OverView

This application note explains the way to build the secure BL1(1 st Bootloader) and BL2(2nd Bootloader) images in the booting environment of Exynos4212. iROM code(iROM Bootloader) of Exynos4212 confirms to download the BL1 image with checksum, verifies the integrity of the secure BL1 image, decrypts the secure BL1 image, and then iROM goes to BL1. In the BL1, the integrity of the secure BL2 is verified. If the secure BL2 image is verified successfully, BL1 will go to BL2. In order to verify the integrity of the secure image on each stage, iROM code provides the secure library functions to reuse in BL1 and BL2.

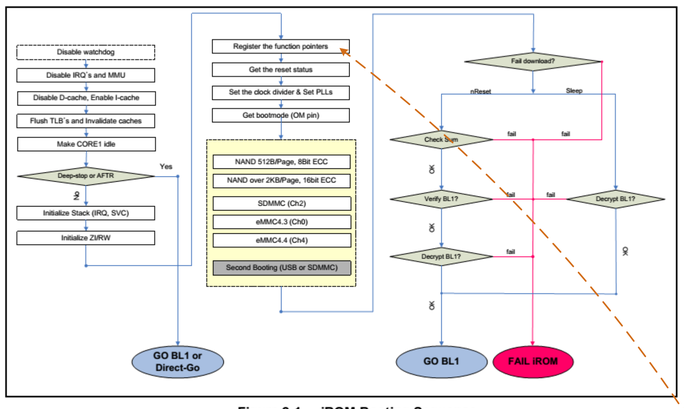

Figure 2-1 iROM Booting Sequence

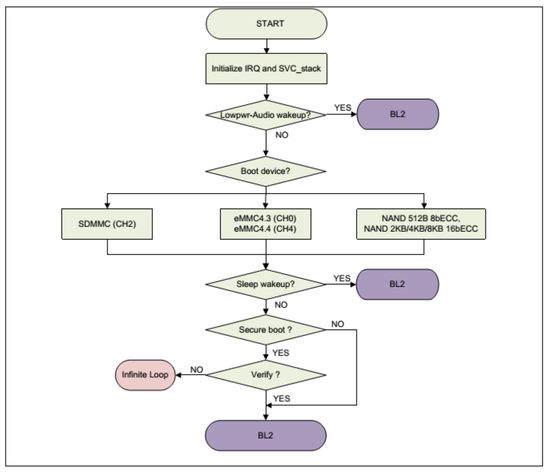

Figure 2-2 Secure BL1 Booting Sequence

作者： 彭东林
邮箱： pengdonglin137@163.com
QQ： 405728433

file: arch/arm/cpu/armv7/start.S

set the cpu to SVC32 mode

cpu_init_crit

invalidate L1 I/D

disable MMU stuff and caches

board/samsung/tiny4412/lowlevel_init.S

lowlevel_init

设置栈sp

点亮LED灯，点亮led0

read_om

判断启动方式，并将表示启动方式的值存到寄存器r3中，最后将r3寄存器中的值存放到INF_REG_BASE+INF_REG3_OFFSET，即0x1002080c中

判断是否需要重定位，原理是判断当前运行的物理地址与编译时的链接地址是否一致，这里_TEXT_BASE的值是0xc3e00000，从这里可以看出u-boot一定启用了MMU，因为物理内存的大小是1G，物理地址的范围是0x40000000~0x80000000

```
ldr     r0, =0xff000fff
bic     r1, pc, r0              /* r0 <- current base addr of code */
ldr     r2, _TEXT_BASE          /* r1 <- original base addr in ram */
bic     r2, r2, r0              /* r0 <- current base addr of code */
cmp     r1, r2                  /* compare r0, r1 */
beq     after_copy              /* r0 == r1 then skip sdram init and u-boot.bin loading */
```

需要重定位

N

Y

点亮led0和led1

enable_mmu

```
/*
 * MMU Table for SMDKC210
 * 0x0000_0000 -- 0x1FFF_FFFF => A:0x0000_0000 -- 0x1FFF_FFFF
 * 0x2000_0000 -- 0x3FFF_FFFF => Not Allowed
 * 0x4000_0000 -- 0x5FFF_FFFF => A:0x4000_0000 -- 0x5FFF_FFFF
 * 0x6000_0000 -- 0xBFFF_FFFF => Not Allowed
 * 0xC000_0000 -- 0xDFFF_FFFF => A:0x4000_0000 -- 0x5FFF_FFFF
 * 0xE000_0000 -- 0xFFFF_FFFF => Not Allowed
 */
```

