(1条消息) u-boot-2021.01（imx6ull）启动流程分析之五：board_init_r函数分析___ASDFGH的博客-CSDN博客

3.4.5 board_init_r

从函数名称也可以知道，它也是负责一些初始化，但它还有一个目的就是通过层层调用之后启动内核。将相关的宏定义简化一下如下：

```
void board_init_r(gd_t *new_gd, ulong dest_addr)
{

        gd->flags &= ~GD_FLG_LOG_READY;

        if (initcall_run_list(init_sequence_r))
                hang();


        hang();
}
```

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

类似地，同样可以找到函数的重点——`init_sequence_r`函数指针数组，将宏定义简化一下如下：

```
static init_fnc_t init_sequence_r[] = {
        initr_trace,
        initr_reloc,
    initr_caches,
    initr_reloc_global_data,
    initr_barrier,
        initr_malloc,
        log_init,
        initr_bootstage,
        initr_console_record,
    initr_of_live,
    initr_dm,
    board_init,
        efi_memory_init,
    initr_binman,
        initr_dm_devices,
```

```
        stdio_init_tables,
        serial_initialize,
        initr_announce,
    INIT_FUNC_WATCHDOG_RESET
    initr_mmc,
    initr_env,
    stdio_add_devices,
        initr_jumptable,
        console_init_r,
    interrupt_init,
    initr_ethaddr,
    board_late_init,
    initr_net,
    run_main_loop,
};
```

函数比较多，所以和`board_init_f`函数一样，也只挑一部分比较重要的函数研究，其余的函数也类似地自行研究即可。

3.4.5.1 serial_initialize：注册各个厂家的串口驱动

```
int serial_initialize(void)
{
        atmel_serial_initialize();
        mcf_serial_initialize();
        mpc85xx_serial_initialize();
        mxc_serial_initialize();
```

```
        ns16550_serial_initialize();
        pl01x_serial_initialize();
        pxa_serial_initialize();
        sh_serial_initialize();
        mtk_serial_initialize();

        serial_assign(default_serial_console()->name);

        return 0;
}
```

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

看到前面那一大坨的`xxx_serial_initialize`，这里包括不同芯片厂家的串口“初始化”，但其实它并不是真正的“初始化”，查看它的内部实现就会发现，里面都是使用`serial_register`来将各个厂商的串口加入到链表。加入到链表也得用才行，那就是`serial_assign`函数：

```
int serial_assign(const char *name)
{
        struct serial_device *s;

        for (s = serial_devices; s; s = s->next) {
                if (strcmp(s->name, name))
                        continue;
                serial_current = s;
                return 0;
        }

        return -EINVAL;
}
```

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

而函数的实参default_serial_console()->name似曾相识，在分析board_init_f函数时就已经接触过了。看了前面这些调用，其实serial_initialize函数的主要功能是将串口注册进链表，然后从链表里“挑”出一个串口来作为stdin/stdout/stderr，imx6ull芯片就很明显会用到mxc_serial_drv驱动，而不像函数名一样误认为是初始化所有厂家的SoC串口。

3.4.5.2 initr_announce：debug一下u-boot重定位后的地址

```
static int initr_announce(void)
{
        debug("Now running in RAM - U-Boot at: %08lx\n", gd->relocaddr);
        return 0;
}
```

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

3.4.5.3 initr_mmc：初始化mmc设备

```
static int initr_mmc(void)
{
        puts("MMC:   ");
        mmc_initialize(gd->bd);
        return 0;
}
```

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

继续往里面探索：

```
int mmc_initialize(struct bd_info *bis)
{
        static int initialized = 0;
        int ret;
        if (initialized)
                return 0;
        initialized = 1;

#if !CONFIG_IS_ENABLED(BLK)
#if !CONFIG_IS_ENABLED(MMC_TINY)
        mmc_list_init();
#endif
#endif
        ret = mmc_probe(bis);
```

```
        if (ret)
                return ret;

#ifndef CONFIG_SPL_BUILD
        print_mmc_devices(',');
#endif

        mmc_do_preinit();
        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

里面主要调用了3个函数，其中mmc_list_init函数初始化链表，后面初始化mmc设备会用到，但不用深入了解，知道作用即可；主要还是看mmc_probe和mmc_do_preinit，先看mmc_probe：

