BOOTLOADER BOOT PROCEDURE

FOR LINUX OS IN I.MX6Q

CONTENTS

CONTENTS

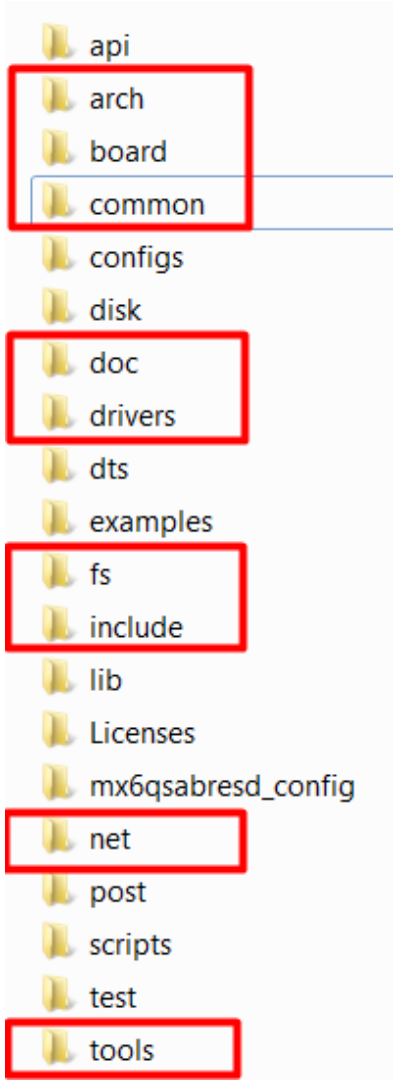
Bootloader Introduction

- Definition
  - A boot loader is a computer program that loads an operating system or some other system software for the computer after completion of the power-on self-tests; it is the loader for the operating system itself.
  - U-boot(Universal Bootloader) is a widely used embedded system Bootloader
- Typically four partitions of embedded storage devices
  - 1st partition: Bootloader
  - 2nd partition: Boot Parameters(Passed from Bootloader to Kernel)
  - 3rd partition: Kernel
  - 4th partition: Root Filesystem

Bootloader	**Parameters**	**Kernel**	**Root Filesystem**

U-boot Directory Structure of the Source Code

- Directory structure in …\tmp\work \imx6qsabresd-poky-linux-gnueabi\u-boot-imx\2015.04-r0\git
  - \arch: different cpu architectures
  - \board: relevant configuration files on board
  - \common: executable C files
  - \doc: u-boot detailed documents
  - \drives: various devices' drives supported by u-boot
  - \fs: several file systems that u-boot supported
  - \include: head files & hardware platform support files
  - \net: network-related protocol code
  - \tools: U-boot generation tool



CONTENTS

i.MX6Q Introduction

- High Level Block Diagram
  - 4x ARM Cortex A9 MPCore™
  - L1 Cache: 32KB Instruction, 32KB Data
  - L2 Cache: Unified instruction and data (1MB)
  - On-chip Memory
    - Boot ROM, including HAB (96 KB)
    - Internal fast access RAM (256 KB)
    - Secure/non-secure RAM (16 KB)
  - Smart DMA
  - Multimedia supported(VPU etc.)

