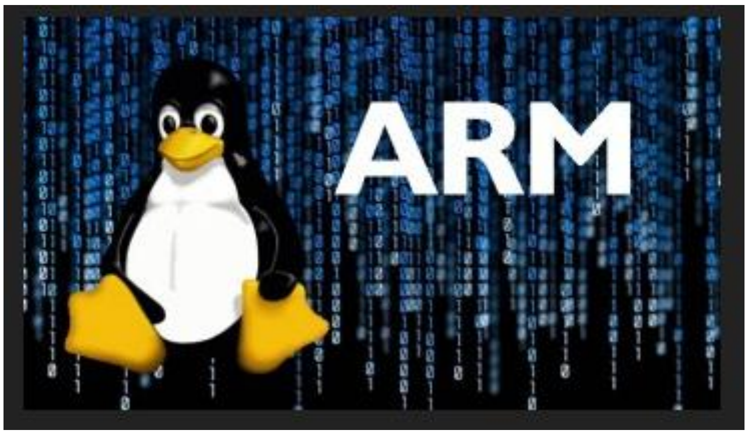
Mob: 91-9535072845 Email:info@zeelogic.com

--------------------------------------------------------------------------------------------------------------------------------------

# ARM Linux Booting Process



# ARM Linux Booting Process

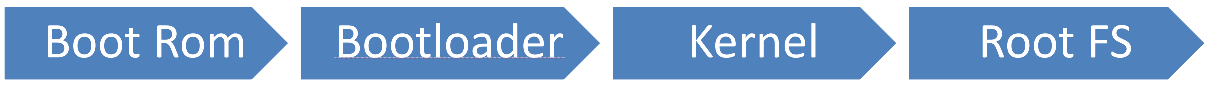
We will look at Linux boot process on ARM based SOC we will take AT91RM9200 system-on-chip, built around the ARM920T ARM Thumb processor.

You can have a look at AT91RM9200 data sheet (specification). You can download the data sheet with the following link. (Download)

1) T91RM9200 data sheet (specification) --->( Download )

2) ARM Architecture Reference Manual -->(Download )

## Linux boot sequence



### Boot loader

a) First Stage Boot Loaders

b) Second stage Bootloaders.

### Linux Kernel Boot Sequence

1) Architecture Dependent boot process

2) Architecture Independent Boot Process.

## Bootloader

The boot loader is a small application that installs the operating system or application

Onto a hardware target. The boot loader only exists up to the point that the operating

System or application is executing, and it is commonly incorporated into the firmware.

## What Happens Immediately After Reset on the Target Board?

When the processor has been reset, it will commence execution at the location of the reset vector within the exception vector table (at either address 0 or 0xFFFF0000). The reset handler code will need to do some, or all of the following:

- In a multi-processor system, put non-primary processors to sleep
- Initialize exception vectors
- Initialize the memory system, including the MMU
- Initialize processor mode stacks and registers
- Initialize variables required by C
- Initialize any critical I/O devices
- Perform any necessary initialization of NEON/VFP

- Enable interrupts
- Change processor mode and/or state
- Handle any set-up required for the Secure world
- Call the main() application.

Boot-loaders are highly processor specific and board specific. Boot-loader will be executed when power is applied to a processor board. Basically it will have some minimal features to load the image and boot it up.

## Joint Test Action Group (JTAG)

This interface may be used to write the boot loader program into boo-table non-volatile

Memory (e.g. flash) Flash Can be either (Nor/Nand Flash).

After system reset, the micro-controller begins to execute code programmed into its non-volatile memory, just like usual processors are using ROM's for booting. In many cases such

Interfaces are implemented by hardwired logic. In other cases such interfaces could be created by software running in integrated on-chip boot ROM from GPIO pins.

We will look at U-boot boot-loader. U-boot is the widely used boot-loader in embedded systems. I will explain code from the u-boot-2010.03 source. You can download U-boot

From the following site.

http://www.denx.de/wiki/U-Boot

## U-boot Source Download

**Command :-** # git clone git://git.denx.de/u-boot.git

## Download Cross Compile tool from website:-

http://www.codesourcery.com/sgpp/lite/arm/portal/package1787/public/arm-none-linux-gnueabi/arm-2007q3-51-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2

## Boot Loader Build Procedure:-

```
export PATH=/home/alex/toolchain/linux-x86/toolchain/arm-eabi-4.4.3/bin:$PATH

sudo make CROSS_COMPILE=arm-none-linux-gnueabi- omap3devkit8000_config

export PATH=/home/aleem/toolchain/linux-x86/toolchain/arm-eabi-4.4.3/bin:$PATH

ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-

make CROSS_COMPILE=arm-none-linux-gnueabi- mrproper

sudo make CROSS_COMPILE=arm-none-linux-gnueabi- omap3devkit8000_config
```

**< After Build Procedure Explain Linker script u-Boot.lds>**

/home/aleem/Devkit-8000/u-boot-1.3.3/board/omap3devkit8000/u-boot.lds

cpu/omap3/start.S

U-boot will be copied to flash disk or internal RAM( if it is less size) and will be

downloaded to RAM and will be executed when the board power is applied to the board. For this board(at91rm9200) the code is always downloaded from device address

0x0000_0000 to the address 0x0000_0000 of the SRAM after remap. That‘s why we have given the start address of the .text section as 0×00000000. If you want to load the code

anywhere in the RAM and want to execute U-boot you need to build you code as position independent code(PIC). Then the instructions addresses will be offset into to PC (cpu

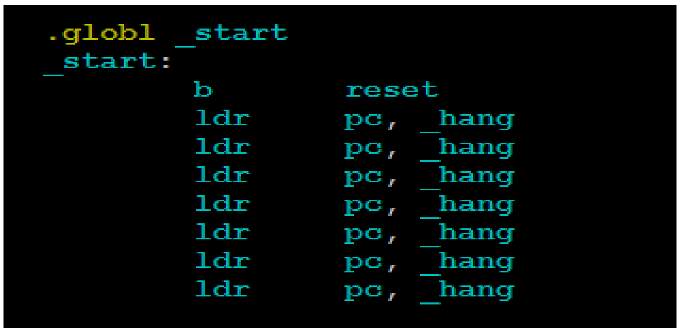
register) value. So the downloaded code must be position-independent or linked at address 0x0000_0000.

## U-Boot Execution

As specified in the linker script U-boot starting function (entry point) will be ***_start():***

**File : -** ***cpu/arm920t/start.S***

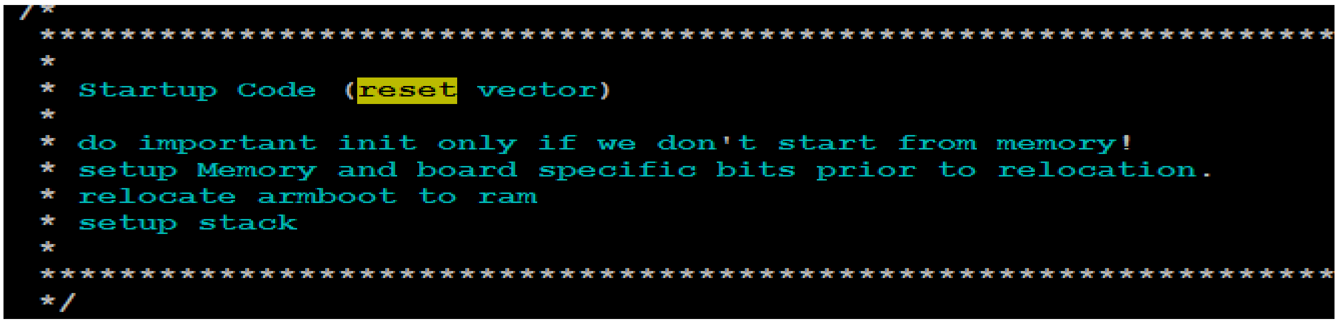
***cpu/arm920t/start.S. --> _start()***



```
 .globl _start
_start:
        b       reset
        ldr     pc, _hang
        ldr     pc, _hang
        ldr     pc, _hang
        ldr     pc, _hang
        ldr     pc, _hang
        ldr     pc, _hang
        ldr     pc, _hang
```

First instruction executed by ***_start()*** is a call ***to _start***: ***"cpu/arm920t/start.S"***.

Which will have a jump instruction to reset.



```
/*
 *******************************************************************
 *
 * Startup Code (reset vector)
 *
 * do important init only if we don't start from memory!
 * setup Memory and board specific bits prior to relocation.
 * relocate armboot to ram
 * setup stack
 *
 *******************************************************************
 */
```

```
/*
 * the actual reset code
 */

reset:
        /*
         * set the cpu to SVC32 mode
         */
        mrs     r0,cpsr             /* Load cpsr register into r0 */
        bic     r0,r0,#0x1f         /* Clear first 5 bits of cpsr(encode processor mode). */
        orr     r0,r0,#0xd3         /* supervisor mode (10011) + disable all interupts (110) */
        msr     cpsr,r0             /* write r0 to cpsr */
```

This code will perform the following task

1) Set the cpu in supervisor mode. The current operating processor status is in the Current Program Status Register (CPSR). The CPSR holds:

- Four ALU flags (Negative, Zero, Carry, and Overflow),
- Two interrupt disable bits (one for each type of interrupt (FIQ and IRQ),
- One bit to indicate ARM or Thumb execution
- Five bits to encode the current processor mode

Check ARM processor data sheet for more details on the CPSR register.

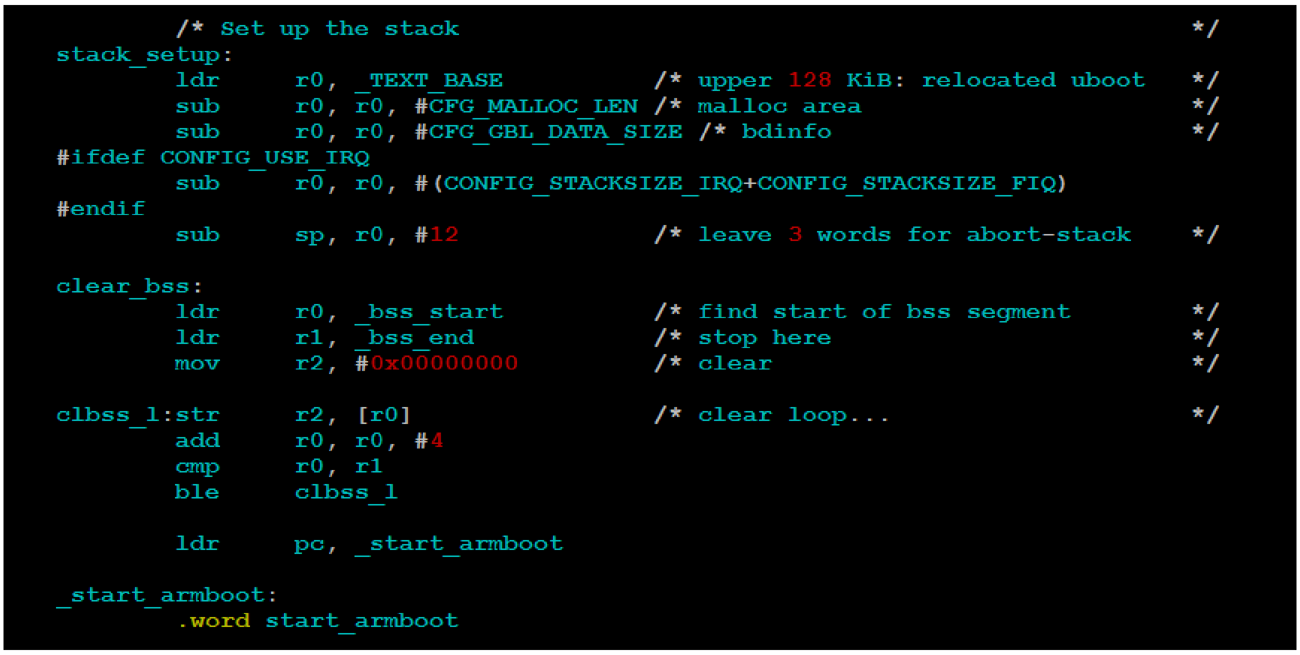
Try to relocate the U-boot code to RAM if we are running from flash.

```
#ifndef CONFIG_SKIP_RELOCATE_UBOOT
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
#endif  /* CONFIG_SKIP_RELOCATE_UBOOT */
```

