here
2 arch/arm/cpu/armv7/omap-common/spl.c:56:void board_init_f(ulong dummy)
3 arch/arm/lib/board.c:262:void board_init_f(ulong bootflag)
```

ROM Code for Execution (cont'd)

➤ Board initialization in C language

```
void board_init_f(ulong bootflag)
{
    bd_t *bd;
    init_fnc_t **init_fnc_ptr;
    gd_t *gd;
    ulong addr, addr_sp;

    /* Pointer is writable since we allocated a register for it */
    gd = (gd_t *) ((CONFIG_SYS_INIT_SP_ADDR) & ~0x07);
    /* compiler optimization barrier needed for GCC >= 3.4 */
    __asm__ __volatile__(": : :memory");

    memset((void *)gd, 0, sizeof(gd_t));

    gd->mon_len = _bss_end_ofs;

    for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
        if ((*init_fnc_ptr)() != 0) {
            hang ();
        }
    }

    debug("monitor len: %08lx\n", gd->mon_len);
    /*
     * Ram is setup, size stored in gd !!
     */
    debug("ramsize: %08lx\n", gd->ram_size);
}
```

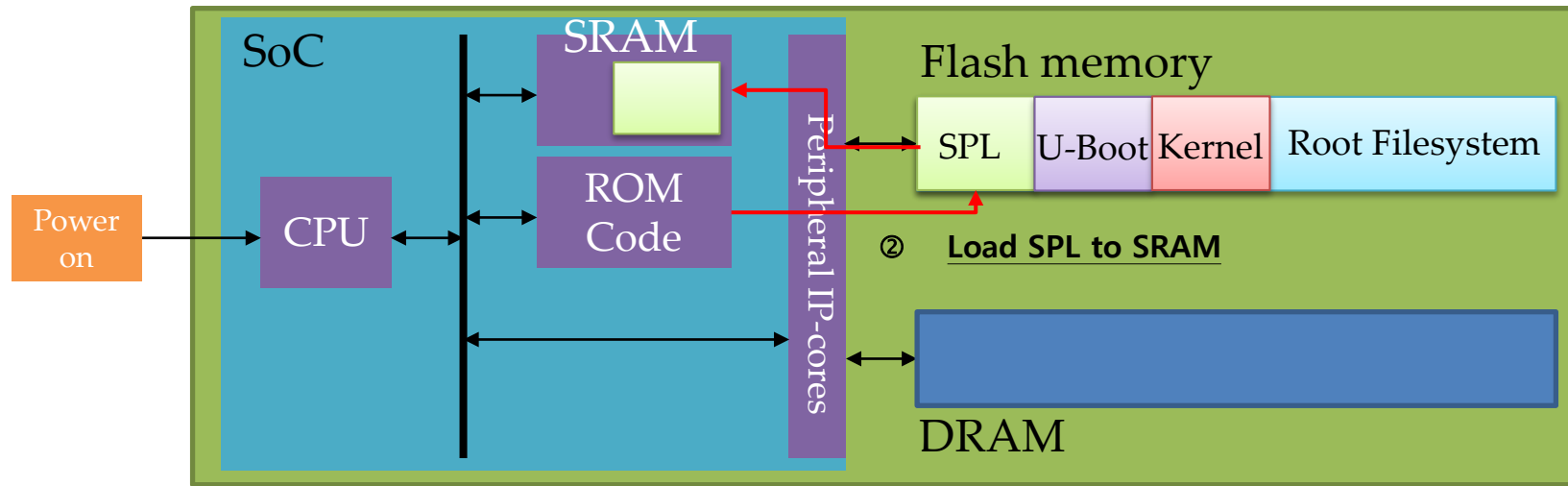
```
void __dram_init_banksz(ulong)
{
}

void dram_init_banksz(void)
{
    __attribute__((weak, alias("__dram_init_banksz")));
}

init_fnc_t *init_sequence[] = {
    #if defined(CONFIG_ARCH_CPU_INIT)
        arch_cpu_init, /* basic arch cpu dependent setup */
    #endif
    #if defined(CONFIG_BOARD_EARLY_INIT_F)
        board_early_init_f,
    #endif
    timer_init, /* initialize timer */
    #ifdef CONFIG_FSL_ESDHC
        get_clocks,
    #endif
    env_init, /* initialize environment */
    init_baudrate, /* initialize baudrate settings */
    serial_init, /* serial communications setup */
    console_init_f, /* stage 1 init of console */
    display_banner, /* say that we are here */
    #if defined(CONFIG_DISPLAY_CPUINFO)
        print_cpuinfo, /* display cpu info (and speed) */
    #endif
    #if defined(CONFIG_DISPLAY_BOARDINFO)
        checkboard, /* display board info */
    #endif
    #if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT_I2C)

