LƯU AN PHÚ

Uboot Architecture and Porting

Facebook group: Cùng Nhau Học Linux Kernel

MỤC LỤC

CHƯƠNG 1. Flow source code của uboot

1.1 Mở đầu

Do hệ điều hành có thể nằm trên các vùng nhớ khác nhau như HDD, flash, mmc, sdcard, thậm chí có thể nằm trên internet. Do vậy khi máy tính khởi động, nó cần chạy 1 chương trình đặc biệt dùng để load hệ điều hành. Chương trình đó gọi là boot loader. Nếu như hệ điều hành có thể lưu trữ tại bất cứ đâu, có thể là bộ nhớ nằm trong máy tính hoặc nằm ngoài máy tính (lưu trữ trên internet) thì boot loader thông thường chỉ lưu trữ trên ROM (Read Only Memory). Khi power on, CPU sẽ tự load boot loader và thực hiện câu lệnh đầu tiên của nó. Boot loader sau đó sẽ khởi tạo các tài nguyên khác của hệ thống như CPU, Ram, ethernet, sau đó nó sẽ tiến hành load hệ điều hành.

Boot loader là tên chung định danh cho loại chương trình được chạy trước hệ điều hành và có nhiệm vụ là load hệ điều hành. Trong thực tế boot loader có rất nhiều loại như Grub dùng cho PC, Uboot dùng cho các thiết bị embedded Linux.

Các công ty khi làm product chạy hệ điều hành Linux hoặc Android thường sẽ phát sinh nhu cầu chỉnh sửa code của uboot do mạch đã được làm lại khác với sample board. Ví dụ như thay đổi port serial mặc định, thay đổi nơi lưu trữ OS từ MMC sang sdcard… Trong tài liệu này mình sẽ trình bày kiến trúc source code của Uboot và cách porting nó cho board mới.

Trong tài liệu này mình sẽ đưa ra sample code và thực hành trên phiên bản Uboot mới nhất và chạy trên board Beagle Bone Black. Các bạn có thể tải source code và làm theo chỉ dẫn của hãng. Đây là link hướng dẫn:

https://www.digikey.com/eewiki/display/linuxonarm/BeagleBone+Black

1.2 Uboot basic

Về bản chất thì Uboot là 1 chương trình vi điều khiển. Nó sử dụng trực tiếp địa chỉ vật lý chứ không thông qua virtual memory như Linux. Quá trình chạy của boot loader được chia làm 3 giai đoạn như sau:

1.2.1. First stage boot loader.

Khi vừa bật nguồn thì CPU sẽ ngay lập tức thực thi 1 chương trình nhỏ được lưu trữ trong ROM. Quá trình CPU nhảy tới thực hiện câu lệnh tại địa chỉ cố định trong ROM được thực thi tự động bởi hardware trong chip và cố định đối với từng loại kiến trúc như Arm, intel… Luồng thực thi của chương trình nằm trong ROM này được gọi là first stage boot loader. Nhiệm vụ của first stage nhằm khởi tạo những tài nguyên tối thiểu như 1 core của CPU, static ram, sdcard (nếu như uboot được lưu trữ trên sdcard). Do chương trình first stage boot loader thông thường được viết bởi hang cung cấp chip nên trong tài liệu này mình sẽ không đi sâu.

1.2.2. Second stage boot loader.

Sau khi first stage thực thi xong, nó sẽ load tiếp chương trình thứ 2 là second stage boot loader được lưu trên scard của beage bone black. Sau khi build xong uboot của beagle bone, chúng ta sẽ có 2 file MLO và uboot.img như hình dưới đây:

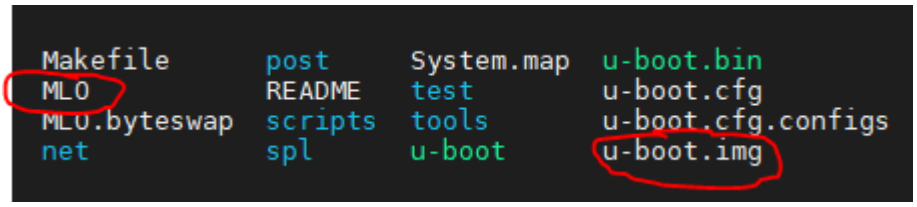


Figure 1 Các file sau khi build uboot

Đây là 2 câu lệnh liên tiếp dùng để ghi uboot vào thẻ nhớ trên board beage bone:

sudo dd if=./u-boot/MLO of=${DISK} count=1 seek=1 bs=128k

sudo dd if=./u-boot/u-boot.img of=${DISK} count=2 seek=1 bs=384k

File chương trình second stage (MLO) được ghi vào block đầu tiên của thẻ sdcard và third stage (u-boot.img) được ghi vào block thứ 2 có địa bắt đầu là 128k. Như vậy sau khi boot rom chạy xong nó sẽ nhảy đến câu lệnh đầu tiên được lưu trữ trên thẻ nhớ và đó chính là câu lệnh đầu tiên của MLO (second stage). Chúng ta sẽ cùng tìm hiểu chi tiết các công việc mà second state thực hiện.

