基于 S3C44B0 的 UBOOT 移植

1 UBOOT 源码结构

此处介绍的 uboot 的版本是 1.1.4。

1.1 根目录

根目录下包含的主要文件有 README、config.mk、Makefile 和编译生成的执行及调试文件。其中，README 对 UBOOT 进行了详细的介绍，对 README 的了解有助于 uboot 的移植。config.mk 和 Makefile 文件是编译配置文件，控制整个的编译过程，移植时对其改动较小。编译生成多个以 u-boot 开头的文件，没有后缀的 u-boot 其实就是.elf 文件，可用于调试，而 u-boot.bin 文件是实际烧写到 FLASH 中的 bootloader 文件。

1.2 board 目录

board 目录下包含的子目录都以开发板的名字命名，如 dave 是以 dave 开发板命名的，其中的文件是 uboot 中的基于 S3C44B0 的相关文件。在移植过程中，需要在此添加自己开发板的子目录。在移植时，所有子目录中的文件都需被改动。下面简单介绍一下其中的文件。

config.mk：其中包含了 uboot 从 FLASH 映射到 SDRAM 中的起始地址。

uboot.lds： 编译后，uboot 各个段的分布及位置，与 ADS 中的 scatter 文件相似。

lowlevel.S：该文件被 cpu 中的 start.S 调用，用于初始化 cpu 的存储器控制器，并设置 cpu 与外围器件（如 Flash、sdram、以太网控制器等）接口的总线时钟。

flash.c： 初始化 flash 及相关读、写、擦操作，主要根据 flash 的 datasheet 改写。

gx44b0.c： 初始化 GPIO，需根据开发板的原理图改写。

1.3 common 目录

uboot 中各种命令的源文件。

1.4 cpu 目录

该目录下包含有以处理器芯片命名的子目录。子目录中与移植相关的文件的 start.S 是 uboot 第一阶段运行的硬件初始化程序，其实现了对 cpu 中各种控制器的初始化设置，总线时钟的设置，初始化中断和堆栈，并将 uboot 自身复制到 sdram 中，为第二阶段的运行做好准备。另外，cpu.c 文件初始化 cpu 和 cache，interrupt.c 设置中断和异常处理，一般不需要改动。serial.c 需根据 cpu 的频率修改相关波特率对应的参数。

1.5 disk 目录

磁盘驱动分区处理源文件。

1.6 doc 目录

uboot 的说明文档。

1.7 drivers 目录

通用设备驱动程序源文件，其中包含一些基本的驱动程序，如网卡、串口、USB 等。

1.8 dtt 目录

温度设备驱动程序源文件。

1.9 examples 目录

应用实例源文件。

1.10 fs 目录

包含了 uboot 所支持的文件系统，其中属于 Linux 文件系统的较多。

1.11 Include 目录

系统头文件所在目录，如目标板对应的头文件 include\configs\B2.h，该文件中包含 s3c44b0 对应的开发板的相关配置定义。移植过程中，需在此目录下添加自己的开发板所对应的头文件。

1.12 lib_xxx 目录

lib_xxx 目录，包含的相关体系结构的源文件，如 lib_arm，则包含了与 arm 体系结构相关的代码，此目录下的 board.c 文件中的 start_armboot 函数是 uboot 第二阶段，即 c 语言阶段，首先执行的代码，由 cpu 目录的对应 start.S 调用。

1.13 net 目录

该目录包含了网络协议相关源文件，如 NFS、TFTP、BOOTP 等。

1.14 post 目录

此目录实现了目标板的上电自检的功能。

1.15 rtc 目录

硬件实时时钟控制器驱动源代码。

1.16 tools 目录

此目录实现了生成 uboot 镜像文件的功能。

2 UBOOT 启动分析

uboot 的启动分为两个阶段，第一阶段由汇编语言实现，主要功能是对 cpu 及板载外围设备的初始化，建立堆栈后启动第二阶段；第二阶段由 c 语言实现，主要功能是实现各种外设的驱动，如 flash 读写、串口驱动、usb 驱动、网卡驱动等，并提供了强大的命令接口，与上位机通信。第二阶段为操作系统的移植与引导提供了基础。

2.1 第一阶段分析

对于 ARM7TDMI 的体系结构来说，系统上电复位后，从 0x00000000 开始执行，因此将 flash 存储器映射到这个地址。上电复位后，cpu 读取 flash 中代码开始执行。第一阶段对应的源文件是 cpu 目录下的 start.S 文件，这部分代码与 cpu 的体系结构密切相关。下面介绍一下 start.S 的执行流程。

