(1条消息) u-boot-2021.01（imx6ull）启动流程分析之三：board_init_f函数分析___ASDFGH的博客-CSDN博客

3.4.2 board_init_f

顾名思义，函数主要工作是早期的一些硬件初始化和设置全局变量gd结构体的成员：

```
void board_init_f(ulong boot_flags)
{
        gd->flags = boot_flags;
        gd->have_console = 0;

        if (initcall_run_list(init_sequence_f))
                hang();

#if !defined(CONFIG_ARM) && !defined(CONFIG_SANDBOX) && \
                !defined(CONFIG_EFI_APP) && !CONFIG_IS_ENABLED(X86_64) && \
                !defined(CONFIG_ARC)

        hang();
#endif
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16

函数内容不多，但是`init_sequence_f`这个函数指针数组内容可不简单，由于数组元素较多，所以将相关宏定义简化之后就是：

```
static const init_fnc_t init_sequence_f[] = {
        setup_mon_len,
    fdtdec_setup,
    trace_early_init,
    initf_malloc,
        log_init,
        initf_bootstage,
    setup_spl_handoff,
        initf_console_record,
    arch_cpu_init,
        mach_cpu_init,
        initf_dm,
        arch_cpu_init_dm,
    board_early_init_f,
    get_clocks,
        timer_init,
    board_postclk_init,
        env_init,
        init_baud_rate,
        serial_init,
        console_init_f,
        display_options,
        display_text_info,
        checkcpu,
    print_cpuinfo,
    show_board_info,
    INIT_FUNC_WATCHDOG_INIT
    misc_init_f,
    INIT_FUNC_WATCHDOG_RESET
    init_func_i2c,
    init_func_vid,
    announce_dram_init,
        dram_init,
    post_init_f,
    init_post,
    ...
        reserve_uboot,
        reserve_malloc,
        reserve_board,
        setup_machine,
        reserve_global_data,
        reserve_fdt,
    ...
        dram_init_banksize,
        show_dram_config,
        setup_bdinfo,
        display_new_sp,
        ...
        reloc_fdt,
        reloc_bootstage,
        reloc_bloblist,
        setup_reloc,
        ...
        NULL,
};
```

- 1
- 2
- 3
- 4

这个数组最终作为函数initcall_run_list的参数，遍历执行这个函数指针数组的每一个元素（函数），所以重点可以放在函数指针数组里的各个元素。篇章问题，这里就不一一探究，挑出个别容易在移植过程中出现的函数即可。

3.4.2.1 setup_mon_len：设置**gd**的**mon_len**成员（代码长度）

去掉宏定义之后的函数定义：

```
static int setup_mon_len(void)
{
        gd->mon_len = (ulong)&__bss_end - CONFIG_SYS_MONITOR_BASE;
        return 0;
}
```

3.4.2.2 fdtdec_setup：如果**u-boot**中使用设备树，则需处理一些相关工作

去掉宏定义之后的函数定义：

```
int fdtdec_setup(void)
{
        int ret;


        gd->fdt_blob = board_fdt_blob_setup();


        gd->fdt_blob = map_sysmem
                (env_get_ulong("fdtcontroladdr", 16,
                               (unsigned long)map_to_sysmem(gd->fdt_blob)), 0);
        ret = fdtdec_prepare_fdt();
        if (!ret)
                ret = fdtdec_board_setup(gd->fdt_blob);
        return ret;
}
```

- 11
- 12
- 13
- 14
- 15
- 16
- 17

函数先是调用board_fdt_blob_setup来设置gd结构体的fdt_blob成员，简化宏定义之后函数如下：

```
__weak void *board_fdt_blob_setup(void)
{
        void *fdt_blob = NULL;

        fdt_blob = (ulong *)&_end;
        return fdt_blob;
}
```

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

由前面的镜像组成可以知道设备树文件是放在u-boot.bin的末尾，所以取它的地址返回即可。 后面继续调用env_get_ulong从环境变量中获取设备树的地址，如果该环境变量没有被设置则返回原本的默认值。最终继续调用fdtdec_prepare_fdt函数来打印一些信息：

