# 前言

从事嵌入式行业多年，虽然因为工作原因接触过嵌入式Linux，也参与过相关产品的底层和应用功能开发，但对于嵌入式Linux的内核，驱动，以及上层开发，仍然停留在初级的水平，没有过系统深入的去总结整理，随着工作年限的递增，越来越感受到这种浮躁感带来的技术面瓶颈。既然发现了问题，自然就要去解决，回想起我踏入嵌入式行业来的经历，正是对STM32芯片以及网络部分的学习总结笔记支撑我走到如今的地步，那么沉淀下来，从嵌入式Linux入门开始整理，层层深入，对嵌入式Linux进行系统的总结也是最符合我目前现状的解决办法，这也是我下定决心放弃日常娱乐，开始本系列的由来。

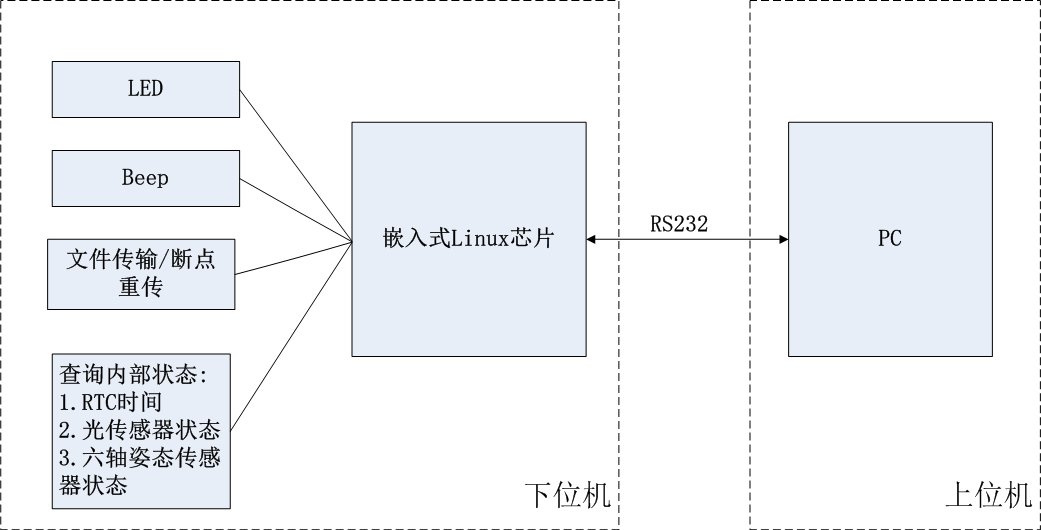
嵌入式Linux的掌握学习是很复杂的过程，从最基础的Linux安装，shell指令的学习和应用，交叉编译环境搭建，C语言开发，Uboot启动维护，Linux内核接口，Linux应用系统接口，在掌握了前面所有知识后，对于一个完整的产品，也只是完成了整个项目基础构建，这些知识不仅对于学习是难点，对于已经掌握的人来用文字描述清楚，特别是系统/软件版本引发的编译，调试问题，复杂应用的指针处理问题，这部分开发经验是文字很难描述的，往往只有自己去动手实践，才能够理解，这种复杂度对于入门者门槛是相当高的。

嵌入式Linux是一门应用开发技术，多练多总结才能积累足够的知识，保证职业生涯顺利走下去。需要耐心和持之以恒的努力，遇到问题是常态，一定不要着急，要善于思考，并培养使用搜索引擎或者官方论坛的解决问题的方法，但找到解决方法只是目的之一，如何从这些方法中总结经验，也是学习中的重要部分，这部分对于初学者尤其重要，切记！这也是我做嵌入式软件开发多年以来来最重要的经验。

对于大部分的学习者来说，可能按照如下的流程，从uboot，驱动，内核开始，先学习外设模块，在理解如何注册字符型设备，然后按照从易到难的顺序在掌握中断和时钟，文件系统，块设备，I2C驱动，LCD驱动，摄像头驱动，网络设备驱动，设备树，然后在学习涉及上层的QT界面，远程访问的网络socket(B/S， C/S框架)，以及应用端的Android平台开发，多线程，多进程同步等知识，这也是大部分开发板或者教程的学习方案，可从我经验来看，如果按照上面的流程是可以覆盖嵌入式Linux的主要工作需求的(部分知识是溢出的)。但是从产品实践的角度来说，这些都只是基础的技术，而不是具体实现产品的方案，事实上，对于刚入门的来说，如何从学习思维转变为工程师开发思维这部分也同样重要，从更高维的角度了解嵌入式Linux开发，这也是本系列的目的。我们先制定一个产品目标(可能不符合现有的产品模型)，所有学习都围绕着此产品来开发。这个系列将不仅仅讲述学习嵌入式，而且也讲述我根据工作积累的开发经验，讲述如何完成项目中的思考，也方便了解嵌入式软件开发的工作是什么，下面正式开始项目的实现吧。