……

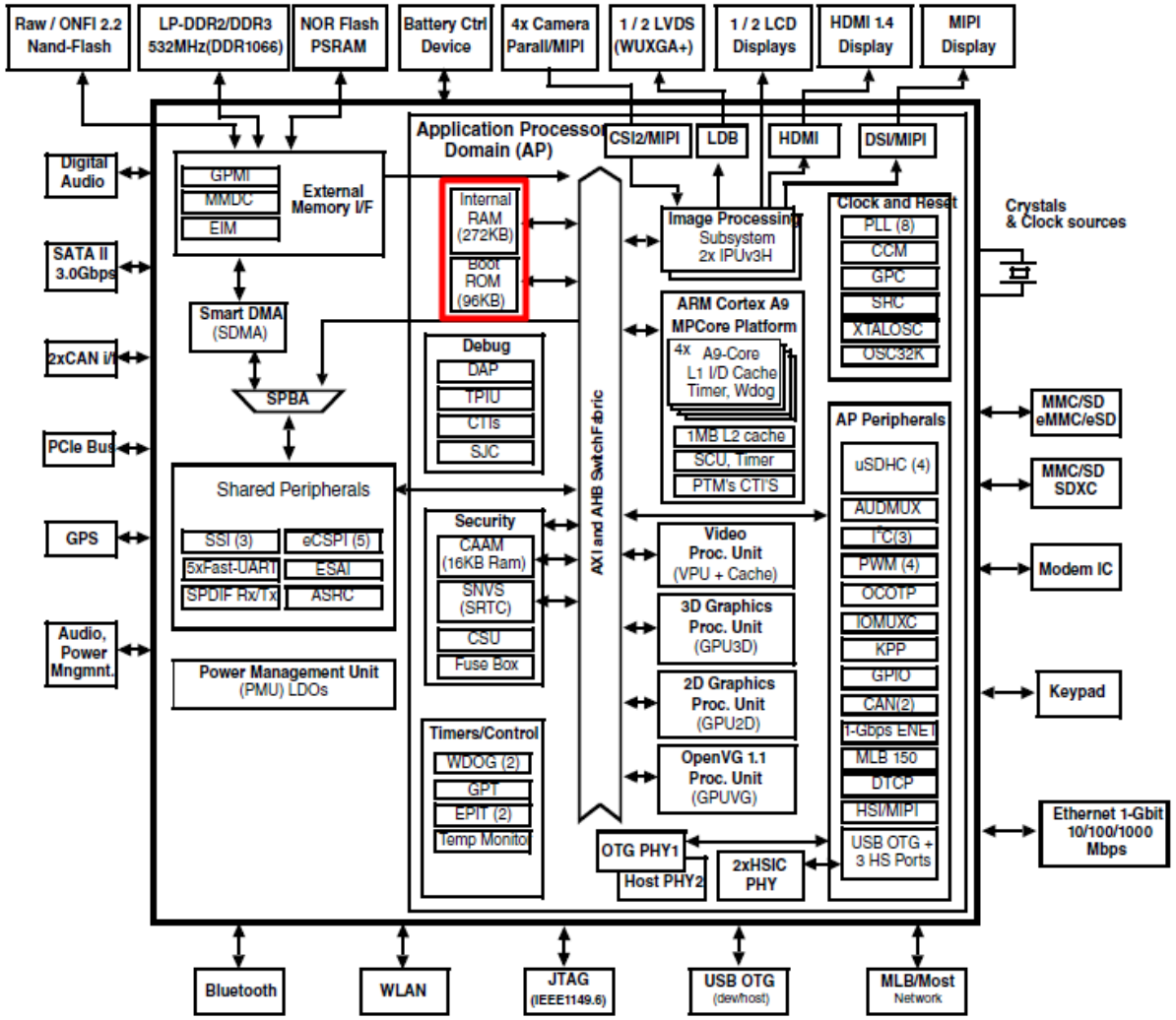


Figure 1-2. Simplified Block Diagram

i.MX6Q Introduction

- Internal ROM and RAM memory map
  - The entire OCRAM region can be used freely post boot
  - The boot ROM includes a feature of enabling the Memory Management Unit (MMU) and caches to improve boot speed.
  - For devices that perform a secure boot, the HAB library may be called by boot stages that execute after ROM code.

