(1条消息) u-boot-2021.01（imx6ull）启动流程分析之二：从执行第1句u-boot代码开始分析 ___ASDFGH的博客-CSDN博客

3、启动流程分析

先预览函数的大概调用框图，后面对每个函数进行分析：

```
_start
        |_ cpu_init_cp15
        |_ cpu_init_crit
        |       |_ lowlevel_init
        |               |_ s_init
        |_ _main
                |_ board_init_f_alloc_reserve
                |_ board_init_f_init_reserve
                |_ board_init_f
                |_ relocate_code
                |_ relocate_vectors
                |_ board_init_r
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12

3.1 _start

查看程序的入口可以从编译生成的u-boot.lds入手，因为lds文件是指定程序各个段的存储位置，查看它的内容：

```
/* file: u-boot.lds */
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
 . = 0x00000000;
 . = ALIGN(4);
 .text :
 {
  *(.__image_copy_start)
  *(.vectors)
  arch/arm/cpu/armv7/start.o (.text*)
 }
...
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

通过以上ENTRY(_start)定义，我们可以找到arm架构的arch/arm/lib/vectors.S文件，摘取部分内容如下：

```
/* file: arch/arm/lib/vectors.S */
        .macro ARM_VECTORS
#ifdef CONFIG_ARCH_K3
        ldr     pc, _reset
#else
        b       reset
#endif
        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
        .endm
...

_start:
#ifdef CONFIG_SYS_DV_NOR_BOOT_CFG
  .word CONFIG_SYS_DV_NOR_BOOT_CFG
#endif
```

```
  ARM_VECTORS
#endif /* !defined(CONFIG_ENABLE_ARM_SOC_BOOT0_HOOK) */
...

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

#ifdef CONFIG_SPL_BUILD
...
#else   /* !CONFIG_SPL_BUILD */
...
undefined_instruction:
        get_bad_stack
        bad_save_user_regs
        bl      do_undefined_instruction
...
#endif  /* CONFIG_SPL_BUILD */
```

根据文件名字也可以看到，这些都是向量表相关的一些跳转和异常处理。在文件开始部分使用汇编宏`.macro ARM_VECTORS ... .endm`定义了`ARM_VECTORS`，然后在程序的入口`_start`标志处引用了这个宏。

这个宏涉及了异常处理，看下如图Cortex A7的异常向量表:

Table 11-1 Summary of exception behavior

Normal Vector offset	High vector address	Non-secure	Secure	Hypervisor[a]	Monitor
0x0	0xFFFF0000	Not used	Reset	Reset	Not used
0x4	0xFFFF0004	UNDEFINED instruction	UNDEFINED instruction	UNDEFINED instruction from Hyp mode.	Not used
0x8	0xFFFF0008	Supervisor Call	Supervisor Call	Secure Monitor Call	Secure Monitor Call
0xC	0xFFFF000C	Prefetch Abort	Prefetch Abort	Prefetch Abort from Hyp mode.	Prefetch Abort
0x10	0xFFFF0010	Data Abort	Data Abort	Data Abort from Hyp mode,	Data Abort
0x14	0xFFFF0014	Not used	Not used	Hyp mode entry	Not used
0x18	0xFFFF0018	IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
0x1C	0xFFFF001C	FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt

举个例子，当发生FIQ快速中断时，硬件决定直接跳到如表格展示的0x1C地址去，但是该地址处无法处理过多的指令，因为这些向量表的地址都是连续的。所以干脆在对应的地址处使用`ldr pc, xxx`再次跳转到其他地址去处理。

但在分析启动流程时，我们只关心**u-boot**的第一句代码——**`b reset`**跳转到复位语句，reset标志就在对应的cpu架构目录`arch/arm/cpu/armv7/start.S`中定义，摘取部分内容如下：

```
/* file: arch/arm/cpu/armv7/start.S */
reset:
        /* Allow the board to save important registers */
        b       save_boot_params
save_boot_params_ret:
#ifdef CONFIG_ARMV7_LPAE
/*
 * check for Hypervisor support
 */
        mrc     p15, 0, r0, c0, c1, 1           @ read ID_PFR1
        and     r0, r0, #CPUID_ARM_VIRT_MASK    @ mask virtualization bits
        cmp     r0, #(1 << CPUID_ARM_VIRT_SHIFT)
        beq     switch_to_hypervisor
switch_to_hypervisor_ret:
#endif
        /*
         * 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 vector:
 * (OMAP4 spl TEXT_BASE is not 32 byte aligned.
 * Continue to use ROM code vector only in OMAP4 spl)
 */
#if !(defined(CONFIG_OMAP44XX) && defined(CONFIG_SPL_BUILD))
        /* Set V=0 in CP15 SCTLR register - for VBAR to point to vector */
        mrc     p15, 0, r0, c1, c0, 0   @ Read CP15 SCTLR Register
        bic     r0, #CR_V               @ V = 0
        mcr     p15, 0, r0, c1, c0, 0   @ Write CP15 SCTLR Register

#ifdef CONFIG_HAS_VBAR
        /* Set vector address in CP15 VBAR register */
        ldr     r0, =_start
        mcr     p15, 0, r0, c12, c0, 0  @Set VBAR
#endif
#endif

        /* the mask ROM code should have PLL and others stable */
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
#ifdef CONFIG_CPU_V7A
```

```
        bl      cpu_init_cp15
#endif
#ifndef CONFIG_SKIP_LOWLEVEL_INIT_ONLY
        bl      cpu_init_crit
#endif
#endif

        bl      _main
```