上面列出的是在设置mmu时设置的内存映射关系，重点看：0xC000_0000 -- 0xDFFF_FFFF => A:0x4000_0000 -- 0x5FFF_FFFF
在编译出的u-boot中我们把链接起始地址设置到了0xc3e00000，实际物理地址范围是0x40000000~0x80000000，通过将u-boot反编译，可以看到从0xc0000000~0xdfffffff完全够用了。

```
/*
 * uart_asm_init: Initialize UART in asm mode, 115200bps fixed.
 * void uart_asm_init(void)
 */
        .globl uart_asm_init
uart_asm_init:

        /* set GPIO to enable UART */
        @ GPIO setting for UART for UART0/1/2/3
        ldr     r0, =0x11400000
        ldr     r1, =0x22222222
        str     r1, [r0]
        ldr     r0, =0x11400020
        ldr     r1, =0x222222
        str     r1, [r0]

        ldr     r0, =S5PV310_CLOCK_BASE
        ldr     r1, =CLK_SRC_PERIL0_VAL
        ldr     r2, =CLK_SRC_PERIL0_OFFSET
        str     r1, [r0, r2]
        ldr     r1, =CLK_DIV_PERIL0_VAL
        ldr     r2, =CLK_DIV_PERIL0_OFFSET
        str     r1, [r0, r2]

        ldr     r0, =S5PV310_UART_CONSOLE_BASE
        ldr     r1, =0x111
        str     r1, [r0, #UFCON_OFFSET]

        mov     r1, #0x3
        str     r1, [r0, #ULCON_OFFSET]

        ldr     r1, =0x3c5
        str     r1, [r0, #UCON_OFFSET]
        // 设置波特率为115200
        ldr     r1, =UART_UBRDIV_VAL
        str     r1, [r0, #UBRDIV_OFFSET]

        ldr     r1, =UART_UDIVSLOT_VAL
        str     r1, [r0, #UDIVSLOT_OFFSET]
        // 向串口终端中打印字符 'O'
        ldr     r1, =0x4f4f4f4f
        str     r1, [r0, #UTXH_OFFSET]          @'O'

        mov     pc, lr
```

system_clock_init
// 初始化系统时钟

/* Memory initialize */
mem_ctrl_asm_init
// 初始化内存

/* init uart for debug */
uart_asm_init
// 初始化串口，用于调试

tzpc_init

load_uboot

```
#define BOOT_ONENAND        0x1
#define BOOT_NAND           0x40000
#define BOOT_MMCSD          0x3
#define BOOT_NOR            0x4
#define BOOT_SEC_DEV        0x5
#define BOOT_EMMC           0x6
#define BOOT_EMMC_4_4       0x7
```

读取0x1002080c的值，依次来判断从哪读取uboot到内存，这里以从SD卡启动为例分析

mmcsd_boot

movi_uboot_copy

SDMMC_ReadBlocks(MOVI_UBOOT_POS, MOVI_UBOOT_BLKCNT, CONFIG_PHY_UBOOT_BASE);

```
#define SDMMC_ReadBlocks(uStartBlk, uNumOfBlks, uDstAddr)               \
        (((void(*)(u32, u32, u32*))(*((u32 *)EXTERNAL_FUNC_ADDRESS)))(uStartBlk, uNumOfBlks, uDstAddr))
```

// 其中EXTERNAL_FUNC_ADDRESS的值是0x02020030，这是isram的地址，在这个地址上存放的是一个函数指针，用于从sd卡中读取数据到内存中，这个函数指针是由IROM中的代码设置的

```
/* store second boot information in u-boot C level variable */
ldr     r0, =CONFIG_PHY_UBOOT_BASE
sub     r0, r0, #8
ldr     r1, [r0]
ldr     r0, _second_boot_info
str     r1, [r0]
```

其中，CONFIG_PHY_UBOOT_BASE是0x43e00000，而_second_boot_info中存放的是变量second_boot_info，这个变量可以在c代码中访问，second_boot_info的初始值是0xffffffff

file: arch/arm/lib/board.c
board_init_f

1. 将uboot代码段拷贝到内存中计算的位置处
2. 对其中的符号以及变量的地址重新设置，因为此时的位置跟编译链接时的位置不同了，所以需要修改其中符号和变量的地址为新的地址

relocate_code

clear_bss

board_init_r

main_loop

u_boot_hush_start ()

s = getenv ("bootcmd")

abortboot (bootdelay)

Y

parse_file_outer()

parse_stream_outer(&input, FLAG_PARSE_SEMICOLON)

N

parse_string_outer(s, FLAG_PARSE_SEMICOLON | FLAG_EXIT_FROM_LOOP);
解析并执行bootcmd环境变量中的命令，引导启动Linux

下面这个循环不会退出，等待用户输入命令，然后解析命令执行，这种方法跟以前的uboot处理用户输入并执行的方法不太一样，因为它定义了宏CONFIG_SYS_HUSH_PARSER

```
do {
        ctx.type = flag;
        initialize_context(&ctx);
        update_ifs_map();
        if (!(flag & FLAG_PARSE_SEMICOLON) || (flag & FLAG_REPARSING)) mapset((uchar *)";S&|", 0);
        inp->promptmode=1;
        rcode = parse_stream(&temp, &ctx, inp, '\n');

        if (rcode == 1) flag_repeat = 0;

        if (rcode != 1 && ctx.old_flag != 0) {
                syntax();

                flag_repeat = 0;

        }
        if (rcode != 1 && ctx.old_flag == 0) {
                done_word(&temp, &ctx);
                done_pipe(&ctx,PIPE_SEQ);
                code = run_list(ctx.list_head);
                if (code == -2) {       /* exit */
                        b_free(&temp);
                        code = 0;
                        /* XXX hackish way to not allow exit from main loop */
                        if (inp->peek == file_peek) {
                                printf("exit not allowed from main input shell.\n");
                                continue;
                        }
                        break;
                }
                if (code == -1)
                  flag_repeat = 0;
        } else {
                if (ctx.old_flag != 0) {
                        free(ctx.stack);
                        b_reset(&temp);
                }
                if (inp->__promptme == 0) printf("<INTERRUPT>\n");
                inp->__promptme = 1;
                temp.nonnull = 0;
                temp.quote = 0;
                inp->p = NULL;
                free_pipe_list(ctx.list_head,0);
        }
        b_free(&temp);
} while (rcode != -1 && !(flag & FLAG_EXIT_FROM_LOOP));   /* loop on syntax errors, return on EOF */
```