➢ Setup the stack now. U-boot code has been relocated to ***_TEXT_BASE***

---

***(Defined in board/kb9202/configs.mk)***. Stack will be setup below this address.



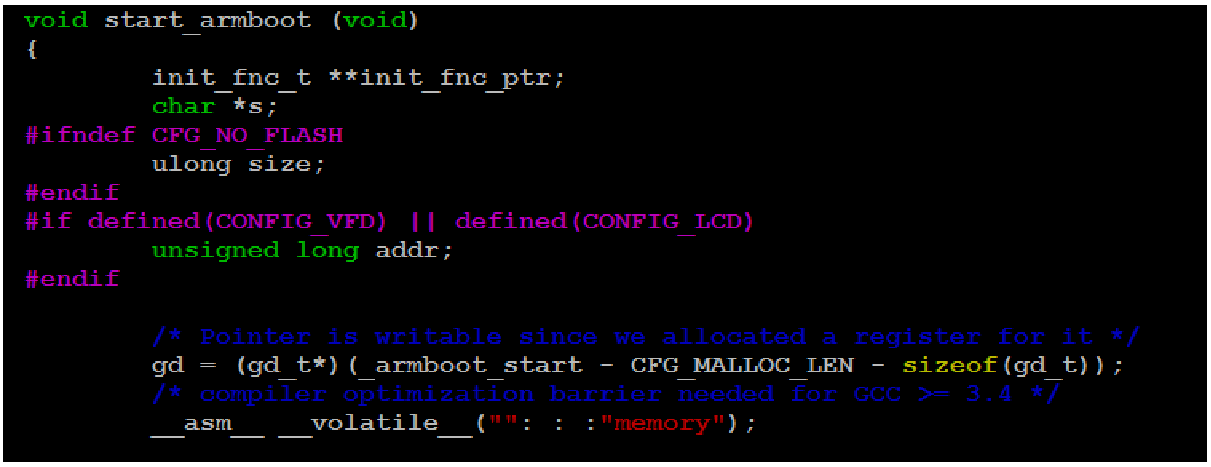
```
        /* Set up the stack                                         */
stack_setup:
        ldr     r0, _TEXT_BASE          /* upper 128 KiB: relocated uboot   */
        sub     r0, r0, #CFG_MALLOC_LEN /* malloc area                      */
        sub     r0, r0, #CFG_GBL_DATA_SIZE /* bdinfo                        */
#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
        ble     clbss_l

        ldr     pc, _start_armboot

_start_armboot:
        .word start_armboot
```

Call ***start_armboot*()**, a C function**.**

***start_armboot()*** function is defined in ***lib_arm/board.c*** which is common function for all arm based boards. The following are the tasks performed by ***start_armboot().***
***Void start_armboot(void).***

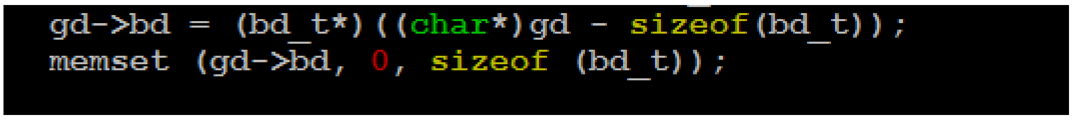
## Void start_armboot(void)

1) Allocate memory for a global data structure ***gd_t***. This is defined in
   ***include/asm-arm/global_data.h***.
   When we setup the stack we left some space for ***gd_t*** data structure
*( CONFIG_SYS_GBL_DATA_SIZE)* below the *_TEXT_BASE* where U-boot has been relocated.



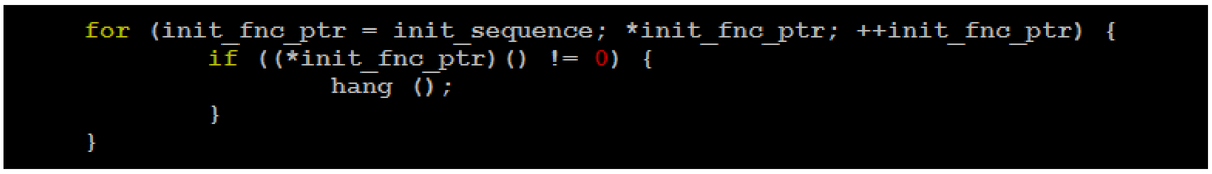
```
void start_armboot (void)
{
        init_fnc_t **init_fnc_ptr;
        char *s;
#ifndef CFG_NO_FLASH
        ulong size;
#endif
#if defined(CONFIG_VFD) || defined(CONFIG_LCD)
        unsigned long addr;
#endif

        /* Pointer is writable since we allocated a register for it */
        gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
        /* compiler optimization barrier needed for GCC >= 3.4 */
        __asm__ __volatile__("": : :"memory");
```

2) Some information like architecture number (unique board id), boot params that have to be passed to kernel image and baud rate etc are stored in a data structure called ***bd_t*** whose pointer is stored in ***gd_t***. Allocate memory for this ***bd_t*** after the ***gd_t***.



```
gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
memset (gd->bd, 0, sizeof (bd_t));
```

3) Call a set of functions which initialize all subsystems.

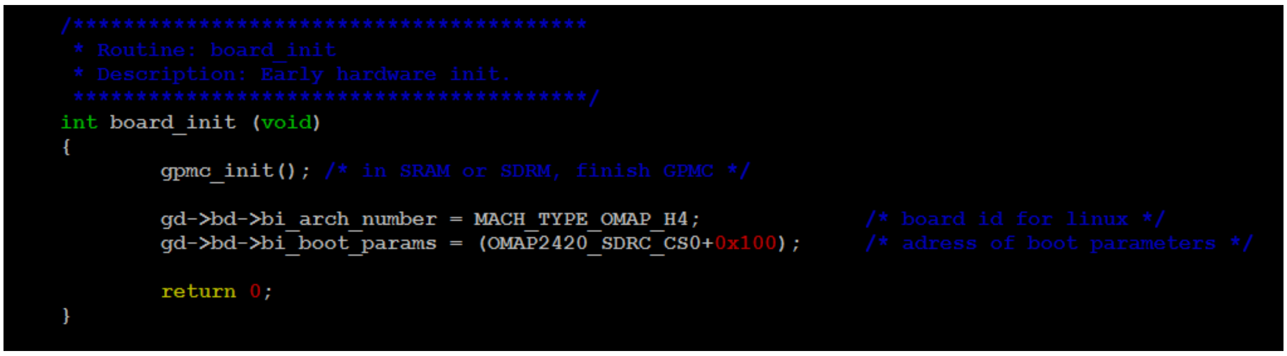


```
    for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
            if ((*init_fnc_ptr)() != 0) {
                    hang ();
            }
    }
```

init_sequence is an array of function pointers defined in ***lib_arm/board.c***. The above loop takes each function pointer and calls it.
The following are some of important functions called.

## 4) board_init():

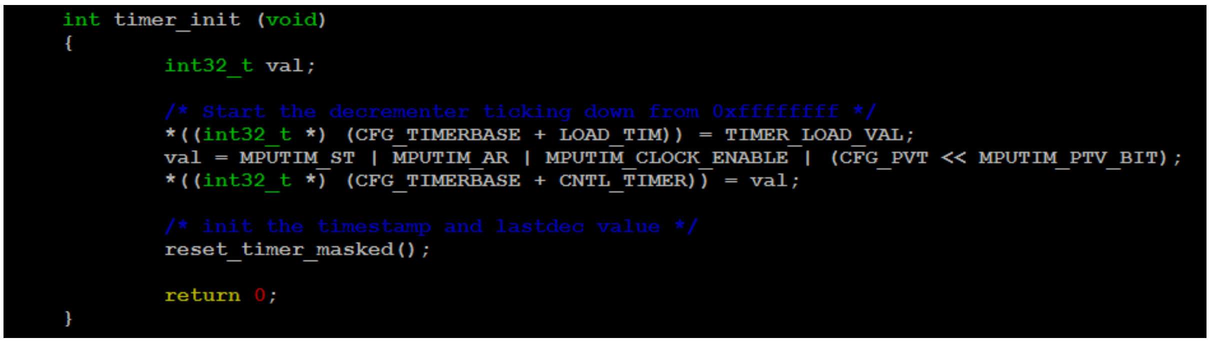
This is board specific function and should definitely be defined by each board. This function should some board specific initialization if there are any. When you are porting u-boot to a new board you must define this function. For **omap2420h4.c** board this function is defined < *"**board/omap2420h4/omap2420h4.c**"*>. This function just sets its board number and tells where the boot params for Linux are stored.



```
/*********************************************
 * Routine: board_init
 * Description: Early hardware init.
 *********************************************/
int board_init (void)
{
        gpmc_init(); /* in SRAM or SDRM, finish GPMC */

        gd->bd->bi_arch_number = MACH_TYPE_OMAP_H4;          /* board id for linux */
        gd->bd->bi_boot_params = (OMAP2420_SDRC_CS0+0x100);   /* adress of boot parameters */

        return 0;
}
```

## 5) timer_init():

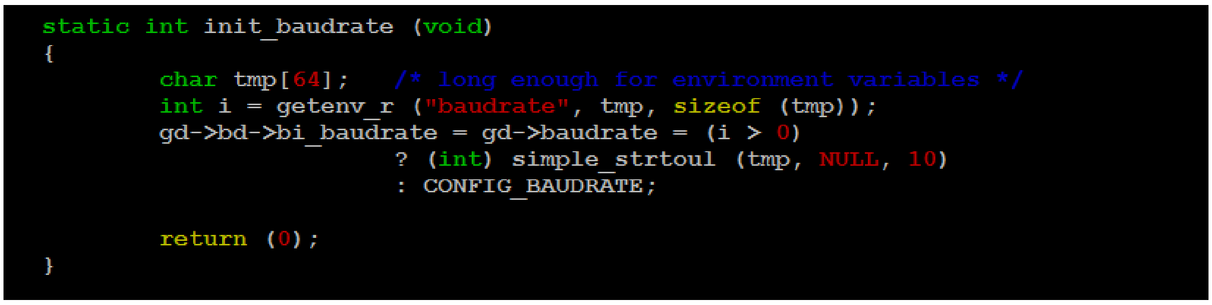
This is a cpu specific function and each cpu code must define it. It should basically initialize the timer services in the cpu. ***timer_init()*** for ***arm926ejs/cpu*** defined in ***"cpu/arm926ejs/omap/timer.c"***



```
int timer_init (void)
{
        int32_t val;

        /* Start the decrementer ticking down from 0xffffffff */
        *((int32_t *) (CFG_TIMERBASE + LOAD_TIM)) = TIMER_LOAD_VAL;
        val = MPUTIM_ST | MPUTIM_AR | MPUTIM_CLOCK_ENABLE | (CFG_PVT << MPUTIM_PTV_BIT);
        *((int32_t *) (CFG_TIMERBASE + CNTL_TIMER)) = val;

        /* init the timestamp and lastdec value */
        reset_timer_masked();

        return 0;
}
```

## init_baudrate():

This architecture specific function defines the default baud rate for the serial port communication. For ARM this function is defined in "***lib_arm/board.c***".



```
static int init_baudrate (void)
{
        char tmp[64];   /* long enough for environment variables */
        int i = getenv_r ("baudrate", tmp, sizeof (tmp));
        gd->bd->bi_baudrate = gd->baudrate = (i > 0)
                        ? (int) simple_strtoul (tmp, NULL, 10)
                        : CONFIG_BAUDRATE;

        return (0);
}
```

If the 'baudrate' enviromentvarable is set baud rate is taken from that, otherwise, taken from the CONFIG_BAUDRATE macro defined in board specific header file ***"include/configs/omap2420h4.h".***

#define CONFIG_BAUDRATE         115200

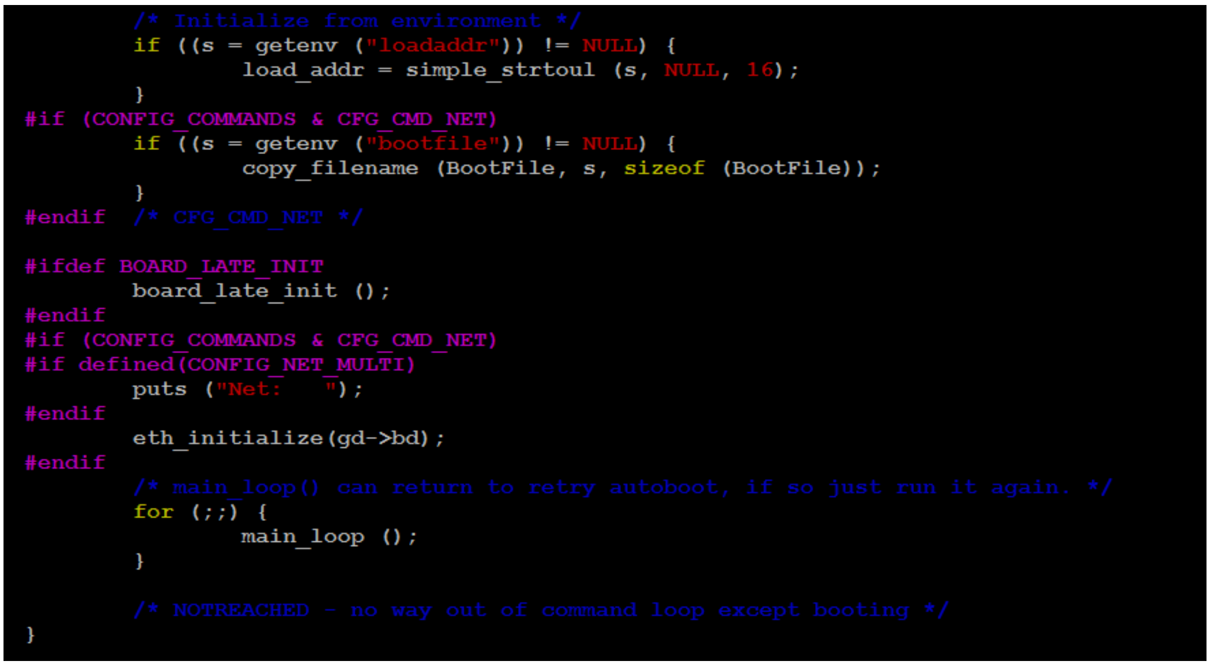
**serial_init():** ***"common/serial.c"***

This is a common function called to setup the serial port. This function internally calls cpu or board specific ***serial_init()*** function

```
int serial_init (void)
{
        if (!(gd->flags & GD_FLG_RELOC) || !serial_current) {
                struct serial_device *dev = default_serial_console ();

                return dev->init ();
        }

        return serial_current->init ();
}
```

As of now we did not relocate the u-boot code to ram and is running from flash. GD_FLG_RELOC will be set when the u-boot is relocated to RAM. serial_current will point to an object of type ***struct serial_device*** of current serial device that we are using. As we have not yet initialized any serial device serial_current will be NULL. Default ***_serial_console()*** is a function which points to ***__default_serial_console()*** defined in ***common/serial.c***.