Những câu lệnh đầu tiên mà second stage thực hiện sẽ nằm trong file *arch/arm/cpu/armv7/start.S*. Trong source code u-boot có rất nhiều file start.S, mỗi một loại chip sẽ sử dụng 1 file start.S khác nhau. File này do nhà sản xuất chip như Arm hoặc Intel viết riêng cho chip của họ. Đối với Beagle bone sử dụng armv7 nên file start.S của nó sẽ có đường dẫn như trên. Dưới đây là 1 đoạn code nhỏ của start.S:

```
82          /* the mask ROM code should have PLL and others stable */
83  #ifndef CONFIG_SKIP_LOWLEVEL_INIT
84  #ifdef CONFIG_CPU_V7A
85          bl      cpu_init_cp15
86  #endif
87  #ifndef CONFIG_SKIP_LOWLEVEL_INIT_ONLY
88          bl      cpu_init_crit
89  #endif
90  #endif
91
92          bl      _main
93
```

Vì chưa được khởi tạo tài nguyên như memory nên start.S thực thi với rất nhiều giới hạn. Nó không truy cập vào bộ nhớ, không sử dụng variable. Thay vào đó nó chỉ thao tác trên các thanh ghi mà thôi. Start.S làm các công việc rất cơ bản như khởi tạo cpu ở mức basic… Nhiệm vụ của nó là chuẩn bị tài nguyên để thực thi hàm _main.

Hàm _main nằm trong file *arch/arm/lib/crt0_64.S*. Code của nó như sau:

```
 * 1. Set up initial environment for calling board_init_f().
 *    This environment only provides a stack and a place to store
 *    the GD ('global data') structure, both located in some readily
 *    available RAM (SRAM, locked cache...). In this context, VARIABLE
 *    global data, initialized or not (BSS), are UNAVAILABLE; only
 *    CONSTANT initialized data are available. GD should be zeroed
 *    before board_init_f() is called.
 *
 * 2. Call board_init_f(). This function prepares the hardware for
 *    execution from system RAM (DRAM, DDR...) As system RAM may not
 *    be available yet, , board_init_f() must use the current GD to
 *    store any data which must be passed on to later stages. These
 *    data include the relocation destination, the future stack, and
 *    the future GD location.
 *
 * 3. Set up intermediate environment where the stack and GD are the
 *    ones allocated by board_init_f() in system RAM, but BSS and
 *    initialized non-const data are still not available.
 *
 * 4a.For U-Boot proper (not SPL), call relocate_code(). This function
 *    relocates U-Boot from its current location into the relocation
 *    destination computed by board_init_f().
 *
 * 4b.For SPL, board_init_f() just returns (to crt0). There is no
 *    code relocation in SPL.
 *
 * 5. Set up final environment for calling board_init_r(). This
 *    environment has BSS (initialized to 0), initialized non-const
 *    data (initialized to their intended value), and stack in system
 *    RAM (for SPL moving the stack and GD into RAM is optional - see
 *    CONFIG_SPL_STACK_R). GD has retained values set by board_init_f().
 *
```

```
#endif
        bic     sp, x0, #0xf    /* 16-byte alignment for ABI compliance */
        mov     x0, sp
        bl      board_init_f_alloc_reserve
        mov     sp, x0
        /* set up gd here, outside any C code */
        mov     x18, x0
        bl      board_init_f_init_reserve

        mov     x0, #0
        bl      board_init_f
```

```
/*
 * Clear BSS section
 */
        ldr     x0, =__bss_start                /* this is auto-relocated! */
        ldr     x1, =__bss_end                  /* this is auto-relocated! */
clear_loop:
        str     xzr, [x0], #8
        cmp     x0, x1
        b.lo    clear_loop

        /* call board_init_r(gd_t *id, ulong dest_addr) */
        mov     x0, x18                         /* gd_t */
        ldr     x1, [x18, #GD_RELOCADDR]        /* dest_addr */
        b       board_init_r                    /* PC relative jump */
```

Nhiệm vụ của hàm _main là khởi tạo tiếp các tài nguyên cần thiết như memory rồi gọi hàm board_init_f. Sau đó nó sẽ gọi hàm board_init_r.