# 项目架构初探

## 系统架构



## 硬件说明

正点原子的I.MX6U-ALPHA开发平台，256MB(DDR3)+256MB/512MB(NAND)核心板。涉及硬件 RS232，GPIO，I2C，SPI, ADC, DAC

## 代码路径

详细代码见:https://github.com/zc110747/remote\_manage

## 软件说明

1.上位机软件支持串口通讯，双机通讯需要制定协议(可使用自定义协议或者Modbus)，支持界面化管理(目前定义使用QT开发, 与后续的完善计划有关)

2.支持文件传输，文件传输支持断点重传(传输后文件位于指定文件夹，初步定义为/usr/download)

3.能够查询内部的一些数据，除显示已经列出状态外，支持后期扩展查询其他状态

## 任务分解

1. uboot，内核和文件系统的编译，下载和调试，并集成ssh方便传输应用文件调试

2. 分模块完成驱动的开发调试，不过为了方便测试及后期集成，需要同步完成串口驱动，串口通讯协议定义及上位机的软件框架

3. 后期的综合性功能调试和应用开发(如协议扩展问题，状态查询到界面显示，考虑到协议数据的复用, 后期该数据可能用于网页界面的状态显示或者QT界面的控制)

## 参考资料

宋宝华《Linux设备驱动开发详解 -- 基于最新的Linux4.0内核》第四章Linux内核模块

## 内核模块初探

本节作为整个系列的起点，重点当然是上面的项目规划和任务分解，不过为了让文章更丰富，我们可以初步体验下Linux下的应用和编程，下面代码将执行在Ubuntu系统，PC端，事实上PC端的Ubuntu可以验证很多实现，如加载驱动和设备，实现QT界面，进行网络通讯的应用端测试，所以一定不要忽略这个优势，本小节的代码都是在PC端测试完成，用于体验内核模块开发的特征。作为内核模块，可以通关Kernel编译时加入到内核中，也可以通过insmod/rmmod动态的加载到系统中，为了满足Linux系统的访问，内核模块就需要实现接口用于Linux访问，开发者只要按照规则用C语言实现这些需要的接口，在按照一定的规则编译后，就可以使用lsmod/rmmod来加载和移除自定义的模块，这套规则就是我们掌握内核模块需要学习的知识，下面来熟悉Linux内核的相关接口。

### 必须模块

模块加载函数：

module\_init(func)

模块卸载函数:

module\_exit(func)

模块许可声明：

MODULE\_LICENSE("xxx") 支持的许可有:

"GPL", "GPL V2", "GPL and additional right", "Dual BSCD/GPL", "DUAL MPL/GPL", "Proprietary"

### 可选模块

模块参数 -- 模块加载时传递变量 module\_param(name, charp, S\_IRUGO);

模块导出符号 --用于将符号导出，用于其它内核模块使用。

EXPORT\_SYSMBOL(func)/EXPORT\_SYSMBOL\_GPL(func)

**注意:Linux内核2.6增加了函数校验机制，后续模块需要引入时要在Module.symvers下添加导入函数内核的路径和symbol**。

模块作者 -- MODULE\_AUTHOR("xxx")

模块描述 -- MODULE\_DESCRIPTION("xxx")

模块版本 -- MODULE\_VERSION("xxx")

模块别名 -- MODULE\_ALIAS("xxx")

模块设备表 -- MODULE\_DEVICE\_TABLE

对于USB或者PCI设备需要支持，表示支持的设备，这部分比较复杂，这里就不在多说，后续如果用到，在详细去说明。

在了解上述模块的基础上，就可以实现如下的模块代码：

//hello.ko

#include <linux/init.h>

#include <linux/module.h>

//extern int add\_integar(int a, int b);

static char \*buf = "driver";

module\_param(buf, charp, S\_IRUGO); //模块参数

static int \_\_init hello\_init(void)

{

int dat = 3; //int dat = add\_integar(5, 6);

printk(KERN\_WARNING "hello world enter, %s, %d\n", buf, dat);

return 0;