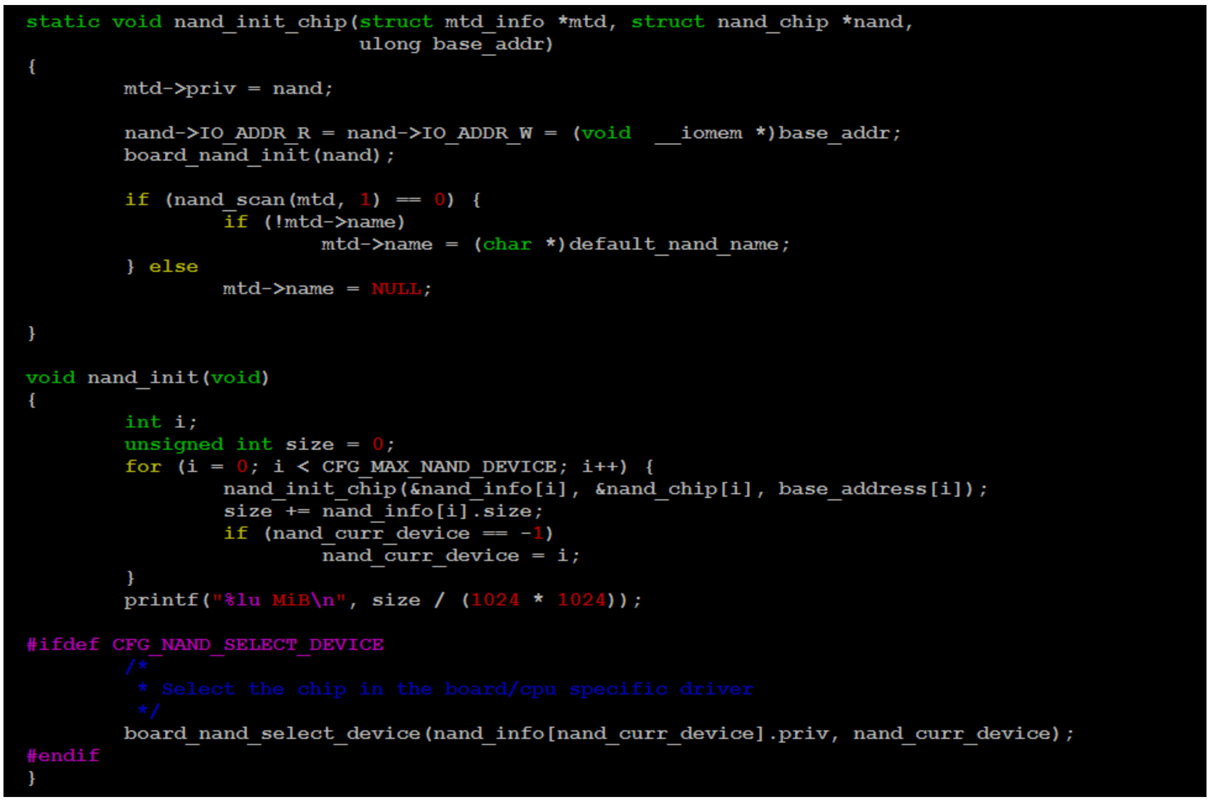
```
   void start_armboot (void)
   {
           init_fnc_t **init_fnc_ptr;
           char *s;
 #if (CONFIG_COMMANDS & CFG_CMD_NAND)
          puts ("NAND:  ");
          nand_init();            /* go init the NAND */
 #endif

 #ifdef CONFIG_HAS_DATAFLASH
          AT91F_DataflashInit();
          dataflash_print_info();
 #endif

          /* initialize environment */
          env_relocate ();

          devices_init ();        /* get the devices list going. */

 #ifdef CONFIG_CMC_PU2
          load_sernum_ethaddr ();
 #endif /* CONFIG_CMC_PU2 */

          jumptable_init ();

          console_init_r ();      /* fully init console as a device */

 #if defined(CONFIG_MISC_INIT_R)
          /* miscellaneous platform dependent initialisations */
          misc_init_r ();
 #endif

          /* enable exceptions */
          enable_interrupts ();
```



```
        /* Initialize from environment */
        if ((s = getenv ("loadaddr")) != NULL) {
                load_addr = simple_strtoul (s, NULL, 16);
        }
#if (CONFIG_COMMANDS & CFG_CMD_NET)
        if ((s = getenv ("bootfile")) != NULL) {
                copy_filename (BootFile, s, sizeof (BootFile));
        }
#endif  /* CFG_CMD_NET */

#ifdef BOARD_LATE_INIT
        board_late_init ();
#endif
#if (CONFIG_COMMANDS & CFG_CMD_NET)
#if defined(CONFIG_NET_MULTI)
        puts ("Net:   ");
#endif
        eth_initialize(gd->bd);
#endif
        /* main_loop() can return to retry autoboot, if so just run it again. */
        for (;;) {
                main_loop ();
        }

        /* NOTREACHED - no way out of command loop except booting */
}
```

## 6) Initialize NAND if configured.

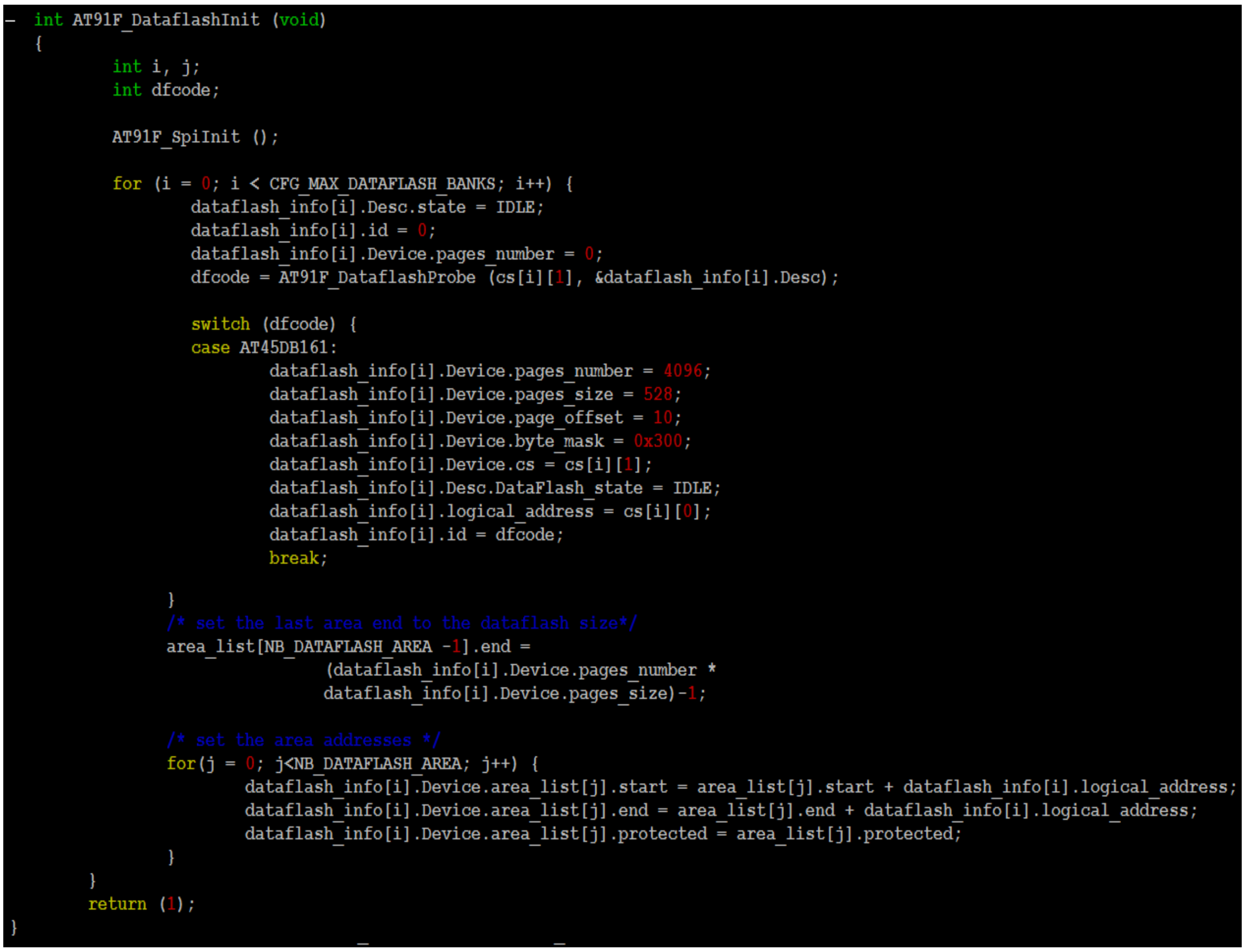
***nand_init()*** function is defined in ***drivers/nand/nand.c***



```
static void nand_init_chip(struct mtd_info *mtd, struct nand_chip *nand,
                           ulong base_addr)
{
        mtd->priv = nand;

        nand->IO_ADDR_R = nand->IO_ADDR_W = (void  __iomem *)base_addr;
        board_nand_init(nand);

        if (nand_scan(mtd, 1) == 0) {
                if (!mtd->name)
                        mtd->name = (char *)default_nand_name;
        } else
                mtd->name = NULL;

}

void nand_init(void)
{
        int i;
        unsigned int size = 0;
        for (i = 0; i < CFG_MAX_NAND_DEVICE; i++) {
                nand_init_chip(&nand_info[i], &nand_chip[i], base_address[i]);
                size += nand_info[i].size;
                if (nand_curr_device == -1)
                        nand_curr_device = i;
        }
        printf("%lu MiB\n", size / (1024 * 1024));

#ifdef CFG_NAND_SELECT_DEVICE
        /*
         * Select the chip in the board/cpu specific driver
         */
        board_nand_select_device(nand_info[nand_curr_device].priv, nand_curr_device);
#endif
}
```

**7)** Initialize if dataflash is configured

**#ifdef CONFIG_HAS_DATAFLASH**



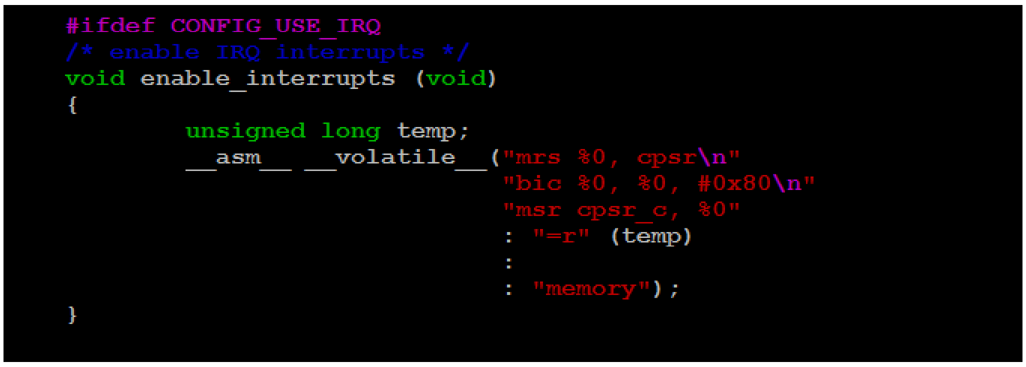
```
- int AT91F_DataflashInit (void)
  {
          int i, j;
          int dfcode;

          AT91F_SpiInit ();

          for (i = 0; i < CFG_MAX_DATAFLASH_BANKS; i++) {
                  dataflash_info[i].Desc.state = IDLE;
                  dataflash_info[i].id = 0;
                  dataflash_info[i].Device.pages_number = 0;
                  dfcode = AT91F_DataflashProbe (cs[i][1], &dataflash_info[i].Desc);

                  switch (dfcode) {
                  case AT45DB161:
                          dataflash_info[i].Device.pages_number = 4096;
                          dataflash_info[i].Device.pages_size = 528;
                          dataflash_info[i].Device.page_offset = 10;
                          dataflash_info[i].Device.byte_mask = 0x300;
                          dataflash_info[i].Device.cs = cs[i][1];
                          dataflash_info[i].Desc.DataFlash_state = IDLE;
                          dataflash_info[i].logical_address = cs[i][0];
                          dataflash_info[i].id = dfcode;
                          break;

                }
                /* set the last area end to the dataflash size*/
                area_list[NB_DATAFLASH_AREA -1].end =
                                (dataflash_info[i].Device.pages_number *
                                dataflash_info[i].Device.pages_size)-1;

                /* set the area addresses */
                for(j = 0; j<NB_DATAFLASH_AREA; j++) {
                        dataflash_info[i].Device.area_list[j].start = area_list[j].start + dataflash_info[i].logical_address;
                        dataflash_info[i].Device.area_list[j].end = area_list[j].end + dataflash_info[i].logical_address;
                        dataflash_info[i].Device.area_list[j].protected = area_list[j].protected;
                }
        }
        return (1);
}
```

8) Get all input and output devices list.
   ***stdio_init ();*** /* get the devices list */
   ***stdio_init()*** is defined in common/stdio.c.