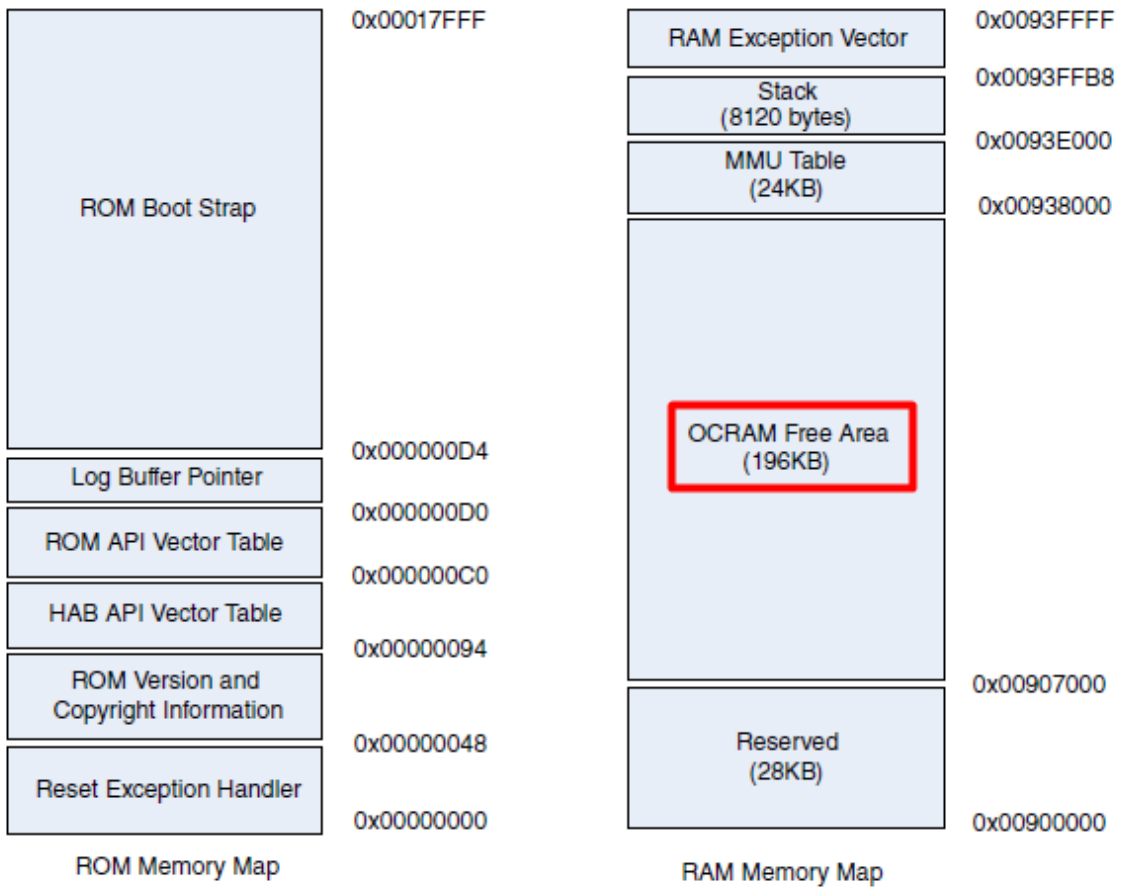


Figure 8-3. Internal ROM and RAM memory map for i.MX 6Dual/6Quad

i.MX6Q Introduction

- On normal boot, the core's behavior is defined by the Boot Mode pins settings

- Three boot mode is selected based on the binary value stored in the internal BOOT_MODE register.
  - Boot From Fuses
  - Serial Downloader
  - Internal Boot

Table 8-1. Boot MODE Pin Settings

BOOT_MODE[1:0]	Boot Type
00	Boot From Fuses
01	Serial Downloader
10	Internal Boot
11	Reserved

i.MX6Q Introduction

- The Chip supports the following boot Flash devices
  - NOR Flash
  - OneNAND Flash
  - Raw NAND
  - SD/MMC/eSD/SDXC/eMMC4.4
  - EEPROM
  - Serial ATA (SATA)
- The selection of external boot device type is controlled by BOOT_CFG1[7:4] eFUSEs.

Table 8-7. Boot Device Selection

BOOT_CFG1[7:4]	Boot Device
0000	NOR/OneNAND (EIM)
0001	Reserved
0010	SSD/Hard Disk (SATA)
0011	Serial ROM (SPI)
010x	SD/eSD/SDXC
011x	MMC/eMMC
1xxx	Raw NAND

i.MX6Q Introduction

- A user's program image(u-boot) consists of:
  - Image vector table
    - A list of pointers that the ROM examin to determine where other components of the program image are located
  - Boot data
    - A table indicating the program image location, program image size in bytes, and the plugin flag
  - Device configuration data
    - IC configuration data
  - User code and data
    - u-boot code and data segments

