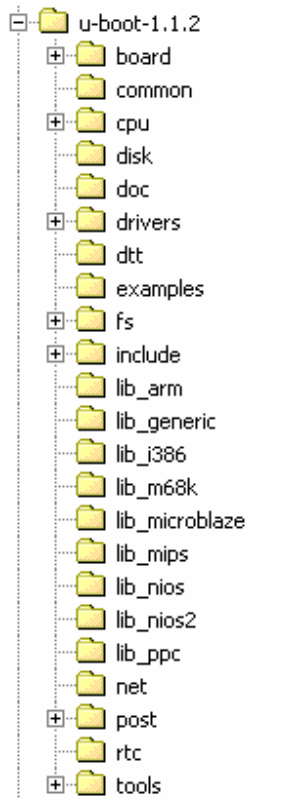
U-Boot 源代码分析

刘通平 Homepage: http://www.cs.umass.edu/~tonyliu/

为了更好的分析 U-Boot，下面列出了 U-Boot 根目录源码树。

4.2.1 源码树



从根目录树可以看出，U-Boot 源代码主要包含以下几部份：

- 与目标板相关的代码，对应于 board 目录。
- 公共代码，对应于 common 目录。
- 与 CPU 相关的代码，对应于 cpu 目录。
- 磁盘驱动和磁盘分区相关的处理代码，对应于 disk 目录。
- 说明文档，对应于 doc 目录。
- 关键的驱动程序，对应于 drivers 目录。
- 数字温度计和自动调温装置的驱动，对应于 dtt 目录。
- 简单应用的例程，对应于 example 目录。
- 头文件，对应于 include 目录。
- 体系结构相关的代码，对应于 lib_arm, lib_i386, lib_m68k, lib_microblaze, lib_mips, lib_nios, lib_nios2, lib_ppc 目录。
- 体系结构无关的公共代码，对应于 lib_generic 目录。

- 网络传输代码，对应于 net 目录。
- 上电自测代码，对应于 post 目录。
- 实时时钟驱动程序，对应于 rtc 目录。
- U-Boot 常用工具，对应于 tools 目录。

4.2.2 与目标板相关的代码

在 U-Boot 中，目标板相关的代码都位于/board 目录，在该目录下，列出了当前版本的 U-Boot 所能支持的所有目标板。目前 U-Boot 支持大多数比较常见的目标板，象 MOTOROLA 的 FADS 板，一般我们设计的主板和这些主板都是大同小异，不会有太大的差别。

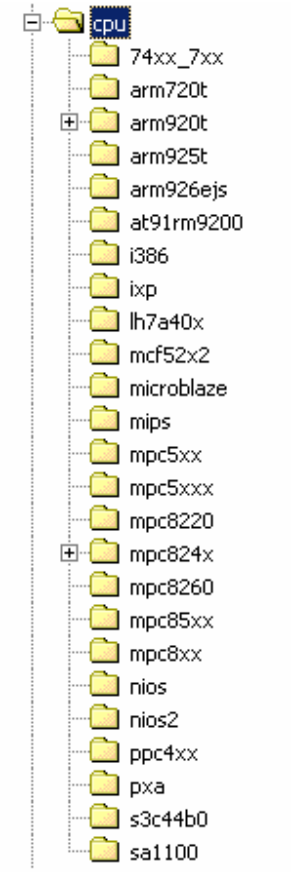
源代码树中/board 下的每个文件夹都对应一个目标板板，比如/board/fads 目录对应的就是 MOTOROLA 的 FADS 评估板。在 U-Boot-1.1.2 中，/board 目录下包含 162 个文件夹，这说明支持 162 种常用的目标板。

在具体目标板对应的目录中，一般至少都包含以下文件：

- config.mk，用于设置程序连接的起始地址等。
- u-boot.lds，链接脚本文件，指定了目标文件的链接规则。
- Makefile
- 目标板相关的初始化代码，包括 SDRAM 和 Flash 的初始化代码。