9) Call ***console_init_r ()*** to setup the console info like where the input(stdin) should be taken and where the output(stdout) to be returned.
   ***console_init_r ()***;     /* fully init console as a device */

## 10 ) Enable the interrupts " *enable_interrupts ()* "

For arm boards is defined in ***lib_arm/interrupts.c***. Writing 0×80 into cpsr register will enable the interrupts. ***enable_interrupts()*** is the following assembly code**.<*cpu/arm926ejs/interrupts.c*>**



```
#ifdef CONFIG_USE_IRQ
/* enable IRQ interrupts */
void enable_interrupts (void)
{
        unsigned long temp;
        __asm__ __volatile__("mrs %0, cpsr\n"
                             "bic %0, %0, #0x80\n"
                             "msr cpsr_c, %0"
                             : "=r" (temp)
                             :
                             : "memory");
}
```

**11)** Get the 'loadaddr' environment variable defined in u-boot through setenv. Kernel image is loaded at the load address.

```
        /* Initialize from environment */
        if ((s = getenv ("loadaddr")) != NULL) {
                load_addr = simple_strtoul (s, NULL, 16);
        }
```

**12)** All initialization has been done, now enter a main loop where we accept commands from the user and execute them.

```
        /* main_loop() can return to retry autoboot, if so just run it again. */
        for (;;) {
                main_loop ();
        }

        /* NOTREACHED - no way out of command loop except booting */
}
```

***main_loop()*** function is defined in "***common/main.c".*** This function is common for all boards When the U-boot is booting up it will take a default command that is given in the U-boot environment variable *bootcmd* and executes that command. If this variable is not defined U-boot will provide U-boot prompt where user can enter the commands. U-boot can be made to go to its boot prompt even if the *bootcmd* is defined by giving some bootdelay.
U-boot waits until the bootdelay expires or user presses any key. If user does not press any key U-boot will execute the default command defined in *bootcmd* else U-boot goes to its prompt.

```
    void main_loop (void)
    {
#ifdef CONFIG_BOOTCOUNT_LIMIT
        if (bootlimit && (bootcount > bootlimit)) {
                printf ("Warning: Bootlimit (%u) exceeded. Using altbootcmd.\n",
                        (unsigned)bootlimit);
                s = getenv ("altbootcmd");
        }
        else
#endif /* CONFIG_BOOTCOUNT_LIMIT */
                s = getenv ("bootcmd");

        debug ("### main_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");

        if (bootdelay >= 0 && s && !abortboot (bootdelay)) {
         for (;;) {
 #ifdef CONFIG_BOOT_RETRY_TIME
                if (rc >= 0) {
                        /* Saw enough of a valid command to
                         * restart the timeout.
                         */
                        reset_cmd_timeout();
                }
 #endif
                len = readline (CFG_PROMPT);

                 if (len == -1)
                        puts ("<INTERRUPT>\n");
                 else
                        rc = run_command (lastcommand, flag);

                 if (rc <= 0) {
                        /* invalid command or not repeatable, forget it */
                        lastcommand[0] = 0;
                 }
          }
  #endif /*CFG_HUSH_PARSER*/
  }
```

If U-boot is interrupted by user by pressing a key U-boot enters into an infinite while loop and accepts user commands and executes them.

CONFIG_BOOTDELAY is the number of seconds the u-boot should wait for user interrupt before it takes its default action. CONFIG_BOOTDELAY is defined in the board specific file header file, in this case in ***"include/configs/omap2420h4.h"***.

As we have explained each U-boot command is an object of type struct cmd_tbl_s. The command name and function to be executed will be stored in this structure. All Commands structures are kept in memory at particular memroy, **__u_boot_cmd_start** and ***__u_boot_cmd_end*** contain start and end address of this memory section called command table.
***run_command()*** function takes the command name and finds the data structure belonging to this command in the command table and calls the corresponding command function. Let's assume that *bootcmd* is not configured and U-boot provided the prompt to enter commands. And also assume that the linux image is loaded into the ram in a perticular address using tftpboot command or using some ymodem command and *bootm* command is given at the U-boot prompt.

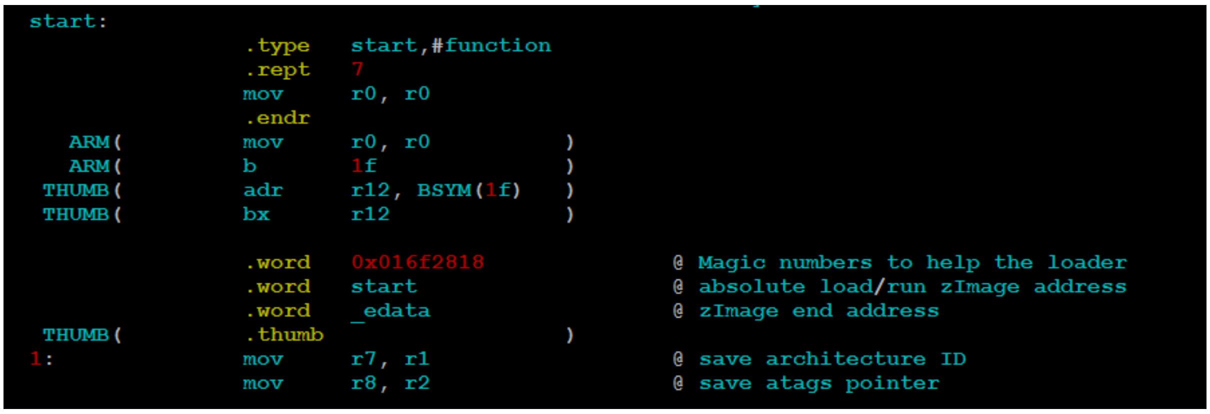
kb-9202# bootm 0×00280000
The function do_bootm() that is responsible for *executing bootm command. Bootm* loads kernel image. The function do_bootm() that is responsible for *executing bootm command. bootm*loads kernel image.

## kernel image (zImage) decompression

- ***arch/arm/boot/compressed/head.S***

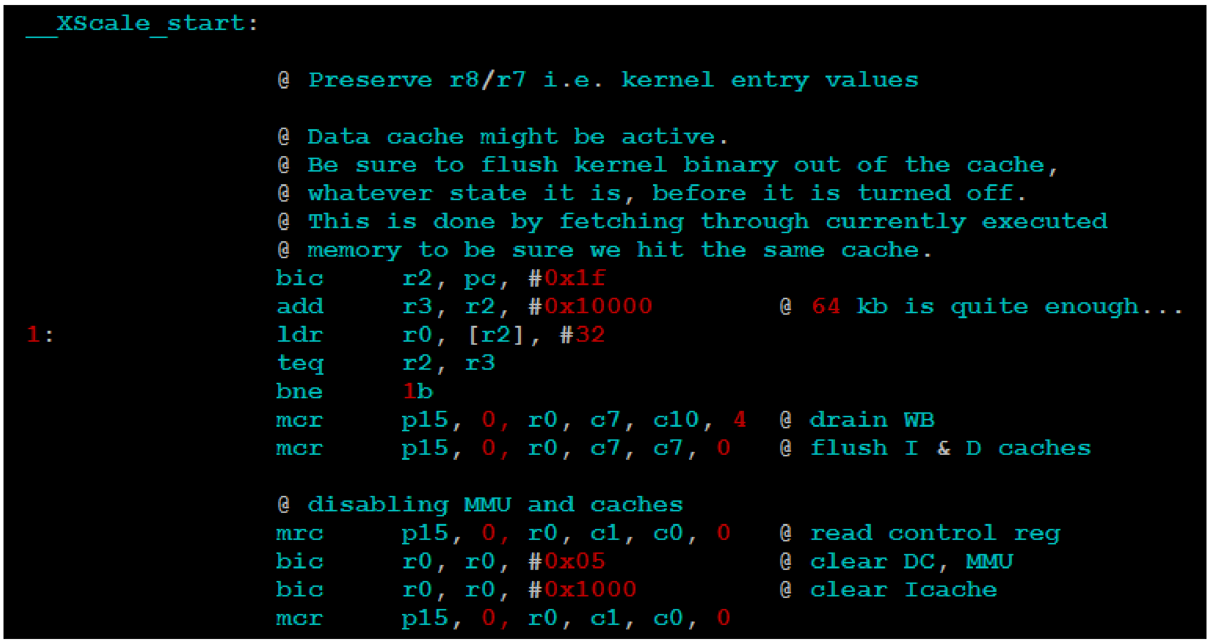
- First code executed, jumped to by the bootloader, at label "start"
- Save contents of registers r1 and r2 in r7 and r8 to save off architecture ID and atags pointer passed in by bootloader.

File :- "***arch/arm/boot/compressed/head.S***"



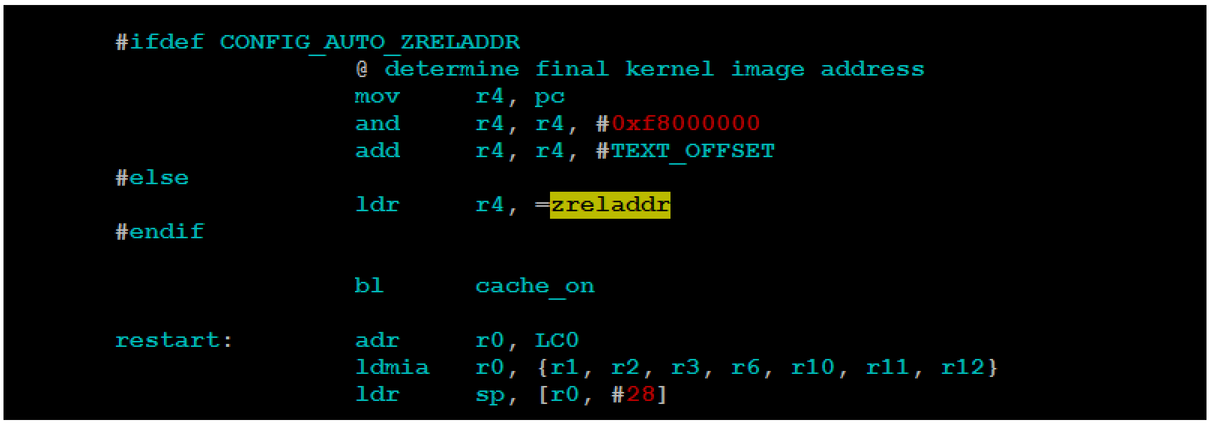
```
start:
                .type   start,#function
                .rept   7
                mov     r0, r0
                .endr
   ARM(         mov     r0, r0          )
   ARM(         b       1f              )
 THUMB(         adr     r12, BSYM(1f)   )
 THUMB(         bx      r12             )

                .word   0x016f2818              @ Magic numbers to help the loader
                .word   start                   @ absolute load/run zImage address
                .word   _edata                  @ zImage end address
 THUMB(         .thumb                  )
1:              mov     r7, r1                  @ save architecture ID
                mov     r8, r2                  @ save atags pointer
```

- execute arch-specific code
  - ***arch/arm/boot/compressed/head-xscale.S*** or other arch-specific code file
  - added to build in *"**arch/arm/boot/compressed/Makefile".***
  - linked into ***head.S*** by linker section declaration: .section "start"
  - flush cache, turn off cache and MMU


```
__XScale_start:

                @ Preserve r8/r7 i.e. kernel entry values

                @ Data cache might be active.
                @ Be sure to flush kernel binary out of the cache,
                @ whatever state it is, before it is turned off.
                @ This is done by fetching through currently executed
                @ memory to be sure we hit the same cache.
                bic     r2, pc, #0x1f
                add     r3, r2, #0x10000        @ 64 kb is quite enough...
1:              ldr     r0, [r2], #32
                teq     r2, r3
                bne     1b
                mcr     p15, 0, r0, c7, c10, 4  @ drain WB
                mcr     p15, 0, r0, c7, c7, 0   @ flush I & D caches

                @ disabling MMU and caches
                mrc     p15, 0, r0, c1, c0, 0   @ read control reg
                bic     r0, r0, #0x05           @ clear DC, MMU
                bic     r0, r0, #0x1000         @ clear Icache
                mcr     p15, 0, r0, c1, c0, 0
```

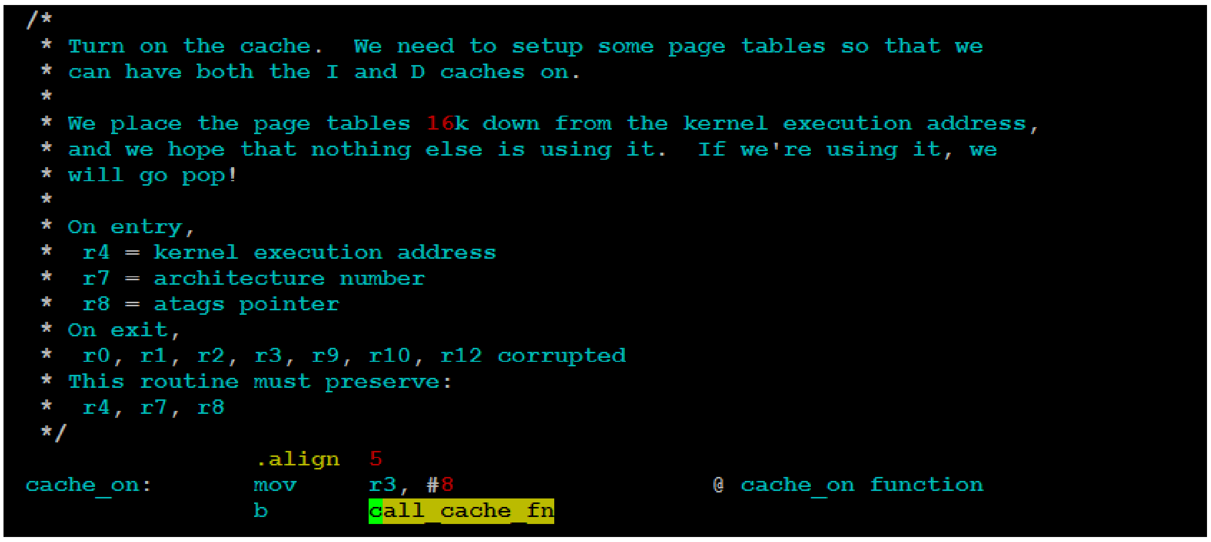
- load registers with stored parameters
  - sp = stack pointer for decompression code
  - r4 = zreladdr = kernel entry point physical address.

