嵌入式linux开发

# 嵌入式开发环境构建

硬件：在主机上编译bootloader，然后通过JTAG烧写入单板。

软件：在主机上编译嵌入式linux内核，通过BootLoader烧入单板或者直接启动。

应用：在主机上编译各类应用程序，单板启动内核后通过NFS运行他们，经过验证后再烧入单板。

主机与单板的三种连接：JTAG、串口、网络

主机linux操作系统上网络服务配置与启动：FTP、SSH、NFS

1、主机linux系统安装

2、主机linux网络与服务配置

3、交叉编译工具链安装

4、目标板的bootloader、linux内核、文件系统录入

# 嵌入式编程基础

交叉编译工具选项说明

Makefile介绍

常用ARM汇编指令即ATPS规则

ldm{cond}<addressing\_mode> <m>{1} <register list> {^}

相对跳转指令：b,bl

数据传送指令：mov, 地址读取伪指令：ldr

汇编语言中有三种操作数：寄存器操作数、存储器操作数和立即数

内存访问指令：ldr(从内存到寄存器), str(从寄存器到内存), ldm(批量内存访问质量，从内存到寄存器), stm(批量内存访问质量，从寄存器到内存)；

加减指令：add, sub

程序状态寄存器访问指令：msr， mrs

其他伪指令：

.extern main

.text

.\_global \_start

\_start:

# Windows与linux环境下的工具和命令使用

Windows环境下工具、命令使用

Linux环境下工具、命令使用

# GPIO接口

## GPIO硬件介绍

**GPIO寄存器**

通过寄存器来操作GPIO引脚：GPxCON寄存器（控制引脚的模式），GPxDAT寄存器（决定对应引脚的输入输出数据），GPxUP寄存器（决定相应引脚是否使用上拉电阻）

备注：上拉、下拉电阻的作用是当引脚处于第三态（即高阻态）时，它的电平由上拉、下拉电阻确定。

**访问硬件方式：**①通过寄存器；②通过总线；

**引脚操作：**单个引脚的操作无外乎三种，输出高低电平、检测引脚状态、中断；对某个引脚的操作一般通过读写寄存器来完成

**以总线形式访问硬件：**例如CPU访问nor flash；

地址对齐的访问：指的是访问的地址必须是2的倍数

例如CPU要读取16为数据，访问的地址0x2就是对齐的；访问的地址0x1就是没有对齐，因为没有对齐，CPU就会发起两次读操作，分别访问地址0x0得到字节D0、D1，和访问地址0x2得到字节D2、D3，最后组合D1、D2成16位数获得需要读的16为数据。

## GPIO操作实例

编写源程序、生成可执行程序、烧写程序、运行程序

# 存储控制器

“存储控制器”提供了访问外部设备所需的信号。

8根片选线+27根地址线

CPU内部寄存器地址

# 内存管理单元

内存管理单元（MMU）负责虚拟地址到物理地址的映射，并提供硬件机制的内存访问权限检查。

虚拟地址空间、虚拟地址、物理地址空间、物理地址

段、大页、小页、极小页

虚拟地址空间——物理地址空间：建立映射关系

虚拟地址VA、变换后的虚拟地址MVA，物理地址PA

# 移植u-boot

## Bootloader概念

一段初始化硬件设备、准备好软件环境的程序。关闭看门狗、改变系统时钟、初始化存储控制器、将更多代码复制到内存中等

可以增强BootLoader的功能：增加网络功能、从PC上通过串口或者网络下载文件、烧写文件、将flash上压缩的文件解压后在运行，这就是更强大的BootLoader，也称为Monitor。

**BootLoader启动：**CPU上电后会从某个地址开始执行。BootLoader就存放在这个地址开始处，这样一上电就可以执行。

启动加载模式和下载模式

嵌入式linux从软件角度通常分为4个层次：引导加载程序、Linux内核、文件系统、应用程序

**BootLoader结构和启动过程：**启动过程可以分为但阶段和多阶段两种；一般情况下是两阶段的启动；

第一个阶段：硬件设备初始化、为加载BootLoader第二阶段代码准备RAM空间、复制BootLoader第二阶段代码到RAM中、设置好栈、跳转到第二阶段的C入口处。