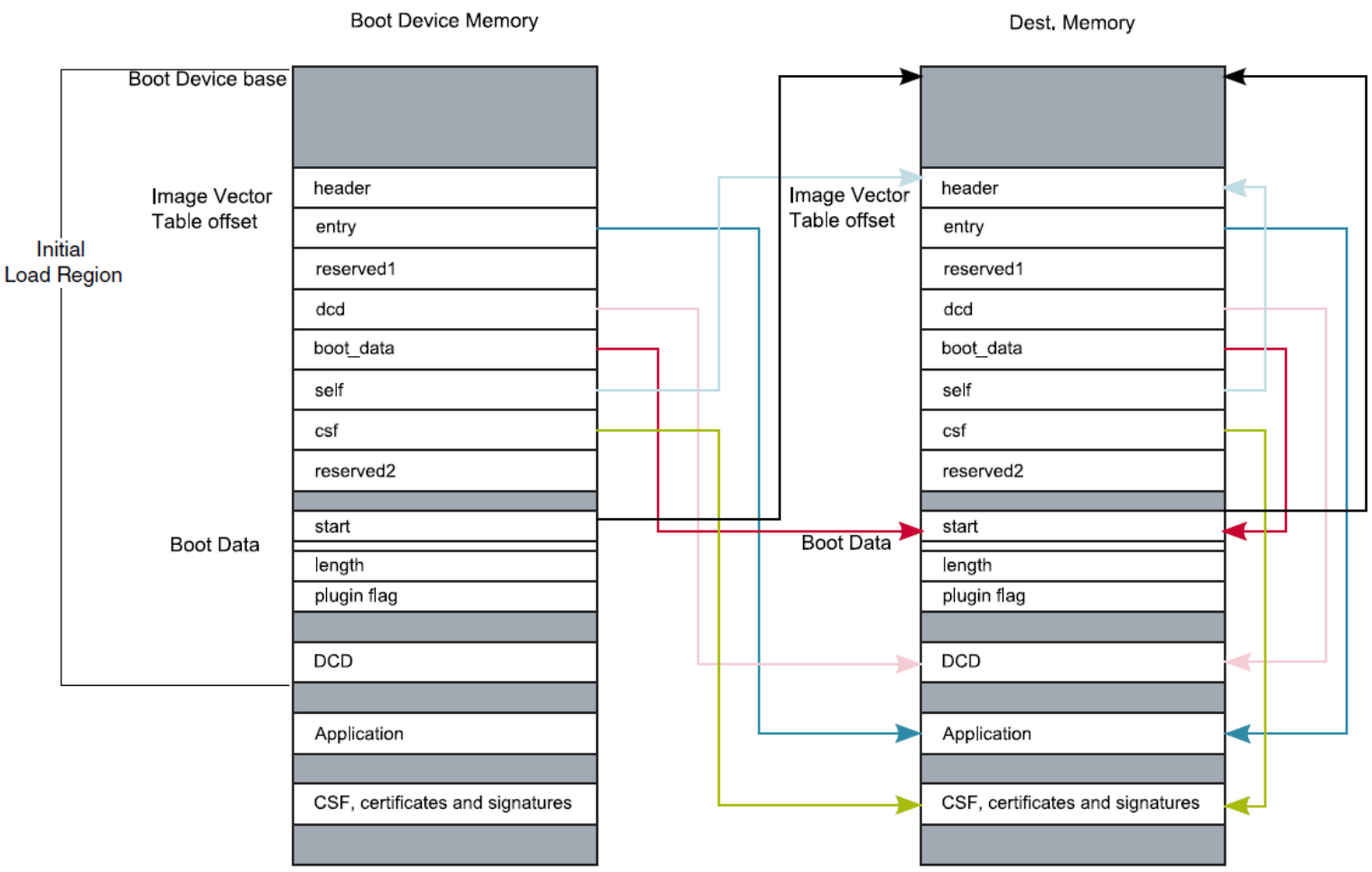
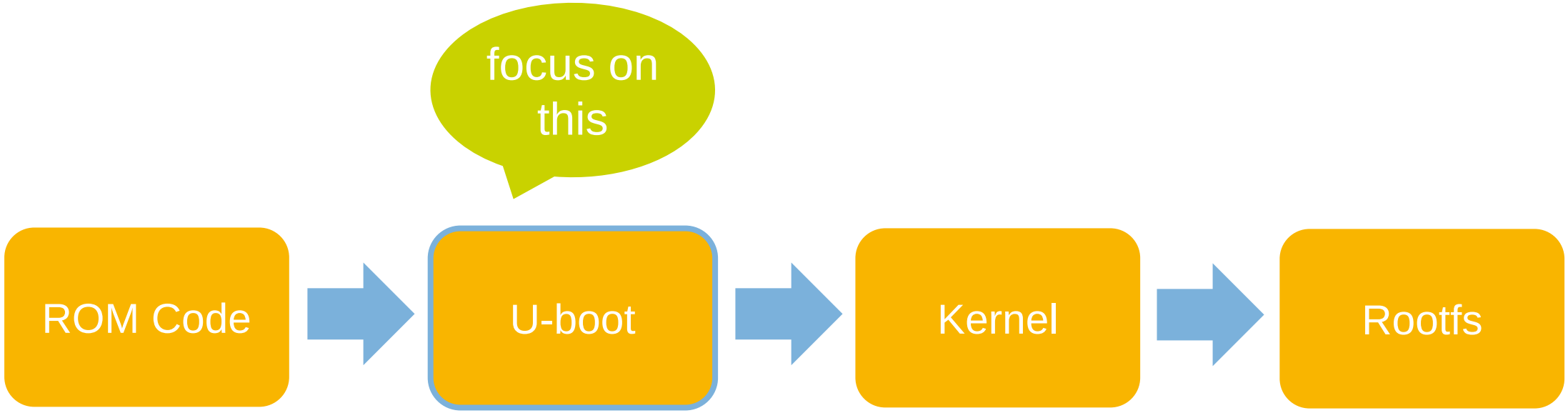


Figure 8-21. Image Vector Table

Linux OS Boot Process



First Stage of Boot Sequence(Assembly Language)

- Setup exception vectors
  - /arch/arm/lib/vectors.S
- Set cpu to SVC32 mode & disable FIQ and IRQ
  - /arch/arm/cpu/armv7/start.S
- Setup CP15 registers (cache, MMU, TLBs)
  - /arch/arm/cpu/armv7/start.S
- Setup important registers and memory timing
  - /arch/arm/cpu/armv7/lowlevel_init.S

First Stage of Boot Sequence(Assembly Language)

- Setup exception vectors
  - /arch/arm/lib/vectors.S

- Set cpu to SVC32 mode & disable FIQ and IRQ
  - /arch/arm/cpu/armv7/start.S

- Setup CP15 registers (cache, MMU, TLBs)
  - /arch/arm/cpu/armv7/start.S

- Setup important registers and memory timing
  - /arch/arm/cpu/armv7/lowlevel_init.S

```
_start:

#ifdef CONFIG_SYS_DV_NOR_BOOT_CFG
        .word   CONFIG_SYS_DV_NOR_BOOT_CFG
#endif

        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
```