Check if running at link address, and fix up global offset table if not


```
#ifdef CONFIG_AUTO_ZRELADDR
                @ determine final kernel image address
                mov     r4, pc
                and     r4, r4, #0xf8000000
                add     r4, r4, #TEXT_OFFSET
#else
                ldr     r4, =zreladdr
#endif

                bl      cache_on

restart:        adr     r0, LC0
                ldmia   r0, {r1, r2, r3, r6, r10, r11, r12}
                ldr     sp, [r0, #28]
```

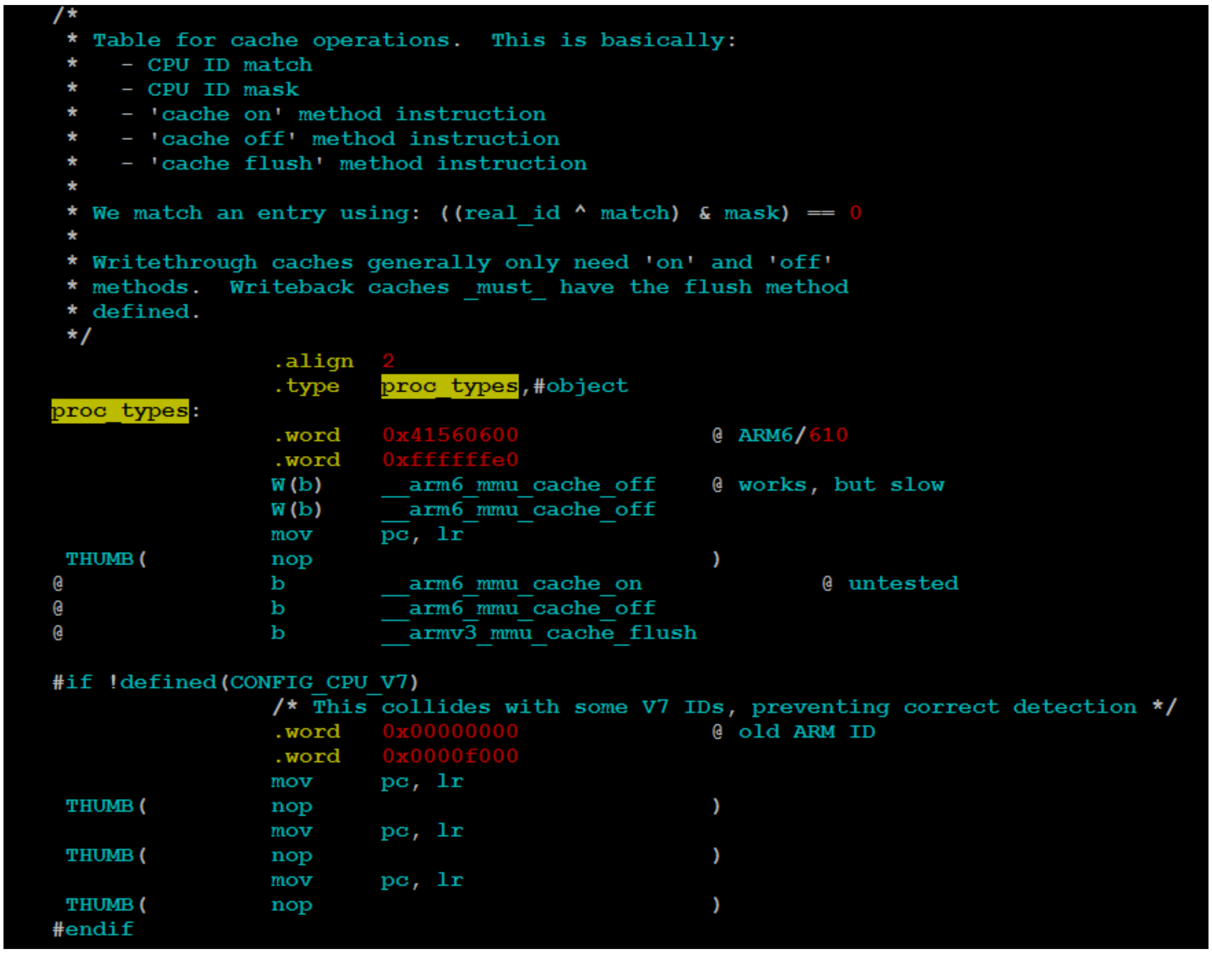
- zero decompression bss
- call ***cache_on*** to turn on cache
- defined at ***arch/arm/boot/compressed/head.S***
- call ***call_cache_fn*** to turn on cache as appropriate for processor variant.

Defined at **File: -** ***"arch/arm/boot/compressed/head.S"***



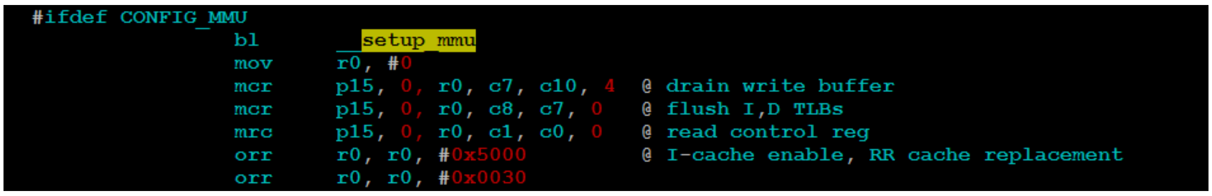
```
/*
 * Turn on the cache.  We need to setup some page tables so that we
 * can have both the I and D caches on.
 *
 * We place the page tables 16k down from the kernel execution address,
 * and we hope that nothing else is using it.  If we're using it, we
 * will go pop!
 *
 * On entry,
 *  r4 = kernel execution address
 *  r7 = architecture number
 *  r8 = atags pointer
 * On exit,
 *  r0, r1, r2, r3, r9, r10, r12 corrupted
 * This routine must preserve:
 *  r4, r7, r8
 */
                .align  5
cache_on:       mov     r3, #8                  @ cache_on function
                b       call_cache_fn
```

- Walk through proc_types list  until find corresponding processor
- call cache-on function in list item corresponding to processor
- For ARMv5tej core, ***cache_on*** function is ***__armv4_mmu_cache_on***.

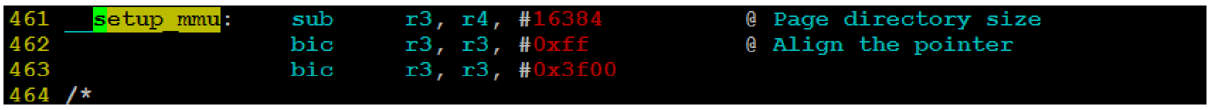


```
/*
 * Table for cache operations.  This is basically:
 *   - CPU ID match
 *   - CPU ID mask
 *   - 'cache on' method instruction
 *   - 'cache off' method instruction
 *   - 'cache flush' method instruction
 *
 * We match an entry using: ((real_id ^ match) & mask) == 0
 *
 * Writethrough caches generally only need 'on' and 'off'
 * methods.  Writeback caches _must_ have the flush method
 * defined.
 */
                .align  2
                .type   proc_types,#object
proc_types:
                .word   0x41560600              @ ARM6/610
                .word   0xffffffe0
                W(b)    __arm6_mmu_cache_off    @ works, but slow
                W(b)    __arm6_mmu_cache_off
                mov     pc, lr
 THUMB(         nop                             )
@               b       __arm6_mmu_cache_on             @ untested
@               b       __arm6_mmu_cache_off
@               b       __armv3_mmu_cache_flush

#if !defined(CONFIG_CPU_V7)
                /* This collides with some V7 IDs, preventing correct detection */
                .word   0x00000000              @ old ARM ID
                .word   0x0000f000
                mov     pc, lr
 THUMB(         nop                             )
                mov     pc, lr
 THUMB(         nop                             )
                mov     pc, lr
 THUMB(         nop                             )
#endif
```

- Call ***setup_mmu*** to set up initial page tables since MMU must be on for cache to be on
- turn on cache and MMU

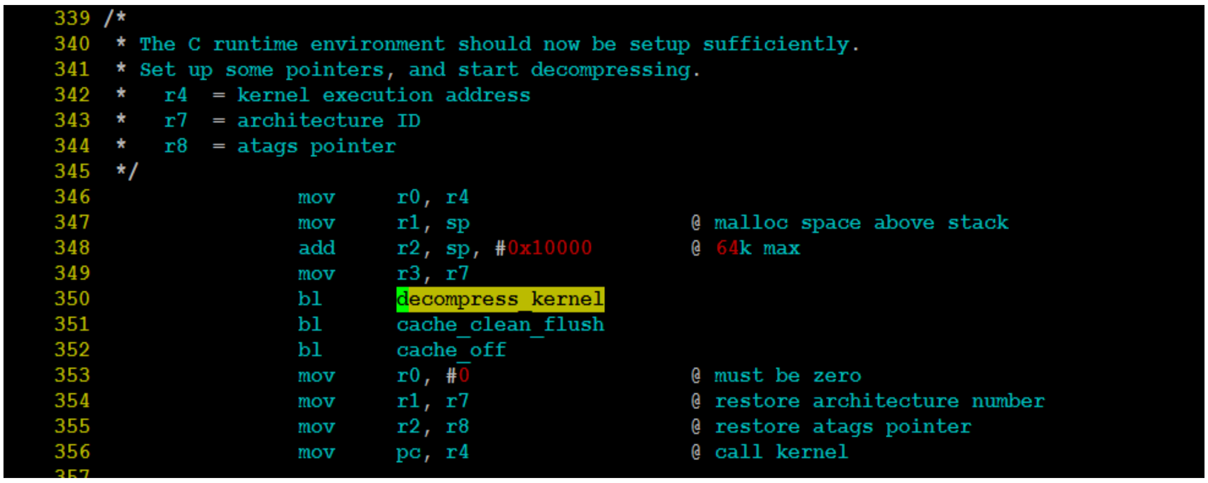


```
#ifdef CONFIG_MMU
                bl      __setup_mmu
                mov     r0, #0
                mcr     p15, 0, r0, c7, c10, 4  @ drain write buffer
                mcr     p15, 0, r0, c8, c7, 0   @ flush I,D TLBs
                mrc     p15, 0, r0, c1, c0, 0   @ read control reg
                orr     r0, r0, #0x5000         @ I-cache enable, RR cache replacement
                orr     r0, r0, #0x0030
```