```
int fdtdec_prepare_fdt(void)
{
        if (!gd->fdt_blob || ((uintptr_t)gd->fdt_blob & 3) ||
            fdt_check_header(gd->fdt_blob)) {
#ifdef CONFIG_SPL_BUILD
                puts("Missing DTB\n");
#else
                puts("No valid device tree binary found - please append one to U-Boot binary, use u-boot-dtb.bin or define CONFIG_OF_EMBED. For sandbox, use -d <file.dtb>\n");
# ifdef DEBUG
                if (gd->fdt_blob) {
                        printf("fdt_blob=%p\n", gd->fdt_blob);
                        print_buffer((ulong)gd->fdt_blob, gd->fdt_blob, 4,
                                     32, 0);
                }
# endif
#endif
                return -1;
        }
        return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21

3.4.2.3 board_early_init_f：配置串口引脚

```
int board_early_init_f(void)
{
        setup_iomux_uart();

        return 0;
}
```

```
#define UART_PAD_CTRL  (PAD_CTL_PKE | PAD_CTL_PUE |             \
        PAD_CTL_PUS_100K_UP | PAD_CTL_SPEED_MED |               \
        PAD_CTL_DSE_40ohm   | PAD_CTL_SRE_FAST  | PAD_CTL_HYS)

static iomux_v3_cfg_t const uart1_pads[] = {
        MX6_PAD_UART1_TX_DATA__UART1_DCE_TX | MUX_PAD_CTRL(UART_PAD_CTRL),
        MX6_PAD_UART1_RX_DATA__UART1_DCE_RX | MUX_PAD_CTRL(UART_PAD_CTRL),
};

static void setup_iomux_uart(void)
{
        imx_iomux_v3_setup_multiple_pads(uart1_pads, ARRAY_SIZE(uart1_pads));
}
```

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

```
    • 8
    • 9
    • 10
    • 11
    • 12
    • 13
    • 14
    • 15
    • 16
    • 17
    • 18
    • 19
    • 20
    • 21
    • 22
```

发现函数目的是串口uart1引脚的复用配置，至于函数imx_iomux_v3_setup_multiple_pads就不继续往下追踪了，这里只需要知道函数的功能即可，函数一般不会有太大问题，如果程序运行不了再继续往下追，否则会篇幅太长。

3.4.2.4 init_baud_rate：设置波特率

```
static int init_baud_rate(void)
{
        gd->baudrate = env_get_ulong("baudrate", 10, CONFIG_BAUDRATE);
        return 0;
}
```

```
    • 1
    • 2
    • 3
    • 4
    • 5
    • 6
```

从env_get_ulong函数定义可以知道它是从环境变量里获取十进制的baudrate值，如果没有则使用默认的CONFIG_BAUDRATE。

3.4.2.5 serial_init：初始化串口

（如果不知道函数在哪个文件中定义，可以通过u-boot.map和各个子目录下的.o文件快速定位。）

```
int serial_init(void)
{
        gd->flags |= GD_FLG_SERIAL_READY;
        return get_current()->start();
}


static struct serial_device *get_current(void)
{
        struct serial_device *dev;

        if (!(gd->flags & GD_FLG_RELOC))
                dev = default_serial_console();
        else if (!serial_current)
                dev = default_serial_console();
        else
                dev = serial_current;


        if (!dev) {
#ifdef CONFIG_SPL_BUILD
                puts("Cannot find console\n");
                hang();
#else
                panic("Cannot find console\n");
#endif
        }

        return dev;
}
```

```
    • 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

get_current函数的一进来就是判断是否已经代码重定位了，很明显，执行到这里还没有，所以条件为真，于是通过default_serial_console来获取默认的串口设备，按照u-boot.map文件和各个子目录下的.o文件也不难发现它的定义：

```
__weak struct serial_device *default_serial_console(void)
{
        return &mxc_serial_drv;
}


static struct serial_device mxc_serial_drv = {
        .name   = "mxc_serial",
        .start  = mxc_serial_init,
        .stop   = NULL,
        .setbrg = mxc_serial_setbrg,
        .putc   = mxc_serial_putc,
        .puts   = default_serial_puts,
        .getc   = mxc_serial_getc,
        .tstc   = mxc_serial_tstc,
};
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17