Hàm board_init_f của second stage có source code nằm ở file *arch/arm/mach-omap2/hwinit-common.c.* Source code của file start.S do các hãng chip viết, còn source code của board_init_f do các hãng làm SoC (system on chip) viết. Do cả 2 được build

cùng vào 1 chương trình là MLO nên chúng có thể gọi được nhau. Sau đây là source code của hàm board_init_f trong first stage:

```
237 #ifdef CONFIG_SPL_BUILD
238 void board_init_f(ulong dummy)
239 {
240         early_system_init();
241 #ifdef CONFIG_BOARD_EARLY_INIT_F
242         board_early_init_f();
243 #endif
244         /* For regular u-boot sdram_init() is called from dram_init() */
245         sdram_init();
246         gd->ram_size = omap_sdram_size();
247 }
248 #endif
```

Board_init_f sẽ khởi tạo sdram chính là bộ nhớ cached L2 của chip. Ngoài ra nó có thể khởi tạo 1 số tài nguyên quan trọng khác. Sau khi board_init_f thực hiện xong, _main sẽ gọi tiếp hàm board_init_r. Board_init_r của second stage có source code nằm ở file: *common/spl/spl.c*

```
635 #ifdef CONFIG_CPU_V7M
636         spl_image.entry_point |= 0x1;
637 #endif
638         switch (spl_image.os) {
639         case IH_OS_U_BOOT:
640                 debug("Jumping to U-Boot\n");
641                 break;
642 #if CONFIG_IS_ENABLED(ATF)
643         case IH_OS_ARM_TRUSTED_FIRMWARE:
644                 debug("Jumping to U-Boot via ARM Trusted Firmware\n");
645                 spl_invoke_atf(&spl_image);
646                 break;
647 #endif
648 #if CONFIG_IS_ENABLED(OPTEE)
649         case IH_OS_TEE:
650                 debug("Jumping to U-Boot via OP-TEE\n");
651                 spl_optee_entry(NULL, NULL, spl_image.fdt_addr,
652                                 (void *)spl_image.entry_point);
653                 break;
```

Board_init_r sẽ khởi tạo thêm 1 số thứ nữa, cuối cùng nó sẽ load image của chương trình third stage boot loader (chính là file u-boot.img) vào sdram. Tham số spl_image trong hình chính là struct chứa thông tin của file MLO như start_address, end_address trên sdcard. Như mình đã nói ở trên thì u-boot.img sẽ có địa chỉ trên sdcard ngay sau end_address của MLO.

1.2.3. Third stage boot loader

Sau khi u-boot.img được load vào dram, nó sẽ chiếm toàn quyền sử dụng hệ thống. Trong hệ thống không còn sự tồn tại của chương trình second state MLO nữa. So với second stage thì chương trình third stage sẽ làm được nhiều việc hơn do hệ thống đã khởi tạo được nhiều thứ hơn như đã có dram, có thể sử dụng hàm malloc,…

u-boot.img sẽ gọi hàm board_init_f và board_init_r trong source code của nó. Mỗi hàm này đều có 1 cặp, trong đó 1 cái sẽ nằm trong MLO và cái còn lại sẽ nằm trong u-boot.img. Hàm nằm trong u-boot.img sẽ làm được nhiều việc hơn so với MLO. Các hàm được build cùng với chương trình first stage sẽ có chỉ thị tiền xử lý CONFIG_SPL_BUILD ở đầu. Giống như hình bên dưới:

```
237 #ifdef CONFIG_SPL_BUILD
238 void board_init_f(ulong dummy)
239 {
240         early_system_init();
241 #ifdef CONFIG_BOARD_EARLY_INIT_F
242         board_early_init_f();
243 #endif
244         /* For regular u-boot sdram_init() is called from dram_init() */
245         sdram_init();
246         gd->ram_size = omap_sdram_size();
247 }
248 #endif
249
```