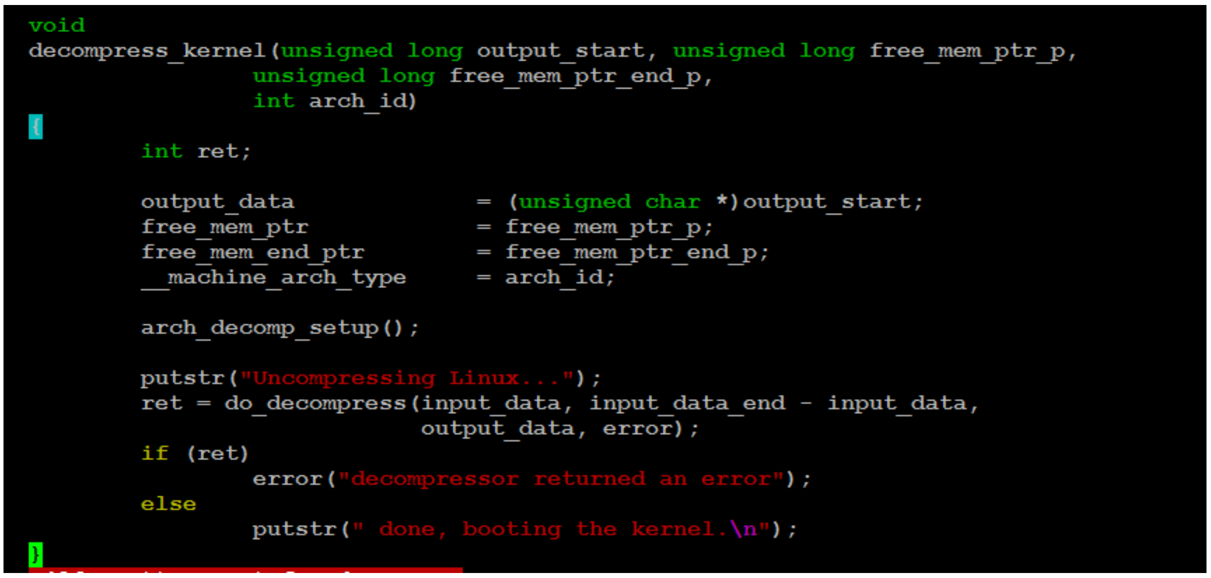
```
__setup_mmu:    sub     r3, r4, #16384          @ Page directory size
                bic     r3, r3, #0xff           @ Align the pointer
                bic     r3, r3, #0x3f00
/*
```

- ➢ Check to make sure won't overwrite image during decompression; assume not for this trace
- ➢ Call ***decompress_kernel*** to decompress kernel to RAM.



```
/*
 * The C runtime environment should now be setup sufficiently.
 * Set up some pointers, and start decompressing.
 *   r4  = kernel execution address
 *   r7  = architecture ID
 *   r8  = atags pointer
 */
                mov     r0, r4
                mov     r1, sp                  @ malloc space above stack
                add     r2, sp, #0x10000        @ 64k max
                mov     r3, r7
                bl      decompress_kernel
                bl      cache_clean_flush
                bl      cache_off
                mov     r0, #0                  @ must be zero
                mov     r1, r7                  @ restore architecture number
                mov     r2, r8                  @ restore atags pointer
                mov     pc, r4                  @ call kernel
```

***"arch/arm/boot/compressed/misc.c"***



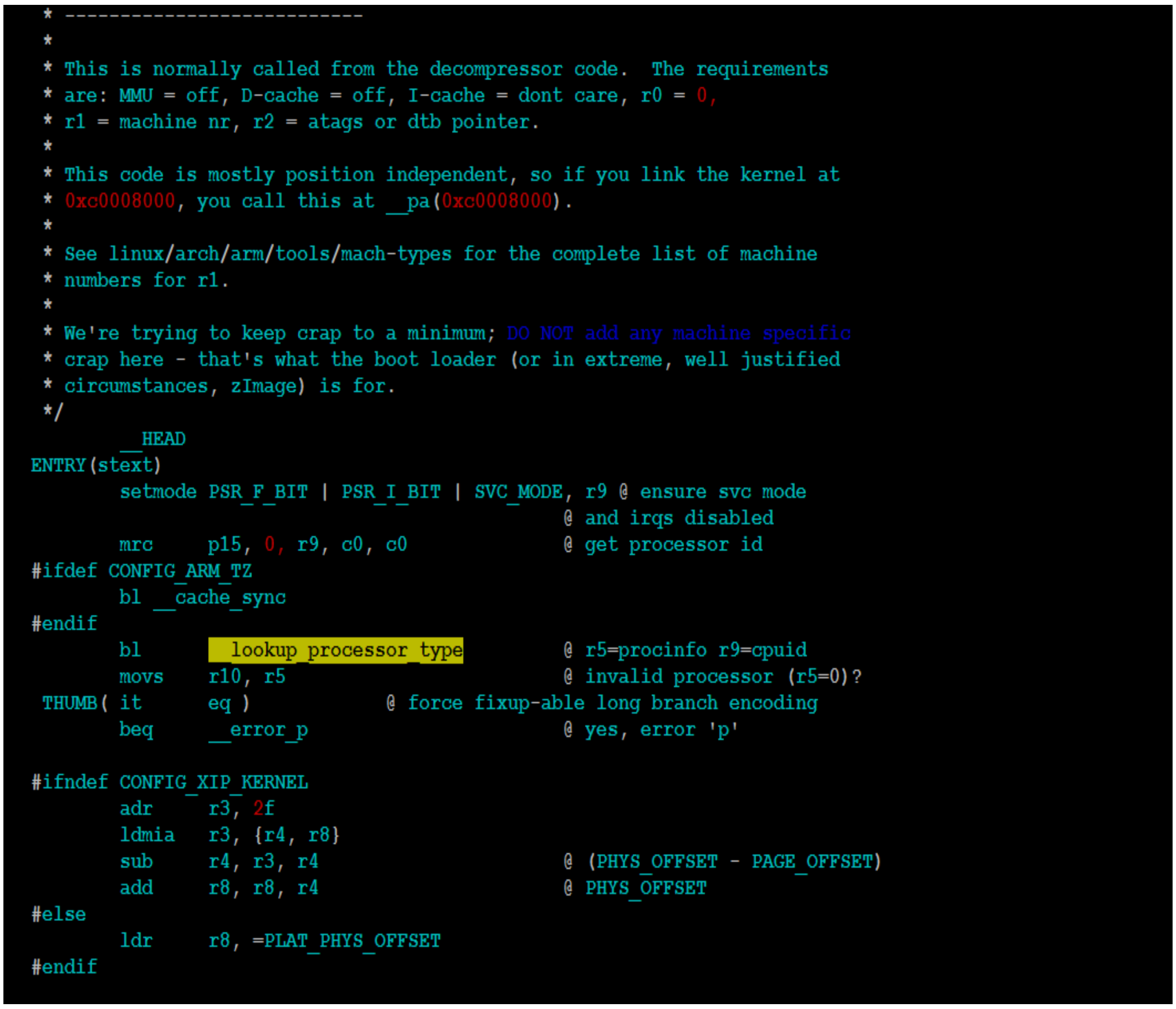
```
void
decompress_kernel(unsigned long output_start, unsigned long free_mem_ptr_p,
                unsigned long free_mem_ptr_end_p,
                int arch_id)
{
        int ret;

        output_data             = (unsigned char *)output_start;
        free_mem_ptr            = free_mem_ptr_p;
        free_mem_end_ptr        = free_mem_ptr_end_p;
        __machine_arch_type     = arch_id;

        arch_decomp_setup();

        putstr("Uncompressing Linux...");
        ret = do_decompress(input_data, input_data_end - input_data,
                            output_data, error);
        if (ret)
                error("decompressor returned an error");
        else
                putstr(" done, booting the kernel.\n");
}
```

- ➢ branch to **call_kernel**
- Call ***cache_clean_flush*** to flush cache contents to RAM
- Call cache_off to turn cache off as expected by kernel initialization routines
- Jump to start of kernel in RAM
- Jump to address in r4 = zreladdr from previous load
  - zreladdr = ZRELADDR = zreladdr-y
  - zreladdr-y specified ***in arch/arm/mach-omap2/Makefile.boot***
  - (zreladdr-y     := 0x80008000   )

# ARM-specific kernel code

**File :-** ***"arch/arm/kernel/head.S"***

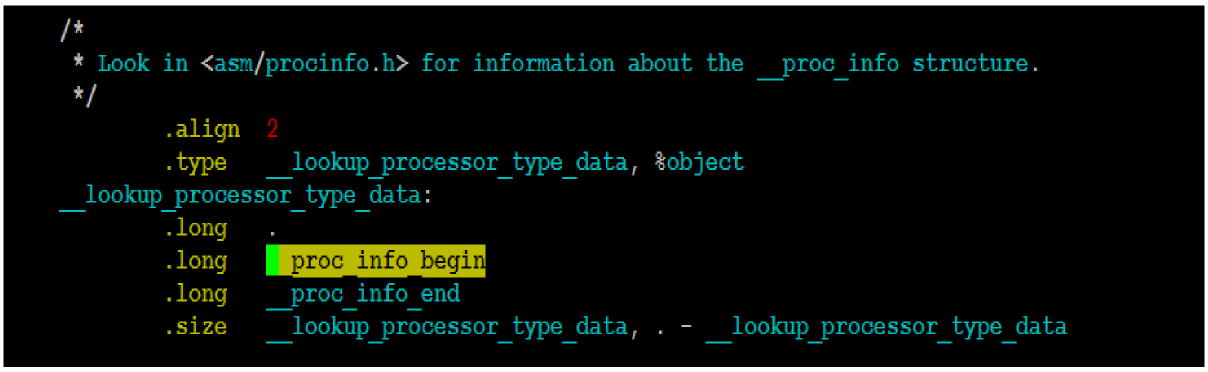
call ***__lookup_processor_type*** defined in “***arch/arm/kernel/head-common.S”***



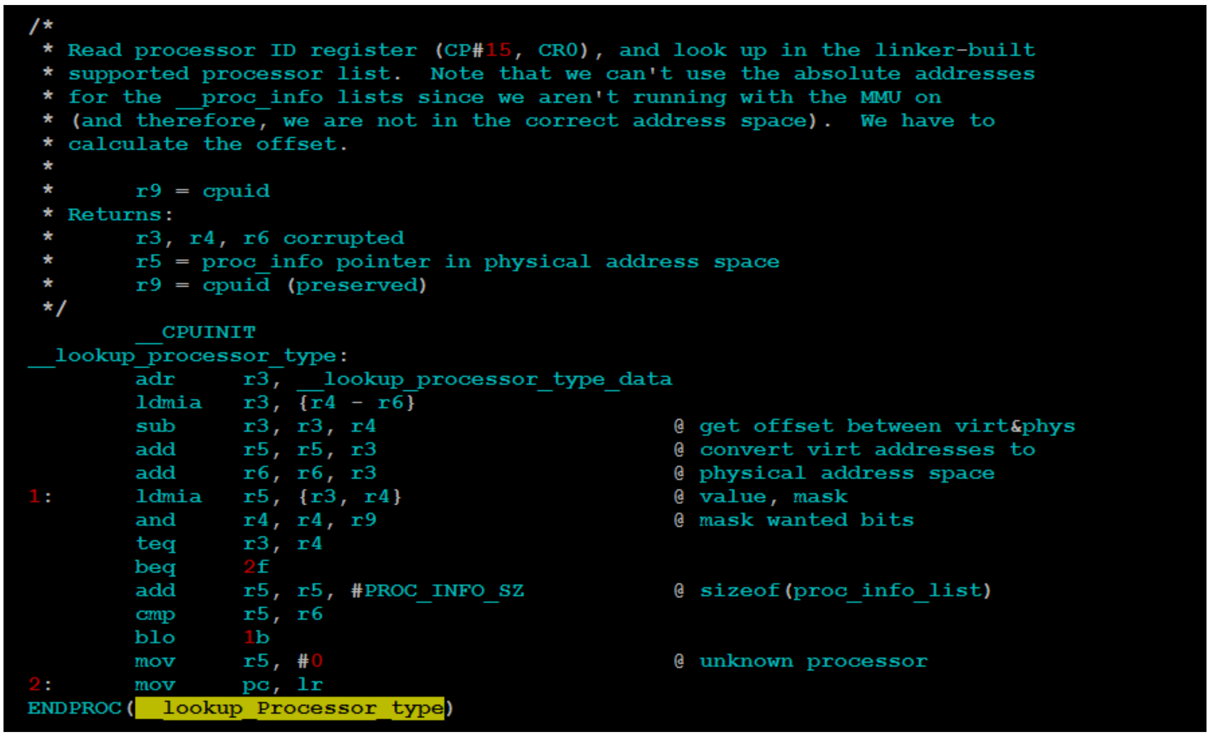
```
 * -------------------------
 *
 * This is normally called from the decompressor code.  The requirements
 * are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
 * r1 = machine nr, r2 = atags or dtb pointer.
 *
 * This code is mostly position independent, so if you link the kernel at
 * 0xc0008000, you call this at __pa(0xc0008000).
 *
 * See linux/arch/arm/tools/mach-types for the complete list of machine
 * numbers for r1.
 *
 * We're trying to keep crap to a minimum; DO NOT add any machine specific
 * crap here - that's what the boot loader (or in extreme, well justified
 * circumstances, zImage) is for.
 */
        __HEAD
ENTRY(stext)
        setmode PSR_F_BIT | PSR_I_BIT | SVC_MODE, r9 @ ensure svc mode
                                                @ and irqs disabled
        mrc     p15, 0, r9, c0, c0              @ get processor id
#ifdef CONFIG_ARM_TZ
        bl      __cache_sync
#endif
        bl      __lookup_processor_type         @ r5=procinfo r9=cpuid
        movs    r10, r5                         @ invalid processor (r5=0)?
 THUMB( it      eq )            @ force fixup-able long branch encoding
        beq     __error_p                       @ yes, error 'p'

#ifndef CONFIG_XIP_KERNEL
        adr     r3, 2f
        ldmia   r3, {r4, r8}
        sub     r4, r3, r4                      @ (PHYS_OFFSET - PAGE_OFFSET)
        add     r8, r8, r4                      @ PHYS_OFFSET
#else
        ldr     r8, =PLAT_PHYS_OFFSET
#endif
```

