u－boot 移植步骤详解

1 U-Boot 简介

U-Boot，全称 Universal Boot Loader，是遵循 GPL 条款的开放源码项目。从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来。其源码目录、编译形式与 Linux 内核很相似，事实上，不少 U-Boot 源码就是相应的 Linux 内核源程序的简化，尤其是一些设备的驱动程序，这从 U-Boot 源码的注释中能体现这一点。但是 U-Boot 不仅仅支持嵌入式 Linux 系统的引导，当前，它还支持 NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS 嵌入式操作系统。其目前要支持的目标操作系统是 OpenBSD, NetBSD, FreeBSD,4.4BSD, Linux, SVR4, Esix, Solaris, Irix, SCO, Dell, NCR, VxWorks, LynxOS, pSOS, QNX, RTEMS, ARTOS。这是 U-Boot 中 Universal 的一层含义，另外一层含义则是 U-Boot 除了支持 PowerPC 系列的处理器外，还能支持 MIPS、 x86、ARM、NIOS、XScale 等诸多常用系列的处理器。这两个特点正是 U-Boot 项目的开发目标，即支持尽可能多的嵌入式处理器和嵌入式操作系统。就目前来看，U-Boot 对 PowerPC 系列处理器支持最为丰富，对 Linux 的支持最完善。其它系列的处理器和操作系统基本是在 2002 年 11 月 PPCBOOT 改名为 U-Boot 后逐步扩充的。从 PPCBOOT 向 U-Boot 的顺利过渡，很大程度上归功于 U-Boot 的维护人德国 DENX 软件工程中心 Wolfgang Denk[以下简称 W.D]本人精湛专业水平和持着不懈的努力。当前，U-Boot 项目正在他的领军之下，众多有志于开放源码 BOOT LOADER 移植工作的嵌入式开发人员正如火如荼地将各个不同系列嵌入式处理器的移植工作不断展开和深入，以支持更多的嵌入式操作系统的装载与引导。

选择 U-Boot 的理由：

① 开放源码；

② 支持多种嵌入式操作系统内核，如 Linux、NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS；

③ 支持多个处理器系列，如 PowerPC、ARM、x86、MIPS、XScale；

④ 较高的可靠性和稳定性；

④ 较高的可靠性和稳定性；

⑤ 高度灵活的功能设置，适合 U-Boot 调试、操作系统不同引导要求、产品发布等；

⑥ 丰富的设备驱动源码，如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等；

⑦ 较为丰富的开发调试文档与强大的网络技术支持；

2 U-Boot 主要目录结构

- board 目标板相关文件，主要包含 SDRAM、FLASH 驱动；
- common 独立于处理器体系结构的通用代码，如内存大小探测与故障检测；

- cpu 与处理器相关的文件。如 mpc8xx 子目录下含串口、网口、LCD 驱动及中断初始化等文件；
- driver 通用设备驱动，如 CFI FLASH 驱动（目前对 INTEL FLASH 支持较好）
- doc U-Boot 的说明文档；
- examples 可在 U-Boot 下运行的示例程序；如 hello_world.c,timer.c；
- include U-Boot 头文件；尤其 configs 子目录下与目标板相关的配置头文件是移植过程中经常要修改的文件；
- lib_xxx 处理器体系相关的文件，如 lib_ppc, lib_arm 目录分别包含与 PowerPC、ARM 体系结构相关的文件；
- net 与网络功能相关的文件目录，如 bootp,nfs,tftp；
- post 上电自检文件目录。尚有待于进一步完善；
- rtc RTC 驱动程序；
- tools 用于创建 U-Boot S-RECORD 和 BIN 镜像文件的工具；

3 U-Boot 支持的主要功能

U-Boot 可支持的主要功能列表

系统引导 支持 NFS 挂载、RAMDISK（压缩或非压缩）形式的根文件系统

支持 NFS 挂载、从 FLASH 中引导压缩或非压缩系统内核；

基本辅助功能 强大的操作系统接口功能；可灵活设置、传递多个关键参数给操作系统，适合系统在不同开发阶段的调试要求与产品发布，尤对 Linux 支持最为强劲；

支持目标板环境参数多种存储方式，如 FLASH、NVRAM、EEPROM；

CRC32 校验，可校验 FLASH 中内核、RAMDISK 镜像文件是否完好；

设备驱动 串口、SDRAM、FLASH、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI、RTC 等驱动支持；