最终会调用到mxc_serial_init函数，往里面继续探究看看：

```
static int mxc_serial_init(void)
{
        _mxc_serial_init(mxc_base, false);

        serial_setbrg();

        return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

继续往里探究，其中mxc_base的定义如下：

```
#define mxc_base        ((struct mxc_uart *)CONFIG_MXC_UART_BASE)
```

- 1
- 2

还没能看清楚使用的是哪个串口，那就继续往下追：

```
#define CONFIG_MXC_UART_BASE            UART1_BASE
```

- 1
- 2

看完参数看函数，看看_mxc_serial_init如何实现：

```
static void _mxc_serial_init(struct mxc_uart *base, int use_dte)
{
        writel(0, &base->cr1);
        writel(0, &base->cr2);

        while (!(readl(&base->cr2) & UCR2_SRST));

        if (use_dte)
                writel(0x404 | UCR3_ADNIMP, &base->cr3);
        else
                writel(0x704 | UCR3_ADNIMP, &base->cr3);

        writel(0x704 | UCR3_ADNIMP, &base->cr3);
        writel(0x8000, &base->cr4);
        writel(0x2b, &base->esc);
        writel(0, &base->tim);

        writel(0, &base->ts);
}
```

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

- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20

最终就是直接操作寄存器了。到此为止，串口初始化函数`serial_init`基本追踪完毕了。

3.4.2.6 display_options：打印u-boot版本、编译时间

接着查看一下u-boot启动时一系列打印辅助信息函数，我们可以通过这些打印信息确定程序是否能执行到这里或者串口等问题。

```
int display_options(void)
{
        char buf[DISPLAY_OPTIONS_BANNER_LENGTH];

        display_options_get_banner(true, buf, sizeof(buf));
        printf("%s", buf);

        return 0;
}


char *display_options_get_banner(bool newlines, char *buf, int size)
{
        return display_options_get_banner_priv(newlines, BUILD_TAG, buf, size);
}


char *display_options_get_banner_priv(bool newlines, const char *build_tag,
                                      char *buf, int size)
{
        int len;

        len = snprintf(buf, size, "%s%s", newlines ? "\n\n" : "",
                       version_string);
        ...
        return buf;
}


const char __weak version_string[] = U_BOOT_VERSION_STRING;


#define U_BOOT_VERSION_STRING U_BOOT_VERSION " (" U_BOOT_DATE " - " \
        U_BOOT_TIME " " U_BOOT_TZ ")" CONFIG_IDENT_STRING


#define U_BOOT_DATE "Feb 05 2021"
#define U_BOOT_TIME "17:14:08"
#define U_BOOT_TZ "+0800"
#define U_BOOT_DMI_DATE "02/05/2021"
#define U_BOOT_BUILD_DATE 0x20210205
#define U_BOOT_EPOCH 1612516448
```

- 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

最终就是串口打印出u-boot的版本以及编译时间等等，继续查看下一个打印函数。

3.4.2.7 display_text_info：打印u-boot的地址

```
static int display_text_info(void)
{
#if !defined(CONFIG_SANDBOX) && !defined(CONFIG_EFI_APP)
        ulong bss_start, bss_end, text_base;

        bss_start = (ulong)&__bss_start;
        bss_end = (ulong)&__bss_end;

#ifdef CONFIG_SYS_TEXT_BASE
        text_base = CONFIG_SYS_TEXT_BASE;
#else
        text_base = CONFIG_SYS_MONITOR_BASE;
#endif

        debug("U-Boot code: %08lX -> %08lX  BSS: -> %08lX\n",
              text_base, bss_start, bss_end);
#endif

        return 0;
}
```

这个函数就只是debug一下u-boot的代码在内存中的位置而已，相对上面的简单一点。

3.4.2.8 print_cpuinfo：打印CPU信息

```
int print_cpuinfo(void)
{
        u32 cpurev;
        __maybe_unused u32 max_freq;

        cpurev = get_cpu_rev();