2.1.1 异常中断向量表

```
.globl _start
_start:    b           reset
add  pc, pc, #0x0c000000
add  pc, pc, #0x0c000000
add  pc, pc, #0x0c000000
add  pc, pc, #0x0c000000
add  pc, pc, #0x0c000000
```

```
add  pc, pc, #0x0c000000
add  pc, pc, #0x0c000000
.balignl 16,0xdeadbeef
```

这是 uboot 中开始的一段代码，功能是实现异常中断向量跳转的设置。_start 是系统复位位置，在 uboot.lds 文件中指定，为 0x00000000。系统复位后，b 指令跳转到 reset 出开始执行，从而进入第二阶段继续执行。下面的 add pc, pc, #0x0c0000000 用于跳转到 sdram 中的中断处理程序处。b 指令的寻址空间是 32M，而 add 的寻址空间是 4G，reset 同处在 start.S 中，因此只需 b 指令就可寻址，但另外 6 个中断处理函数在 sdram 中，因此需要用有较大寻址空间能力的指令进行跳转。

当一个中断发生时，ARM 会强制把 PC 指针设置为对应中断类型地址处，该处的指令会跳转到自己的中断处理处继续执行。在 S3C44B0 的 datasheet 中有如下一段汇编代码：

```
ENTRY
 b ResetHandler ; 0x00
 b HandlerUndef ; 0x04
 b HandlerSWI ; 0x08
 b HandlerPabort ; 0x0c
 b HandlerDabort ; 0x10
 b . ; 0x14
 b HandlerIRQ ; 0x18
 b HandlerFIQ ; 0x1c
```

如，当发生 SWI 中断是，ARM 会跳转到 0x08 处，执行该处的指令，跳转到 HandlerSWI 处进行中断处理。不可以将中断处理程序写在 0x08 地址处，因为 ARM 要求，异常中断必须放在 0x00 开始的连续地址空间内。

.balignl 16,0xdeadbeef 指令的意思是将地址对齐为 16 的倍数，并用 0xdeadbeef 填充跳过的字节。

2.1.2 定义变量

```
_TEXT_BASE:
.word      TEXT_BASE
.globl _armboot_start
_armboot_start:
.word _start
/*These are defined in the board-specific linker script.*/
.globl _bss_start
_bss_start:
.word __bss_start
.globl _bss_end
_bss_end:
.word _end
#ifdef CONFIG_USE_IRQ
/* IRQ stack memory (calculated at run-time) */
```

```
.globl IRQ_STACK_START
IRQ_STACK_START:
.word   0x0badc0de
/* IRQ stack memory (calculated at run-time) */
.globl FIQ_STACK_START
FIQ_STACK_START:
.word 0x0badc0de
#endif
```

此处的作用是定义变量，以备接下来的程序使用。如

```
.globl _bss_start
_bss_start:
.word __bss_start
```

的意思是定义全局变量_bss_start，并有__bss_start 为他赋值，__bss_start 的定义在 u-boot.lds 中。其中，bss 段(Block Started by Symbol segment)是指用来存放程序中未 初始化的全局变量的一块内存区域。此处赋值变量的定义可以在 board 目录下的 config.mk 和 u-boot.lds 中找到。

2.1.3 初始化 cpu 寄存器

```
cpu_init_crit:
/* disable watch dog */
ldr     r0, =WTCON
ldr     r1, =0x0
str     r1, [r0]
/*mask all IRQs by clearing all bits in the INTMRs*/
ldr     r1,=INTMSK
ldr     r0, =0x03fffeff
str     r0, [r1]
ldr     r1, =INTCON
ldr     r0, =0x05
str     r0, [r1]
/* Set Clock Control Register */
ldr     r1, =LOCKTIME
ldrb    r0, =800
strb    r0, [r1]
ldr     r1, =PLLCON
#if CONFIG_S3C44B0_CLOCK_SPEED==66
ldr     r0, =0x34031  /* 66MHz (Quartz=11MHz) */
#elif CONFIG_S3C44B0_CLOCK_SPEED==75
ldr     r0, =0x610c1 /*B2: Xtal=20mhz Fclk=75MHz    */
#else
# error CONFIG_S3C44B0_CLOCK_SPEED undefined
#endif
str     r0, [r1]
```

```
ldr     r1,=CLKCON
ldr     r0, =0x7ff8
str     r0, [r1]
mov     pc, lr
```