上电自检功能 SDRAM、FLASH 大小自动检测；SDRAM 故障检测；CPU 型号；

特殊功能 XIP 内核引导；

4 移植前的准备

（1）、首先读读 uboot 自带的 readme 文件，了解了一个大概。

（2）、看看 common.h，这个文件定义了一些基本的东西，并包含了一些必要的头文件。再看看 flash.h，这个文件里面定义了 flash_info_t 为一个 struct。包含了 flash 的一些属性定义。并且定义了所有的 flash 的属性，其中，AMD 的有：AMD_ID_LV320B，定义为"#define AMD_ID_LV320B 0x22F922F9"。

（3）、对于".borad/at91rm9200dk/flash.c"的修改，有以下的方面：

"void flash_identification(flash_info_t *info)"这个函数的目的是确认 flash 的型号。

注意的是，这个函数里面有一些宏定义，直接读写了 flash。并获得 ID 号。
（4）、修改：”./board/at91rm9200dk/config.mk”为
TEXT_BASE=0x21f80000 为 TEXT_BASE=0x21f00000 （当然，你应该根据自己的板子来修改，和一级 boot 的定义的一致即可）。
（5）、再修改”./include/configs/at91rm9200dk.h”为
修改 flash 和 SDRAM 的大小。
（6）、另外一个要修改的文件是：
./borad/at91rm9200dk/flash.c。这个文件修改的部分比较的多。
a. 首先是 OrgDef 的定义，加上目前的 flash。
b. 接下来，修改”#define FLASH_BANK_SIZE 0x200000”为自己 flash 的 容量
c. 在修改函数 flash_identification(flash_info_t * info)里面的打印信息，这部分将在 u-boot 启动的时候显示。
d. 然后修改函数 flash_init(void)里面对一些变量的赋值。
e. 最后修改的是函数 flash_print_info(flash_info_t * info)里面实际打印的函数信息。
f. 还有一个函数需要修改，就是：“flash_erase”，这个函数要检测先前知道的 flash 类型是否匹配，否则，直接就返回了。把这里给注释掉。

（7）、接下来看看 SDRAM 的修改。
这个里面对于“SIZE”的定义都是基于字节计算的。
只要修改”./include/configs/at91rm9200dk.h”里面的
“#define PHYS_SDRAM_SIZE 0X200000”就可以了。注意，SIZE 是以字节为单位的。
（8）、还有一个地方要注意
就是按照目前的设定，一级 boot 把 u_boot 加载到了 SDRAM 的空间为：21F00000 -> 21F16B10，这恰好是 SDRAM 的高端部分。另外，BSS 为 21F1AE34。

（9）、编译后，可以写入 flash 了。
a. 压缩这个 u-boot.bin
“gzip –c u-boot.bin > u-boot.gz”
压缩后的文件大小为：
43Kbytes
b. 接着把 boot.bin 和 u-boot.gz 烧到 flash 里面去。
Boot.bin 大约 11kBytes，在 flash 的 0x1000 0000 ~ 0x1000 3fff

5 U-Boot 移植过程
① 获得发布的最新版本 U-Boot 源码，与 Linux 内核源码类似，也是 bzip2 的压缩格式。可从 U-Boot 的官方网站 http://sourceforge.net/projects/U-Boot 上获得；
② 阅读相关文档，主要是 U-Boot 源码根目录下的 README 文档和 U-Boot 官方网站的

DULG（The DENX U-Boot and Linux Guide）文档
http://www.denx.de/twiki/bin/view/DULG/Manual。尤其是 DULG 文档，从如何安装建立交叉开发环境和解决 U-Boot 移植中常见问题都一一给出详尽的说明；
③ 订阅 U-Boot 用户邮件列表
http://lists.sourceforge.net/lists/listinfo/u-boot-users。在移植 U-Boot 过程中遇有问题，在参考相关文档和搜索 U-Boot-User 邮件档案库
http://sourceforge.net/mailarchive/forum.php?forum_id=12898 仍不能解决的情况下，第一时间提交所遇到的这些问题，众多热心的 U-Boot 开发人员会乐于迅速排查问题，而且很有可能，W.D 本人会直接参与指导；
④ 在建立的开发环境下进行移植工作。绝大多数的开发环境是交叉开发环境。在这方面，DENX 和 MontaVista 均提供了完整的开发工具集；
⑤ 在目标板与开发主机间接入硬件调试器。这是进行 U-Boot 移植应当具备且非常关键的调试工具。因为在整个 U-Boot 的移植工作中,尤其是初始阶段，硬件调试器是我们了解目标板真实运行状态的唯一途径。在这方面，W.D 本人和众多嵌入式开发人员倾向于使用 BDI2000。一方面，其价格不如 ICE 调试器昂贵，同时其可靠性高，功能强大，完全能胜任移植和调试 U-Boot。另外，网上也有不少关于 BDI2000 调试方面的参考文档。
⑥ 如果在参考开发板上移植 U-Boot，可能需要移除目标板上已有的 BOOT LOADER。可以根据板上 BOOT LOADER 的说明文档，先着手解决在移除当前 BOOT LOADER 的情况下，如何进行恢复。以便今后在需要场合能重新装入原先的 BOOT LOADER。