}

module\_init(hello\_init); //模块加载函数

static void \_\_exit hello\_exit(void)

{

printk(KERN\_WARNING "hello world exit\n");

}

module\_exit(hello\_exit); 　　　　　　　　　　　　 //模块卸载函数

MODULE\_AUTHOR("ZC");　　　　　　　　　　　　 //模块作者

MODULE\_LICENSE("GPL v2"); //模块许可协议

MODULE\_DESCRIPTION("a simple hello module"); //模块许描述

MODULE\_ALIAS("a simplest module"); //模块别名

使用Makefile文件如下：

ifeq ($(KERNELRELEASE),)

KDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

modules:

$(MAKE) -C $(KDIR) M=$(PWD) modules

modules\_install:

$(MAKE) -C $(KDIR) M=$(PWD) modules\_install

clean:

rm -rf \*.o \*.ko .depend \*.mod.o \*.mod.c modules.\*

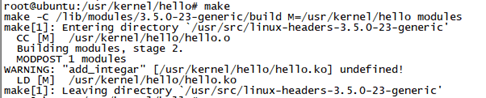
.PHONY:modules modules\_install clean

else

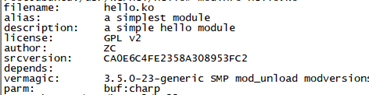
obj-m :=hello.o

endif

保存后，使用make指令既可以编译，如果遇到编译错误，请先查看文章最后的备注，未包含问题请搜索或者留言，编译结果如图所示。



之后执行指令modinfo hello.ko即可查看当前的模块信息。



如果无法查看信息，可通过dmesg查看加载信息。

https://img2020.cnblogs.com/blog/786514/202004/786514-20200421191734642-1990850423.png

### 内核模块的跨模块调用

上一节可以解决我们遇到的大部分内核实现问题，但某些时候我们可能需要一些公共内核模块，提供接口给大部分模块使用，这就涉及到内核模块的跨模块调用。

对于跨核模块调用的实现，对于调用的模块，主要包含2步：

在代码实现中添加extern int add\_integar(int a, int b);

在编译环境下修改Module.symvers, 添加被链接模块的地址，函数校验值(可通过查看被链接模块编译环境下的Module.symvers内复制即可)

对于被链接的模块，代码实现如下:

//math.ko

#include <linux/init.h>

#include <linux/module.h>

static int \_\_init math\_init(void)

{

printk(KERN\_WARNING "math enter\n");

return 0;

}

module\_init(math\_init);

static void \_\_exit math\_exit(void)

{

printk(KERN\_WARNING "math exit\n");

}

module\_exit(math\_exit);

int add\_integar(int a, int b)

{

return a+b;

}

EXPORT\_SYMBOL(add\_integar);

int sub\_integar(int a, int b)

{

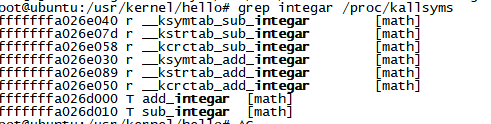
return a-b;

}

EXPORT\_SYMBOL(sub\_integar);

MODULE\_LICENSE("GPL V2");

编译Makefile同上，需要将**obj-m :=hello.o**修改为**obj-m :=math.o**，执行make编译完成该文件，并通过insmod加载完模块后，可通过**grep integar /proc/kallsyms** 查看加载在内核中的符号，状态如下：



然后加载insmod hello.ko, 即可跨文件调用该接口。如此，便初步完成对Linux内核模块的学习。

## 备注

1.**内核编译名称必须为Makefile，否则编译会出错**

　 make[2]: \*\*\* No rule to make target `/usr/kernel/hello/Makefile'. Stop.

　 make[1]: \*\*\* [\_module\_/usr/kernel/hello] Error 2

　 make[1]: Leaving directory `/usr/src/linux-headers-3.5.0-23-generic'

**2.Makefile的内容，如果编译多个文件obj-m :=hello.o test.o**

**3.Makefile中，指令必须以Tab对齐，否则编译会异常。**

4.**printk不打印，一般来说输出的KERNEL\_INFO为超过最大输出值，可直接通过dmesg，在系统信息内查看**。

**5.内核跨文件访问接口**

除EXPORT\_SYSMBOL外，在编译时Module.symvers需要包含对应函数的校验值，路径

0x13db98c9 sub\_integar /usr/kernel/math/math EXPORT\_SYMBOL

0xe1626dee add\_integar /usr/kernel/math/math EXPORT\_SYMBOL

否则编译时报警告

WARNING: "add\_integar" [/usr/kernel/hello/hello.ko] undefined!