在调用 cpu_init_crit 前，先设置 cpsr 进入 SVC 模式，之后通过指令 bl 跳转到 cpu_init_crit 处，继续执行。在 cpu_init_crit 中执行的动作分别是关闭看门狗，屏蔽所有其他中断，除了 TIMER5 中断，因为在 uClinux 下，用定时器 5 产生时钟节拍，为了能够使 uClinux 运行，将其打开。在重新设置好 PLLCON 之后，新的 PLL 时钟使用之前，PLL 会锁住时钟一会，这个时间就是 LOCKTIME，它由 LOCKTIME 寄存器设定，可以设置为 0xfff，也可以取值 t_lock * Fin（t_lock 为 200us）。根据 datasheet 提供的计算公示，设置 PLL。通过设置 CLKCON 使能各个功能的时钟。最后，将 lr 的值赋给 pc，回到 bl cpu_init_crit 下一条指令处继续执行。

2.1.4 初始化 memory 控制器

```
/*ldr r0, =MEMORY_CONFIG*/
mov r0, pc
ldr r1, =(0x38+4)
sub r0, r0, r1
ldmia r0, {r1-r13}
ldr     r0, =0x01c80000
stmia r0, {r1-r13}
mov pc, lr
```

调用 cpu_init_crit 返回后，继续调用 lowlevel_init 初始化总线的宽度及访问时序，lowlevel_init.S 文件在 board 目录下相关开发板的子目录中。lowlevel_init 的主要作用是，初始化存储空间，根据开发板所接的外围器件（如 flash、sdram、网卡控制器等）与 S3C44B0 连接情况，设置总线访问宽度和访问失序等。

在对存储空间进行初始化时，用到了 ARM 的多址寄存器寻址的指令，即 ldmia 和 stmia。如：

ldmia r1!, {r2-r7, r12};将 r1 单元中的数据读出到 r2-r7，r12 中，r1 自动加 1

stmia r0!, {r3-r6, r10};将 r3-r6, r10 中的数据保存到 r0 指向的地址，r0 自动加 1

lowlevel_init.S 中通过调用指令 ldmia 和 stmia，将初始化数据赋值到以 0x01c80000 开始的连续地址上。

列出的三条指令 mov r0, pc；ldr r1, =(0x38+4)；sub r0, r0, r1 是将放置初始化数据的地址赋值给 r0，即文件中 MEMORY_CONFIG 处。如何设置初始化数据，取决于硬件的连接情况，网上的一篇文章对此阐述的很透彻，它也解决了我的疑惑。

http://blog.csdn.net/axx1611/archive/2008/04/29/2342529.aspx

2.1.5 UBOOT 映射到 SDRAM

```
#ifndef CONFIG_SKIP_RELOCATE_UBOOT
 relocate:    /* relocate U-Boot to RAM */
 adr   r0, _start       /* r0 <- current position of code */
```

```
ldr r1, _TEXT_BASE/* test if we run from flash or RAM */
cmp    r0, r1    /* don't reloc during debug */
beq         stack_setup
ldr     r2, _armboot_start
ldr     r3, _bss_start
sub     r2, r3, r2          /* r2 <- size of armboot      */
add     r2, r0, r2          /* r2 <- source end address */
copy_loop:
ldmia r0!, {r3-r10}/* copy from source address [r0]    */
stmia r1!, {r3-r10}    /* copy to      target address [r1]    */
cmp   r0, r2 /* until source end addreee [r2]        */
ble     copy_loop
/* now copy to sram the interrupt vector */
adr     r0, real_vectors
add     r2, r0, #1024
ldr     r1, =0x0c000000
add     r1, r1, #0x08
vector_copy_loop:
ldmia r0!, {r3-r10}
stmia r1!, {r3-r10}
cmp   r0, r2
ble     vector_copy_loop
#endif       /* CONFIG_SKIP_RELOCATE_UBOOT */
```

此段代码用来实现将 flash 中的 uboot 移植到 sdram 中，flash 中起始地址由_start 指定，复制到 sdram 中起始地址由_TEXT_BASE 指定。通过 cmp r0, r1 确定是否需要将当前的 uboot 移植到 sdram 中，一般来说，在移植调试阶段，在没有仿真器的情况下，可以通过开发板自带的 bootloader 将 uboot 写到 sdram 中的_TEXT_BASE 处进行，这样不用反复烧写 flash。如果 uboot 通过串口等写入 sdram 中_TEXT_BASE 处，那么 uboot 的起始地址就和目标地址相等，这样便无需执行搬移的工作，直接跳转到 stack_setup 处执行。

uboot 从 flash 到 sdram 的映射工作是通过几个寄存器实现的。r0 存放 uboot 在 flash 中起始地址；r1 存放映射到 sdram 中起始地址；因为 bss 段与_armboot_start 中间的大小便是 uboot 的大小，所以 r2 中存放 uboot 在 flash 中结束地址。通过指令对 ldmia 和 stmia，及中间寄存器，便可将 flash 中的 uboot 搬移到 sdram 中。