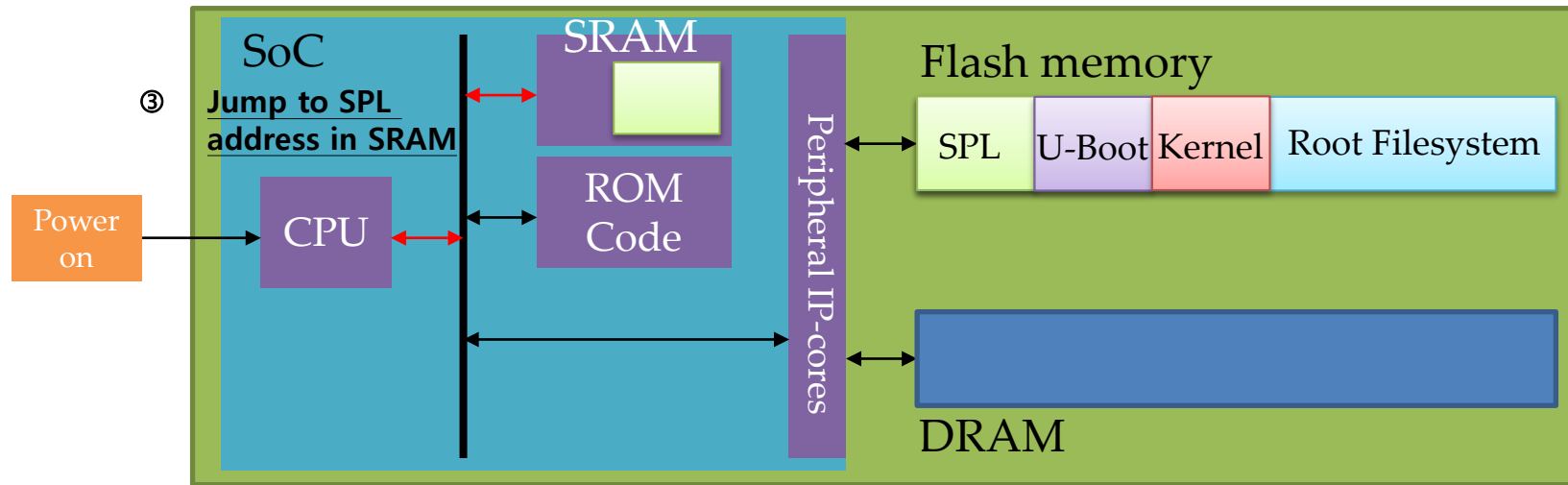
```

Flow for Boot Process



- SRAM does not need to be initialized and it is ready after power-on
- Load SPL to SRAM

Flow for Boot Process



- CPU executes the SPL code in SRAM to initialize DRAM and clock.

Set CPU to SVC

➤ In `./arch/arm/cpu/<HW>/<Model>/start.S`

```
/*  
    31  30  29  28  -----  7   6   -   4   3   2   1   0  
    N   Z   C   V           I   F           M4  M3  M2  M1  M0  
  
                                1   0   0   0   0      User Mode  
                                1   0   0   0   1      FIQ Mode  
                                1   0   0   1   0      IRQ Mode  
                                1   0   0   1   1      SVC Mode  
                                1   0   1   1   1      ABT Mode  
                                1   1   0   1   1      UND Mode  
                                1   1   1   1   1      SYS Mode  
  