6. U-Boot 移植方法
当前，对于 U-Boot 的移植方法，大致分为两种。一种是先用 BDI2000 创建目标板初始运行环境，将 U- Boot 镜像文件 u-boot.bin 下载到目标板 RAM 中的指定位置，然后，用 BDI2000 进行跟踪调试。其好处是不用将 U-Boot 镜像文件烧写到 FLASH 中去。但弊端在于对移植开发人员的移植调试技能要求较高，BDI2000 的配置文件较为复杂。另外一种方法是用 BDI2000 先将 U-Boot 镜像文件烧写到 FLASH 中去，然后利用 GDB 和 BDI2000 进行调试。这种方法所用 BDI2000 的配置文件较为简单，调试过程与 U-Boot 移植后运行过程相吻合，即 U-Boot 先从 FLASH 中运行，再重载至 RAM 中相应位置，并从那里正式投入运行。唯一感到有些麻烦的就是需要不断烧写 FLASH。但考虑到 FLASH 常规擦写次数基本为 10 万次左右，作为移植 U-Boot，不会占用太多的次数，应该不会为 FLASH 烧写有什么担忧。同时，W. D 本人也极力推荐使用后一种方法。笔者建议，除非 U-Boot 移植资深人士或有强有力的技术支持，建议采用第二种移植方法。

7. U-Boot 移植主要修改的文件
从移植 U-Boot 最小要求－U-Boot 能正常启动的角度出发，主要考虑修改如下文件：
① <目标板>.h 头文件，如 include/configs/RPXlite.h。可以是 U-Boot 源码中已有的

目标板头文件，也可以是新命名的配置头文件；大多数的寄存器参数都是在这一文件中设置完成的；

② <目标板>.c 文件，如 board/RPXlite/RPXlite.c。它是 SDRAM 的驱动程序，主要完成 SDRAM 的 UPM 表设置，上电初始化。

③ FLASH 的驱动程序，如 board/RPXlite/flash.c，或 common/cfi_flash.c。可在参考已有 FLASH 驱动的基础上，结合目标板 FLASH 数据手册，进行适当修改；

④ 串口驱动，如修改 cpu/mpc8xx/serial.c 串口收发器芯片使能部分。

8. U-Boot 移植要点

① BDI2000 的配置文件。如果采用第二种移植方法，即先烧入 FLASH 的方法，配置项只需很少几个，就可以进行 U-Boot 的烧写与调试了。对 PPC 8xx 系列的主板，可参考 DULG 文档中 TQM8xx 的配置文件进行相应的修改。下面，笔者以美国 Embedded Planet 公司的 RPXlite DW 板为例，给出在嵌入式 Linux 交叉开发环境下的 BDI2000 参考配置文件以作参考：

```
; bdiGDB configuration file for RPXlite DW or LITE_DW
; ----------------------------------------
[INIT]
; init core register
WSPR 149 0x2002000F ;DER : set debug enable register
; WSPR 149 0x2006000F ;DER : enable SYSIE for BDI flash program
WSPR 638 0xFA200000 ;IMMR : internal memory at 0xFA200000
WM32 0xFA200004 0xFFFFFF89 ;SYPCR
[TARGET]
CPUCLOCK 40000000 ;the CPU clock rate after processing the init list
BDIMODE AGENT ;the BDI working mode (LOADONLY | AGENT)
BREAKMODE HARD ;SOFT or HARD, HARD uses PPC hardware breakpoints
[HOST]
IP 173.60.120.5
FILE uImage.litedw
FORMAT BIN
LOAD MANUAL ;load code MANUAL or AUTO after reset
DEBUGPORT 2001
START 0x0100
[FLASH]
CHIPTYPE AM29BX8 ;;Flash type (AM29F | AM29BX8 | AM29BX16 | I28BX8 |
I28BX16)
```