4.2.3 与 CPU 相关的代码

CPU 相关的代码位于/cpu 目录下，在该目录下，列出了当前版本的 U-Boot 所能支持的所有 CPU 类型。和目标板相关的代码类似，该目录的每个子目录都对应一种具体的 CPU 类型。该目录的目录树如下图所示：



由上图可以看出，U-Boot-1.1.2 已经支持 PowerPC, ARM, Freescale, Xscale，ColdFire，Altera，Xscale，StrongARM 等序列的 CPU，因此可以看出 U-Boot 是一种被广泛使用的 Bootloader。

在具体 CPU 对应的目录中，一般包含以下文件：

- config.mk，该文件主要包含一些编译选项，该文件将被相同目录下的 Makefile 所引用。
- Makefile
- start.S, 启动代码。整个 U-Boot 映像的入口点。
- 其它一些 CPU 初始化相关代码。

4.2.4 头文件

头文件位于源码树下的 include/下，其中各种主板的配置文件位于/include/configs/文件夹中，比如 FADS860T 主板的配置文件为/include/configs/FADS860T.h；

目录/include/asm-*** 是一些比较底层的头文件，编译时会根据不同的配置与/include/configs/asm 建立一个符号链接。

4.2.5 公共代码

除了和主板，CPU 特性相关的代码外，其他大部分都是公用的代码，位于/common 目录下面，比如 U-Boot 的命令解析代码 command.c，U-Boot 环境变量处理代码 environment.c 等都位

于该目录下。

该目录下的代码构成了 U-Boot 的基本框架，也是 U-Boot 的精髓所在。因为板级相关的代码和 CPU 相关的代码往往都是由 CPU 厂商按 U-Boot 的规则提供的，目的是推广目标板或芯片的，但这些公共的代码则是由 Denx 项目组成员开发和维护的。其目的提供一套不同的目标板的共性，比如在引导时与目标板的信息的交互，环境变量的设置等，这样就可以省去具体 BSP 包开发人员很多的工作量。而且 U-Boot 的这些公共文件做的非常完善，应该能够满足大部分嵌入式系统开发的要求，比如支持各式各样的命令，这些都是 U-Boot 的非常突出的优点。

利用 U-Boot 机制，不同 BSP 包的开发者只需关心和板级相关，CPU 相关的设置及其所用器件的驱动开发，然后通过某种方式把其添加到 U-Boot 中，其他工作都由 U-Boot 完成了。本书的 4.7 节将具体阐述如何把 BSP 包添加到 U-Boot 程序中。

该目录下主要包含以下几类文件：

- 命令解析文件
- 环境变量相关文件
- 驱动和调试文件
- C 函数入口文件 main.c

命令解析文件：命令解析文件包含 command.c 和所有以 cmd_打头的文件，其中 command.c 是所有以“cmd_”打头命令文件的入口文件。

环境变量文件：包括 environment.c 和所有以”env_”打头的文件，其中 environment.c 是环境变量相关的入口文件。

C 函数入口文件 main.c：当 CPU 和板级初始化后，将调用 main_loop()进入一个无限循环，等待界面发出命令或者直接执行预设的某些命令。Main_loop()函数如下所示：

```
void main_loop (void)
{
    static char lastcommand[CFG_CBSIZE] = { 0, };
    int len;
    int rc = 1;
    int flag;
    char *s;
    int bootdelay;

    s = getenv ("bootdelay");
    bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;

    debug ("### main_loop entered: bootdelay=%d\n\n", bootdelay);

    s = getenv ("bootcmd");

    debug ("### main_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");

    if (bootdelay >= 0 && s && !abortboot (bootdelay)) {

        run_command (s, 0);
    }

    /*
     * Main Loop for Monitor Command Processing
```

```
     */
    for (;;) {
        len = readline (CFG_PROMPT);

        flag = 0;  /* assume no special flags for now */
        if (len > 0)
            strcpy (lastcommand, console_buffer);
        else if (len == 0)
            flag |= CMD_FLAG_REPEAT;

        if (len == -1)
            puts ("<INTERRUPT>\n");
        else
            rc = run_command (lastcommand, flag);

        if (rc <= 0) {
            /* invalid command or not repeatable, forget it */
            lastcommand[0] = 0;
        }
    }
}
```

