**AM335x uboot 移植**

2013-06-11 22:36:39|  分类： [am335x-艾默生](http://blog.163.com/shaohj_1999@126/blog/#m=0&t=1&c=fks_084069081085082069085094081095082085085067082094083066) |  标签：[uboot](http://blog.163.com/shaohj_1999@126/blog/#m=0&t=3&c=uboot)  [am335x](http://blog.163.com/shaohj_1999@126/blog/#m=0&t=3&c=am335x)  [移植](http://blog.163.com/shaohj_1999@126/blog/#m=0&t=3&c=移植)  |举报|字号 订阅

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 言从 个函数开始 行 语 这 执

board\_init\_f  在Board.c (arch\arm\lib)文件中

init\_sequence数 ， 行初始化 组继续进

relocate\_code  函数，重新回到 汇编 start.s文件中

relocate\_code：在start.s文件中

主要 了 实现 uboot的重新定位代 ，和 码 bss段的清零

ldr r0, \_board\_init\_r\_ofs   通 个方式 取 过这 获 board\_init\_r函数入口

adr r1, \_start

add lr, r0, r1

mov pc, lr   最后 里个到 这 C函数，board\_init\_r里面

board\_init\_r   第2 段的初始化，在 阶 board.c文件中

enable\_caches    个函数没有做什么内容 这

board\_init      第2次初始化平台， 里可以初始化其他内容了，和平台相关 这

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 境 量 环 变 /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 中 码

cpsw\_register      /driver/net/cpsw.c 注册一个以太网设备

main\_loop();  入控制台 进

init\_sequence数 里的内容 组

#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 的定 器， 开始运行 认 时 刚

uboot ，那个倒 多少 后按空格 入 时 计时 时间 进uboot

#ifdef CONFIG\_FSL\_ESDHC

get\_clocks,

#endif

env\_init, Env\_nand.c (common)  uboot默 的参数，都在 里初始化 认 这

init\_baudrate, 本文件中，初始化uboot使用的默 的串口波特率 认

serial\_init,   Serial.c (drivers\serial)初始化串口，在UBOOT中，使用的默 串口名 认

字是叫NS16550，可能是因 原来就存在此代 的 故，然后 用了 为 码缘 继续调 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， 出一些信息，代表运行到了 里 输 这

#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 用 继续调 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;   个 是一个 构体 这还 结

unsigned long flags;

unsigned long baudrate;  UBOOT串口 端所使用的串口波特率 终

unsigned long have\_console; 串口控制台是否初始化 ？ 过=1初始化过

//UBOOT 境 量相关的 量 环 变 变

unsigned long env\_addr; 个是地址，存放了 这 uboot使用的 境 量参数存放在内 环 变

存中 的地址，例如 了 对应 记录uboot的ip地址， 入 进uboot默 的延 等，和 认 时时间

default\_environment 起来 对应

unsigned long env\_valid; 默 的 境 量是否有效， 认环 变 =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 指 栈 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)

movr4, r0 /\* save addr\_sp \*/             新的堆栈地址

movr5, 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

movr6, r2 /\* save addr of destination \*/    目标地址

/\* Set up the stack     \*/

stack\_setup:

movsp, r4           设置堆栈

adr r0, \_start      uboot的起始运行地址

cmpr0, 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

beqclear\_bss /\* skip relocation \*/

movr1, 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

addr2, 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]    \*/

cmpr0, 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

addr10, 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

addr2, 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 \*/

addr3, 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! \*/

addr0, 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]

andr7, r1, #0xff

这里判断是相对位移还是绝对位移,然后修改

cmpr7, #23 /\* relative fixup? \*/

beqfixrel

cmpr7, #2 /\* absolute fixup? \*/

beqfixabs

/\* 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 \*/

movr1, r1, LSR #4 /\* r1 <- symbol index in .dynsym \*/

addr1, r10, r1 /\* r1 <- address of symbol in table \*/

ldr r1, [r1, #4] /\* r1 <- symbol value \*/

addr1, 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]

addr1, 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

addr2, r2, #8 /\* each rel.dyn entry is 8 bytes \*/

cmpr2, 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