第二阶段：初始化本阶段要用到的硬件设备、检查系统内存映射、将内核映像和根文件系统映像从flash复制到RAM空间中、为内核设置启动参数、调用内核。

检测内核映射：确定内存使用情况和他们的地址

调用内核前必备条件：

1、内核寄存器设置

2、CPU工作模式

3、Cache和MMU设置

C语言重点：将绝对地址转化为函数指针以及跳转到指定位置出执行的技巧；首先通过强制类型转换将一个绝对地址转化为一个指针，然后在以函数指针的方式进行调用。

void (\*theKernel)(int zero, int arch, u32 params\_addr) = (void (\*)(int, int, u32) ) KERNEL\_RAM\_BASE;

**BootLoader与内核的交互：**单向交互、BootLoader将参数传递给内核、不能同时运行、BootLoader将参数放在某个约定的地方，再启动内核，内核从约定地方读取参数；

（通过约定的参数存放位置、规定参数的结构来实现BootLoader和内核的交互。）（标记列表）（标记列表以ATAG\_CORE开始，以ATAG\_NONE结束）

**标记数据结构：**由一个tag\_header结构和一个联合组成。tag\_header结构表示标记的类型和长度；对于不同的类型的标记使用不同的联合。

**u-boot分析与移植：支持多种CPU、多种类型开发板、支持多种嵌入式操作系统内核、支持NFS挂载、RAMDISK形式的根文件系统、支持从flash中引导压缩和非压缩系统内核、可灵活设置和传递多个参数给操作系统、支持目标板环境变量的多种存储方式、CRC32校验、上电自检**

## U-Boot分析与移植

### u-boot源码结构

根目录下有26个子目录、可以分为4类：

1. 平台相关的或者开发板相关的
2. 通用函数
3. 通用设备驱动程序
4. U-boot工具、示例程序、文档

Common目录下是一些通用函数，提供了一些通用指令。这些函数、指令被封装了，将与平台和开发板相关的代码用宏和外部函数来代替。二这些宏和外部函数，如果与平台相关，就在下一层次的CPU/xxx目录中实现；如果与开发板相关，就在下一层次的board/xxx目录中实现。

### u-boot的配置、编译、连接过程：

通过阅读Makefile来了解某款开发板、使用哪些文件、哪个文件首先执行、可执行文件占用内存情况。

make <board\_name>\_config #生成目标开发板的默认配置

make all #生成3个文件：U-Boot.bin, U-Boot, U-Boot.srec ;其中U-Boot.bin可以烧录入Nor-flash中运行

备注：编译u-boot成功后，还会在tool子目录下生成一些工具，如mkimage等。将他们复制到/usr/local/bin目录下，以后就可以直接使用。在编译内核时，就会自动使用mkimage来生成U-Boot格式的内核映像文件uImage。

#### 配置过程：

1. 确定开发板名称
2. 创建到平台/开发板相关的头文件的链接
3. 创建顶层Makefile包含的文件include/config.mk
4. 创建开发板相关的头文件

在board目录下新建一个开发板<board\_name>目录，则在include/config目录下也要建立一个<board\_name>.h，里面存放的就是开发板<board\_name>的配置信息。

配置文件中有以下两类宏：

①一类是选项，前缀为CONFIG\_，用于选择CPU、SOC、开发板类型、设置系统时钟、选择设备驱动

②第二类是参数，前缀为CFG\_，用于设置malloc缓冲池大小，U-Boot提示符，U-Boot下载文件是的默认加载地址、flash的起始地址等。

CONFIG\_除了设置一些参数外，主要用来设置U-Boot的功能、选择使用文件的那一部分，而CFG\_用来设置更细的参数。

备注：u-boot配置过程可以理解为两个关键过程，一是开发板的选择（配置顶层目录中的Makefile，mkconfig文件），二是开发板的配置（配置<board\_name>.h文件）

make smdk2410\_config #进行配置

1) make会执行Makefile文件，

2) smdk2410\_config是传递给Makefile文件的参数

3) 通过这个参数来执行并设定mkconfig的文件的参数，从而完成目标开发板的通用配置。

4) 所有开发板通用的配置设置都在mkconfig中，特定开发板具体应用配置都在<board\_name>.h中，需要提前配置好。

#### 编译、链接过程：

make all #进行编译

1. 确定开发板类型、编译器类型
2. U-Boot 目标文件的添加、平台/开发板相关的各个目录和通用目录下的相应库添加（OBJS、LIBS所代表的的.o，.a文件就是U-Boot的构成）
3. 当所有的OBJS、LIBS所表示的.o，.a文件都生成后，就剩下最后的链接。

