UBOOT 的编译命令

直接一次性编译

```
make O=am335x CROSS_COMPILE=arm-arago-linux-gnueabi ARCH=arm
am335x_evm
```

配置

```
make ARCH=arm CROSS_COMPILE=arm-arago-linux-gnueabi- am335x_evm_config
```

编译

```
make ARCH=arm CROSS_COMPILE=arm-arago-linux-gnueabi-
```

清理

```
make clean ARCH=arm CROSS_COMPILE=arm-arago-linux-gnueabi-
make distclean ARCH=arm CROSS_COMPILE=arm-arago-linux-gnueabi-
```

编译器环境变量的设置

这个环境变量是 TI 的 SDK 包里面带的编译器，不是之前的 arm-gcc-

```
export PATH=$PATH:/mnt/disk1/ti-sdk-am335x-evm-05.05.00.00/linux-devkit/bin/
```

UBOOT 里面的 MLO（u-boot-spl）

如果使用 NAND 启动，那么这个文件就是相当于 NBOOT，进行第一次的引导

这个 MLO 实际上就是 u-boot-spl.bin 生成的，

在编译完 uboot 后，SPL 的目录里面会产生了许多的.o 文件，这里文件就是 uboot 的文件，

可以打开 Makefile，有一些对应的宏定义，可以取消，减少 MLO 文件的大小

UBOOT 的链接脚本 lds

UBOOT\arch\arm\cpu\armv7\u-boot.lds

正常运行 UBOOT 的 lds

UBOOT\arch\arm\cpu\armv7\omap-common\u-boot.lds

这个是 nboot，加载 uboot 用

有 2 个 lds，不同的作用，注意要区别开

增加新的单板支持

在 boards.cfg 文件中，找到加入，例如

单板名字 arm armv7 对应 board 的目录 ti ti81xx

以后就可以执行 make 单板名字 来生成 uboot，这里被 ti 改写了，所以不是原版的 uboot 生成方法

一些代码的定位

u-boot-2011.09-psp04.06.00.08/arch/arm/cpu/armv7

这个目录下的几个文件，start.s 这个是程序的入口执行文件

u-boot-2011.09-psp04.06.00.08/arch/arm/cpu/armv7/omap-common

u-boot-2011.09-psp04.06.00.08/arch/arm/cpu/armv7/ti81xx

这 2 个目录是和平台板子相关，AM335X 是 ti81xx 的版本

以上都是和 CPU 有关

u-boot-2011.09-psp04.06.00.08/arch/lib

ARM 平台的公用代码

u-boot-2011.09-psp04.06.00.08/lib

通用的库代码，无论什么平台都编译

board/ti/xxx 这个目录就是单板的配置

Makefile 文件分析

在终端中输入后

make ARCH=arm CROSS_COMPILE=arm-arago-linux-gnueabi- am335x_evm_config

命令后，会生成 3 个文件

1、.boards.depend

2、include/config.h

```
/* Automatically generated - do not edit */
#define CONFIG_BOARDDIR board/ti/am335x
#include <config_cmd_defaults.h>
#include <config_defaults.h>
#include <configs/am335x_evm.h>
#include <asm/config.h>
```


3、include/config.mk

```
ARCH   = arm
CPU    = armv7
BOARD  = am335x
VENDOR = ti
SOC    = ti81xx
```

终端中会输出

```
awk '(NF && $1 !~ /^#/) { print $1 ": " $1 "_config; $(MAKE)" }' boards.cfg > .boards.depend
Configuring for am335x_evm board...
```

支持的配置 am335x 配置有

```
am335x_evm
am335x_evm_restore_flash
am335x_evm_spiboot
```


解析 Makefile 文件，

```
sinclude $(obj).boards.depend

$(obj).boards.depend:   boards.cfg

    awk '(NF && $$1 !~ /^#/) { print $$1 ": " $$1 "_config; $$(MAKE)" }' $< > $@
```