First Stage of Boot Sequence(Assembly Language)

- Setup exception vectors
  - /arch/arm/lib/vectors.S

- Set cpu to SVC32 mode & disable FIQ and IRQ
  - /arch/arm/cpu/armv7/start.S

```
save_boot_params_ret:
        /*
         * disable interrupts (FIQ and IRQ), also set the cpu to SVC32 mode,
         * except if in HYP mode already
         */
        mrs     r0, cpsr
        and     r1, r0, #0x1f           @ mask mode bits
        teq     r1, #0x1a               @ test for HYP mode
        bicne   r0, r0, #0x1f           @ clear all mode bits
        orrne   r0, r0, #0x13           @ set SVC mode
        orr     r0, r0, #0xc0           @ disable FIQ and IRQ
        msr     cpsr,r0
```

- Setup CP15 registers (cache, MMU, TLBs)
  - /arch/arm/cpu/armv7/start.S

- Setup important registers and memory timing
  - /arch/arm/cpu/armv7/lowlevel_init.S

First Stage of Boot Sequence(Assembly Language)

- Setup exception vectors
  - /arch/arm/lib/vectors.S
- Set cpu to SVC32 mode & disable FIQ and IRQ
  - /arch/arm/cpu/armv7/start.S
- Setup CP15 registers (cache, MMU, TLBs)
  - /arch/arm/cpu/armv7/start.S
- Setup important registers and memory timing
  - /arch/arm/cpu/armv7/lowlevel_init.S

```
/*************************************************************************
 *
 * cpu_init_cp15
 *
 * Setup CP15 registers (cache, MMU, TLBs). The I-cache is turned on unless
 * CONFIG_SYS_ICACHE_OFF is defined.
 *
 *************************************************************************/
ENTRY(cpu_init_cp15)
        /*
         * Invalidate L1 I/D
         */
        mov     r0, #                   @ set up for MCR
        mcr     p15, 0, r0, c8, c7, 0   @ invalidate TLBs
        mcr     p15, 0, r0, c7, c5, 0   @ invalidate icache
        mcr     p15, 0, r0, c7, c5, 6   @ invalidate BP array
        mcr     p15, 0, r0, c7, c10, 4  @ DSB
        mcr     p15, 0, r0, c7, c5, 4   @ ISB

        /*
         * disable MMU stuff and caches
         */
        mrc     p15, 0, r0, c1, c0, 0
        bic     r0, r0, #0x00002000     @ clear bits 13 (--V-)
        bic     r0, r0, #0x00000007     @ clear bits 2:0 (-CAM)
        orr     r0, r0, #0x00000002     @ set bit 1 (--A-) Align
        orr     r0, r0, #0x00000800     @ set bit 11 (Z---) BTB
#ifdef CONFIG_SYS_ICACHE_OFF
        bic     r0, r0, #0x00001000     @ clear bit 12 (I) I-cache
#else
        orr     r0, r0, #0x00001000     @ set bit 12 (I) I-cache
#endif
```

First Stage of Boot Sequence(Assembly Language)

- Setup exception vectors
  - /arch/arm/lib/vectors.S
- Set cpu to SVC32 mode & disable FIQ and IRQ
  - /arch/arm/cpu/armv7/start.S
- Setup CP15 registers (cache, MMU, TLBs)
  - /arch/arm/cpu/armv7/start.S
- Setup important registers and memory timing
  - /arch/arm/cpu/armv7/lowlevel_init.S

```
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
/*************************************************************************
 *
 * CPU_init_critical registers
 *
 * setup important registers
 * setup memory timing
 *
 *************************************************************************
ENTRY(cpu_init_crit)
        /*
         * Jump to board specific initialization...
         * The Mask ROM will have already initialized
         * basic memory. Go here to bump up clock rate and handle
         * wake up conditions.
         */
        b       lowlevel_init           @ go setup pll,mux,memory
ENDPROC(cpu_init_crit)
#endif
```

Second Stage of Boot Sequence(Assembly + C Language)

- Setup initial environment for calling board_init_f() (stack, GD structure and unavailable BSS)
  - /arch/arm/lib/crt0.S
- Call board_init_f() to prepare the hardware for execution from system RAM

(arch_cpu_init, board_early_init_f, timer_init, env_init, init_baud_rate, serial_init, etc.)