CHIPSIZE 0x400000 ;;The size of one flash chip in bytes
BUSWIDTH 32 ;The width of the flash memory bus in bits (8 | 16 | 32)
WORKSPACE 0xFA202000 ; RAM buffer for fast flash programming
FILE u-boot.bin ;The file to program
FORMAT BIN 0x00000000
ERASE 0x00000000 BLOCK
ERASE 0x00008000 BLOCK
ERASE 0x00010000 BLOCK
ERASE 0x00018000 BLOCK
[REGS]
DMM1 0xFA200000
FILE reg823.def

② U-Boot 移植参考板。这是进行 U-Boot 移植首先要明确的。可以根据目标板上 CPU、FLASH、SDRAM 的情况，以尽可能相一致为原则，先找出一个与所移植目标板为同一个或同一系列处理器的 U-Boot 支持板为移植参考板。如 RPXlite DW 板可选择 U-Boot 源码中 RPXlite 板作为 U-Boot 移植参考板。对 U-Boot 移植新手，建议依照循序渐进的原则，目标板文件名暂时先用移植参考板的名称，在逐步熟悉 U-Boot 移植基础上，再考虑给目标板重新命名。在实际移植过程中，可用 Linux 命令查找移植参考板的特定代码，如 grep –r RPXlite ./ 可确定出在 U-Boot 中与 RPXlite 板有关的代码，依此对照目标板实际进行屏蔽或修改。同时应不局限于移植参考板中的代码，要广泛借鉴 U-Boot 中已有的代码更好地实现一些具体的功能。

③ U-Boot 烧写地址。不同目标板，对 U-Boot 在 FLASH 中存放地址要求不尽相同。事实上，这是由处理器中断复位向量来决定的，与主板硬件相关，对 MPC8xx 主板来讲，就是由硬件配置字(HRCW)决定的。也就是说，U-Boot 烧写具体位置是由硬件决定的，而不是程序设计来选择的。程序中相应 U-Boot 起始地址必须与硬件所确定的硬件复位向量相吻合；如 RPXlite DW 板的中断复位向量设置为 0x00000100。因此， U-Boot 的 BIN 镜像文件必须烧写到 FLASH 的起始位置。事实上，大多数的 PPC 系列的处理器中断复位向量是 0x00000100 和 0xfff00100。这也是一般所说的高位启动和低位启动的 BOOT LOADER 所在位置。可通过修改 U-Boot 源码<目标板>.h 头文件中 CFG_MONITOR_BASE 和 board/<目标板>/config.mk 中的 TEXT_BASE 的设置来与硬件配置相对应。

④ CPU 寄存器参数设置。根据处理器系列、类型不同，寄存器名称与作用有一定差别。必须根据目标板的实际，进行合理配置。一个较为可行和有效的方法，就是借鉴参考移植板的配置，再根据目标板实际，进行合理修改。这是一个较费功夫和考验耐力的过程，需要仔细对照处理器各寄存器定义、参考设置、目标板实际作出选择并不断测试。MPC8xx 处理器较为关键的寄存器设置为 SIUMCR、PLPRCR、SCCR、BRx、ORx。

⑤ 串口调试。能从串口输出信息，即使是乱码，也可以说 U-Boot 移植取得了实质性突破。依据笔者调试经历，串口是否有输出，除了与串口驱动相关外，还与 FLASH 相关的寄存器设置有关。因为 U-Boot 是从 FLASH 中被引导启动的，如果 FLASH 设置不正确，U-Boot 代码读取和执行就会出现一些问题。因此，还需要就 FLASH 的相关寄存器设置进行一些参数调试。同时，要注意串口收发芯片相关引脚工作波形。依据笔者调试情况，如果串口无输出或出现乱码，一种可能就是该芯片损坏或工作不正常。

⑥ 与启动 FLASH 相关的寄存器 BR0、OR0 的参数设置。应根据目标板 FLASH 的数据手册与 BR0 和 OR0 的相关位含义进行合理设置。这不仅关系到 FLASH 能否正常工作，而且与串口调试有直接的关联。

⑦ 关于 CPLD 电路。目标板上是否有 CPLD 电路丝毫不会影响 U-Boot 的移植与嵌入式操作系统的正常运行。事实上，CPLD 电路是一个集中将板上电路的一些逻辑关系可编程设置的一种实现方法。其本身所起的作用就是实现一些目标板所需的脉冲信号和电路逻辑，其功能完全可以用一些逻辑电路与 CPU 口线来实现。