（1） 首先将获取环境变量中有关 bootdelay 的定义。此参数说明了启动时预留的人工干预的时间，若在指定的时间内无人干预，那么 U-Boot 将执行默认的启动命令序列，即执行环境变量中”bootcmd”中相关的设置。当然，若”bootcmd”没有定义时，系统将打印提示符界面，等待用户输入具体的指令去干预系统的行为。

（2） 通过调用 getenv("bootcmd")去获取环境变量中”bootcmd”相关的设置，然后调用 run_command()去执行相应的启动命令序列。run_command()的详细信息参见“命令”一节。

（3） 进入一个无限循环，在此循环中首先调用 readline(CFG_PROMPT)从终端获取相应的命令。

（4） 然后判断相应命令的长度，若中断输入的仅仅是一个回车的操作符时，U-Boot 将把其解释成“重复上次操作”的命令，因此置位相应的 flag 为“flag |= CMD_FLAG_REPEAT;”，但并未改变全局变量 lastcommand。但当命令的长度不为 0 时，意味着用户确实输入了相应的命令，因此将调用 strcpy(lastcommand, console_buffer)把全局变量置位为当前的命令，以此实现了命令的更替。

（5） 调用 run_command()去解释执行相应的命令。对于 U-Boot 而言总共有两类的命令，一类命令执行后能够返回的，对于此类命令，该处还将判断返回值，若返回值不对，证明这是一条无效的命令，把全局变量 lastcommand 置为空值，因此下次若想再以回车来执行无效命令时，就不会再去执行一遍无效命令。第二类命令是“有去无回”的命令，类似于”bootm 0x100000000”或者”go 0x10000000”，这类指令最终将去启动内核，因此系统将不再返回。

4.2.6 网络传输代码

网络传输代码位于目录中的/net/下面，象 arp.c ,bootp.c,eth.c,tftp.c 等都在这里，BOOTP，TFTP，以太网初始化等功能的实现也是由这些代码完成的。

4.2.7 Makefile 文件

Makefile 文件位于 U-Boot 的根目录下，是整个编译的控制主文件，后面我们会讲到如何修改该文件以添加我们自己主板的编译控制。

4.2.8 U-Boot 工具集

U-Boot 工具集提供了一些工具，比如 mkimage 等。对于嵌入式开发而言，mkimage 比较常用，而其它工具比较少用，此处着重分析一下 mkimage 及其基本用法。

Mkimage 用以制作 U-Boot 可辨识的映像，包括文件系统映像和内核映像等。mkimage 主要是在原映像的头部添加一个单元，以说明该文件的类型，目标板类型，文件大小，加载地址，名字等信息。

mkimage 的基本用法如下：

“-l” 指定打开文件的方式，当指定了”-l”时，将以只读方式打开指定的文件。

“-A” 设置目标体系结构，因为不同的目标平台 U-Boot 和内核的交互数据不一致，因此此处对目标平台加以指定，当前 U-Boot 支持的目标体系结构有"alpha", "arm", "x86", "ia64", "m68k", "microblaze", "mips", "mips64", "ppc", "s390", "sh", "sparc", "sparc64" 等。

” –C ”设置当前映像的压缩形式，U-Boot 支持：未压缩的映像”none”，bzip2 的压缩格式” bzip2” 和 gzip 的压缩格式”gzip”。

“-T”设置映像类型，当前 U-Boot 支持的映像有：

- "filesystem"－文件系统映像，可以利用 U-Boot 将文件系统放入某个位置。
- "firmware"－固件映像，固件映像通常被直接烧入到 Flash 中的二进制文件。
- "kernel"－内核映像，内核映像是嵌入式操作系统的映像，通常在加载操作系统映像后，控制将不再放回到 U-Boot 中。
- "multi"－多文件映像，多文件映像通常包含多个映像，比如同时包含操作系统映像和 ramdisk 映像。多文件映像将同时包含不同映像的信息。
- "ramdisk"－ramdisk 映像，ramdisk 映像有点像数据块，启动是一些启动参数将传递给内核。
- "script"－脚本文件，通常包含 U-Boot 支持的命令序列。当希望 U-Boot 作为一