  - /common/board_f.c
- Setup intermediate environment where the stack and GD are the ones allocated by board_init_f() in system RAM
  - /arch/arm/lib/crt0.S
- Call relocate_code() to relocate U-Boot into destination computed by board_init_f()
  - /arch/arm/lib/relocate.S
- Setup final environment for calling board_init_r()
  - /arch/arm/lib/crt0.S
- Branch to board_init_r() to initialize GD structure and various peripherals, execute an endless main_loop() function

(initr_env, audio_add_devices, console_init_r, interrupt_init, initr_net, main_loop, etc)

  - /common/board_r.c

Second Stage of Boot Sequence(Assembly + C Language)

- Setup initial environment for calling board_init_f()

(stack, GD structure and unavailable BSS)

  – /arch/arm/lib/crt0.S

```
ENTRY(_main)

/*
 * Set up initial C runtime environment and call board_init_f(0).
 */

#if defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)
        ldr     sp, =(CONFIG_SPL_STACK)
#else
        ldr     sp, =(CONFIG_SYS_INIT_SP_ADDR)
#endif
        bic     sp, sp, #7      /* 8-byte alignment for ABI compliance */
        mov     r2, sp
        sub     sp, sp, #GD_SIZE        /* allocate one GD above SP */
        bic     sp, sp, #7      /* 8-byte alignment for ABI compliance */
        mov     r9, sp          /* GD is above SP */
        mov     r1, sp
        mov     r0, #0
clr_gd:
        cmp     r1, r2                  /* while not at end of GD */
        strlo   r0, [r1]                /* clear 32-bit GD word */
        addlo   r1, r1, #4              /* move to next */
        blo     clr_gd
#if defined(CONFIG_SYS_MALLOC_F_LEN)
        sub     sp, sp, #CONFIG_SYS_MALLOC_F_LEN
        str     sp, [r9, #GD_MALLOC_BASE]
#endif
        /* mov r0, #0 not needed due to above code */
        bl      board_init_f
```

Second Stage of Boot Sequence(Assembly + C Language)

- Call board_init_f() to prepare the hardware for execution from system RAM (arch_cpu_init, board_early_init_f, timer_init, env_init, init_baud_rate, serial_init, etc.)
  - /common/board_f.c
  - /lib/initcall.c

```c
void board_init_f(ulong boot_flags)
{
#ifdef CONFIG_SYS_GENERIC_GLOBAL_DATA
        /*
         * For some archtectures, global data is initialized and used before
         * calling this function. The data should be preserved. For others,
         * CONFIG_SYS_GENERIC_GLOBAL_DATA should be defined and use the stack
         * here to host global data until relocation.
         */
        gd_t data;

        gd = &data;

        /*
         * Clear global data before it is accessed at debug print
         * in initcall_run_list. Otherwise the debug print probably
         * get the wrong vaule of gd->have_console.
         */
        zero_global_data();
#endif

        gd->flags = boot_flags;
        gd->have_console = 0;

        if (initcall_run_list(init_sequence_f))
                hang();

#if !defined(CONFIG_ARM) && !defined(CONFIG_SANDBOX)
        /* NOTREACHED - jump_to_copy() does not return */
        hang();
#endif
}
```

```c
int initcall_run_list(const init_fnc_t init_sequence[])
{
        const init_fnc_t *init_fnc_ptr;

        for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
                unsigned long reloc_ofs = 0;
                int ret;

                if (gd->flags & GD_FLG_RELOC)
                        reloc_ofs = gd->reloc_off;
                debug("initcall: %p", (char *)*init_fnc_ptr - reloc_ofs);
                if (gd->flags & GD_FLG_RELOC)
                        debug(" (relocated to %p)\n", (char *)*init_fnc_ptr);
                else
                        debug("\n");
                ret = (*init_fnc_ptr)();
                if (ret) {
                        printf("initcall sequence %p failed at call %p (err=%d)\n",
                                init_sequence,
                                (char *)*init_fnc_ptr - reloc_ofs, ret);
                        return -1;
                }
        }
        return 0;
}
```

Second Stage of Boot Sequence(Assembly + C Language)

- Setup intermediate environment where the stack and GD are the ones allocated by board_init_f() in system RAM
  - /arch/arm/lib/crt0.S

```
#if ! defined(CONFIG_SPL_BUILD)

/*
 * Set up intermediate environment (new sp and gd) and call
 * relocate_code(addr_moni). Trick here is that we'll return
 * 'here' but relocated.
 */

        ldr     sp, [r9, #GD_START_ADDR_SP]     /* sp = gd->start_addr_sp */
        bic     sp, sp, #7      /* 8-byte alignment for ABI compliance */
        ldr     r9, [r9, #GD_BD]                /* r9 = gd->bd */
        sub     r9, r9, #GD_SIZE                /* new GD is below bd */

        adr     lr, here
        ldr     r0, [r9, #GD_RELOC_OFF]         /* r0 = gd->reloc_off */
        add     lr, lr, r0
        ldr     r0, [r9, #GD_RELOCADDR]         /* r0 = gd->relocaddr */
        b       relocate_code
```

Second Stage of Boot Sequence(Assembly + C Language)