⑧ SDRAM 的驱动。串口能输出以后，U-Boot 移植是否顺利基本取决于 SDRAM 的驱动是否正确。与串口调试相比，这部分工作更为核心，难度更大。 MPC8xx 目标板 SDRAM 驱动涉及三部分。一是相关寄存器的设置；二是 UPM 表；三是 SDRAM 上电初始化过程。任何一部分有问题，都会影响 U- Boot、嵌入式操作系统甚至应用程序的稳定、可靠运行。所以说，SDRAM 的驱动不仅关系到 U-Boot 本身能否正常运行，而且还与后续部分相关，是相当关键的部分。

⑨ 补充功能的添加。在获得一个能工作的 U-Boot 后，就可以根据目标板和实际开发需要，添加一些其它功能支持。如以太网、LCD、NVRAM 等。与串口和 SDRAM 调试相比，在已有基础之上，这些功能添加还是较为容易的。大多只是在参考现有源码的基础上，进行一些修改和配置。

另外，如果在自主设计的主板上移植 U-Boot，那么除了考虑上述软件因素以外，还需要排查目标板硬件可能存在的问题。如原理设计、PCB 布线、元件好坏。在移植过程中，敏锐判断出故障态是硬件还是软件问题，往往是关系到项目进度甚至移植成败的关键，相应难度会增加许多。

下面以移植 u-boot 到 44B0 开发板的步骤为例，移植中上仅需要修改和硬件相关的部分。在代码结构上：

1) 在 board 目录下创建 ev44b0ii 目录，创建 ev44b0ii.c 以及 flash.c,memsetup.S,u-boot.lds 等。不需要从零开始，可选择一个相似的目录，直接复制过来，修改文件名以及内容。我在移植 u-boot 过程中，选择的是 ep7312 目录。由于 u-boot 已经包含基于 s3c24b0 的开发板目录，作为参考，也可以复制相应的目录。

2) 在 cpu 目录下创建 arm7tdmi 目录,主要包含 start.S，interrupts.c 以及

cpu.c,serial.c 几个文件。同样不需要从零开始建立文件，直接从 arm720t 复制，然后修改相应内容。

3) 在 include/configs 目录下添加 ev44b0ii.h，在这里放上全局的宏定义等。

4) 找到 u-boot 根目录下 Makefile 修改加入

ev44b0ii_config：unconfig

@./mkconfig $(@:_config=) arm arm7tdmi ev44b0ii

5) 运行 make ev44bii_config,如果没有错误就可以开始硬件相关代码移植的工作

3. u-boot 的体系结构

1) 总体结构

u-boot 是一个层次式结构。从上图也可以看出，做移植工作的软件人员应当提供串口驱动（UART Driver）,以太网驱动(Ethernet Driver),Flash 驱动（Flash 驱动）,USB 驱动（USB Driver）。目前，通过 USB 口下载程序显得不是十分必要，所以暂时没有移植 USB 驱动。驱动层之上是 u-boot 的应用，command 通过串口提供人机界面。我们可以使用一些命令做一些常用的工作，比如内存查看命令 md。

Kermit 应用主要用来支持使用串口通过超级终端下载应用程序。TFTP 则是通过网络方式来下载应用程序，例如 uclinux 操作系统。

2) 内存分布

在 flash rom 中内存分布图 ev44b0ii 的 flash 大小 2M(8bits),现在将 0-40000 共 256k 作为 u-boot 的存储空间。由于 u-boot 中有一些环境变量，例如 ip 地址，引导文件名等，可在命令行通过 setenv 配置好,通过 saveenv 保存在 40000-50000（共 64k）这段空间里。如果存在保存好的环境变量，u-boot 引导将直接使用这些环境变量。正如从代码分析中可以看到，我们会把 flash 引导代码搬移到 DRAM 中运行。下图给出 u-boot 的代码在 DRAM 中的位置。引导代码 u-boot 将从 0x0000 0000 处搬移到 0x0C700000 处。特别注意的由于 ev44b0ii uclinux 中断向量程序地址在 0x0c00 0000 处，所以不能将程序下载到 0x0c00 0000 出，通常下载到 0x0c08 0000 处。

4. start.S 代码结构

1) 定义入口

一个可执行的 Image 必须有一个入口点并且只能有一个唯一的全局入口，通常这个入口放在 Rom(flash)的 0x0 地址。例如 start.S 中的

.globl _start

_start:

值得注意的是你必须告诉编译器知道这个入口，这个工作主要是修改连接器脚本文件（lds）。

2) 设置异常向量(Exception Vector)