Hàm board_init_f của third stage có source code nằm ở file *common/board_f.c*.

```
986 void board_init_f(ulong boot_flags)
987 {
988         debug("PhuLA board_init_f\n");
989         gd->flags = boot_flags;
990         gd->have_console = 0;
991
992         if (initcall_run_list(init_sequence_f))
993                 hang();
994
995 #if !defined(CONFIG_ARM) && !defined(CONFIG_SANDBOX) && \
996                 !defined(CONFIG_EFI_APP) && !CONFIG_IS_ENABLED(X86_64) && \
997                 !defined(CONFIG_ARC)
998         /* NOTREACHED - jump_to_copy() does not return */
999         hang();
1000 #endif
```

```
837 static const init_fnc_t init_sequence_f[] = {
838         setup_mon_len,
839 #ifdef CONFIG_OF_CONTROL
840         fdtdec_setup,
841 #endif
842 #ifdef CONFIG_TRACE
843         trace_early_init,
844 #endif
845         initf_malloc,
846         log_init,
847         initf_bootstage,        /* uses its own ti
848 #ifdef CONFIG_BLOBLIST
849         bloblist_init,
850 #endif
851         setup_spl_handoff,
852         initf_console_record,
853 #if defined(CONFIG_HAVE_FSP)
854         arch_fsp_init,
855 #endif
856         arch_cpu_init,          /* basic arch cpu
857         mach_cpu_init,          /* SoC/machine dep
858         initf_dm,
859         arch_cpu_init_dm,
860 #if defined(CONFIG_BOARD_EARLY_INIT_F)
861         board_early_init_f,
862 #endif
```

Nhiệm vụ của board_init_f là khởi tạo các tài nguyên dựa trên tài nguyên bộ nhớ ít ỏi là sdram ram. Luồng gọi sẽ như sau: board_init_f -> init_sequence_f -> khởi tạo từng module. Thông thường các hàm khởi tạo hardware sẽ gọi xuống driver của từng hãng khác nhau. Ví dụ như đối với beagle bone: init_sequence_f -> serial_init -> probe (*drivers/serial/serial_omap.c*).

Sau khi board_init_f chạy xong, lúc này hệ thống đã khởi tạo được nhiều tài nguyên hơn. Ví dụ như đã sử dụng được DRAM, dùng được malloc, device tree…

Lúc này _main tiếp tục gọi board_init_r để khởi tạo nốt những module hardware còn lại. Tương tự như trên thì board_init_r cũng gọi xuống các file driver của board beagle bone theo luồng như sau board_init_r -> init_sequence_r -> khởi tạo từng module:

```
830 void board_init_r(gd_t *new_gd, ulong dest_addr)
831 {
832         /*
833          * Set up the new global data pointer. So far only x86 does this
834          * here.
835          * TODO(sjg@chromium.org): Consider doing this for all archs, or
836          * dropping the new_gd parameter.
837          */
838 #if CONFIG_IS_ENABLED(X86_64)
839         arch_setup_gd(new_gd);
840 #endif
841
842 #ifdef CONFIG_NEEDS_MANUAL_RELOC
843         int i;
844 #endif
845
846 #if !defined(CONFIG_X86) && !defined(CONFIG_ARM) && !defined(CONFIG_ARM64)
847         gd = new_gd;
848 #endif
849         gd->flags &= ~GD_FLG_LOG_READY;
850
851 #ifdef CONFIG_NEEDS_MANUAL_RELOC
852         for (i = 0; i < ARRAY_SIZE(init_sequence_r); i++)
853                 init_sequence_r[i] += gd->reloc_off;
854 #endif
855
856         if (initcall_run_list(init_sequence_r))
857                 hang();
858
859         /* NOTREACHED - run_main_loop() does not return */
860         hang();
861 }
```

```
641 static init_fnc_t init_sequence_r[] = {
642         initr_trace,
643         initr_reloc,
644         /* TODO: could x86/PPC have this also perhaps? */
645 #ifdef CONFIG_ARM
646         initr_caches,
647         /* Note: For Freescale LS2 SoCs, new MMU table is created in DDR.
648          *       A temporary mapping of IFC high region is since removed,
649          *       so environmental variables in NOR flash is not available
650          *       until board_init() is called below to remap IFC to high
651          *       region.
652          */
653 #endif
654         initr_reloc_global_data,
655 #if defined(CONFIG_SYS_INIT_RAM_LOCK) && defined(CONFIG_E500)
656         initr_unlock_ram_in_cache,
657 #endif
658         initr_barrier,
659         initr_malloc,
660         log_init,
661         initr_bootstage,        /* Needs malloc() but has its own timer */
662         initr_console_record,
663 #ifdef CONFIG_SYS_NONCACHED_MEMORY
664         initr_noncached,
665 #endif
666         bootstage_relocate,
667 #ifdef CONFIG_OF_LIVE
668         initr_of_live,
669 #endif
670 #ifdef CONFIG_DM
671         initr_dm,
```

Sau khi khởi tạo xong hết hardware, board_init_r sẽ gọi ra hàm run_main_loop. Hàm này sẽ khởi tạo hệ thống command line của u-boot. Vậy command line trên uboot dùng để làm gì? Chúng ta sẽ cùng tìm hiểu ở phần tiếp theo.