➢ Search list of supported processor types ***__proc_info_begin***
Kernel may be built to support more than one processor type
List of ***proc_info_list*** structs defined in

**File**: - ***"arch/arm/include/asm/procinfo.h"***



```
/*
 * Look in <asm/procinfo.h> for information about the __proc_info structure.
 */
        .align  2
        .type   __lookup_processor_type_data, %object
__lookup_processor_type_data:
        .long   .
        .long   __proc_info_begin
        .long   __proc_info_end
        .size   __lookup_processor_type_data, . - __lookup_processor_type_data
```



```
/*
 * Read processor ID register (CP#15, CR0), and look up in the linker-built
 * supported processor list.  Note that we can't use the absolute addresses
 * for the __proc_info lists since we aren't running with the MMU on
 * (and therefore, we are not in the correct address space).  We have to
 * calculate the offset.
 *
 *      r9 = cpuid
 * Returns:
 *      r3, r4, r6 corrupted
 *      r5 = proc_info pointer in physical address space
 *      r9 = cpuid (preserved)
 */
        __CPUINIT
__lookup_processor_type:
        adr     r3, __lookup_processor_type_data
        ldmia   r3, {r4 - r6}
        sub     r3, r3, r4                      @ get offset between virt&phys
        add     r5, r5, r3                      @ convert virt addresses to
        add     r6, r6, r3                      @ physical address space
1:      ldmia   r5, {r3, r4}                    @ value, mask
        and     r4, r4, r9                      @ mask wanted bits
        teq     r3, r4
        beq     2f
        add     r5, r5, #PROC_INFO_SZ           @ sizeof(proc_info_list)
        cmp     r5, r6
        blo     1b
        mov     r5, #0                          @ unknown processor
2:      mov     pc, lr
ENDPROC(__lookup_Processor_type)
```

Search list of supported processor types ***__proc_info_begin***

- ➢ Kernel may be built to support more than one processor type.
- ➢ List of ***__arm926_proc_info*** structs

Defined in *"**arch/arm/mm/proc-arm926.S**"* and other corresponding proc-*.S files
Linked into list by section declaration: .section *"**.proc.info.init**"*

Return pointer to ***proc_info_list*** struct corresponding to processor if found, or loop in error if not call ***__machine_arch_type*** defined in *"**arch/arm/kernel/head-common.S**"*

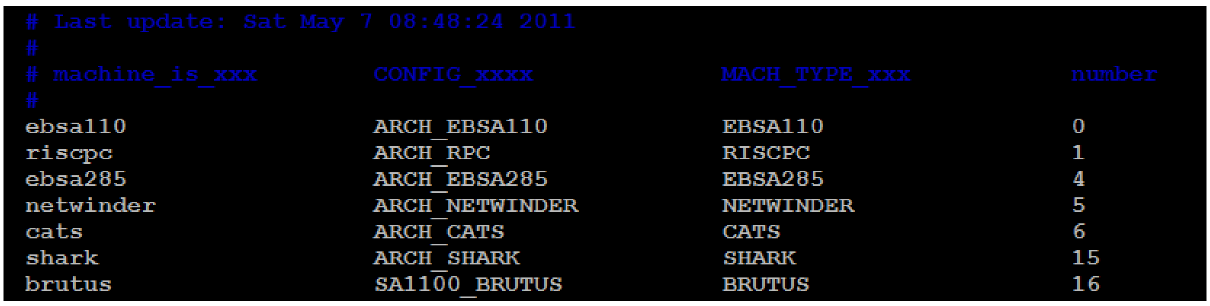
Which gets assigned in ( ***__machine_arch_type***     = ***arch_id*** ) in

**<u>File:-</u>** *"**arch/arm/boot/compressed/misc.c**"*

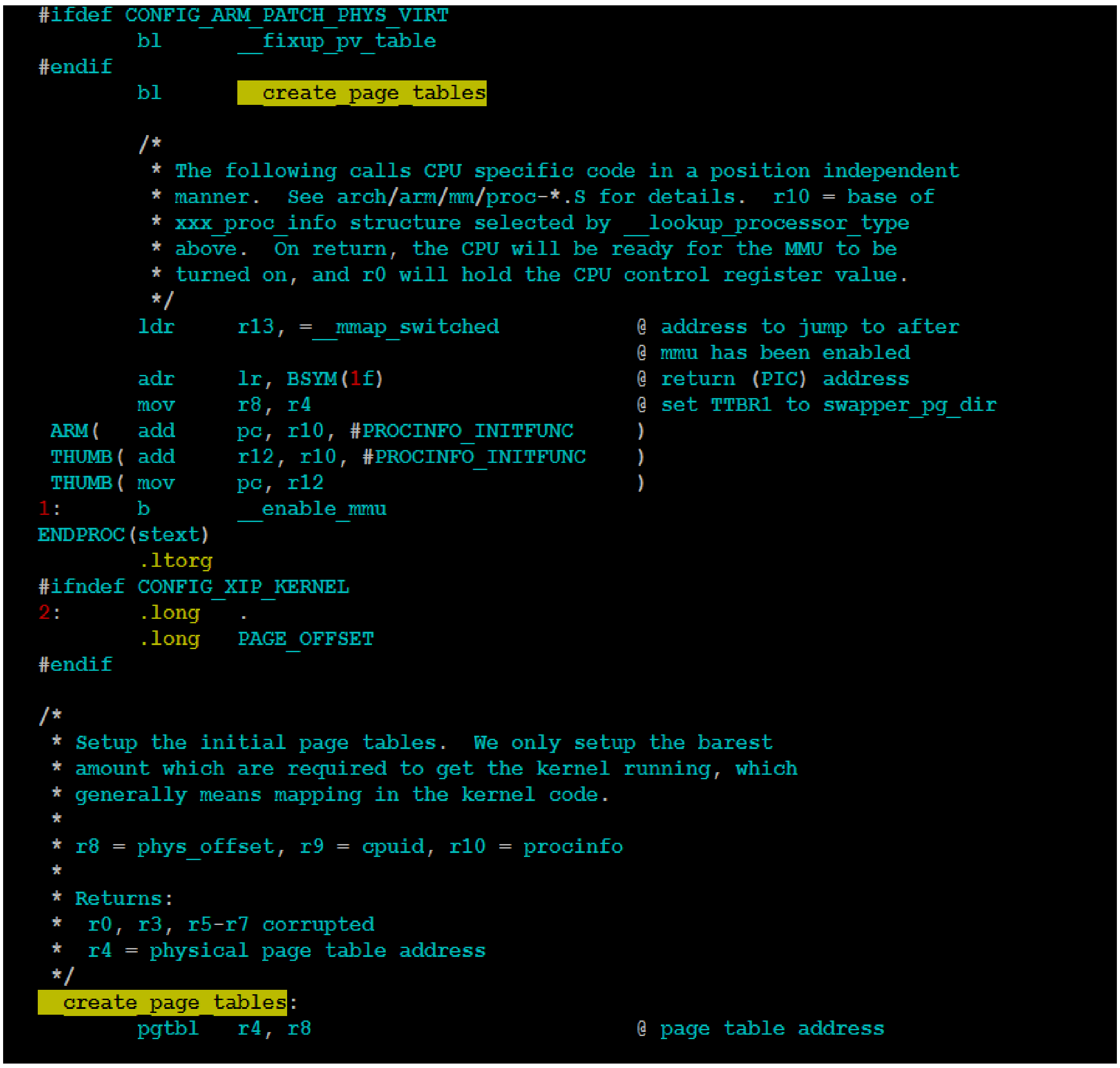
Search list of supported machines (boards)

- ➢ kernel may be built to support more than one board
- ➢ list of ***machine_desc*** structs defined in **("*arch/arm/include/asm/mach/arch.h*" )**

All machine type is defined in (**"*arch/arm/tools/mach-types*")** which is a Database of machine macros and numbers.
linked into list by section declaration that's part of MACHINE_DESC macro
return pointer to ***machine_desc*** struct corresponding to machine (board).



```
# Last update: Sat May 7 08:48:24 2011
#
# machine_is_xxx        CONFIG_xxxx             MACH_TYPE_xxx           number
#
ebsa110                 ARCH_EBSA110            EBSA110                 0
riscpc                  ARCH_RPC                RISCPC                  1
ebsa285                 ARCH_EBSA285            EBSA285                 4
netwinder               ARCH_NETWINDER          NETWINDER               5
cats                    ARCH_CATS               CATS                    6
shark                   ARCH_SHARK              SHARK                   15
brutus                  SA1100_BRUTUS           BRUTUS                  16
```



```
#ifdef CONFIG_ARM_PATCH_PHYS_VIRT
        bl      __fixup_pv_table
#endif
        bl      __create_page_tables

        /*
         * The following calls CPU specific code in a position independent
         * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
         * xxx_proc_info structure selected by __lookup_processor_type
         * above.  On return, the CPU will be ready for the MMU to be
         * turned on, and r0 will hold the CPU control register value.
         */
        ldr     r13, =__mmap_switched           @ address to jump to after
                                                @ mmu has been enabled
        adr     lr, BSYM(1f)                    @ return (PIC) address
        mov     r8, r4                          @ set TTBR1 to swapper_pg_dir
 ARM(   add     pc, r10, #PROCINFO_INITFUNC     )
 THUMB( add     r12, r10, #PROCINFO_INITFUNC    )
 THUMB( mov     pc, r12                         )
1:      b       __enable_mmu
ENDPROC(stext)
        .ltorg
#ifndef CONFIG_XIP_KERNEL
2:      .long   .
        .long   PAGE_OFFSET
#endif

/*
 * Setup the initial page tables.  We only setup the barest
 * amount which are required to get the kernel running, which
 * generally means mapping in the kernel code.
 *
 * r8 = phys_offset, r9 = cpuid, r10 = procinfo
 *
 * Returns:
 *  r0, r3, r5-r7 corrupted
 *  r4 = physical page table address
 */
__create_page_tables:
        pgtbl   r4, r8                          @ page table address
```

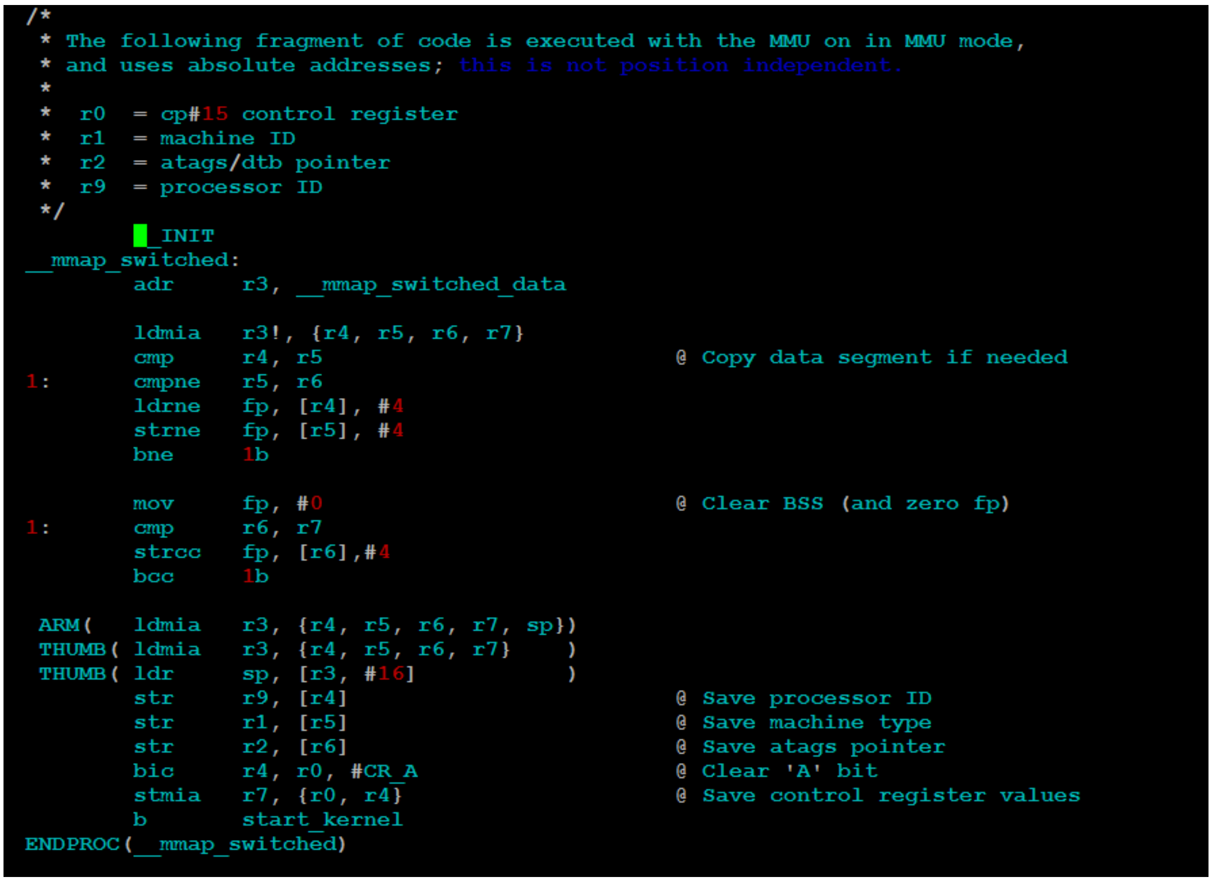
- call ***__create_page_tables*** to set up initial MMU tables
- set lr to ***__enable_mmu***, r13 to address of ***__switch_data***

  lr and r13 used for jumps after the following calls

  ***__switch_data*** defined in **arch/arm/kernel/head-common.S**

- call the ***__cpu_flush*** function pointer in the previously returned proc_info_list struct .
  Offset is #PROCINFO_INITFUNC into struct
  - This function is ***__arm926_setup*** for the ARM 926EJ-S, defined in ***<arch/arm/mm/proc-arm926.S>***
  - initialize caches, write buffer
  - jump to lr, previously set to address of ***__enable_mmu***