异常向量表，也可称为中断向量表，必须是从 0 地址开始，连续的存放。如下面的就包括了复位(reset),未定义处理（undef）,软件中断(SWI),预去指令错误(Pabort),数据错误(Dabort),保留，以及 IRQ,FIQ 等。注意这里的值必须与 uclinux 的 vector_base 一致。这就是说如果 uclinux 中 vector_base(include/armnommu/proc-armv/system.h)定义为 0x0c00 0000,则 HandleUndef 应该在
0x0c00 0004。

```
b reset //for debug
ldr pc,=HandleUndef
ldr pc,=HandleSWI
ldr pc,=HandlePabort
ldr pc,=HandleDabort
b .
ldr pc,=HandleIRQ
ldr pc,=HandleFIQ
ldr pc,=HandleEINT0 /*mGA H/W interrupt vector table*/
ldr pc,=HandleEINT1
ldr pc,=HandleEINT2
ldr pc,=HandleEINT3
ldr pc,=HandleEINT4567
ldr pc,=HandleTICK /*mGA*/
b .
b .
ldr pc,=HandleZDMA0 /*mGB*/
ldr pc,=HandleZDMA1
ldr pc,=HandleBDMA0
ldr pc,=HandleBDMA1
ldr pc,=HandleWDT
ldr pc,=HandleUERR01 /*mGB*/
b .
b .
ldr pc,=HandleTIMER0 /*mGC*/
ldr pc,=HandleTIMER1
ldr pc,=HandleTIMER2
ldr pc,=HandleTIMER3
ldr pc,=HandleTIMER4
ldr pc,=HandleTIMER5 /*mGC*/
b .
```

```
b .
ldr pc,=HandleURXD0 /*mGD*/
ldr pc,=HandleURXD1
ldr pc,=HandleIIC
ldr pc,=HandleSIO
ldr pc,=HandleUTXD0
ldr pc,=HandleUTXD1 /*mGD*/
b .
b .
ldr pc,=HandleRTC /*mGKA*/
b .
b .
b .
b .
b . /*mGKA*/
b .
b .
ldr pc,=HandleADC /*mGKB*/
b .
b .
b .
b .
b . /*mGKB*/
b .
b .
ldr pc,=EnterPWDN
```

作为对照：请看以上标记的值：

```
.equ HandleReset, 0xc000000
.equ HandleUndef,0xc000004
.equ HandleSWI, 0xc000008
.equ HandlePabort, 0xc00000c
.equ HandleDabort, 0xc000010
.equ HandleReserved, 0xc000014
.equ HandleIRQ, 0xc000018
.equ HandleFIQ, 0xc00001c
/*the value is different with an address you think it may be.
*IntVectorTable */
```

```
.equ HandleADC, 0xc000020
.equ HandleRTC, 0xc000024
.equ HandleUTXD1, 0xc000028
.equ HandleUTXD0, 0xc00002c
.equ HandleSIO, 0xc000030
.equ HandleIIC, 0xc000034
.equ HandleURXD1, 0xc000038
.equ HandleURXD0, 0xc00003c
.equ HandleTIMER5, 0xc000040
.equ HandleTIMER4, 0xc000044
.equ HandleTIMER3, 0xc000048
.equ HandleTIMER2, 0xc00004c
.equ HandleTIMER1, 0xc000050
.equ HandleTIMER0, 0xc000054
.equ HandleUERR01, 0xc000058
.equ HandleWDT, 0xc00005c
.equ HandleBDMA1, 0xc000060
.equ HandleBDMA0, 0xc000064
.equ HandleZDMA1, 0xc000068
.equ HandleZDMA0, 0xc00006c
.equ HandleTICK, 0xc000070
.equ HandleEINT4567, 0xc000074
.equ HandleEINT3, 0xc000078
.equ HandleEINT2, 0xc00007c
.equ HandleEINT1, 0xc000080
.equ HandleEINT0, 0xc000084
```

3) 初始化 CPU 相关的 pll,clock,中断控制寄存器

依次为关闭 watch dog timer,关闭中断，设置 LockTime，PLL(phase lock loop),以及时钟。

这些值（除了 LOCKTIME）都可从 Samsung 44b0 的手册中查到。

```
ldr r0,WTCON //watch dog disable
ldr r1,=0x0
str r1,[r0]
ldr r0,INTMSK
ldr r1,MASKALL //all interrupt disable
str r1,[r0]
/*****************************************************
```

* Set clock control registers *
**********/
ldr r0,LOCKTIME
ldr r1,=800 // count = t_lock * Fin (t_lock=200us, Fin=4MHz) = 800
str r1,[r0]
ldr r0,PLLCON /*temporary setting of PLL*/
ldr r1,PLLCON_DAT /*Fin=10MHz,Fout=40MHz or 60MHz*/
str r1,[r0]
ldr r0,CLKCON
ldr r1,=0x7ff8 //All unit block CLK enable
str r1,[r0]