安装模块时出错

[ 9091.025357] hello: no symbol version for add\_integar

[ 9091.025360] hello: Unknown symbol add\_integar (err -22)

# Linux系统命令，交叉编译

千里之行，始于足下。虽然本系列立足于应用角度讲述如何去学习嵌入式Linux，但对于基础仍然是不可或缺的，甚至这部分也穿插于整个项目的实现过程，这部分知识是需要系统的去了解的，但更重要的是在实践中积累总结，Linux是复杂的系统，虽然现在图形化也在进步，但基于命令行的主要访问方式对于开发者来说仍然是不可或缺的，占据重要的地位，对于大部分熟悉Windows界面化操作的用户来说入门是有些别扭的，就像我刚入门的时候也是很抗拒命令行，vim的，可是在熟悉后，发现命令行用起来也十分的爽快，对于命令行的学习基本没有捷径，无论是鸟哥的私房菜，还是专门讲述Shell语法的书籍，最后归结到原点，还是要多加练习，并整理总结，这是从菜鸟到高手的必经之路，下面正式开始这部分的讲解吧。

## Linux常用命令整理

**sudo su** 获取root权限

**clear** 清除当前界面

**ifconfig** 网络相关执行

ifconfig eth0 up

ifconfig eth1 up

**mkdir -p filepath** 创建路径，可递归创建

**apt-get install filename** 安装指定文件

**alias ll='ls -alF'** 重新定义指令

**ls /dev/\*** 查询当前的设备

**ls /dev/sd\*** 查询当前的是的sd卡设备

**ps -a | grep ssh\*** 查询当前执行的后台应用， grep可以限制名称

**kill -9 id** 关闭指定ID的后台应用

**tar -xvf filename** 解压到当前文件夹，后面可指定目录

**tar -vcjf filename.tar.bz2 \*** 压缩目录下文件和文件夹到指定的文件

**cat /proc/devices** 查询当前的设备总线

**scp -r file\_name system\_usr@ip\_addr:/filepath**

例如: scp -r uart\_proto root@192.168.1.251:/usr/app

通过ssh快速上传文件到指定地址

**insmod/rmmod/modprobe/lsmod** 加载/删除/带关联加载/显示内核模块

**modinfo xx.ko** 列出模块的信息

**mknode /dev/... c main\_id slave\_id**

例如: mknod /dev/led c 1 0

根据主从设备号创建设备节点

**ls /proc/slabinfo** 查看内存占用情况

**free -m** 查询内存的占用

**ln -s 原始路径 链接路径 生成文件链接，用于其它方式的访问**

## 交叉编译环境构建

在上一章我们已经根据项目需求确定了后续的实现目标:首先就要熟悉自己的平台，完成交叉环境编译环境的构建，并进行嵌入式Linux系统编译，下载和执行，理论上来说，这部分的内容其实是相当复杂的，如何选择指定的编译器，uboot的执行和裁剪，配置内核满足应用需求，设备树的构建，文件系统的加载流程，这些都是需要去掌握的重要知识。但是在本项目的实现流程我们不能够在这部分花太多时间，并不是他们不重要，而是对于产品来说，这部分是成体系的东西，不应该在最初的时候花费太大的精力去理解细节，某些时候使用官方或者开发板厂商提供的资源包，以方便快速开发作为基础是比较合理的方法。当应用开发一段时间，在各方面有着基础之后，遇到问题在反过来去理解和掌握，化整为零，即可以增强收获感，也可以，学与用结合，事半功倍。本次编译都会使用正点原子提供的修改后的内核和系统文件，另外**参考原子文档内的章节会讲解的更详细(具体细节按照原章节掌握更合理)**，这里因为篇幅原因只讲解流程中用到的知识，如果后续设计需要，则会根据应用需求来裁剪，后续参考资料：

1. 正点原子《Linux驱动开发指南说明》 第四章 开发环境搭建
2. 正点原子《Linux驱动开发指南说明》 第三十章 U-Boot使用实验
3. 正点原子《Linux驱动开发指南说明》 第三十七章 Linux内核移植
4. 正点原子《Linux驱动开发指南说明》 第三十八章 根文件系统移植

在大致理解上述资料后，就可以开始正式的交叉编译环境构建了：

选择 正点原子资料盘A盘 > 5、开发工具 > 1、交叉编译器中已经下载好的编译工具。

并根据Ubuntu系统位数的不同选择指定的编译器，如我安装的系统是64位，选择gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf.tar.xz文件