接下来的一段代码，是将中断向量表映射到 sdram 中去。这里的中断向量表与此前提到的 0x00000000 处的异常中断向量表向对应，成为二级中断向量表，当发生中断时，ARM 跳转到 0x00 开始的对应中断入口出，执行此处的指令跳转到 sdram 中，在 sdram 中的二级中断向量表处，跳转到具体的中断处理函数中。此处复制到 sdram 中的起始地址是 0x0c000000+0x08，其中 0x0c000000~0x0c100000 之间的 128k 空间用于存放中断向量表、动态分配的系统空间和 uboot 中的 dbinfo 信息。在 sdram 中起始地址处所加的 0x08，可由 2.1.1 中的指令

```
add     pc, pc, #0x0c000000
```

解释，由于采用流水线结构的原因，此处取出的 pc 值等于当前指令地址加 0x08，因此需要将 sdram 中的中断向量表的起始地址设置为 0x0c000000+0x08。

2.1.6 堆栈指针设置

```
/* Set up the stack    */
stack_setup:
ldr r0, _TEXT_BASE /* upper 128 KiB:relocated uboot */
sub r0, r0, #CFG_MALLOC_LEN  /*malloc area*/
sub    r0, r0, #CFG_GBL_DATA_SIZE /* bdinfo*/
#ifdef CONFIG_USE_IRQ
sub  r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
sub    sp, r0, #12      /* leave 3 words for abort-stack      */
```

此段代码实现了对堆栈指针的初始化，此堆栈的指针设置在_TEXT_BASE 地址之前，即 sdram 中的 uboot 起始地址前，sp 指针的设置为 uboot 留出了内存分配空间+dbinfo 结构体空间+为 abort-stack 预留 3 个字。

2.1.7 进入第二阶段

```
ldr    pc, _start_armboot
```

将第二阶段 c 程序的起始地址赋值给 pc，uboot 对目标板的初始化结束，进入第二阶段继续执行。

2.2 第二阶段分析

在第一阶段的 start.S 中，通过讲_start_armboot 赋给 pc，将 uboot 的启动阶段跳转到第二阶段，第二阶段是在 sdram 中的一段 c 语言程序，主要用来实现对各种硬件设备的驱动，及实现 uboot 的命令控制功能。此部分主要分析 board.c 文件中的 start_armboot 函数，即 c 语言的入口函数。

2.2.1 初始化全局数据结构

start_armboot 的第一部分实现对全局数据结构 gd_t 的初始化。通过声明 DECLARE_GLOBAL_DATA_PTR 指定用 r8 寄存器作为此全局变量的存储地址。gd_t 用于存储 uboot 运行时所需要的配置信息。

```
gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
```

设置 gd 指针在 sdram 中的地址，_armboot_start 是 uboot 在 sdram 中起始地址，CFG_MALLOC_LEN 是动态分配所需要的空间。gd 指针所指地址是 0x0C100000（在 board 下对应目标板目录的 config.mk 文件中定义）－ CFG_MALLOC_LEN（在 include\configs 的相应目标板配置头文件中定义）－ gd_t 结构体大小。在结构体 gd_t 中的 bd_t 用于保存开发板配置的基本信息。此处与 start.S 中的 stack_setup 部分相关。

2.2.2 通用初始化函数

此处调用函数数组 init_sequence 中的各个初始化函数。

2.2.2.1 cpu_init()

函数 cpu_init 用来实现对 cpu 的初始化，其主要功能是调用 icache_enable 函数初始化内部缓冲区及 cache。

```
void icache_enable (void)
{
  ulong reg;
  s3c44b0_flush_cache();
  /*
      Init cache
      Non-cacheable area (everything outside RAM)
      0x0000:0000 - 0x0C00:0000
   */
  NCACHBE0 = 0xC0000000;
  NCACHBE1 = 0x00000000;
  /*
      Enable chache
  */
  reg = SYSCFG;
  reg |= 0x00000006; /* 8kB */
  SYSCFG = reg;
}
```

函数 icache_enable 调用 s3c44b0_flush_cache 清空缓冲区，通过设置 SYSCFG 为 0x06 启用 cpu 内部的 8K cache。因为对于一些操作不需要进行 cache 的缓冲，因此设置一段 non-cacheable 区域，用于非缓冲。NCACHBE0 = 0xC0000000;用来实现此功能。在 s3c44b0 的 datasheet 中也有对设置 Non-Cacheable Area 的说明：

Usually a cache stores any data in the whole system memory area, but sometimes it needs a non-cacheable area, because the cache cannot keep track of the external memory device whose contents are changed without read/write operation.