- Call relocate_code() to relocate U-Boot into destination computed by board_init_f()
  - /arch/arm/lib/relocate.S

```
ENTRY(relocate_code)
        ldr     r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
        subs    r4, r0, r1              /* r4 <- relocation offset */
        beq     relocate_done           /* skip relocation */
        ldr     r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */

copy_loop:
        ldmia   r1!, {r10-r11}          /* copy from source address [r1]    */
        stmia   r0!, {r10-r11}          /* copy to   target address [r0]    */
        cmp     r1, r2                  /* until source end address [r2]    */
        blo     copy_loop

        /*
         * fix .rel.dyn relocations
         */
        ldr     r2, =__rel_dyn_start    /* r2 <- SRC &__rel_dyn_start */
        ldr     r3, =__rel_dyn_end      /* r3 <- SRC &__rel_dyn_end */
fixloop:
        ldmia   r2!, {r0-r1}            /* (r0,r1) <- (SRC location,fixup) */
        and     r1, r1, #0xff
        cmp     r1, #23                 /* relative fixup? */
        bne     fixnext

        /* relative fix: increase location by offset */
        add     r0, r0, r4
        ldr     r1, [r0]
        add     r1, r1, r4
        str     r1, [r0]
fixnext:
        cmp     r2, r3
        blo     fixloop

relocate_done:
```

Second Stage of Boot Sequence(Assembly + C Language)

- Setup final environment for calling board_init_r()
  - /arch/arm/lib/crt0.S

```
/* Set up final (full) environment */

        bl      c_runtime_cpu_setup     /* we still call old routine here */
#endif
#if !defined(CONFIG_SPL_BUILD) || defined(CONFIG_SPL_FRAMEWORK)
# ifdef CONFIG_SPL_BUILD
        /* Use a DRAM stack for the rest of SPL, if requested */
        bl      spl_relocate_stack_gd
        cmp     r0, #0
        movne   sp, r0
# endif
        ldr     r0, =__bss_start        /* this is auto-relocated! */

#ifdef CONFIG_USE_ARCH_MEMSET
        ldr     r3, =__bss_end          /* this is auto-relocated! */
        mov     r1, #0x00000000         /* prepare zero to clear BSS */

        subs    r2, r3, r0              /* r2 = memset len */
        bl      memset
#else
        ldr     r1, =__bss_end          /* this is auto-relocated! */
        mov     r2, #0x00000000         /* prepare zero to clear BSS */

clbss_l:cmp     r0, r1                  /* while not at end of BSS */
        strlo   r2, [r0]                /* clear 32-bit BSS word */
        addlo   r0, r0, #4              /* move to next */
        blo     clbss_l
#endif

#if ! defined(CONFIG_SPL_BUILD)
        bl coloured_LED_init
        bl red_led_on
#endif
```

Second Stage of Boot Sequence(Assembly + C Language)

- Branch to board_init_r() to initialize GD structure and various peripherals, execute an endless main_loop() function

(audio_add_devices, console_init_r, interrupt_init, initr_net, main_loop, etc)

  - /common/board_r.c
  - /lib/initcall.c

```c
void board_init_r(gd_t *new_gd, ulong dest_addr)
{
#ifdef CONFIG_NEEDS_MANUAL_RELOC
        int i;
#endif

#ifdef CONFIG_AVR32
        mmu_init_r(dest_addr);
#endif

#if !defined(CONFIG_X86) && !defined(CONFIG_ARM) && !defined(CONFIG_ARM64)
        gd = new_gd;
#endif

#ifdef CONFIG_NEEDS_MANUAL_RELOC
        for (i = 0; i < ARRAY_SIZE(init_sequence_r); i++)
                init_sequence_r[i] += gd->reloc_off;
#endif

        if (initcall_run_list(init_sequence_r))
                hang();

        /* NOTREACHED - run_main_loop() does not return */
        hang();
}
```

```c
int initcall_run_list(const init_fnc_t init_sequence[])
{
        const init_fnc_t *init_fnc_ptr;

        for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
                unsigned long reloc_ofs = 0;
                int ret;

                if (gd->flags & GD_FLG_RELOC)
                        reloc_ofs = gd->reloc_off;
                debug("initcall: %p", (char *)*init_fnc_ptr - reloc_ofs);
                if (gd->flags & GD_FLG_RELOC)
                        debug(" (relocated to %p)\n", (char *)*init_fnc_ptr);
                else
                        debug("\n");
                ret = (*init_fnc_ptr)();
                if (ret) {
                        printf("initcall sequence %p failed at call %p (err=%d)\n",
                                init_sequence,
                                (char *)*init_fnc_ptr - reloc_ofs, ret);
                        return -1;
                }
        }
        return 0;
}
```

Second Stage of Boot Sequence(Assembly + C Language)

- Branch to board_init_r() to initialize GD structure and various peripherals, execute an endless main_loop() function