首先就是先跳转到同文件中的`save_boot_params`:

```
/* file: arch/arm/cpu/armv7/start.S */
ENTRY(save_boot_params)
        b       save_boot_params_ret            @ back to my caller
ENDPROC(save_boot_params)
        .weak   save_boot_params
```

然而发现它并没有作任何处理，又继续跳转回来。接着就是根据宏`CONFIG_ARMV7_LPAE`判断是否需要支持Hypervisor模式，但没有定义所以不需要关心；然后就是设置cpsr寄存器让CPU进入SVC32管理模式并且关闭FIQ快速中断和IRQ中断；

往下就是使用CR_V的值（在`arch/arm/include/asm/system.h`中定义）来设置CP15协处理器，目的是支持向量表的重定位，接着就是设置新的向量表地址了。

再往下就是跳转执行`cpu_init_cp15`、`cpu_init_crit`和`_main`，下面就逐个分析。

3.2 cpu_init_cp15

它在同文件中定义，主要是设置CP15协处理器，目的是初始化并设置“数据缓存dcache”和“指令缓存icache”。其中，dcache是务必关闭的，因为u-boot程序更多时候是和硬件在打交道，所以尽可能少用dcache中的缓存数据，目的是实时读写硬件，而icache则根据“SYS_ICACHE_OFF”是否配置来决定。其次函数工作内容还有关闭MMU等等，代码如下:

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

```
ENTRY(cpu_init_cp15)
        /*
         * Invalidate L1 I/D
         */
        mov     r0, #0                  @ 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
#if CONFIG_IS_ENABLED(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
        mcr     p15, 0, r0, c1, c0, 0
...
        mov     r5, lr                  @ Store my Caller
...
        mov     pc, r5                  @ back to my caller
ENDPROC(cpu_init_cp15)
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31

执行完`cpu_init_cp15`函数接着继续跳转到`cpu_init_crit`函数。

3.3 cpu_init_crit

它也是在同文件中定义，根据官方注释也可以清楚地看到它的目的是设置重要的寄存器等等，往里看看就知道了：

```
/* file: arch/arm/cpu/armv7/start.S */
/*************************************************************************
 *
 * 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)
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18

看到它就只是一句跳转到`lowlevel_init`函数。

3.3.1 lowlevel_init

```
/* file: arch/arm/cpu/armv7/lowlevel_init.S */
WEAK(lowlevel_init)
        /*
         * Setup a temporary stack. Global data is not available yet.
         */
#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 */
#ifdef CONFIG_SPL_DM
        mov     r9, #0
#else
        /*
         * Set up global data for boards that still need it. This will be
         * removed soon.
         */
#ifdef CONFIG_SPL_BUILD
        ldr     r9, =gdata
#else
        sub     sp, sp, #GD_SIZE
        bic     sp, sp, #7
        mov     r9, sp
#endif
#endif
        /*
         * Save the old lr(passed in ip) and the current lr to stack
         */
        push    {ip, lr}

        /*
         * Call the very early init function. This should do only the
         * absolute bare minimum to get started. It should not:
         *
         * - set up DRAM
         * - use global_data
         * - clear BSS
         * - try to start a console
         *
         * For boards with SPL this should be empty since SPL can do all of
         * this init in the SPL board_init_f() function which is called
         * immediately after this.
         */
        bl      s_init
        pop     {ip, pc}
ENDPROC(lowlevel_init)
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35

- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47

可以看到，函数里面就是先设置sp栈地址并进行8字节对齐，目的是后面跳转到s_init函数。

3.3.1.1 s_init

```
void s_init(void)
{
        struct anatop_regs *anatop = (struct anatop_regs *)ANATOP_BASE_ADDR;
        struct mxc_ccm_reg *ccm = (struct mxc_ccm_reg *)CCM_BASE_ADDR;
        u32 mask480;
        u32 mask528;
        u32 reg, periph1, periph2;

        if (is_mx6sx() || is_mx6ul() || is_mx6ull() || is_mx6sll())
                return;
...
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