- ***__enable_mmu***. Set page table pointer (TTB) in MMU hardware so it knows where to start page-table walks
- Enable MMU so running with virtual addresses
- Jump to r13, previously set to address of ***__switch_data,*** whose first field is address of ***__mmap_switched***. ***__switch_data*** defined in *"**arch/arm/kernel/head-common.S**"*



```
/*
 * The following fragment of code is executed with the MMU on in MMU mode,
 * and uses absolute addresses; this is not position independent.
 *
 *  r0  = cp#15 control register
 *  r1  = machine ID
 *  r2  = atags/dtb pointer
 *  r9  = processor ID
 */
        __INIT
__mmap_switched:
        adr     r3, __mmap_switched_data

        ldmia   r3!, {r4, r5, r6, r7}
        cmp     r4, r5                          @ Copy data segment if needed
1:      cmpne   r5, r6
        ldrne   fp, [r4], #4
        strne   fp, [r5], #4
        bne     1b

        mov     fp, #0                          @ Clear BSS (and zero fp)
1:      cmp     r6, r7
        strcc   fp, [r6],#4
        bcc     1b

 ARM(   ldmia   r3, {r4, r5, r6, r7, sp})
 THUMB( ldmia   r3, {r4, r5, r6, r7}     )
 THUMB( ldr     sp, [r3, #16]            )
        str     r9, [r4]                        @ Save processor ID
        str     r1, [r5]                        @ Save machine type
        str     r2, [r6]                        @ Save atags pointer
        bic     r4, r0, #CR_A                   @ Clear 'A' bit
        stmia   r7, {r0, r4}                    @ Save control register values
        b       start_kernel
ENDPROC(__mmap_switched)
```

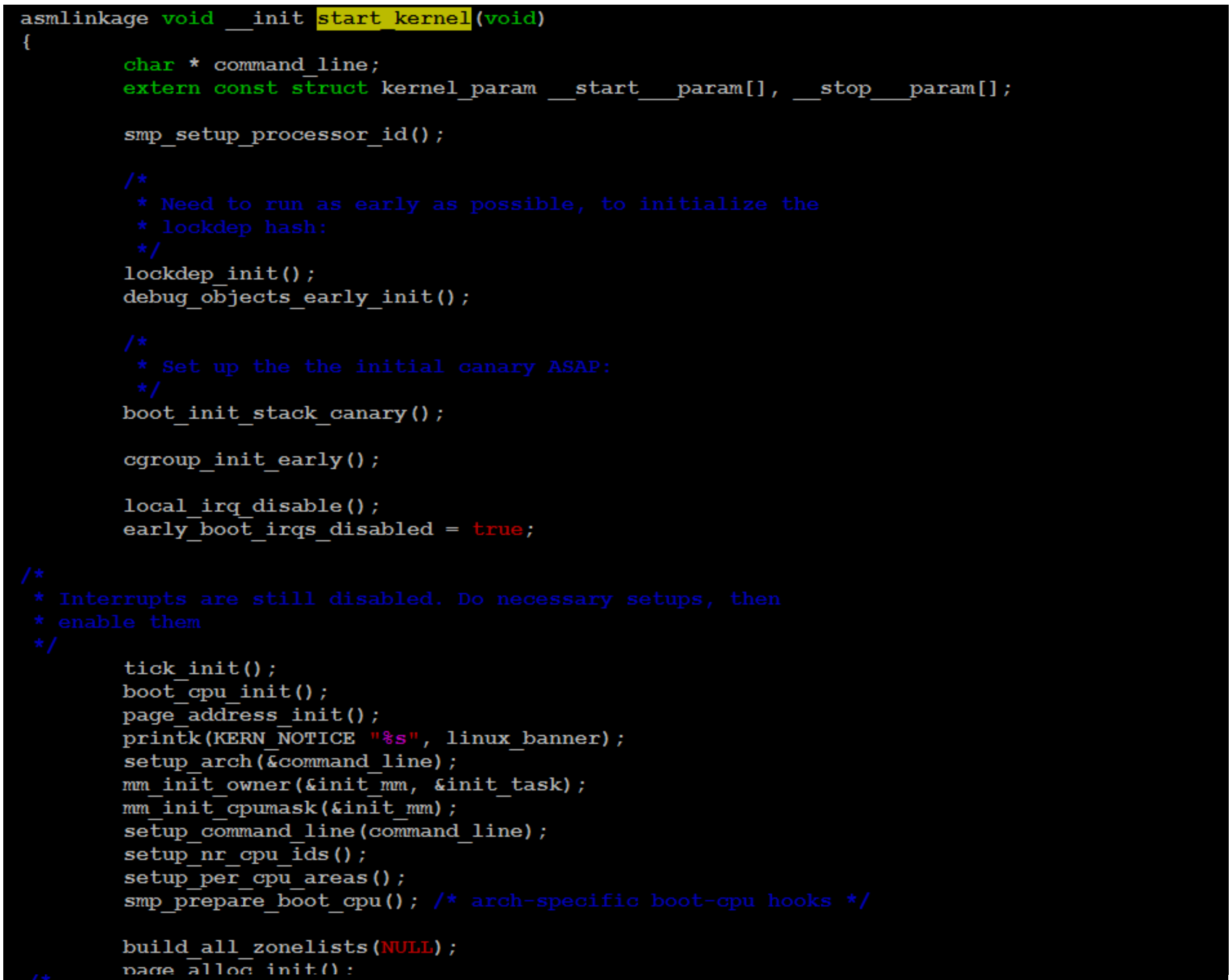
- ***__mmap_switched*** in File "**arch/arm/kernel/head-common.S**"
  - copy data segment to RAM
  - zero BSS
  - branch to ***start_kernel*** "***linux/init/main.c***"

## Processor-independent kernel code

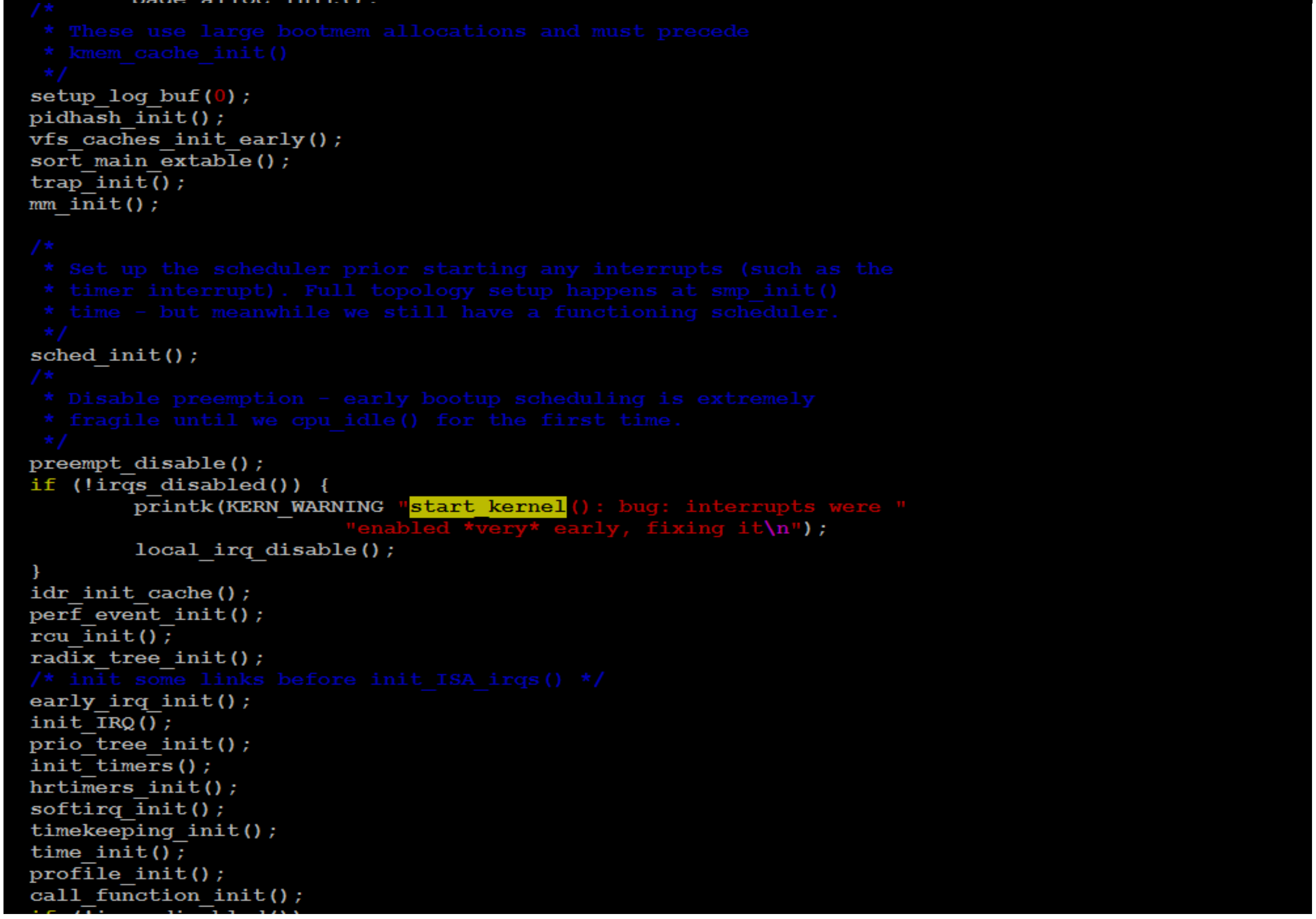
***init/main.c***: ***start_kernel()*** The function completes the initialization of the Linux kernel. Nearly every kernel component is initialized by this function;

### To mention a few of them:

- The Page Tables are initialized by invoking the ***paging_init( )*** function.
- The page descriptors are initialized by the ***kmem_init( ), free_area_init( ),*** and ***mem_init( )*** functions.
- The final initialization of the IDT is performed by invoking ***trap_init( )*** and ***init_IRQ( )***.
- The slab allocator is initialized by the ***kmem_cache_init( )*** and ***kmem_cache_sizes_init( )*** functions.
- The system date and time are initialized by the ***time_init( )*** function.
- The kernel thread for process 1 is created by invoking the ***kernel_thread( )*** function. In turn, this kernel thread creates the other kernel threads and executes the ***/sbin/init*** program.



```
asmlinkage void __init start_kernel(void)
{
        char * command_line;
        extern const struct kernel_param __start___param[], __stop___param[];

        smp_setup_processor_id();

        /*
         * Need to run as early as possible, to initialize the
         * lockdep hash:
         */
        lockdep_init();
        debug_objects_early_init();

        /*
         * Set up the the initial canary ASAP:
         */
        boot_init_stack_canary();

        cgroup_init_early();

        local_irq_disable();
        early_boot_irqs_disabled = true;

/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
        tick_init();
        boot_cpu_init();
        page_address_init();
        printk(KERN_NOTICE "%s", linux_banner);
        setup_arch(&command_line);
        mm_init_owner(&init_mm, &init_task);
        mm_init_cpumask(&init_mm);
        setup_command_line(command_line);
        setup_nr_cpu_ids();
        setup_per_cpu_areas();
        smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */

        build_all_zonelists(NULL);
        page_alloc_init();
```



```
/*
 * These use large bootmem allocations and must precede
 * kmem_cache_init()
 */
setup_log_buf(0);
pidhash_init();
vfs_caches_init_early();
sort_main_extable();
trap_init();
mm_init();

/*
 * Set up the scheduler prior starting any interrupts (such as the
 * timer interrupt). Full topology setup happens at smp_init()
 * time - but meanwhile we still have a functioning scheduler.
 */
sched_init();
/*
 * Disable preemption - early bootup scheduling is extremely
 * fragile until we cpu_idle() for the first time.
 */
preempt_disable();
if (!irqs_disabled()) {
        printk(KERN_WARNING "start_kernel(): bug: interrupts were "
                        "enabled *very* early, fixing it\n");
        local_irq_disable();
}
idr_init_cache();
perf_event_init();
rcu_init();
radix_tree_init();
/* init some links before init_ISA_irqs() */
early_irq_init();
init_IRQ();
prio_tree_init();
init_timers();
hrtimers_init();
softirq_init();
timekeeping_init();
time_init();
profile_init();
call_function_init();
```