2.2.2.2 board_init()

初始化函数 board_init 实现对控制器的 GPIO 的配置 ，此处需要根据开发板的连接情况进行设置。函数 board_init 也是移植时需要修改的函数。

2.2.2.3 interrupt_init()

设置 Timer1 定时器，使其能自动装载计数值，定时产生中断信号。

2.2.2.4 env_init()

设置 uboot 环境变量，如果环境变量存储在 flash 中，则该函数的位置在 Env_flash.c 中。此处，将配置参数在 flash 中的地址赋值给全局变量结构体 gd_t 中的 env_addr，即 env_ptr->data 为 CFG_ENV_ADDR，并设置 env_valid 为 1。

```
gd->env_addr  = (ulong)&(env_ptr->data);
gd->env_valid = 1;
```

而具体实现将 flash 中的配置参数 copy 到 sdram 中是在 env_relocate 中。

2.2.2.5 其他函数

1. init_baudrate()：初始化波特率，将配置参数中的波特率赋值给 gd。
2. serial_init()：根据 1 中的波特率，调用 serial_init 初始化串口参数。
3. console_init_f()：初始化控制台。
4. display_banner()：打印 uboot 的起始地址及 bss 段的边界地址。
5. dram_init()：配置 sdram 的起始地址和大小。
6. display_dram_config()：打印标题信息。

2.2.3 FLASH 初始化

start_armboot 调用 flash_init 实现对 Flash 的初始化，此处和移植相关，需要在移植时根据开发板进行改写。在 flash_init 中，读取 flash 的厂商 ID 和设备 ID，配置 uboot 中的 flash ID 及其扇区的个数和大小，并根据已知的扇区数及 flash 的起始地址，初始化每个扇区的起始地址，最后返回 flash 的大小。

2.2.4 env_relocate 重定向配置参数

环境变量重定向函数，此函数实现将 flash 中的配置参数重定向到 sdram 中。通过 env_ptr = (env_t *)malloc (CFG_ENV_SIZE)，在 sdram 中分配一段内存，并将地址赋值给 env_ptr。随后调用函数 env_relocate_spec，在其中实现 memcpy (env_ptr, (void*)flash_addr, CFG_ENV_SIZE) ， flash_addr 的值为 CFG_ENV_ADDR，即将 flash 中的配置参数复制到 env_ptr 处。最后，将配置参数在 sdram 中起始地址赋值给 gd->env_addr。这样，便可在 sdram 中访问配置参数。

2.2.5 MAIN_LOOP

调用 main_loop 进入主循环，等待处理命令。

3 U-BOOT 移植

移植 uboot 的版本是 1.1.4，目标板是 GX44B0，移植 uboot 涉及的文件有：

1. board\gx44b0\lowlevel_init.S
2. board\gx44b0\gx44b0.c
3. board\gx44b0\flash.c
4. cpu\s3c44b0\start.S
5. cpu\s3c44b0\serial.c
6. drivers\rtl8019.h
7. drivers\rtl8019.c
8. include\configs\gx44b0.h
9. lib_arm\board.c

编译环境的话，最好还是用 linux，我开始时用的是 cygwin，但是编译的时候会遇到很多与移植无关的问题，解决起来比较费时间，所以我在 VMware Workstation 上装了 linux，作为编译环境。

调试主要用的是 AXD+JTAG，虽然不像 Mutli ICE 那样功能强大，不过对移植 uboot 还是很有帮助的。

3.1 创建目录和文件

1. 在 board 目录下创建目录 gx44b0，并将 board\dave\B2 目录下的文件及 board\dave\B2\common 目录复制到 gx44b0 目录下，将其中所有的 B2 改为 gx44b0。
2. 在目录 include\configs 下，建立文件 gx44b0.h，并将 B2.h 文件的内容复制到 gx44b0.h，将其中的 B2 改为 gx44b0。

3.2 修改移植文件

3.2.1 编译配置文件

1. 修改根目录下的 Makefile 文件，在 S3C44B0 Systems 下面添加：
   gx44b0_config : unconfig
   @./mkconfig $(@:_config=) arm s3c44b0 gx44b0
2. 修改 cpu\s3c44b0\config.mk 文件，
   替换
   PLATFORM_CPPFLAGS +=$(call cc-option,-mapcs-32,-mabi=apcs-gnu)
   为