我们搜索 am335x_evm 并没有在 makefile 文件中找到对应的关键字，但是发现 boards.cfg 中有此关键词，

可能 am335x_evm_config 输入后，是到 boards.cfg 中寻找的

通过 TI CCS 调试 uboot

通过 makefile 文件建立 ccs 工程

1、打开 CCS，选择 File->New->Project

2、打开新建窗口后，选择 c/c++下的 Makefile Project with Existing Code

3、点击下一步，选择 uboot 的存放的目录，然后点击完成

4、等待右下角的进度，一直到达 100%后，在继续操作

5、最后在属性设置里，取消 C 的自动编译

调试的时候，如果目标代码是汇编，则不会现实源码，只有 C 才会现实出

uboot 的代码运行流程

u-boot-2011.09-psp04.06.00.08/arch/arm/cpu/armv7

start.s 入口运行文件

bl save_boot_params 跳转到 lowlevel_init.S

该文件在(arch\arm\cpu\armv7\omap-common)，如果是 MLO 则会定义 CONFIG_SPL_BUILD 宏，uboot 没有定义，保存 CPU ROM 的参数到一个地方中

bl cpu_init_crit SPL 里面调用，初始化底层相关的（函数就在本文件中最下面）

bl lowlevel_init 保存旧的堆栈指针，设置新的堆栈指针

在 lowlevel_init.S (arch\arm\cpu\armv7\omap-common)文件中

bl s_init 此函数在 Evm.c (board\ti\am335xwecon)文件中，

不同的平台初始化不同的内容

初始化堆栈指针

跳转到 C 语言，board_init_f，C 语言从这个函数开始执行

board_init_f 在 Board.c (arch\arm\lib)文件中

init_sequence 数组，继续进行初始化

relocate_code 汇编函数，重新回到 start.s 文件中

relocate_code：在 start.s 文件中

主要实现了 uboot 的重新定位代码，和 bss 段的清零

```
ldr r0, _board_init_r_ofs    通过这个方式获取 board_init_r 函数入口
adr r1, _start
add lr, r0, r1
mov pc, lr    最后这里个到 C 函数，board_init_r 里面
```

board_init_r 第 2 阶段的初始化，在 board.c 文件中

```
enable_caches    这个函数没有做什么内容
board_init       第 2 次初始化平台，这里可以初始化其他内容了，和平台相关
mem_malloc_init  初始化 malloc 内存
nand_init        初始化 NAND (CONFIG_CMD_NAND 需要定义)
mmc_initialize   初始化 SD (CONFIG_GENERIC_MMC 需要定义)
```

/drivers/mmc/mmc.c
文件中
board_mmc_init 平台相关
omap_mmc_init (ID) /drivers/mmc/omap_hsmmc.c
文件中
env_relocate 初始化环境变量 /common/env_common.c
文件中
stdio_init
jumptable_init
console_init_r
misc_init_r 平台相关，目前不知道什么用
interrupt_init
enable_interrupts
eth_initialize(gd->bd); 初始化以太网 /net/eth.c 文件中
board_eth_init 平台相关的代码中
cpsw_register /driver/net/cpsw.c 注册一个以太网设备
main_loop(); 进入控制台

init_sequence 数组里的内容

```
#if defined(CONFIG_ARCH_CPU_INIT)
    arch_cpu_init,        /* basic arch cpu dependent setup */
#endif
#if defined(CONFIG_BOARD_EARLY_INIT_F)
    board_early_init_f,
#endif
    timer_init,  Timer.c (arch¥arm¥cpu¥armv7¥omap-common)默认的定时器，刚开始运行uboot时，那个倒计时多少时间后按空格进入uboot
#ifdef CONFIG_FSL_ESDHC
    get_clocks,
#endif
    env_init,         Env_nand.c (common)  uboot默认的参数，都在这里初始化
    init_baudrate, 本文件中，初始化uboot使用的默认的串口波特率
    serial_init,    Serial.c (drivers¥serial)初始化串口，在UBOOT中，使用的默认串口名字是叫NS16550，可能是因为原来就存在此代码的缘故，然后继续调用了ns16550.c里面的函数初始化串口
    console_init_f, Console.c (common)初始化控制台
    display_banner, 本文件中，输出一些信息，代表运行到了这里
#if defined(CONFIG_DISPLAY_CPUINFO)
```

```
    print_cpuinfo,        /* display cpu info (and speed) */
#endif
#if defined(CONFIG_DISPLAY_BOARDINFO)
    checkboard,       /* display board info */
#endif
#if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT_I2C)
    init_func_i2c, 本文件中, 初始化IIC, 然后继续调用Omap24xx_i2c.c (drivers¥i2c),
具体的CPU IIC初始化的实现
#endif
    dram_init,        /* configure available RAM banks */
```