4) 初始化内存控制器

内存控制器，主要通过设置 13 个从 1c80000 开始的寄存器来设置，包括总线宽度，8 个内存 bank，bank 大小，sclk,以及两个 bank mode。

/**********
* Set memory control registers *
**********/
memsetup:
adr r0,SMRDATA
ldmia r0,{r1-r13}
ldr r0,=0x01c80000 //BWSCON Address
stmia r0,{r1-r13}

5) 将 rom 中的程序复制到 RAM 中

首先利用 PC 取得 bootloader 在 flash 的起始地址，再通过标号之差计算出这个程序代码的大小。这些标号，编译器会在连接（link）的时候生成正确的分布的值。取得正确信息后，通过寄存器(r3 到 r10)做为复制的中间媒介，将代码复制到 RAM 中。

relocate:
/*
* relocate armboot to RAM
*/
adr r0, _start /* r0 <- current position of code */
ldr r2, _armboot_start
ldr r3, _armboot_end
sub r2, r3, r2 /* r2 <- size of armboot */
ldr r1, _TEXT_BASE /* r1 <- destination address */
add r2, r0, r2 /* r2 <- source end address */

```
/*
* r0 = source address
* r1 = target address
* r2 = source end address
*/
copy_loop:
ldmia r0!, {r3-r10}
stmia r1!, {r3-r10}
cmp r0, r2
ble copy_loop
```

6) 初始化堆栈

进入各种模式设置相应模式的堆栈。

```
InitStacks:
/*Don't use DRAM,such as stmfd,ldmfd......
SVCstack is initialized before*/
mrs r0,cpsr
bic r0,r0,#0X1F
orr r1,r0,#0xDB /*UNDEFMODE|NOINT*/
msr cpsr,r1 /*UndefMode*/
ldr sp,UndefStack
orr r1,r0,#0XD7 /*ABORTMODE|NOINT*/
msr cpsr,r1 /*AbortMode*/
ldr sp,AbortStack
orr r1,r0,#0XD2 /*IRQMODE|NOINT*/
msr cpsr,r1 /*IRQMode*/
ldr sp,IRQStack
orr r1,r0,#0XD1 /*FIQMODE|NOINT*/
msr cpsr,r1 /*FIQMode*/
ldr sp,FIQStack
bic r0,r0,#0XDF /*MODEMASK|NOINT*/
orr r1,r0,#0X13
msr cpsr,r1 /*SVCMode*/
ldr sp,SVCStack
```

7) 转到 RAM 中执行

使用指令 ldr,pc,RAM 中 C 函数地址就可以转到 RAM 中去执行。

5. 系统初始化部分

1. 串口部分

串口的设置主要包括初始化串口部分，值得注意的串口的 Baudrate 与时钟 MCLK 有很大关系，是通过：rUBRDIV0=((int)(MCLK/16./(gd ->baudrate) + 0.5) -1)计算得出。这可以在手册中查到。其他的函数包括发送，接收。这个时候没有中断，是通过循环等待来判断是否动作完成。

例如，接收函数：

```
while(!(rUTRSTAT0 & 0x1)); //Receive data read
return RdURXH0();
```

2. 时钟部分

实现了延时函数 udelay。

这里的 get_timer 由于没有使用中断，是使用全局变量来累加的。

3. flash 部分

flash 作为内存的一部分，读肯定没有问题，关键是 flash 的写部分。

Flash 的写必须先擦除，然后再写。

```
unsigned long flash_init (void)
{
int i;
u16 manId,devId;
//first we init it as unknown,even if you forget assign it below,it's not a problem
for (i=0; i < CFG_MAX_FLASH_BANKS; ++i){
flash_info[i].flash_id = FLASH_UNKNOWN;
flash_info[i].sector_count=CFG_MAX_FLASH_SECT;
}
/*check manId,devId*/
_RESET();
_WR(0x555,0xaa);
_WR(0x2aa,0x55);
_WR(0x555,0x90);
manId=_RD(0x0);
_WR(0x555,0xaa);
_WR(0x2aa,0x55);
_WR(0x555,0x90);
devId=_RD(0x1);
_RESET();
printf("flashn");
printf("Manufacture ID=%4x(0x0004), Device
ID(0x22c4)=%4xn",manId,devId);
if(manId!=0x0004 && devId!=0x22c4){
```

```
printf("flash check falilure");
return 0;
}else{
for (i=0; i < CFG_MAX_FLASH_BANKS; ++i){
flash_info[i].flash_id=FLASH_AM160T;/*In fact it is fujitu,I only don't want to
modify common files*/
}
}
/* Setup offsets */
flash_get_offsets (CFG_FLASH_BASE, &flash_info[0]);
/* zhangyy comment
#if CFG_MONITOR_BASE >= CFG_FLASH_BASE
//onitor protection ON by default
flash_protect(FLAG_PROTECT_SET,
CFG_MONITOR_BASE,
CFG_MONITOR_BASE+monitor_flash_len-1,
&flash_info[0]);
#endif
*/
flash_info[0].size =PHYS_FLASH_SIZE;
return (PHYS_FLASH_SIZE);
}
```