(audio_add_devices, console_init_r, interrupt_init, initr_net, mian_loop, etc)

- /common/board_r.c
- /common/main.c

```c
static int run_main_loop(void)
{
#ifdef CONFIG_SANDBOX
        sandbox_main_loop_init();
#endif
        /* main_loop() can return to retry autoboot, if so just run it again */
        for (;;)
                main_loop();
        return 0;
}
```

```c
/* We come here after U-Boot is initialised and ready to process commands */
void main_loop(void)
{
        const char *s;

        bootstage_mark_name(BOOTSTAGE_ID_MAIN_LOOP, "main_loop");

#ifndef CONFIG_SYS_GENERIC_BOARD
        puts("Warning: Your board does not use generic board. Please read\n");
        puts("doc/README.generic-board and take action. Boards not\n");
        puts("upgraded by the late 2014 may break or be removed.\n");
#endif

        modem_init();
#ifdef CONFIG_VERSION_VARIABLE
        setenv("ver", version_string);  /* set version variable */
#endif /* CONFIG_VERSION_VARIABLE */

        cli_init();

        run_preboot_environment_command();

#if defined(CONFIG_UPDATE_TFTP)
        update_tftp(0UL);
#endif /* CONFIG_UPDATE_TFTP */

        s = bootdelay_process();
        if (cli_process_fdt(&s))
                cli_secure_boot_cmd(s);

        autoboot_command(s);

        cli_loop();
}
```

Second Stage of Boot Sequence(Assembly + C Language)

- Branch to board_init_r() to initialize GD structure and various peripherals, execute an endless main_loop() function

(audio_add_devices, console_init_r, interrupt_init, initr_net, main_loop, etc)

  - /common/board_r.c
  - /common/autoboot.c

```c
void autoboot_command(const char *s)
{
        debug("### main_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");

        if (stored_bootdelay != -1 && s && !abortboot(stored_bootdelay)) {
#if defined(CONFIG_AUTOBOOT_KEYED) && !defined(CONFIG_AUTOBOOT_KEYED_CTRLC)
                int prev = disable_ctrlc(1);    /* disable Control C checking */
#endif

                run_command_list(s, -1, 0);

#if defined(CONFIG_AUTOBOOT_KEYED) && !defined(CONFIG_AUTOBOOT_KEYED_CTRLC)
                disable_ctrlc(prev);    /* restore Control C checking */
#endif
        }

#ifdef CONFIG_MENUKEY
        if (menukey == CONFIG_MENUKEY) {
                s = getenv("menucmd");
                if (s)
                        run_command_list(s, -1, 0);
        }
#endif /* CONFIG_MENUKEY */
}
```

```c
static int abortboot(int bootdelay)
{
#ifdef CONFIG_AUTOBOOT_KEYED
        return abortboot_keyed(bootdelay);
#else
        return abortboot_normal(bootdelay);
#endif
}
```

Second Stage of Boot Sequence(Assembly + C Language)

- Branch to board_init_r() to initialize GD structure and various peripherals, execute an endless main_loop() function

(audio_add_devices, console_init_r, interrupt_init, initr_net, main_loop, etc)

- /common/board_r.c
- /common/autoboot.c

```c
static int abortboot_normal(int bootdelay)
{
        int abort = 0;
        unsigned long ts;

#ifdef CONFIG_MENUPROMPT
        printf(CONFIG_MENUPROMPT);
#else
        if (bootdelay >= 0)
                printf("Hit any key to stop autoboot: %2d ", bootdelay);
#endif

#if defined CONFIG_ZERO_BOOTDELAY_CHECK
        /*
         * Check if key already pressed
         * Don't check if bootdelay < 0
         */
        if (bootdelay >= 0) {
                if (tstc()) {   /* we got a key press   */
                        (void) getc();  /* consume input        */
                        puts("\b\b\b 0");
                        abort = 1;      /* don't auto boot      */
                }
        }
#endif
```

After several steps, u-boot execution is completed!

```
COM4 - PuTTY

U-Boot 2015.04-imx_v2015.04_4.1.15_1.0.0_ga+gd7d7c43 (Jun 17 2016 - 15:56:01)

CPU:   Freescale i.MX6Q rev1.2 at 792 MHz
CPU:   Temperature 45 C
Reset cause: unknown reset
Board: MX6-SabreSD
I2C:   ready
DRAM:  1 GiB
PMIC:  PFUZE100 ID=0x10
MMC:   FSL_SDHC: 0, FSL_SDHC: 1, FSL_SDHC: 2
No panel detected: default to Hannstar-XGA
Display: Hannstar-XGA (1024x768)
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc2(part 0) is current device
Net:   FEC [PRIME]
Normal Boot
Hit any key to stop autoboot:  0
switch to partitions #0, OK
mmc2(part 0) is current device
reading boot.scr
```