```
static int mmc_probe(struct bd_info *bis)
{
        if (board_mmc_init(bis) < 0)
                cpu_mmc_init(bis);

        return 0;
}
```

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

只有两个函数：board_mmc_init和cpu_mmc_init；

```
__weak int board_mmc_init(struct bd_info *bis)
{
        return -1;
}
```

- 1
- 2
- 3
- 4
- 5

所以判断返回之后还会调用到cpu_mmc_init:

```
int cpu_mmc_init(struct bd_info *bis)
{
        return fsl_esdhc_mmc_init(bis);
}
```

- 1
- 2
- 3
- 4
- 5

继续看调用:

```
int fsl_esdhc_mmc_init(struct bd_info *bis)
{
        struct fsl_esdhc_cfg *cfg;

        cfg = calloc(sizeof(struct fsl_esdhc_cfg), 1);
        cfg->esdhc_base = CONFIG_SYS_FSL_ESDHC_ADDR;
        cfg->sdhc_clk = gd->arch.sdhc_clk;
        return fsl_esdhc_initialize(bis, cfg);
}
```

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

函数里面使用“基地址”和“时钟频率”传递到fsl_esdhc_initialize函数中初始化，函数里面调用的内容分析起来比较多，就只列出个别重要的操作:

```
fsl_esdhc_initialize(bis, cfg);
    fsl_esdhc_cfg_to_priv(cfg, priv);
        priv->esdhc_regs = (struct fsl_esdhc *)(unsigned long)(cfg->esdhc_base);
        priv->bus_width = cfg->max_bus_width;
        priv->sdhc_clk = cfg->sdhc_clk;
        ...
    fsl_esdhc_init(priv, plat);
                regs = priv->esdhc_regs;
                esdhc_reset(regs);
                esdhc_setbits32(&regs->sysctl, ...
```

```
                esdhc_write32(&regs->mixctrl, 0);
                cfg = &plat->cfg;
                cfg->name = "FSL_SDHC";
                cfg->ops = &esdhc_ops;
                        .getcd          = esdhc_getcd,
                        .init           = esdhc_init,
                        .send_cmd       = esdhc_send_cmd,
                        .set_ios        = esdhc_set_ios,
                ...
        mmc_create(&plat->cfg, priv);
                mmc->cfg = cfg;
                mmc->priv = priv;
                bdesc = mmc_get_blk_desc(mmc);
                bdesc->if_type = IF_TYPE_MMC;
                bdesc->removable = 1;
                bdesc->devnum = mmc_get_next_devnum();
                bdesc->block_read = mmc_bread;
                bdesc->block_write = mmc_bwrite;
                bdesc->block_erase = mmc_berase;
                mmc_list_add(mmc);
```

需要注意的是，上面这些内容主要是配置Soc里面的eSDHC控制器和一些mmc驱动的属性和操作函数。也就是说mmc设备还没初始化，前面说的`mmc_do_preinit`函数还没研究呢，回去看看：

```
void mmc_do_preinit(void)
{
        struct udevice *dev;
        struct uclass *uc;
        int ret;
```

```
        ret = uclass_get(UCLASS_MMC, &uc);
        if (ret)
                return;
        uclass_foreach_dev(dev, uc) {
                struct mmc *m = mmc_get_mmc_dev(dev);

                if (!m)
                        continue;
                if (m->preinit)
                        mmc_start_init(m);
        }
}
```

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

前面的遍历是为了得到找到mmc设备，一旦找到该设备，就进行以下函数调用：

```
mmc_start_init(m);
        mmc_get_op_cond(mmc);
                mmc_power_init(mmc);
                mmc_power_cycle(mmc);
                mmc->cfg->ops->init(mmc);
                mmc_set_initial_state(mmc);
                mmc_go_idle(mmc);
                ...
```

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

可以看到，前面配置的那些属性和函数在这里就用上了。

3.4.5.4 initr_ethaddr：设置网卡mac地址

```
static int initr_ethaddr(void)
{
        struct bd_info *bd = gd->bd;


        eth_env_get_enetaddr("ethaddr", bd->bi_enetaddr);

        return 0;
}
```

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

从函数定义可以知道函数从环境变量`ethaddr`来获取值来设置`bd->bi_enetaddr`。

3.4.5.5 initr_env：初始化环境变量