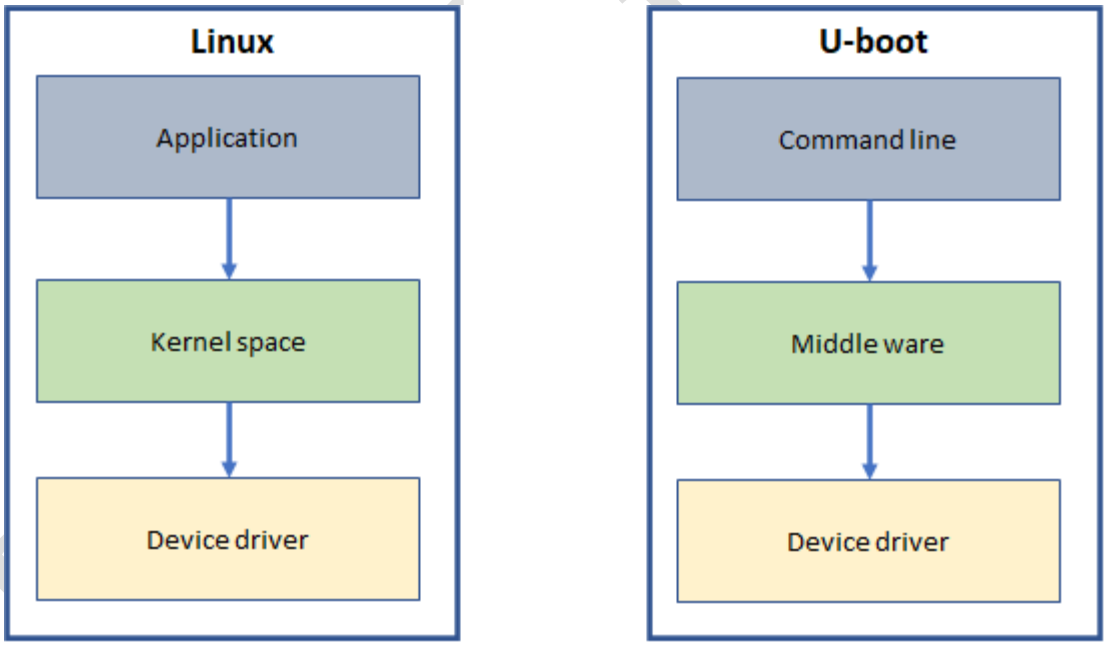
Lưu ý:

1. 1 số tài liệu trên mạng cho rằng hàm board_init_f trong file common/board_f.c và board_init_r trong file common/board_r.c thuộc về luồng second stage (MLO) tuy nhiên mình không nghĩ như vậy. Do source code của 2 hàm này khá dài không phù hợp với kích thước của MLO, ngoài ra trong hàm board_init_f có khởi tạo device tree mà device tree là data được đính kèm ở cuối file u-boot.img. Do vậy mình kết luận rằng chúng thuộc về third stage.

2. u-boot khởi tạo dram thông qua luồng init_sequence_f -> dram_init. Trong dram_init sẽ get ra size của bộ nhớ ram trong hệ thống rồi truyền giá trị đó cho Linux thông qua kernel parameter. Nếu sau khi các bạn tích hợp thêm ram vào board mà hệ thống nhận không đủ ram thì có thể kiểm tra trong hàm này.