PLATFORM_CPPFLAGS +=$(call cc-option,-mapcs-32,$(call cc-option, -mabi=apcs-gnu))

3. 修改根目录下的 config.mk 文件，打开调试信息选项
   DBGFLAGS= -gdwarf-2
   这样，编译结束后，在根目录下生成的 u-boot 文件，便是可以用 axd 进行调试的文件。

3.2.2 开发板配置文件

开发板的配置信息主要体现在文件 include\configs\gx44b0.h 中，要修改的配置参数有：

1. CONFIG_S3C44B0_CLOCK_SPEED：根据需要配置时钟频率；
2. CFG_ENV_SIZE：uboot 启动时环境变量所需空间。
   U-boot 中的 saveenv 命令执行时会将 setenv 命令设置的环境变量存储在 Flash，而 CFG_ENV_SIZE 则是环境变量存储空间的大小，CFG_ENV_SIZE 应和 Flash 块的大小一致，对于 NOR FLASH SST39VF1601 来说，它的块大小为 64KB，因此有 #define CFG_ENV_SIZE (64*1024)。
3. CFG_MALLOC_LEN：堆大小＋环境变量数据区大小。
4. CONFIG_DRIVER_RTL8019：用 CONFIG_DRIVER_RTL8019 替换 CONFIG_DRIVER_LAN91C96，因为我的网卡是 RTL8019。
5. CONFIG_RTL8019_BASE：网卡的基地址，只有基地址设置正确了，处理器才能正确操作 rtl8019 中寄存器。影响基地址确定的因素有：
   1） AEN 与 44b0 的哪个 nGCS 相连；
   2） IOS0~3 是否悬空，如果拉高或者拉低，需加上对应的偏移量；
   3） 如果 SA8~SA9 拉高，需加上 0x300。
   针对我的开发板的连接情况，设置基地址为 0x06000300。
6. CONFIG_SMC_USE_16_BIT：修改 CONFIG_SMC_USE_32_BIT 为 16BIT。此处定义的是网卡的数据总线与处理器的连接情况，对于 RTL8019 来说，网卡总线的位宽分为 16bit 和 8bit，具体选择取决于两个因素：IOCS16B 管脚和 DCR 寄存器的第 0 位，并优先取决于 DCR 的第 0 位。
7. CONFIG_COMMANDS：设置 uboot 所支持的命令，将其中的 CFG_CMD_EEPROM 改为 CFG_CMD_FLASH。如果想测试网卡，用开发板 ping 主机的话，还需要添加 CFG_CMD_PING，另外要注意的是，主机 ping 不通开发板，只能有开发板 ping 主机。
8. CFG_PROMPT：uboot 启动后在超级终端打印出的提示符形式。
9. CONFIG_NR_DRAM_BANKS：分配给 sdram 的 bank 数。
10. PHYS_SDRAM_1：sdram 的起始地址，sdram 的管脚 nCSC 芯片与 44b0 的 nGCS6 相连，所以起始地址为 0x0c000000。
11. PHYS_SDRAM_1_SIZE：sdram 的容量。
12. PHYS_FLASH_1：flash 的起始地址，为 0x00000000
13. PHYS_FLASH_1_SIZE：flash 的容量。
14. CFG_MAX_FLASH_SECT：flash 上的扇区数。SST39VF1601 为 2MB 的 nor flash，从 datasheet 中得知，sector 2Kword，即一个 sector 的大小是 2KB；block 32Kword，即一个 block 的大小是 64KB。这两个参数的目的是，flash 擦除

是按照 sector 或者 block 的方式进行的，flash.c 中对这两种擦除方式都有支持。可以选用 block 的方式：(32*64KB)=2MB；或者 sector 方式：(512*2KB)=2MB。这里选用 block 的方式，即 #define CFG_MAX_FLASH_SECT 32。

15. CFG_FLASH_ADDR0 CFG_FLASH_ADDR1：flash 的读写指令。
16. CFG_ENV_IS_IN_EEPROM：修改为 CFG_ENV_IS_IN_FLASH。
17. CFG_MONITOR_BASE：#define CFG_MONITOR_BASE PHYS_SDRAM_1 用于去掉 envcrc.c 中对 ENV_IS_EMBEDDED 的预定义，以消除环境变量无法保存的问题。
18. CFG_ENV_ADDR：环境变量的首地址。
19. CFG_ENV_OFFSET：环境变量相对 flash 起始地址的偏移量。