*/  
mrs r0, cpsr  
bic r0, r0, #0x1f  
orr r0, r0, #0xd3
```

Load board_init_r and Enter Stage 2

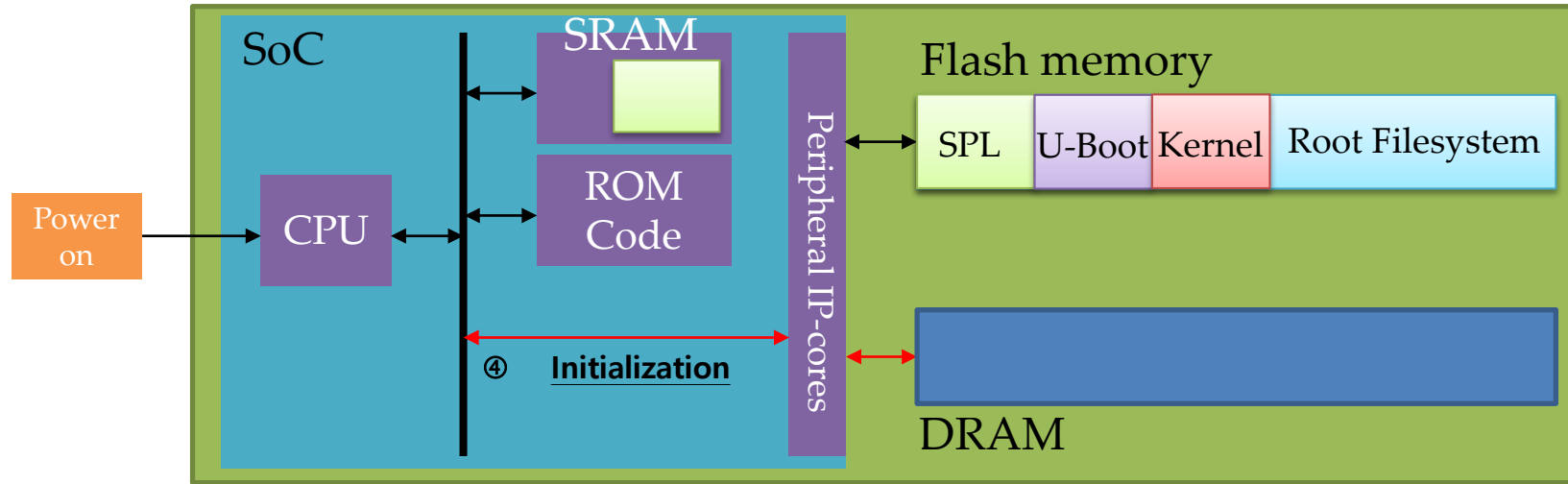
➤ In ./arch/arm/cpu/<HW>/<Model>/start.S