个”Real Shell”时，把这些配置脚本都放入”script”映像中。

- "standalone"－单独的应用程序映像，standalone 可在 U-Boot 提供的环境下直接运行，程序运行完成后一般还将返回 U-Boot。

“-a”设置映像的加载地址，即希望 U-Boot 把该映像加载到什么位置。

“-d”设置输入映像文件，即希望对那个映像文件进行操作。

“-e”设置映像的入口点，这个选项并不是对所有映像都有效。通常对于内核映像而言，设置内核映像的入口点后，系统将把设置硬件寄存器中的 IP 寄存器（对 ARM 来说）为入口地址，然后跳转到该地址继续执行。但对不能直接运行的文件系统映像而言，设置了该地址并无实际意义。

“-n”设置映像的名字，设置了映像的名字是为了给用户一个清楚的描述。

其它的选项不太常用了。

当 uImage 映像制作成功后，可被直接烧入 Flash 中。当 U-Boot 成功运行后，可通过命令行参数让 U-Boot 启动某个内核。其中，最常用的命令行是 U-Boot 命令”bootm”，当 U-Boot 收到”bootm”命令时，将对指定地址的映像进行解析。这个解析过程是在 U-Boot 的 common/cmd_bootm.c 的 do_bootm 函数中定义的。整个解析的过程如下所示：

（1） 首先调用 memmove (&header, (char *)addr, sizeof(image_header_t))从给定地址搬移 sizeof(image_header_t)字节的数据到给定的 header 处。即实际上，mkimage 就是在原来映像的头部填充了 sizeof(image_header_t)字节的数据，即映像头。

（2） 判断头部的 magic 值，magic 值用以简单的判断整个映像的有效性。当 magic 值和 mkimage 默认填充的 magic 值不匹配时，认为整个映像是一个无效映像或者映像已经被破坏，因此程序不再往下执行，简单的输出信息"Bad Magic Number"。因此，若要想 U-Boot 能够辨识映像，必须用 mkimage 工具对现存的映像进行打包，内核编译时直接生成 zImage, vmlinux 等映像是没法直接被 U-Boot 辨识的。

（3） 对映像的头部进行检验，因为 mkimage 在生成映像时曾经生成了映像头的校验和，保存在 hdr->ih_hcrc 中，此处重新对映像头进行计算校验和，并与存储在映像头中的校验和进行比较，以此判断映像头是否已经被破坏。若校验失败，打印"Bad Header Checksum"而直接退出。

（4） 判断 CPU 类型是否有效。

（5） 判断映像的类型，根据不同的映像类型采取不同的动作，比如对于 standalone 的映像，需要把 bootm 的第三个参数作为映像的加载地址。

（6） 若映像为压缩映像，则调用 U-Boot 提供的 gunzip() 或者 BZ2_bzBuffToBuffDecompress 进行解压，解压的地址也就是 mkimage 中指定的加载地址。若映像为未压缩映像时，直接调用 memmove(to, from, tail)把实际的映像搬移到指定的加载地址处。

（7） 当搬移或者解压过程完成后，若相应的映像为内核映像的话将调用 do_bootm_linux() 进行加载内核。do_bootm_linux()程序对应不同的体系结构，有不同的实现方法。对于 ARM 体系结构而言，do_bootm_linux()程序在 lib_arm/armlinux.c 中实现，在一序

列的判断之后准备命令行参数等，当 Linux 启动时将利用这些命令行参数定义内核启动的行为。