3.2.3 GPIO 初始化文件

文件 board\gx44b0\gx44b0.c 主要用来实现对开发板 GPIO 的设置，具体的配置需根据 s3c44b0 与设备的连接情况完成。

3.2.4 网卡驱动文件

Gx44b0 的开发板使用的网卡芯片是 rtl8019，因此移植网卡部分需要修改的文件有 drivers\rtl8019.h 和 drivers\rtl8019.c。rtl8019.h 文件的修改涉及到地址和寄存器的配置，只需简单的修改一下就可以了。

由于 gx44b0 开发板没有外接 EEPROM 93c46，因此需要对 rtl8019.c 文件进行一些改动。93c46 使用来存放网卡的物理地址等信息的，一上电，rtl8019 便会将 mac 地址读入自身的 ram 中，供驱动程序使用。因为开发板上没有 93c46，所以上电后，读入 rtl8019 的 mac 地址为 0。uboot1.1.4 中，在每次使用网络功能前，都会对 mac 地址进行校验，其校验方法是将环境变量中存储的物理地址与 0 比较，如果 mac 地址为 0 的话，就会报错*** ERROR: `ethaddr' not set，因此需要对 rtl8019.c 文件做些改动，以避免该问题。

我们在初始话环境变量时，已经设置好了 mac 地址，可是在 uboot 启动时，为什么打印出的 mac 地址信息是 0 呢？原因是，函数 start_armboot 在初始话设置时，会调用 rtl8019_get_enetaddr 打印 mac 地址信息，其实现语句是：

```
printf ("MAC: ");
    for (i = 0; i < 6; i++) {
        temp = get_reg (RTL8019_DMA_DATA);
        *addr++ = temp;
        temp = get_reg (RTL8019_DMA_DATA);
        printf ("%x:", temp);
    }
```

这里的 get_reg (RTL8019_DMA_DATA)功能是到 rtl8019 的 ram 中去读取 mac 地址，但由于开发板没有 eeprom，所以其读取的 mac 地址为 0。除此之外，它还将读出的地址赋值给参数 addr，而 addr 就是环境变量中存储 mac 地址的部分 gd->bd->bi_enetaddr，也就是说，此函数将我们设置好的 mac 地址更新为了 0。这

样，在后面的程序对 mac 带地址进行校验的时候，使用的是 0，而不是我们事先设置好的 mac 地址。

为了避免这个问题，我在 rtl8019.c 文件中，又增加了一个函数 rtl8019_get_enetaddr，它与原有的 rtl8019_get_enetaddr 函数通过环境变量 CONFIG_RTL8019_EEPROM 进行区分，如果在 gx44b0.h 文件中，定义了此环境变量，则调用 uboot 中原有的函数，否则，调用另外添加的函数。新增的函数 rtl8019_get_enetaddr，只是简单的从参数 addr 中读取 mac 地址，并打印出来。

```
void rtl8019_get_enetaddr(uchar *addr)
{
    unsigned char i;
    printf("RTL8019 MAC: ");
    for(i = 0; i < 6; i ++){
        printf("%.2x%c", addr[i], (i == 5)?' ':':');
    }
    printf("\b\n");
}
```

最后，在 start_armboot 函数中加入：

```
#ifdef CONFIG_DRIVER_RTL8019
rtl8019_get_enetaddr (gd->bd->bi_enetaddr);
#endif
```

3.2.5 串口配置文件

Uboot 通过串口向超级终端打印启动信息，并与用户交互，因此需要串口的驱动程序。对于串口驱动文件 cpu\gx44b0\serial.c，我们只需要在 case 中加入所需的波特率，并设置好 divisor。其中，divisor 与波特率和 MCLK 之间的关系是：

UBRDIVn = (int)(MCLK/(bps x 16)+0.5) -1

3.2.6 Flash 驱动文件

在 board.c 的 start_armboot 函数中，会调用 flash_init 对 flash 进行初始化设置，函数 flash_init 所在的文件是 board\gx44b0\flash.c。

对 flash 的移植，首先是将 board\gx44b0\comon\flash.c 文件合并到 board\gx44b0\flash.c 中。flash_init 会调用函数 flash_get_size 获取 flash 的大小，修改操作 flash 的指令部分

```
addr2[CFG_FLASH_ADDR0] = (CFG_FLASH_WORD_SIZE)0x00AA00AA;
addr2[CFG_FLASH_ADDR1] = (CFG_FLASH_WORD_SIZE)0x00550055;
addr2[CFG_FLASH_ADDR0] = (CFG_FLASH_WORD_SIZE)0x00900009;
```