1.2.4. Hệ thống command line của u-boot

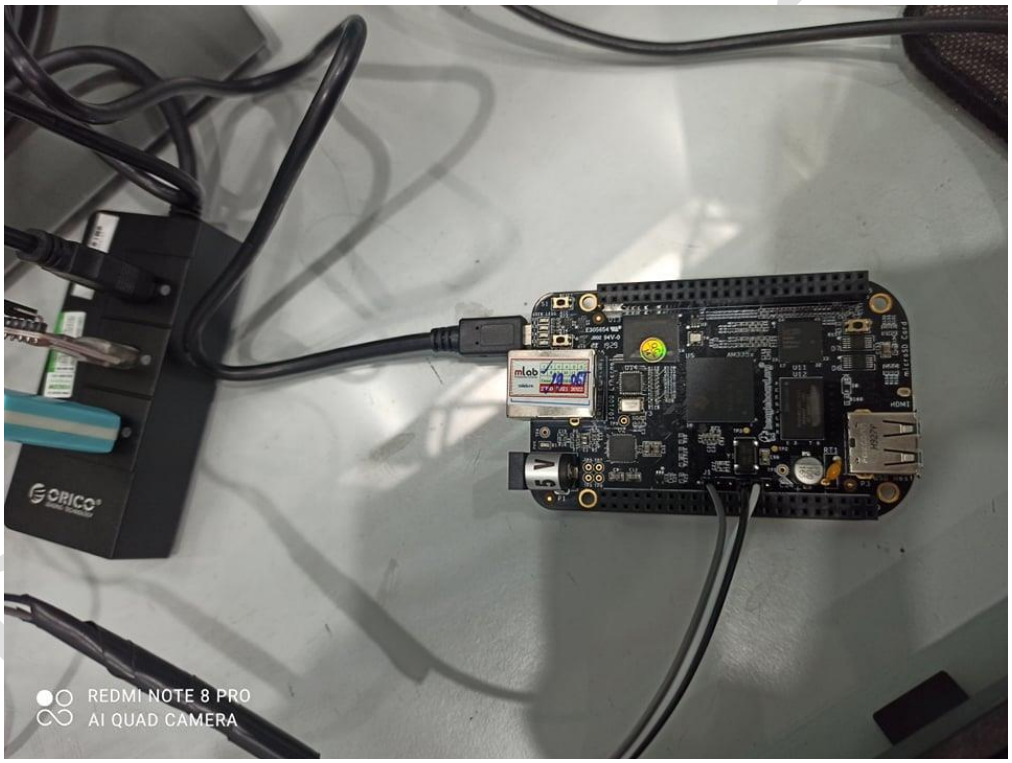
Nếu như Linux được phân chia làm 3 tầng bao gồm application, kernel, driver thì u-boot cũng tương tự như vậy.



Sau khi khởi tạo hardware và middle ware thông qua 2 hàm board_init_f và board_init_r thì u-uboot sẽ đi vào 1 vòng lặp. Lúc này người dùng có thể thao tác gõ command line để điều khiển hệ thống. Muốn vào được giao diện này thì chúng ta phải connect PC vào board qua cổng serial và ngắt quá trình auto boot của hệ thống bằng phím space. Giao diện thao tác command line sẽ giống như hình sau:

```
U-Boot 2019.04-00001-g9af4e99 (Feb 12 2021 - 00:37:27 -0800)

CPU  : AM335X-GP rev 2.1
I2C:   ready
DRAM:  512 MiB
No match for driver 'omap_hsmmc'
No match for driver 'omap_hsmmc'
Some drivers were not found
Reset Source: Global external warm reset has occurred.
Reset Source: Power-on reset has occurred.
RTC 32KCLK Source: External.
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
Loading Environment from EXT4... ** No partition table - mmc 0 **
Board: BeagleBone Black
<ethaddr> not set. Validating first E-fuse MAC
BeagleBone Black:
BeagleBone Cape EEPROM: no EEPROM at address: 0x54
BeagleBone Cape EEPROM: no EEPROM at address: 0x55
BeagleBone Cape EEPROM: no EEPROM at address: 0x56
BeagleBone Cape EEPROM: no EEPROM at address: 0x57
Net:   eth0: MII MODE
cpsw, usb_ether
Press SPACE to abort autoboot in 0 seconds
=>
```



Sau khi ghi u-boot mới build được vào thẻ nhớ, các bạn lưu ý ấn phím s2 trên board để chương trình boot-rom jump sang uboot ở sdcard. Nếu không board sẽ vẫn sử dụng u-boot cũ trên eMMC. Mỗi lần build thì thời gian của hệ thống build cũng được đính kèm vào u-boot.img nên các bạn có thể thấy trên hình ở dòng U-Boot 2019.04-00001-g9af4e99 (Feb 12 2021 - 00:37:27 -0800).