#if defined(CONFIG_IMX_THERMAL) || defined(CONFIG_IMX_TMU)
        struct udevice *thermal_dev;
        int cpu_tmp, minc, maxc, ret;

        printf("CPU:   Freescale i.MX%s rev%d.%d",
               get_imx_type((cpurev & 0x1FF000) >> 12),
               (cpurev & 0x000F0) >> 4,
               (cpurev & 0x0000F) >> 0);
        max_freq = get_cpu_speed_grade_hz();
        if (!max_freq || max_freq == mxc_get_clock(MXC_ARM_CLK)) {
                printf(" at %dMHz\n", mxc_get_clock(MXC_ARM_CLK) / 1000000);
        } else {
                printf(" %d MHz (running at %d MHz)\n", max_freq / 1000000,
                       mxc_get_clock(MXC_ARM_CLK) / 1000000);
        }
#else
        ...
#endif

#if defined(CONFIG_IMX_THERMAL) || defined(CONFIG_IMX_TMU)
        puts("CPU:   ");
        switch (get_cpu_temp_grade(&minc, &maxc)) {
        case TEMP_AUTOMOTIVE:
                puts("Automotive temperature grade ");
                break;
        case TEMP_INDUSTRIAL:
                puts("Industrial temperature grade ");
                break;
        case TEMP_EXTCOMMERCIAL:
                puts("Extended Commercial temperature grade ");
                break;
        default:
                puts("Commercial temperature grade ");
                break;
        }
        printf("(%dC to %dC)", minc, maxc);
        ret = uclass_get_device(UCLASS_THERMAL, 0, &thermal_dev);
        if (!ret) {
                ret = thermal_get_temp(thermal_dev, &cpu_tmp);

                if (!ret)
                        printf(" at %dC", cpu_tmp);
                else
                        debug(" - invalid sensor data\n");
```

```
        } else {
                debug(" - invalid sensor device\n");
        }
        puts("\n");
#endif

        printf("Reset cause: %s\n", get_reset_cause());
        return 0;
}
```

3.4.2.9 show_board_info：打印单板信息

```
int __weak show_board_info(void)
{
#ifdef CONFIG_OF_CONTROL
        DECLARE_GLOBAL_DATA_PTR;
        const char *model;

        model = fdt_getprop(gd->fdt_blob, 0, "model", NULL);

        if (model)
                printf("Model: %s\n", model);
#endif

        return checkboard();
}
```

由于imx6ull默认配置中定义了`CONFIG_OF_CONTROL`，也就是在u-boot中也使用了设备树文件，所以从设备树中获取“model”属性并打印出来，然后继续调用了`checkboard`函数：

```
int checkboard(void)
{
        if (is_cpu_type(MXC_CPU_MX6ULZ))
                puts("Board: MX6ULZ 14x14 EVK\n");
        else
                puts("Board: MX6ULL 14x14 EVK\n");

        return 0;
}
```

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

3.4.2.10 init_func_i2c：初始化**i2c**并打印

```
static int init_func_i2c(void)
{
        puts("I2C:   ");
        i2c_init_all();
        puts("ready\n");
        return 0;
}
```

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

3.4.2.11 announce_dram_init：打印**“DRAM: ”**字符串

```
static int announce_dram_init(void)
{
        puts("DRAM:  ");
        return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6

3.4.2.12 dram_init：设置**gd**结构体的**ram_size**成员

```
int dram_init(void)
{
        gd->ram_size = imx_ddr_size();

        return 0;
}
```

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

函数并非初始化，其实就只是根据MMDC的配置来设置一下前面所说的gd全局变量。

3.4.2.13 reserve_fdt：预留设备树文件的内存，并设置设备树新的内存地址到**gd->new_fdt**

```
static int reserve_fdt(void)
{
#ifndef CONFIG_OF_EMBED

        if (gd->fdt_blob) {
                gd->fdt_size = ALIGN(fdt_totalsize(gd->fdt_blob), 32);

                gd->start_addr_sp = reserve_stack_aligned(gd->fdt_size);
                gd->new_fdt = map_sysmem(gd->start_addr_sp, gd->fdt_size);
                debug("Reserving %lu Bytes for FDT at: %08lx\n",
                      gd->fdt_size, gd->start_addr_sp);
        }
#endif

        return 0;
}
```