flash_init 完成初始化部分，这里的主要目的是检验 flash 的型号是否正确。

```
int flash_erase (flash_info_t *info, int s_first, int s_last)
{
volatile unsigned char *addr = (volatile unsigned char *)(info->start[0]);
int flag, prot, sect, l_sect;
//ulong start, now, last;
u32 targetAddr;
u32 targetSize;
/*zyy note:It is required and can't be omitted*/
rNCACHBE0=( (0x2000000>>12)<<16 )|(0>>12); //flash area(Bank0) must
be non-cachable
area.
rSYSCFG=rSYSCFG & (~0x8); //write buffer has to be off for proper timing.
if ((s_first < 0) || (s_first > s_last)) {
if (info->flash_id == FLASH_UNKNOWN) {
```

```
printf ("- missingn");
} else {
printf ("- no sectors to erasen");
}
return 1;
}
if ((info->flash_id == FLASH_UNKNOWN) ||
(info->flash_id > FLASH_AMD_COMP)) {
printf ("Can't erase unknown flash type - abortedn");
return 1;
}
prot = 0;
for (sect=s_first; sect<=s_last; ++sect) {
if (info->protect[sect]) {
prot++;
}
}
if (prot) {
printf ("- Warning: %d protected sectors will not be erased!n",
prot);
} else {
printf ("n");
}
l_sect = -1;
/* Disable interrupts which might cause a timeout here */
flag = disable_interrupts();
/* Start erase on unprotected sectors */
for (sect = s_first; sect<=s_last; sect++) {
if (info->protect[sect] == 0) {/* not protected */
targetAddr=0x10000*sect;
if(targetAddr<0x1F0000)
targetSize=0x10000;
else if(targetAddr<0x1F8000)
targetSize=0x8000;
else if(targetAddr<0x1FC000)
targetSize=0x2000;
else
```

```
targetSize=0x4000;
F29LV160_EraseSector(targetAddr);
l_sect = sect;
if(!BlankCheck(targetAddr, targetSize))
printf("BlankCheck Errorn");
}
}
/* re-enable interrupts if necessary */
if (flag)
enable_interrupts();
/* wait at least 80us - let's wait 1 ms */
udelay (1000);
/*
*We wait for the last triggered sector
*/
if (l_sect < 0)
goto DONE;
DONE:
printf (" donen");
return 0;
}
int BlankCheck(int targetAddr,int targetSize)
{
int i,j;
for(i=0;i{
j=*((u16 *)(i+targetAddr));
if( j!=0xffff)
{
printf("E:%x=%xn",(i+targetAddr),j);
return 0;
}
}
return 1;
}
```

flash_erase 擦除 flash,BlankCheck 则检查该部分内容是否擦除成功。

```
/*-----------------------------------------------------------------------
*Write a word to Flash, returns:
```

```
* 0 - OK
* 1 - write timeout
* 2 - Flash not erased
*/
static int write_word (flash_info_t *info, ulong dest, ulong data)
{
volatile u16 *tempPt;
/*zhangyy note:because of compatiblity of function,I use low & hi*/
u16 low = data & 0xffff;
u16 high = (data >> 16) & 0xffff;
low=swap_16(low);
high=swap_16(high);
tempPt=(volatile u16 *)dest;
_WR(0x555,0xaa);
_WR(0x2aa,0x55);
_WR(0x555,0xa0);
*tempPt=high;
_WAIT();
_WR(0x555,0xaa);
_WR(0x2aa,0x55);
_WR(0x555,0xa0);
*(tempPt+1)=low;
_WAIT();
return 0;
}
```

wirte_word 则想 flash 里面写入 unsigned long 类型的 data，因为 flash 一次只能写入 16bits，所以这里分两次写入。