1. 在linux下使用指令sudo su，输入密码后进入root模式
2. 使用指令创建文件夹

**mkdir -p /usr/local/arm**

1. 将gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf.tar.通过SSH Secure Shell Client(SSH支持可参考其它相应文档)上传到创建的/usr/local/arm文件夹下，如果上传失败，要用

**chmod 777 /usr/local/arm**

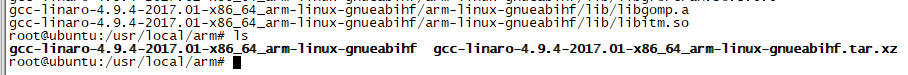
修改路径的权限, 上传之后如图所示



1. 使用指令

tar -xvf gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf.tar.xz

解压到当前路径，如下：



1. 将路径添加到全局变量上，使用vim /etc/profile指令，在末尾添加

export PATH="$PATH:/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf/bin"

同时使用vim /etc/environment，路径如下:

PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf/bin"

开启新的窗口，此时通过

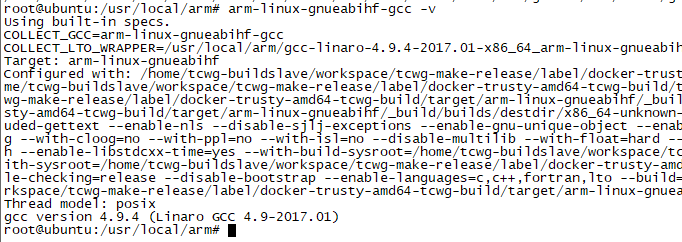
echo $PATH

即可查看全局路径下gcc的路径是否成功添加。

**注意:修改上述路径需要注意不要有语法错误，否则可能导致全局路径丢失，导致系统问题**

1. 最后通过指令，即可查看是否安装成功

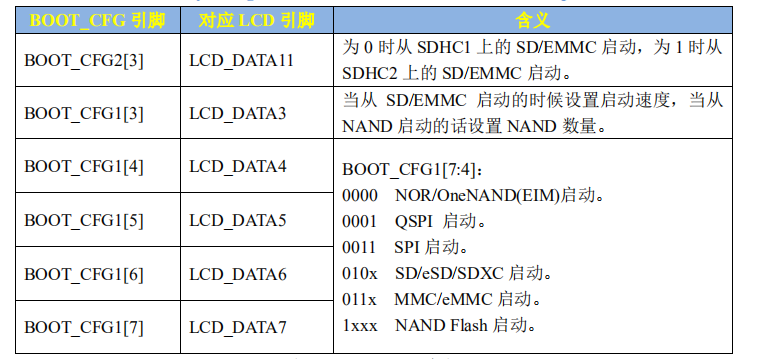
arm-linux-gnueabihf-gcc -v



至此，交叉环境的编译即搭建完成，因为以应用为目的，所以后续都以开发板提供的工具为准，如果想了解编译器和下面的安装包，建议详细去看《Linux驱动开发指南说明》的指定章节。

## Uboot编译和测试

从学习的角度来说，Uboot， 内核，文件系统在整个嵌入式Linux整个体系中占据最大的比重，这部分很重要，当然也是必须要掌握去了解，但是从应用的角度，首先最重要的是去实现需求，这部分初期不深究其实并不影响到实际项目的开发，而如果偏离上来从这地方开始学习，不仅会占用大量的时间，另一方面因为这部分资料比较分散，会很容易就因为没有目标而不知道如何学习下去，这不是入门者的问题，即使像我这样算比较资深的嵌入式工程师，也会同样面临相同的问题，因此不先整个系统为目标，在工程开发中穿插去了解，这种方法是合理且更加高效的。因此这里就以正点原子提供的uboot和linux系统为准，在掌握这部分知识之前，需要确定imx6ull的启动方式，这部分后面会经常用到。



1. 使用光盘资料A盘>1、例程源码>3、正点原子修改后的Uboot和Linux，将解压后的文件上传到Ubuntu到的路径下。
2. 在/usr/code/uboot下使用

tar -xvf uboot-imx-2016.03-2.1.0-g9bd38ef-v1.0.tar.bz2

指令解压，解决后结果如下:

https://img2020.cnblogs.com/blog/786514/202004/786514-20200427225103192-2045211347.png