```
mov r0, r9                /* gd_t */
ldr r1, =(CONFIG_SYS_MALLOC_END) /* dest_addr for malloc heap end */
/* call board init r */
ldr pc, =board_init_r
```

➤ In /u-boot/arch/arm/lib/board.c

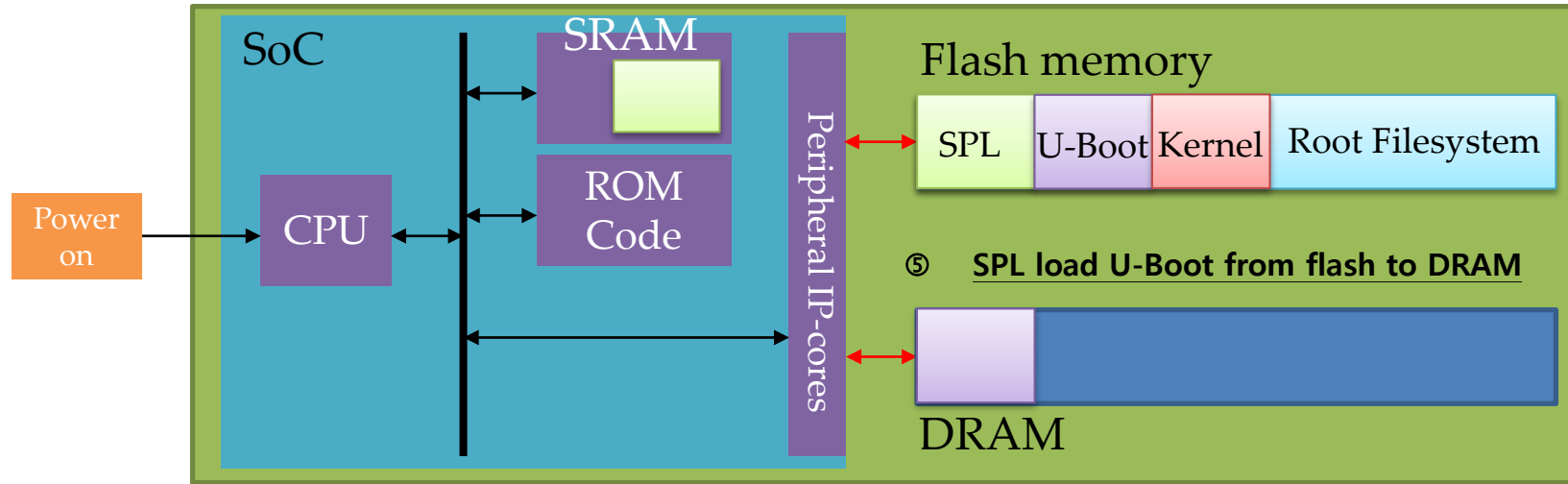
```
init_fnc_t init_sequence_r[] = {
#ifdef CONFIG_SYS_INIT_RAM_LOCK) && defined(CONFIG_E500)
    initr_unlock_ram_in_cache,
#endif
    initr_barrier,
    initr_malloc,
    console_init_m,
```


Flow for Boot Process



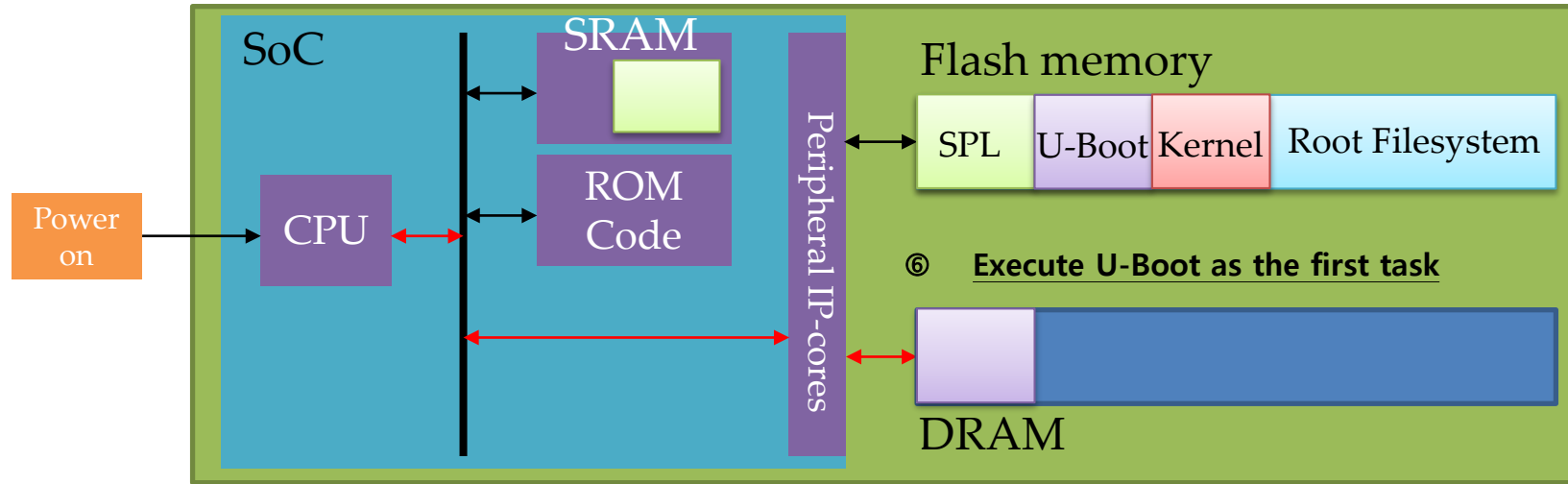
- DRAM space initialization

Flow for Boot Process



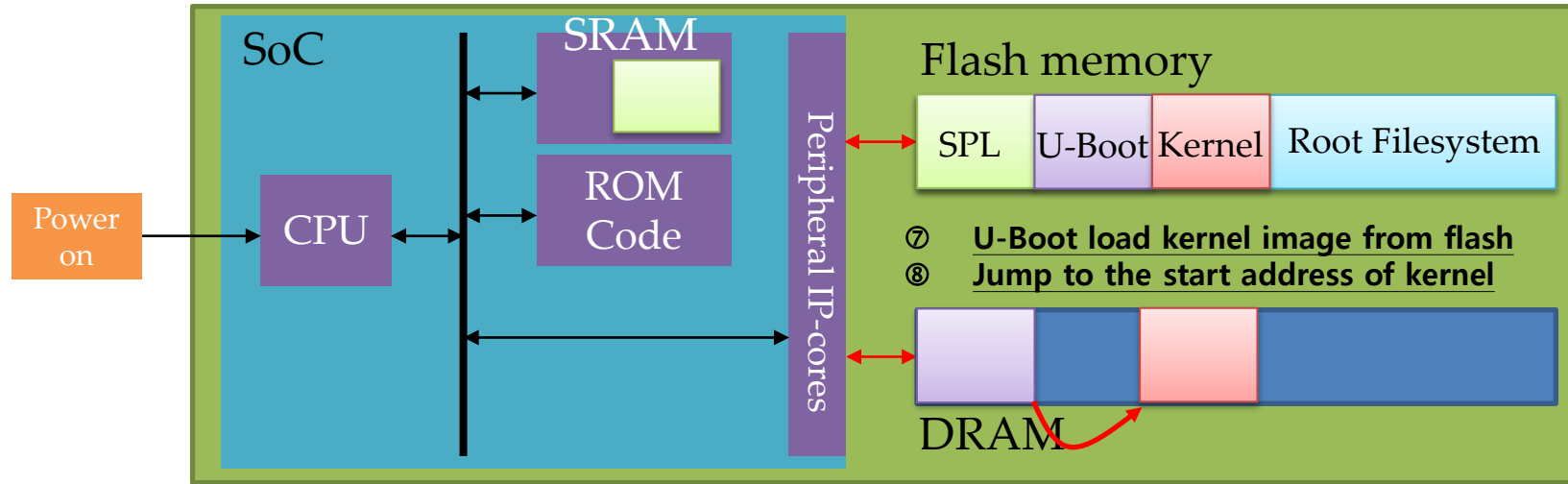
- Execute SPL to load U-Boot from flash memory to DRAM

Flow for Boot Process



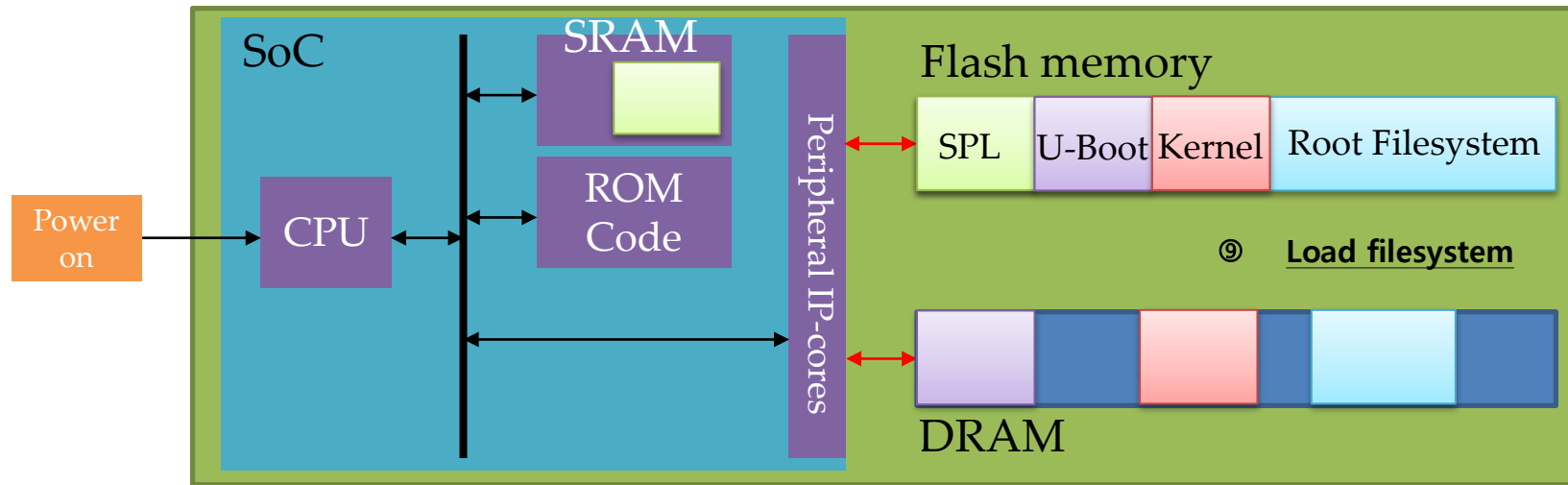
- Jump to the start address of U-Boot and execute its code

Flow for Boot Process



- Load kernel image from flash memory
- Decompress the kernel image in DRAM

Flow for Boot Process



- Load filesystem image from flash memory to DRAM
- Mount the root filesystem

How to Run U-Boot

➤ Step 1: What do we need?

- ✓ Platform information: QEMU to emulate vexpress-a9
- ✓ U-Boot source code from official site
 - <https://www.denx.de/wiki/U-Boot/WebChanges>
- ✓ Linux source code from linux official site
 - <https://www.kernel.org/>

➤ Step 2: Compile U-Boot

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- vexpress_ca9x4_defconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j8
```