U-boot support hàng trăm command line khác nhau như hình dưới đây:

```
=> ?
?          - alias for 'help'
askenv     - get environment variables from stdin
base       - print or set address offset
bdinfo     - print Board Info structure
boot       - boot default, i.e., run 'bootcmd'
bootd      - boot default, i.e., run 'bootcmd'
bootefi    - Boots an EFI payload from memory
bootelf    - Boot from an ELF image in memory
bootm      - boot application image from memory
bootp      - boot image via network using BOOTP/TFTP protocol
bootvx     - Boot vxWorks from an ELF image
bootz      - boot Linux zImage image from memory
btrsubvol  - list subvolumes of a BTRFS filesystem
cmp        - memory compare
coninfo    - print console devices and information
cp         - memory copy
crc32      - checksum calculation
dfu        - Device Firmware Upgrade
dhcp       - boot image via network using DHCP/TFTP protocol
dm         - Driver model low level access
echo       - echo args to console
```

Ngoài ra u-boot cũng có hệ thống biến môi trường giống như Linux. Để show ra giá trị các biến môi trường, các bạn có thể dùng câu lệnh printenv giống Linux.

Trên Linux thì các command line bản chất là các file binary được đặt trong thư mục /bin hoặc /sbin của hệ thống. Tuy nhiên do u-boot là một chương trình vi điều khiển, do vậy các command line của nó là các function trong source code. Mỗi 1 function khi muốn đăng ký thành command line sẽ sử dụng macro U_BOOT_CMD giống như hình dưới:

```
1311 U_BOOT_CMD_COMPLETE(
1312         printenv, CONFIG_SYS_MAXARGS, 1,        do_env_print,
1313         "print environment variables",
1314         "[-a]\n    - print [all] values of all environment variables\n"
1315 #if defined(CONFIG_CMD_NVEDIT_EFI)
1316         "printenv -e [name ...]\n"
1317         "     - print UEFI variable 'name' or all the variables\n"
1318 #endif
1319         "printenv name ...\n"
1320         "     - print value of environment variable 'name'",
1321         var_complete
1322 );
```

Command line của u-boot cũng hỗ trợ help và truyền option khi sử dụng giống như Linux. Hàm do_main_loop sẽ liên tục get input của user thông qua serial, từ đó detect được command line mà user gõ. Ví dụ như user gõ printenv thì hàm do_env_print sẽ được gọi ra để xử lý. Tất cả các command line còn lại đều tương tự như vậy. Một số comman line như đọc data trên mmc, i2c thì sẽ gọi xuống middle ware rồi từ middle ware sẽ call driver của board để thực hiện.

Nếu như quá trình boot không bị interrupt bởi input từ user thì hàm do_main_loop sẽ thực thi 1 loạt command line để execute Linux kernel. Quá trình này giống như chạy shell script. Chúng ta sẽ cùng tìm hiểu kỹ hơn ở phần tiếp theo.

1.2.5. Quá trình load Linux kernel

Để load được image của Linux kernel lên ram thì cần thực thi 1 chuỗi các command line. Mỗi 1 board lại khác nhau đôi chút về các command line cần phải thực thi. Ví dụ như đối với beagle bone black thì cần load file uEnv.txt trên sdcard trước để lấy config

hệ thống. Do vậy mỗi 1 hãng làm board sẽ tạo 1 script riêng và u-boot sẽ thực thi script đấy để load kernel image mà không cần quan tâm bên trong là gì.