1. 以我使用的测试硬件平台(DDR3/256M, NAND/512M)规格为例, 使用的默认配置文件名称为**mx6ull\_14x14\_ddr256\_nand\_defconfig，** 使用如下指令进行编译。

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- distclean

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- mx6ull\_14x14\_ddr256\_nand\_defconfig

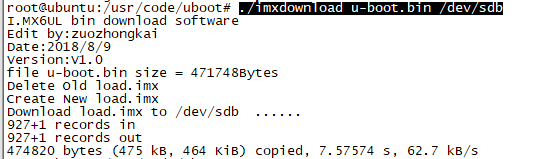
执行结果如下，即用于编译的.config的文件,最后执行编译指令:

make V=1 ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- -j12

如此就完成了编译，然后参考《Linux驱动开发指南说明V1.0》中8.4.3章节，使用imxdownload即可完成下载(具体如何下载参考。使用指令

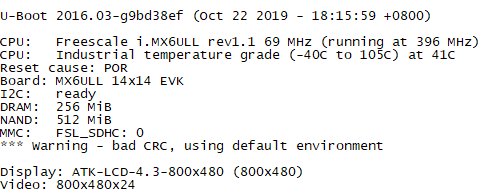
./imxdownload u-boot.bin /dev/sdb

即可实现下载，结果如下：



**至于sdb就是我的SD卡挂载的地址，可通过ls /dev/sd\*查看，此外这个下载会格式化指定的路径，需要格外注意，不要出错。**

将SD卡插在嵌入式Linux平台上，修改Boot为Pin1，Pin7高，复位芯片，打印如下：



到这一步初步完成了uboot的编译已经初步的测试运行情况，此外uboot也支持很多指令，这些后面会了解到，其中关于网络相关的FTP，NFS等支持的指令后面还会用到，就需要通过后面的实践去掌握了解。

## Linux内核编译

1. 使用光盘资料A盘>1、例程源码>3、正点原子修改后的Uboot和Linux，将解压后的文件上传到Ubuntu到的路径下，解压即可。
2. 在/usr/code/uboot下使用

tar -xvf linux-imx-4.1.15-2.1.0-g49efdaa-v1.0.tar.bz2

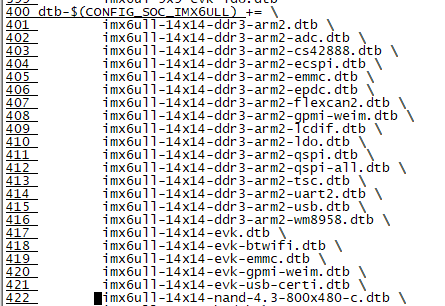
指令解压，解决后结果如下:

1. 使用如下指令

cd arch/arm/boot/dts

vim Makefile

在400多行添加对应的设备树，如我的开发板对应：



使用:wq保存文件即可。

1. 使用如下指令进行编译

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- distclean

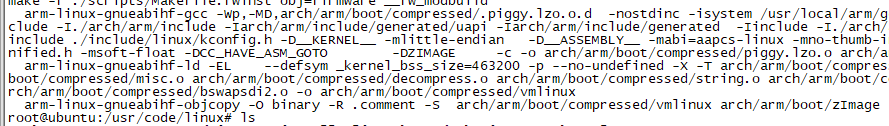
make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- imx\_v7\_defconfig

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- menuconfig

执行结果如，即用于编译的.config的文件,最后执行编译指令:

make V=1 ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- all -j16

如此即完成了编译，编译结果如下：



编译之后文件路径在

arch/arm/boot/zImage

arch/arm/boot/dst/imx6ull-14x14-emmc-4.3-800x480-c.dts

将上述文件和uboot文件通过SSH传输到Windows系统，为后面使用MfgTool工具下载做准备。

## 基于Busybox的文件系统编译

对于大部分嵌入式应用来说，可能上面运行的是基于yocto的支持qt环境的系统，或者支持Android运行的文件系统，但对于入门者来说，特别是初步不需要接触GUI相关应用的需求来说，初步用BusyBox编译的最小文件系统其实已经满足了大部分的需求，这里开始整个文件系统的编译，具体如下:

1. 使用光盘资料A盘>1、例程源码>6、BusyBox源码，将解压后的文件上传到Ubuntu到的路径下，解压即可。
2. 在/usr/code/uboot下使用

tar -xvf busybox-1.29.0.tar.bz2

指令解压，解压后结果如下:

1. 修改Makefile文件，最好使用绝对路径，如果找不编译器，可通过指令echo $PATH查看当前路径是否添加，支持中文字符详细参考文档38.2.2说明。

164 CROSS\_COMPILE ?= /usr/local/arm/gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-

190 ARCH ?= arm

1. 使用指令进行将链接编译到指定的路径

make install CONFIG\_PREFIX=/usr/code/rootfs/nfs/

1. 在编译后的路径添加文件夹，并将支持动态库添加到路径下，参考文档38.2.3说明。

https://img2020.cnblogs.com/blog/786514/202005/786514-20200501220054338-2122447047.png

1. 参考38.2.4章节，创建/etc/init.d/rcS,内容

#!/bin/sh

PATH=/sbin:/bin:/usr/sbin:/usr/bin

LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/lib:/usr/lib

export PATH LD\_LIBRARY\_PATH runlevel

mount -a

mkdir /dev/pts

mount -t devpts devpts /dev/pts

echo /sbin/mdev > /proc/sys/kernel/hotplug

mdev -s

创建文件etc/fstab，内容

#<system> <mount point> <type> <options> <dump> <pass>

proc /proc proc defaults 0 0

tmpfs /tmp tmpfs defaults 0 0

sysfs /sys sysfs defaults 0 0

创建etc/inittab文件，内容

#etc/inittab

::sysinit:/etc/init.d/rcS

console::askfirst:-/bin/sh

::restart:/sbin/init

::ctrlaltdel:/sbin/reboot

::shutdown:/bin/umount -a -r

::shutdown:/sbin/swapoff -a

然后使用打包指令

tar -vcjf rootfs.tar.bz2 \*

获取打包文件： rootfs.tar.bz2, 这就是我们编译打包好的最后需要用于下载的最小文件系统。

## Mgtool文件下载更新

首先mgtool是nxp公司提供用于下载的工具，这并非通用的更新技术，不过如果使用imx系列的芯片，学习如何下载更新也是必须的，这里进行说明

完成上述所有流程，我们就获得了最基础的底层应用结构，包含:

Uboot -- uboot.bin(重命名为imx6ull-14x14-nand-4.3-800x480-c.bin)

Kernel -- zImage, imx6ull-14x14-emmc-4.3-800x480-c.dts

文件系统 -- rootfs.tar.bz2

有了上述软件，就可以进行后续的代码下载，参考39.5章节说明：

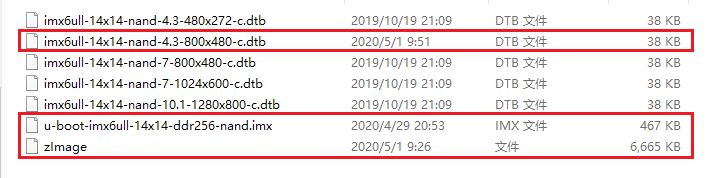
1. MgTool的工具使用原子提供，路径为A盘>5.开发工具>4.正点原子修改过的MFG\_TOOL烧写工具>mfgTool，通关将拨码开关置到仅2为高，然后复位，使用Mfgtool2-NAND-ddr256-NAND.vbs指令先下载测试。
2. MgTool的下载分为两部分，

* 将Profiles/Linux/OS Firmware/firmware下的文件下载到DRAM中，在跳转执行系统
* 与DRAM中运行的系统交互，将Profiles/Linux/OS Firmware/files内的文件通过UTP通讯使用指令将数据更新到NandFlash中

**注意点:files路径下的文件才是要更新的固件，如果替换了firmware，而编译的内核不支持UTP通讯的话，后续就会停在Unconnected位置**

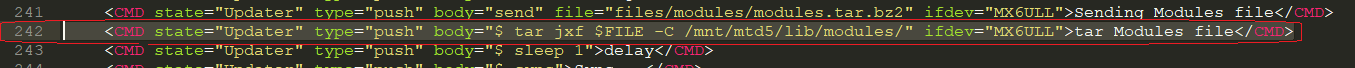
1. 替换files下的uboot，filesystem中的文件，名称要一致，内容如下