```
static int initr_env(void)
{

        if (should_load_env())
                env_relocate();
        else
                env_set_default(NULL, 0);

        if (IS_ENABLED(CONFIG_OF_CONTROL))
                env_set_hex("fdtcontroladdr",
                            (unsigned long)map_to_sysmem(gd->fdt_blob));


        image_load_addr = env_get_ulong("loadaddr", 16, image_load_addr);

        return 0;
}
```

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

- 16
- 17
- 18

函数一开始先是调用should_load_env函数判断是否需要加载环境变量:

```
static int should_load_env(void)
{
        if (IS_ENABLED(CONFIG_OF_CONTROL))
                return fdtdec_get_config_int(gd->fdt_blob,
                                             "load-environment", 1);

        if (IS_ENABLED(CONFIG_DELAY_ENVIRONMENT))
                return 0;

        return 1;
}
```

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

由于imx6ull默认配置中定义了CONFIG_OF_CONTROL宏，所以继续调用fdtdec_get_config_int函数读取设备树信息:

```
int fdtdec_get_config_int(const void *blob, const char *prop_name,
                          int default_val)
{
        int config_node;

        debug("%s: %s\n", __func__, prop_name);
        config_node = fdt_path_offset(blob, "/config");
        if (config_node < 0)
                return default_val;
        return fdtdec_get_int(blob, config_node, prop_name, default_val);
}
```

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

整个函数的调用流程可以大概看出程序是在根节点下的config节点找到"load-environment"属

性，如果找不到则使用默认值1。

回到initr_env函数之后，继续调用env_relocate函数，从名字就可以猜测它是重定位环境变量的。然后接着就是设置环境变量“fdtcontroladdr”，最后获取环境变量“loadaddr”的值来设置全局变量“image_load_addr”，这个变量保存的就是kernel存放的地址。

3.4.5.6 board_late_init：根据cpu型号来设置环境变量

```
int board_late_init(void)
{
#ifdef CONFIG_CMD_BMODE
        add_board_boot_modes(board_boot_modes);
#endif

#ifdef CONFIG_ENV_VARS_UBOOT_RUNTIME_CONFIG
        if (is_cpu_type(MXC_CPU_MX6ULZ))
                env_set("board_name", "ULZ-EVK");
        else
                env_set("board_name", "EVK");
        env_set("board_rev", "14X14");
#endif

        return 0;
}
```

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

3.4.5.7 initr_net：初始化网卡

它是网卡的初始化函数，在使用不同厂家的网卡芯片需要进行修改，函数定义如下：

```
static int initr_net(void)
{
        puts("Net:   ");
        eth_initialize();
#if defined(CONFIG_RESET_PHY_R)
        debug("Reset Ethernet PHY\n");
        reset_phy();
```

```
#endif
        return 0;
}
```

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

往函数eth_initialize里面继续探究：

```
int eth_initialize(void)
{
        int num_devices = 0;
        struct udevice *dev;

        eth_common_init();
    ...
}
```

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

继续往eth_common_init里走：

```
void eth_common_init(void)
{
        bootstage_mark(BOOTSTAGE_ID_NET_ETH_START);
#if defined(CONFIG_MII) || defined(CONFIG_CMD_MII) || defined(CONFIG_PHYLIB)
        miiphy_init();
#endif

#ifdef CONFIG_PHYLIB
        phy_init();
#endif
}
```

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

- 10
- 11
- 12

继续往phy_init里走:

```
int phy_init(void)
{
#ifdef CONFIG_NEEDS_MANUAL_RELOC

        struct list_head *head = &phy_drivers;

        head->next = (void *)head->next + gd->reloc_off;
        head->prev = (void *)head->prev + gd->reloc_off;
#endif

#ifdef CONFIG_B53_SWITCH
        phy_b53_init();
#endif
#ifdef CONFIG_MV88E61XX_SWITCH
        phy_mv88e61xx_init();
#endif
...
}
```

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

由于函数内部是初始化各个网卡厂家的芯片，所以这里仅列出几个。以mv88e61xx系列为例，查看phy_mv88e61xx_init函数内部实现:

```
int phy_mv88e61xx_init(void)
{
        phy_register(&mv88e61xx_driver);
        phy_register(&mv88e609x_driver);
        phy_register(&mv88e6071_driver);

        return 0;
}
```

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

可以看到，在phy_init函数里根据宏定义来决定使用哪个厂商的网卡，而在对应的厂商函数内部就是不同型号的网卡芯片进行注册。这部分对于移植u-boot来说相当重要，尤其是开发板使用与公版不同厂商的网卡芯片。

3.4.5.8 run_main_loop：启动内核/解析命令行输入

一般来说，如果程序执行到这里，后续几乎不会出现什么奇怪的现象了，因为不管哪个SoC，这部分都是属于相同的。对于移植u-boot，后续一般不需要理会，但如果需要添加新功能，还是可以继续往下探究。函数定义如下：

```
static int run_main_loop(void)
{
#ifdef CONFIG_SANDBOX
        sandbox_main_loop_init();
#endif

        for (;;)
                main_loop();
        return 0;
}
```

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

比较显眼的就是main_loop函数，函数调用比较复杂，所以只列出函数调用过程：

```
main_loop
        env_set("ver", ...
        cli_init
        s = bootdelay_process();
                s = env_get("bootdelay");
                bootdelay = s ? (int)...
                bootdelay = fdtdec_get_config_int(gd->fdt_blob, "bootdelay", ...
                s = env_get("bootcmd");
                stored_bootdelay=bootdelay;
                return s;
        cli_process_fdt(&s)
                cli_secure_boot_cmd(s);
```

```
autoboot_command(s);
        if(s && (stored_bootdelay == -2 ||
        (stored_bootdelay != -1 && !abortboot(stored_bootdelay))))
                run_command_list(s, -1, 0);
        parse_string_outer(buff, FLAG_PARSE_SEMICOLON);
                setup_string_in_str(&input, p);
                parse_stream_outer(&input, flag);
                parse_stream(&temp, ...
                    run_list(ctx.list_head);
                        run_list_real(pi);
                            run_pipe_real(pi);
                                cmd_process(flag, child->argc, ...
                                    find_cmd
                                    cmd_call
                                    cmd_usage
cli_loop();
        bootstage_mark(BOOTSTAGE_ID_ENTER_CLI_LOOP);
        parse_file_outer();
                setup_file_in_str(&input);
    parse_stream_outer(&input, FLAG_PARSE_SEMICOLON);
```

- 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

看到上面的函数调用关系可以知道，`main_loop`函数先获取环境变量`bootdelay`和`bootcmd`，如果u-boot配置了`CONFIG_OF_CONTROL`，则在`cli_process_fdt`函数里获取设备树的`bootcmd`和`bootsecure`属性，如果返回的`bootsecure`属性不为0，则进入`cli_secure_boot_cmd`函数解析并执行设备树中的`bootcmd`属性内容；否则进入到`autoboot_command`函数里倒计时，倒计时结束后就会解析并执行环境变量`bootcmd`的内容；如果倒计时过程中检测到串口有输入，就会退出该函数，进入`cli_loop`函数里面解析命令行的输入。

至于find_cmd、cmd_call和cmd_usage三个函数的实现，必须先看u-boot命令在源码中的组织形式：

```
#define U_BOOT_CMD(_name, _maxargs, _rep, _cmd, _usage, _help)          \
        U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, NULL)


#define U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, _comp) \
        ll_entry_declare(struct cmd_tbl, _name, cmd) =                  \
                U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,  \
                                          _usage, _help, _comp);


#define ll_entry_declare(_type, _name, _list)                           \
        _type _u_boot_list_2_##_list##_2_##_name __aligned(4)           \
                        __attribute__((unused,                          \
                        section(".u_boot_list_2_"#_list"_2_"#_name)))

#define U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,          \
                                  _usage, _help, _comp)                 \
                { #_name, _maxargs,                                     \
                  _rep ? cmd_always_repeatable : cmd_never_repeatable,  \
                  _cmd, _usage, _CMD_HELP(_help) _CMD_COMPLETE(_comp) }


# define _CMD_COMPLETE(x) x,
# define _CMD_HELP(x) x,
```

就以“version”命令为例，解析简化之后它的定义：

```
U_BOOT_CMD(
        version,        1,              1,      do_version,
        "print monitor, compiler and linker version",
        ""
```