How to Run U-Boot (cont'd.)

➤ Step 3: Copy zImage and dtb to SD card

```
sudo cp linux-4.14.13/arch/arm/boot/zImage p1/  
sudo cp linux-4.14.13/arch/arm/boot/dts/vexpress-v2*.dtb p1/  
sudo cp -raf ../rootfs/rootfs/* ./p2
```

➤ Step 4.a: Launch U-Boot on QEMU

- ✓ uboot_image=./u-boot-2019.10/u-boot
- ✓ qemu_path=/home/pengdl/work/Qemu/qemu-4.1.0/build/arm-softmmu

```
${qemu_path}/qemu-system-arm -M vexpress-a9 \  
    -m 1024M \  
    -smp 1 \  
    -nographic \  
    -kernel ${uboot_image} \  
    -sd ./uboot.disk
```

How to Run U-Boot (cont'd.)

➤ Step 4.b: In U-Boot, mmc command

```
=> mmc dev 0
switch to partitions #0, OK
mmc0 is current device
=> mmc info
Device: MMC
Manufacturer ID: aa
OEM: 5859
Name: QEMU!
Bus Speed: 6250000
Mode: SD Legacy
Rd Block Len: 512
SD version 2.0
High Capacity: No
Capacity: 1 GiB
Bus Width: 1-bit
Erase Group Size: 512 Bytes
```

```
=> part list mmc 0
.....
=> ls mmc 0:1 or ext4ls mmc 0:1
<DIR> 1024 .
<DIR> 1024 ..
<DIR> 12288 lost+found
          7680720 zImage
          19161 vexpress-v2p-ca15_a7.dtb
          13384 vexpress-v2p-ca15-tcl1.dtb
          12994 vexpress-v2p-ca5s.dtb
          14736 vexpress-v2p-ca9.dtb
=> ls mmc 0:2 or ext4ls mmc 0:2
<DIR> 4096 .
<DIR> 4096 ..
<DIR> 16384 lost+found
<DIR> 4096 bin
<DIR> 4096 dev ...
```


How to Run U-Boot (cont'd.)

➤ Step 5: Load kernel and device tree

```
=> load mmc 0:1 0x60008000 zImage or ext4load mmc 0:1 0x60008000 zImage
7680720 bytes read in 1034 ms (7.1 MiB/s)
=> load mmc 0:1 0x61000000 vexpress-v2p-ca9.dtb or
=> ext4load mmc 0:1 0x61000000 vexpress-v2p-ca9.dtb
14736 bytes read in 67 ms (213.9 KiB/s)
```

➤ Step 6: Set-up Bootargs

```
=> setenv bootargs 'root=/dev/mmcblk0p2 rw rootfstype=ext4 rootwait earlycon
console=tty0 console=ttyAMA0 init=/linuxrc ignore_loglevel'
```

➤ Step 7: Jump to kernel

```
=> bootz 0x60008000 - 0x61000000
```

Device Tree

➤ What is device tree?

- ✓ Device Tree (DT) was created by Open Firmware to allow an operating system at runtime to run on various hardware without hard coding any information
- ✓ It is a data structure and language for describing hardware.