基本编译流程：

1. 首先编译cpu/$(CPU)/start.S
2. 然后，对于平台/开发板相关的每个目录、每个通用目录都使用他们各自的Makefile生成相应的库。
3. 将1)、2)步生成的.o，.a文件按照board/$(BOARDDIR)/config.mk文件中指定的代码段起始地址、board/$(BOARDDIR)/U-Boot.lds链接脚本进行链接。
4. 第3）步得到的ELF格式的U-boot，后面的Makefile还会将它转换为二进制格式的S-Record格式。

### U-Boot启动过程源码分析

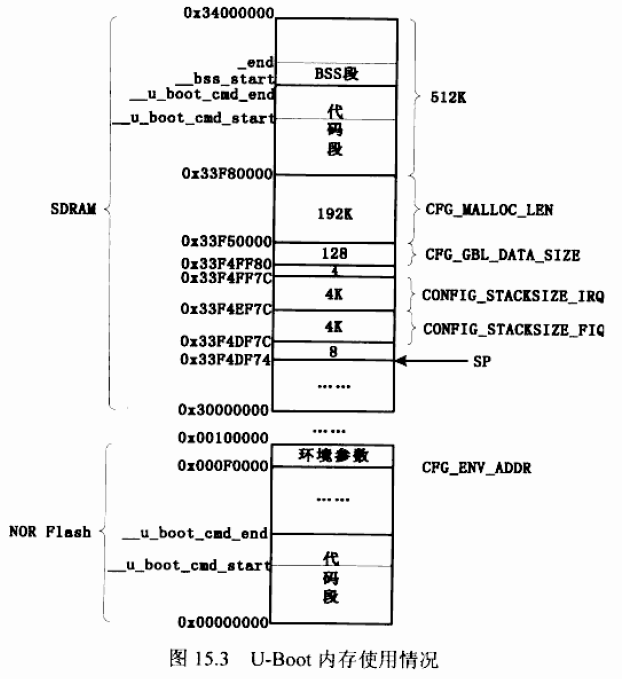
#### 第一阶段

①硬件设备初始化：将CPU设置为管理模式（SVC），关闭看门狗，设置FCLK、HCLK、PCLK的比例，关闭MMU、CACHE；

②为加载BootLoader第二阶段代码准备RAM空间：在start.S中调用lowlevel\_init函数来设置存储控制器，使得外界SDRAM可用。

③为加载BootLoader第二阶段代码准备RAM空间：在start.S中将整个u-boot代码复制到SDRAM中。

④设置好栈：栈设置很灵活，只要让sp寄存器指向一段没有使用的内存即可；



1. 跳转到第二阶段代码的c入口点：

备注：要确实理解第一阶段需要掌握u-boot在SDRAM、Nor-Flash中的存储结构；

#### 第二阶段

第二阶段从lib\_arm/board.c中的start\_armboot函数开始；

初始化本阶段用到的硬件设备

检测系统内存映射

U-Boot命令格式

为内核设置启动参数

### u-boot移植

nand-flash位反转现象比较严重，为保证数据的正确，在读写数据时需要使用ECC校验；在使用过程、传输过程还可能出现坏块。

因此在nor-flash中保存u-boot，在nand-flash中保存内核和文件系统，并在使用u-boot烧写内核、文件系统时进行坏块检查、ECC校验；

移植步骤：

1、新建一个开发板相应的目录和文件

将相似开发板的相关文件复制到新建的对应目录中，并修改相关文件；

首先是文件名修改，然后是顶层Makefile文件修改，对应开发板的Makefile文件修改；

2、修改SDRAM配置

SDRAM的初始化是在u-boot的第一阶段，就是在lowlevel\_init.S文件中设置存储控制器。

3、增加对S3C2440的支持(通过机器类型ID，来实现对多种开发板的支持)