```
);
```

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

经过转化之后的定义就是：

```
struct cmd_tbl _u_boot_list_2_cmd_2_version __aligned(4) \
    __attribute__((unused, section(".u_boot_list_2_cmd_2_version"))) = {
    version, 1, cmd_always_repeatable,
        do_version, "print monitor, compiler and linker version", "", NULL
};
```

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

其中，结构体`cmd_tbl`原型如下：

```
struct cmd_tbl {
        char            *name;
        int             maxargs;
        int             (*cmd_rep)(struct cmd_tbl *cmd, int flags, int argc,
                                   char *const argv[], int *repeatable);

        int             (*cmd)(struct cmd_tbl *cmd, int flags, int argc,
                               char *const argv[]);
        char            *usage;
#ifdef  CONFIG_SYS_LONGHELP
        char            *help;
#endif
#ifdef CONFIG_AUTO_COMPLETE

        int             (*complete)(int argc, char *const argv[],
                                    char last_char, int maxv, char *cmdv[]);
#endif
};
```

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

- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

所以，回到“查找命令”的`find_cmd`函数：

```
struct cmd_tbl *find_cmd(const char *cmd)
{
        struct cmd_tbl *start = ll_entry_start(struct cmd_tbl, cmd);
        const int len = ll_entry_count(struct cmd_tbl, cmd);
        return find_cmd_tbl(cmd, start, len);
}
```

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

里面分别调用了`ll_entry_start`宏和`ll_entry_count`宏来获得命令所在的区间范围：

```
#define ll_entry_start(_type, _list)                                    \
({                                                                      \
        static char start[0] __aligned(4) __attribute__((unused,        \
                section(".u_boot_list_2_"#_list"_1")));                 \
        (_type *)&start;                                                \
})
```

```
#define ll_entry_count(_type, _list)                                    \
({                                                                  \
        _type *start = ll_entry_start(_type, _list);            \
        _type *end = ll_entry_end(_type, _list);                \
        unsigned int _ll_result = end - start;                  \
        _ll_result;                                             \
})

#define ll_entry_end(_type, _list)                                      \
({                                                                      \
        static char end[0] __aligned(4) __attribute__((unused,          \
                section(".u_boot_list_2_"#_list"_3")));                 \
        (_type *)&end;                                                  \
})
```

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

- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23

根据上面宏定义也可以看出回去程序所在的section段的区间里计算_type类型的个数。获得个数之后，`find_cmd`函数里就可以调用`find_cmd_tbl(cmd, start, len);`来找出匹配的命令并且作为返回值:

```
struct cmd_tbl *find_cmd_tbl(const char *cmd, struct cmd_tbl *table,
                             int table_len)
{
    ...
    for (cmdtp = table; cmdtp != table + table_len; cmdtp++) {
                if (strncmp(cmd, cmdtp->name, len) == 0) {
                        if (len == strlen(cmdtp->name))
                                return cmdtp;
                        ...
                }
        }
}
```

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

既然能找出命令所在的结构体的位置了，那调用它的成员函数就比较简单了。首先就是调用该命令的执行函数:

```
static int cmd_call(struct cmd_tbl *cmdtp, int flag, int argc,
                    char *const argv[], int *repeatable)
{
        int result;

        result = cmdtp->cmd_rep(cmdtp, flag, argc, argv, repeatable);
        if (result)
                debug("Command failed, result=%d\n", result);
        return result;
```

```
}
```

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

如果命令输入不正确等情况，那就返回调用者调用其他函数打印的用法:

```
int cmd_usage(const struct cmd_tbl *cmdtp)
{
        printf("%s - %s\n\n", cmdtp->name, cmdtp->usage);

#ifdef  CONFIG_SYS_LONGHELP
        printf("Usage:\n%s ", cmdtp->name);

        if (!cmdtp->help) {
                puts ("- No additional help available.\n");
                return 1;
        }

        puts(cmdtp->help);
        putc('\n');
#endif
        return 1;
}
```

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

至此，启动流程讲得差不多了。但是别忘了u-boot的最终目的是启动内核，所以还没有结束，接着往下看！

未完待续...