- 1
- 2

```
    • 3
    • 4
    • 5
    • 6
    • 7
    • 8
    • 9
    • 10
    • 11
    • 12
    • 13
    • 14
    • 15
    • 16
    • 17
    • 18
    • 19
    • 20
    • 21
```

3.4.2.14 dram_init_banksize：设置**gd**结构体的**dram**信息

```
__weak int dram_init_banksize(void)
{
        gd->bd->bi_dram[0].start = gd->ram_base;
        gd->bd->bi_dram[0].size = get_effective_memsize();

        return 0;
}
```

```
    • 1
    • 2
    • 3
    • 4
    • 5
    • 6
    • 7
    • 8
```

3.4.2.15 show_dram_config：打印内存的大小

```
static int show_dram_config(void)
{
        unsigned long long size;
        int i;

        debug("\nRAM Configuration:\n");
        for (i = size = 0; i < CONFIG_NR_DRAM_BANKS; i++) {
                size += gd->bd->bi_dram[i].size;
                debug("Bank #%d: %llx ", i,
                      (unsigned long long)(gd->bd->bi_dram[i].start));
#ifdef DEBUG
                print_size(gd->bd->bi_dram[i].size, "\n");
#endif
        }
        debug("\nDRAM:  ");

        print_size(size, "");
        board_add_ram_info(0);
        putc('\n');

        return 0;
}
```

```
    • 1
    • 2
    • 3
    • 4
    • 5
    • 6
    • 7
    • 8
    • 9
    • 10
    • 11
    • 12
    • 13
    • 14
    • 15
    • 16
    • 17
    • 18
    • 19
    • 20
    • 21
    • 22
    • 23
```

这个函数才真正的打印出DRAM的大小，如果还想知道单位等等就继续往`print_size`里查看，这里不继续往里探讨。

3.4.2.16 display_new_sp：打印新的**sp**栈内存地址

```
static int display_new_sp(void)
{
        debug("New Stack Pointer is: %08lx\n", gd->start_addr_sp);

        return 0;
}
```

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

3.4.2.17 reloc_fdt：拷贝设备树内容到新的地址去（重定位），并重新设置**gd->fdt_blob**

```
static int reloc_fdt(void)
{
#ifndef CONFIG_OF_EMBED
        if (gd->flags & GD_FLG_SKIP_RELOC)
                return 0;
        if (gd->new_fdt) {
                memcpy(gd->new_fdt, gd->fdt_blob, fdt_totalsize(gd->fdt_blob));
                gd->fdt_blob = gd->new_fdt;
        }
#endif

        return 0;
}
```

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

3.4.2.18 setup_reloc：设置重定位相关的信息，后面会用到

```
static int setup_reloc(void)
{
        if (gd->flags & GD_FLG_SKIP_RELOC) {
                debug("Skipping relocation due to flag\n");
                return 0;
        }

#ifdef CONFIG_SYS_TEXT_BASE
#ifdef ARM
        gd->reloc_off = gd->relocaddr - (unsigned long)__image_copy_start;
#elif defined(CONFIG_M68K)

        gd->reloc_off = gd->relocaddr - (CONFIG_SYS_TEXT_BASE + 0x400);
#elif !defined(CONFIG_SANDBOX)
        gd->reloc_off = gd->relocaddr - CONFIG_SYS_TEXT_BASE;
#endif
#endif
        memcpy(gd->new_gd, (char *)gd, sizeof(gd_t));

        debug("Relocation Offset is: %08lx\n", gd->reloc_off);
        debug("Relocating to %08lx, new gd at %08lx, sp at %08lx\n",
              gd->relocaddr, (ulong)map_to_sysmem(gd->new_gd),
              gd->start_addr_sp);

        return 0;
}
```

- 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

`setup_reloc`函数计算了重定位的偏移地址值赋给`gd->reloc_off`，并且将gd结构体拷贝到前面`reserve_xxx`系列函数计算得到的新地址`gd->new_gd`，这里所涉及的2个gd结构体成员很重要，`board_init_f`函数返回到`_main`函数里马上就用上了：

```
/* file: arch/arm/lib/crt0.S */
        bl      board_init_f

        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
```

所以接下来的任务就是分析`relocate_code`代码重定位了。

未完待续...