为

```
addr2[CFG_FLASH_ADDR0] = (CFG_FLASH_WORD_SIZE)0x00AA;
addr2[CFG_FLASH_ADDR1] = (CFG_FLASH_WORD_SIZE)0x0055;
addr2[CFG_FLASH_ADDR0] = (CFG_FLASH_WORD_SIZE)0x0090;
```

这部分是 flash 功能寄存器的读写时序。

value = addr2[CFG_FLASH_READ0]一句是获取 flash 的厂商 ID，开发板的 flash 芯片是 SST39VF1601，因此其下的 switch 语句不用修改。

value = addr2[CFG_FLASH_READ1]一句是获取 flash 的设备 ID，需在其中添加

```
case (CFG_FLASH_WORD_SIZE)SST_ID_xF1601:
info->flash_id += FLASH_SST1601;
info->sector_count = 32;
info->size = 0x00200000;
break;                          /* => 2 MB           */
```

FLASH_SST1601 在文件 include\flash.h 中定义。

info->start[i] = base + (i * 0x00010000)一句是将 flash 分区，这里是按照 block 来分区的，一个 block 为 64KB。

在 flash_print_info 中加入

```
case FLASH_SST1601: printf("SST39VF1601 (16Mbit, uniform sector size\n");
        break;
```

以使在 uboot 启动时，打印 flash 的信息。

对 flash_erase 和 write_word 的修改与 get_flash_size 中相同，将其 32 位的指令改写为 16 位。

3.2.7 第一阶段启动文件

第一阶段的启动程序，即板载初始化程序是 cpu\s3c44b0\start.S 文件，在本文的第二部分 UBOOT 启动分析里已经做了详细的说明，因此这里只是简单的介绍移植过程。

这部分程序开始是异常中断向量表部分， 8 条跳转指令，不需要改动。网上的一些资料介绍将 add pc, pc, #0x0c000000 改为 ldr pc, _irq 等一些指令，更改后也可以使 uboot 的正常运行，因为当发生异常中断是，会程序会跳转到

```
undefined_instruction:
mov     r6, #3
b   reset
```

处执行重启。但当我在移植成功的 uboot 的基础上移植 uCOSII 时，发现发生异常中断时，不能正确的跳转到 uCOSII 的中断处理函数，而是执行原有的跳转进行重启。所以，我猜想可能是 ldr 指令没有跳转到 sdram 中的中断向量表部分，而是跳转到 flash 中的相应部分进行的。所以，我又将 ldr 改为 add，改后移植的 uCOSII 通过了移植测试。

系统上电后，跳转到 reset 处开始执行，首先将系统设置为 Supervisor 模式，然后跳转到 cpu_init_crit 处，对寄存器进行初始化，这部分的设置还是比较简单的，而且 uboot 原有的设置就可以使用。完成对 cpu 中寄存器的初始化后，程序会跳转到 lowlevel_init 继续执行，此部分在文件 lowlevel_init.S 中，完成对总线时钟进行设置，前面介绍的网址，对理解这部分很有帮助。对此处的移植，可以根据开发板实例程序中的相关部分进行改写。

在以上的初始化结束后，程序运行到 relocate，这部分完成搬运工作，就是将 flash 中的 uboot 复制到 sdram 中，此处也不需要修改。但我在移植时，犯了一个低级的错误，将 ldr r3, _bss_start 一句不小心给注释掉了，结果是在

sdram 中调试 uboot 时可以正常运行，但一烧到 flash 中就有错误，后来发现是那句话被注掉了。因为 uboot 在 sdram 中启动时，根本就不会执行 relocate 的部分，所以对 relocate 部分是不能在 sdram 中调试的（我用的是 AXD+JTAG），所以如果要改动这个部分，还需要多加注意。

在完成对堆栈的空间划分后（stack_setup 处），便会结束第一阶段，跳转到 start_armboot 继续执行。

4 小结

买了开发板后，进行的第一项工作就是移植一个 bootloader，开发板原有的 bootloader 是 armboot，没有源码，只是 bin 文件。上网了解后，说是 armboot 在 2002 年就结束了，被合并到 uboot 中，而其网上对 uboot 的评价很好，支持也比较多，所以决定移植 uboot。在移植过程中，在网上找了很多资料，所以在这里要感谢网络上那些无私奉献的人！

我的这篇文章，只是在前人的经验基础上，把自己在移植过程中遇到的问题和个人的理解加了进去。因此，难免会有疏忽或者错误的地方，不过大家都是学习嘛，只有遇到问题、解决问题才能成长，呵呵。如果写的有什么问题，还请大家多多指教，我的 msn 是 diamondwangyl@hotmail.com。