Script đấy được define bằng macro CONFIG_EXTRA_ENV_SETTINGS nằm trong folder /include/configs. Ví dụ như beagle bone sẽ có file include/configs/omap3_beagle.h của riêng nó. Giải thích một chút về script boot kernel image của beagle bone như sau:

```
#define CONFIG_EXTRA_ENV_SETTINGS \
        MEM_LAYOUT_ENV_SETTINGS \
        "fdtfile=" CONFIG_DEFAULT_FDT_FILE "\0" \
        "fdt_high=0xffffffff\0" \
        "console=ttyO2,115200n8\0" \
        "bootdir=/boot\0" \
        "bootenv=uEnv.txt\0" \
        "bootfile=zImage\0" \
        "bootpart=0:2\0" \
        "bootubivol=rootfs\0" \
        "bootubipart=rootfs\0" \
        "usbtty=cdc_acm\0" \
        "mpurate=auto\0" \
        "buddy=none\0" \
        "camera=none\0" \
        "vram=12M\0" \
        "dvimode=640x480MR-16@60\0" \
        "defaultdisplay=dvi\0" \
        "defaultargs=setenv defargs " \
                "mpurate=${mpurate} " \
                "buddy=${buddy} "\
                "camera=${camera} "\
                "vram=${vram} " \
                "omapfb.mode=dvi:${dvimode} " \
                "omapdss.def_disp=${defaultdisplay}\0" \
        "optargs=\0" \
        "findfdt=" \
                "if test $beaglerev = AxBx; then " \
                        "setenv fdtfile omap3-beagle.dtb; fi; " \
                "if test $beaglerev = Cx; then " \
```

```
        "loadbootenv=fatload mmc ${mmcdev} ${loadaddr} ${bootenv}\0" \
        "ext4bootenv=ext4load mmc ${bootpart} ${loadaddr} ${bootdir}/${bootenv}\0" \
        "importbootenv=echo Importing environment from mmc${mmcdev} ...; " \
                "env import -t ${loadaddr} ${filesize}\0" \
        "mmcbootenv=setenv bootpart ${mmcdev}:${mmcpart}; " \
                "mmc dev ${mmcdev}; " \
                "if mmc rescan; then " \
                        "if run userbutton; then " \
                                "setenv bootenv uEnv.txt;" \
                        "else " \
                                "setenv bootenv user.txt;" \
                        "fi;" \
                        "run loadbootenv && run importbootenv; " \
                        "run ext4bootenv && run importbootenv; " \
                        "if test -n $uenvcmd; then " \
                                "echo Running uenvcmd ...; " \
                                "run uenvcmd; " \
                        "fi; " \
                "fi\0" \
        "validatefdt=" \
                "if test $beaglerev = xMAB; then " \
                        "if test ! -e mmc ${bootpart} ${bootdir}/${fdtfile}; then " \
                                "setenv fdtfile omap3-beagle-xm.dtb; " \
                        "fi; " \
                "fi; \0" \
        "loadimage=ext4load mmc ${bootpart} ${loadaddr} ${bootdir}/${bootfile}\0" \
        "loaddtb=run validatefdt; ext4load mmc ${bootpart} ${fdtaddr} ${bootdir}/${fdtfile}\0" \
        "mmcboot=run mmcbootenv; " \
                "if run loadimage && run loaddtb; then " \
                        "echo Booting ${bootdir}/${bootfile} from mmc ${bootpart} ...; " \
                        "run mmcargs; " \
                        "if test ${bootfile} = uImage; then " \
                                "bootm ${loadaddr} - ${fdtaddr}; " \
                        "fi; " \
                        "if test ${bootfile} = zImage; then " \
                                "bootz ${loadaddr} - ${fdtaddr}; " \
                        "fi; " \
                "fi\0" \
```

Đầu tiên u-boot sẽ đọc file /boot/uEnv.txt để lấy config. Sau đó sẽ lựa chọn load kernel image và dtb file từ mmc hoặc sdcard. Sau khi đọc kernel image và dtb file từ sdcard lên ram uboot sẽ dựa vào định dạng của kernel image và uImage hoặc zImage để tiến hành giải nén. Cuối cùng uboot sẽ gọi command bootm và truyền vào đó địa chỉ của kernel image và dtb file trên ram. Bootm sẽ jump vào câu lệnh đầu tiên của kernel và chuyển toàn bộ quyền điều khiển hệ thống cho nó. Đến đây là u-boot đã hoàn thành nhiệm vụ của mình.

1.2.6. Tổ chức source code của u-boot.

Khi làm việc với u-boot, người lập trình viên phải biết tính năng A có source code nằm ở folder nào. Từ đó thu hẹp phạm vi tìm kiếm. Về cơ bản thì u-boot có tổ chức source code tương tự kernel. Bao gồm những folder chính sau:

1. cmd: Chứa source code của tất cả các command line. Thông thường mỗi command line sẽ là 1 file source C.

2. configs: Chứa file config để generate ra file .config khi build u-boot.

3. Documentation: Chứa hệ thống tài liệu của u-boot.

4. env: Chứa source code để xây dựng ra hệ thống biến môi trương của u-boot.

5. lib: Source code phần middler ware của u-boot.

6. net: Source code liên quan đến tính năng network. U-boot cũng có thể sử dụng được network như ping, sftp,…

7. arch: Source code specific cho từng platform. Device tree cũng được lưu trữ trong folder này.

8. common: Source code phần middler ware của u-boot.

9. drivers: Chứa source code driver của từng board.

10. spl: Chứa 1 phần code để build ra second stage boot loader.

11. tools: Chứa các loại tool dùng trong quá trình build u-boot.

12. fs: Chứa source code của các loại file system dùng trong u-boot. U-boot cũng có tính năng đọc file theo file name như Linux.