UBOOT 用到的结构体

```
register volatile gd_t *gd asm ("r8")
```

此结构体记录了 UBOOT 所有的参数内容

```
typedef struct  global_data {
    bd_t     *bd;  这个还是一个结构体
    unsigned long    flags;
    unsigned long    baudrate;  UBOOT串口终端所使用的串口波特率
    unsigned long    have_console;     串口控制台是否初始化过?=1初始化过

//UBOOT环境变量相关的变量
    unsigned long    env_addr;    这个是地址, 存放了uboot使用的环境变量参数存放在内
存中对应的地址, 例如记录了uboot的ip地址, 进入uboot默认的延时时间等, 和
default_environment对应起来
    unsigned long    env_valid;   默认的环境变量是否有效, =1代表有效

    unsigned long    fb_base;/* base address of frame buffer */
#ifdef CONFIG_FSL_ESDHC
    unsigned long    sdhc_clk;
#endif
#ifdef CONFIG_AT91FAMILY
    /* "static data" needed by at91's clock.c */
    unsigned long    cpu_clk_rate_hz;
    unsigned long    main_clk_rate_hz;
    unsigned long    mck_rate_hz;
    unsigned long    plla_rate_hz;
    unsigned long    pllb_rate_hz;
    unsigned long    at91_pllb_usb_init;
#endif

#ifdef CONFIG_ARM
```

```
    /* "static data" needed by most of timer.c on ARM platforms */
    unsigned long    timer_rate_hz;
    unsigned long    tbl;
    unsigned long    tbu;
    unsigned long long   timer_reset_value;
    unsigned long    lastinc;
#endif

#ifdef CONFIG_IXP425
    unsigned long    timestamp;
#endif
    unsigned long    relocaddr;   /* Start address of U-Boot in RAM */
    phys_size_t  ram_size;    内存的大小
    unsigned long    mon_len;  uboot整个bin文件的大小 代码段+bss段
    unsigned long    irq_sp;  IRQ中断堆栈指针
    unsigned long    start_addr_sp;   /* start_addr_stackpointer */
    unsigned long    reloc_off;
#if !(defined(CONFIG_SYS_ICACHE_OFF) && defined(CONFIG_SYS_DCACHE_OFF))
    unsigned long    tlb_addr;
#endif
    void     **jt;        /* jump table */
    char     env_buf[32]; /* buffer for getenv() before reloc. */
} gd_t;

typedef struct bd_info {
  int            bi_baudrate;  串口控制台波特率
  unsigned long     bi_ip_addr;    IP 地址
  ulong         bi_arch_number;   传递给 LINUX 内核，告诉当前是板子的 ID
  ulong         bi_boot_params;   传递给 linux 内核，告诉其参数存放的位置
  struct                /* RAM configuration */
  {
    ulong start;
    ulong size;

  }bi_dram[CONFIG_NR_DRAM_BANKS];这个用于给 LINUX 传递启动信息的，内存大小和起始地址，如果有多块内存，则这个变量是一个数组
} bd_t;
```