➤ Pros

- ✓ Attempting to eliminate board specific code from drives
- ✓ Ability to cleanly support multiple boards with a single kernel
- ✓ Replaces complex and massive board file

➤ Cons

- ✓ Kernel size increase
- ✓ Increase in boot time

Generate Device Tree

- Need a compiler: Device tree compile (DTC)

```
make ARCH=arm vexpress_ca9x4_defconfig  
make ARCH=arm menuconfig
```

- Generate dtb

- ✓ `scripts/dtc/dtc -I dts -O dtb -o <devicetree name>.dtb <devicetree name>.dts`

Make Menuconfig

```
Linux/arm 3.18.42 Kernel Configuration

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y>
includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]
excluded <M> module < > module capable

(8) Maximum PAGE_SIZE order of alignment for DMA IOMMU buffers
  General setup --->
  [*] Enable loadable module support --->
  *- Enable the block layer --->
    System Type --->
    Bus support --->
    Kernel Features --->
    Boot options --->
    CPU Power Management --->
    Floating point emulation --->
    Userspace binary formats --->
    Power management options --->
  [*] Networking support --->
    Device Drivers --->
    File systems --->
    Kernel hacking --->
    Security options --->
  *- Cryptographic API --->
    Library routines ---->
  [ ] Virtualization ----

<Select>  < Exit >  < Help >  < Save >  < Load >
```

How to Load DTB

- Device tree blob (dtb) is passed to the kernel via U-boot
- U-boot loads the appropriate dtb in to memory
 - ✓ Currently stored at 0x80F80000 (fdtaddr) which is a free region of memory that doesn't overlap with any other memory location
- U-boot boots the kernel using the below command
 - ✓ `bootm/bootz ${loadaddr} - ${fdtaddr};`

DT: Driver Instantiation Example

➤ AM335x generic configuration for i2c0

Driver node name

DT device id. Determines which driver
This DT binding is meant for.

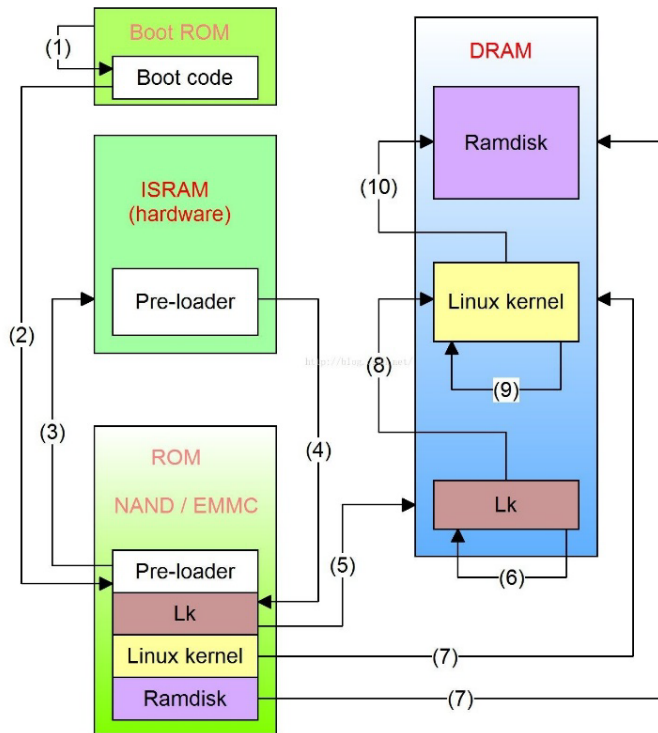
```
i2c0: i2c@44e0b000 {  
    compatible = "ti.omap4-i2c";  
    #address-cells = <1>;  
    #size-cells = <0>;  
    ti,hwmods = "i2c1";  
    reg = <0x44e0b000 0x1000>;  
    interrupts = <70>;  
    status = "disabled";  
};
```

Driver register information. For i2c this
determines memory mapped register
for i2c0

Determines which hardware interrupt
is required

New Bootloader: Little Kernel (lk)

➤ MTK MT6580



Little Kernel (lk)

- What is little kernel?
 - ✓ Open Source software
 - ✓ A tiny operating system
 - ✓ Suited for small embedded devices, bootloaders, and other environments
 - ✓ For Android system
 - ✓ Only 15-20 KB
 - ✓ Available from <https://github.com/littlekernel/lk>
 - ✓ As a bootloader
 - A few ARM SoC manufacturers
 - ✓ As a microkernel
 - Zircon

Outline

- How to Boot an Embedded System (ES)
 - ✓ Grub for x86 Architecture
 - ✓ U-boot for ARM Architecture
 - ✓ How to Implement a Bootloader in an Embedded System.
- Microkernel for Embedded System
 - ✓ What is Process?
 - ✓ The Process Concept in an Embedded System
 - ✓ The Practical Knowledge of Process Management