函数里面会先判断芯片的型号，发现mx6ull不作任何处理就直接返回了，往上返回之后就到了_main函数。

3.4 _main

这个函数非常重要，从函数命名也可以知道它负责主要的工作，它会直接或间接地调用一些函数来初始化硬件和启动内核:

```
/* file: arch/arm/lib/crt0.S */
ENTRY(_main)

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

#if defined(CONFIG_TPL_BUILD) && defined(CONFIG_TPL_NEEDS_SEPARATE_STACK)
        ldr     r0, =(CONFIG_TPL_STACK)
#elif defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)
        ldr     r0, =(CONFIG_SPL_STACK)
#else
        ldr     r0, =(CONFIG_SYS_INIT_SP_ADDR)
#endif
        bic     r0, r0, #7      /* 8-byte alignment for ABI compliance */
        mov     sp, r0
        bl      board_init_f_alloc_reserve
        mov     sp, r0
        /* set up gd here, outside any C code */
        mov     r9, r0
        bl      board_init_f_init_reserve

#if defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_EARLY_BSS)
        CLEAR_BSS
#endif

        mov     r0, #0
        bl      board_init_f

#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     r0, [r9, #GD_START_ADDR_SP]     /* sp = gd->start_addr_sp */
        bic     r0, r0, #7      /* 8-byte alignment for ABI compliance */
        mov     sp, r0
        ldr     r9, [r9, #GD_NEW_GD]            /* r9 <- gd->new_gd */

        adr     lr, here
        ldr     r0, [r9, #GD_RELOC_OFF]         /* r0 = gd->reloc_off */
```

```
        add     lr, lr, r0
#if defined(CONFIG_CPU_V7M)
        orr     lr, #1                          /* As required by Thumb-only */
#endif
        ldr     r0, [r9, #GD_RELOCADDR]         /* r0 = gd->relocaddr */
        b       relocate_code
here:
/*
 * now relocate vectors
 */

        bl      relocate_vectors

/* Set up final (full) environment */

        bl      c_runtime_cpu_setup     /* we still call old routine here */
#endif
#if !defined(CONFIG_SPL_BUILD) || CONFIG_IS_ENABLED(FRAMEWORK)

#if !defined(CONFIG_SPL_BUILD) || !defined(CONFIG_SPL_EARLY_BSS)
        CLEAR_BSS
#endif

# 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
        movne   r9, r0
# endif

#if ! defined(CONFIG_SPL_BUILD)
        bl coloured_LED_init
        bl red_led_on
#endif
        /* call board_init_r(gd_t *id, ulong dest_addr) */
        mov     r0, r9                  /* gd_t */
        ldr     r1, [r9, #GD_RELOCADDR] /* dest_addr */
        /* call board_init_r */
#if CONFIG_IS_ENABLED(SYS_THUMB_BUILD)
        ldr     lr, =board_init_r       /* this is auto-relocated! */
        bx      lr
#else
        ldr     pc, =board_init_r       /* this is auto-relocated! */
#endif
        /* we should not return here. */
#endif

ENDPROC(_main)
```

函数内容有点多，但我们比较感兴趣的主要有以下几句：

```
bl      board_init_f_alloc_reserve
bl      board_init_f_init_reserve
bl      board_init_f
b       relocate_code
bl      relocate_vectors
ldr     pc, =board_init_r
```

前两个函数都是和全局变量gd结构体相关，后四个函数内容更多也比较重要，所以需要重点研究。废话不多说，接下来就开始研究这几个函数。

3.4.1 board_init_f_alloc_reserve、board_init_f_init_reserve

其中，board_init_f_alloc_reserve是为了预留全局变量gd结构体的内存空间，而board_init_f_init_reserve是将gd结构体清0，初始化预留内存空间等等。不妨看下它们的内容，验证下我们的说法：

```
ulong board_init_f_alloc_reserve(ulong top)
{

#if CONFIG_VAL(SYS_MALLOC_F_LEN)
        top -= CONFIG_VAL(SYS_MALLOC_F_LEN);
#endif

        top = rounddown(top-sizeof(struct global_data), 16);

        return top;
}

void board_init_f_init_reserve(ulong base)
{
        struct global_data *gd_ptr;



        gd_ptr = (struct global_data *)base;

        memset(gd_ptr, '\0', sizeof(*gd));


    ...

        if (CONFIG_IS_ENABLED(SYS_REPORT_STACK_F_USAGE))
                board_init_f_init_stack_protection();
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32

其实不只是`board_init_f_init_reserve`函数，接下来很多操作都是设置gd这个全局变量，我们不会探讨它每一个成员设置过程，主要看比较重要的几个即可。

未完待续...