SPL 的代码

/arch/arm/cpu/armv7/start.s 入口

/arch/arm/cpu/armv7/omap-common C 文件，

Uboot 的重定位

relocate_code: 要有 pie 选项(arm-linux-ld 命令中)

```
    mov r4, r0      /* save addr_sp */          新的堆栈地址
    mov r5, r1      /* save addr of gd */       gd_t 变量的内容地址,因为重定位后,这个地址会改变
    mov r6, r2      /* save addr of destination */   目标地址

    /* Set up the stack                                */
stack_setup:
    mov sp, r4          设置堆栈
    adr   r0, _start     uboot 的起始运行地址

    cmp r0, r6
    moveq     r9, #0            /* no relocation. relocation offset(r9) = 0 */   如果代码段已经是目标了,那么不要复制了直接跳转到 clear_bss
    beq  clear_bss        /* skip relocation */

    mov r1, r6                  /* r1 <- scratch for copy_loop */   代码段的目标地址
    ldr    r3, _image_copy_end_ofs                  要复制的长度
    add  r2, r0, r3         /* r2 <- source end address       */   代码段的结束地址
开始循环复制代码段
copy_loop:
    ldmia       r0!, {r9-r10}          /* copy from source address [r0]    */
    stmia       r1!, {r9-r10}          /* copy to   target address [r1]    */
    cmp r0, r2                  /* until source end address [r2]    */
    blo   copy_loop
```

重定位代码，修改代码为新的地址

```
#ifndef CONFIG_SPL_BUILD
```

R0 是目标内存，修改后放入目标的内存中

```
    ldr    r0, _TEXT_BASE           /* r0 <- Text base */  代码段的基地址
    sub   r9, r6, r0        /* r9 <- relocation offset */    重定位的偏移值   r9=代码段目标地址 – 代码段的基地址

    ldr    r10, _dynsym_start_ofs  /* r10 <- sym table ofs */ r10=动态起始地址
    add  r10, r10, r0           /* r10 <- sym table in FLASH */ r10 = 动态符号表的目标存放地址

```

r2 是 rel_dyn 段的内容,这个段就是代表要修改的数据.

这个段都是放一个要修改的值,然后放一个标记.

偏移值分为两种：相对位移和绝对位移

```
    ldr    r2, _rel_dyn_start_ofs    /* r2 <- rel dyn start ofs */ 要修改地址的信息的一些变量都放在这里
    add  r2, r2, r0         /* r2 <- rel dyn start in FLASH */
```

r3 是放结束的地址，用来判断是否结束

```
ldr   r3, _rel_dyn_end_ofs     /* r3 <- rel dyn end ofs */
add  r3, r3, r0        /* r3 <- rel dyn end in FLASH */
```


fixloop:

从 R2 取出要修改的值,

```
ldr   r0, [r2]          /* r0 <- location to fix up, IN FLASH! */
add  r0, r0, r9         /* r0 <- location to fix up in RAM */
```

从 R2 的下一个位置读取一个值,这个值是用来判断是运行 fixrel 还是 fixabs 用

```
ldr   r1, [r2, #4]
and  r7, r1, #0xff
```

这里判断是相对位移还是绝对位移,然后修改

```
cmp r7, #23                /* relative fixup? */
beq  fixrel
cmp r7, #2                 /* absolute fixup? */
beq  fixabs
/* ignore unknown type of fixup */
b     fixnext
```

fixabs: 修改绝对位移，就是运行前才可以确定的

```
/* absolute fix: set location to (offset) symbol value */
mov r1, r1, LSR #4          /* r1 <- symbol index in .dynsym */
add  r1, r10, r1            /* r1 <- address of symbol in table */
ldr   r1, [r1, #4]          /* r1 <- symbol value */
add  r1, r1, r9         /* r1 <- relocated sym addr */
b     fixnext
```


fixrel: 修改相对位移，编译时就确定了地址

```
/* relative fix: increase location by offset */
ldr   r1, [r0]
add  r1, r1, r9
```

fixnext: 写入本次修改的内容，判断是否运行完成，

```
str   r1, [r0]   将 r1 写入目标内存
add  r2, r2, #8         /* each rel.dyn entry is 8 bytes */
cmp r2, r3    判断是否结束,
blo   fixloop   没有结束,继续循环
b     clear_bss  结束了
#endif     /* #ifndef CONFIG_SPL_BUILD */
```

uboot 命令

环境变量

printenv 打印当前的环境变量

运行

go

bootm 0x82000000

boot - boot default, i.e., run 'bootcmd'

根据 bootcmd 环境变量运行命令

bootd - boot default, i.e., run 'bootcmd' 根据 bootcmd 环境变量运行命令

nand flash

以太网

ping 192.168.1.2

tftp