uboot路径下替换：

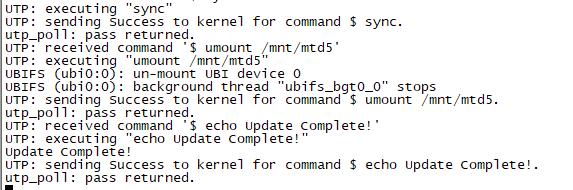


　　filesystem下直接用rootfs替换，

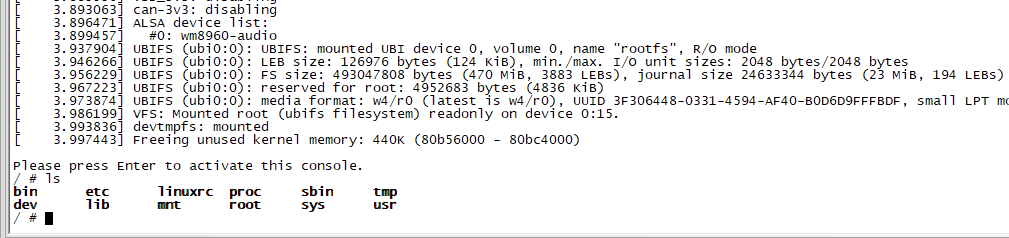
1. 替换后使用Mfgtool2-NAND-ddr256-NAND.vbs烧录即可, 另外如果出现tar error，可以直接删除Profiles/Linux/OS Firmware/ucl2.xml， 删除关于tar的指令



烧录完成后结果如下（通过串口连接电脑，就是打印系统启动信息的串口，波特率115200）:



1. 将boot置为Pin1， Pin5， Pin8为高，复位即可发现系统已经替换成我们编译的最小文件系统。



## 单步更新的方法说明

对于嵌入式系统来说，如果是空的芯片，执行上述的烧写流程是必须的，但是对于已经下载过的芯片，采用上述更新就有些复杂了，这时学会单步更新就比较重要，这里需要重要的工具mtd-untils，不过如果用的最小系统，默认是没有该工具的，需要自己编译实现，具体如下:

### 获取安装的资源包

wget [ftp://ftp.infradead.org/pub/mtd-utils/mtd-utils-2.1.1.tar.bz2](ftp://ftp.infradead.org/pub/mtd-utils/mtd-utils-1.0.0.tar.bz2)

wget <http://www.zlib.net/zlib-1.2.11.tar.gz>

wget <http://www.oberhumer.com/opensource/lzo/download/lzo-2.10.tar.gz>

git clone git://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git

git clone https://www.github.com/facebook/zstd.git

### 交叉编译资源包

#### 交叉编译zlib-1.2.11.tar.gz

mkdir -p lib/zlib

tar -xvf zlib-1.2.11.tar.gz

cd zlib-1.2.11/

CC=arm-linux-gnueabihf-gcc ./configure --prefix=/usr/code/mtd\_utils/lib/zlib --static

make && make install

#### 交叉编译lzo

mkdir -p lib/lzo

tar -xvf lzo-2.10.tar.gz

cd lzo-2.10

./configure CC=arm-linux-gnueabihf-gcc --host=arm-linux --prefix=/usr/code/mtd\_utils/lib/lzo --enable-static

make && make install

#### 交叉编译e2fsprogs

mkdir -p lib/e2fsprogs

cd e2fsprogs/

./configure CC=arm-linux-gnueabihf-gcc --host=arm-linux prefix=/usr/code/mtd\_utils/lib/e2fsprogs

make && make install

#### 交叉编译zstd

mkdir -p lib/zstd

cd zstd/

export CC=arm-linux-gnueabihf-gcc CXX=arm-linux-gnueabihf-g++ LD=arm-linux-gnueabihf-ld RANLIB=arm-linux-gnueabihf-ranlib AR=arm-linux-gnueabihf-ar CFLAGS=-fPIC CXXFLAGS=-fPIC LDFLAGS=-fPIC GYP\_DEFINES="$GYP\_DEFINES target\_arch=armv7"

make && make install

cp -r lib/\* ../lib/zstd

#### 编译mtd-utils

export ZLIB\_CFLAGS=-I/usr/code/mtd\_utils/lib/zlib/include

export ZLIB\_LIBS=-L/usr/code/mtd\_utils/lib/zlib/lib

export LZO\_CFLAGS=-I/usr/code/mtd\_utils/lib/lzo/include

export LZO\_LIBS=-L/usr/code/mtd\_utils/lib/lzo/lib

export UUID\_CFLAGS=-I/usr/code/mtd\_utils/lib/e2fsprogs/include/uuid

export UUID\_LIBS=-L/usr/code/mtd\_utils/lib/e2fsprogs/lib

export ZTSD\_CFLAGS=-I/usr/code/mtd\_utils/lib/zstd

export ZTSD\_LIBS=-L/usr/code/mtd\_utils/lib/zstd

export LDFLAGS="$ZLIB\_LIBS $LZO\_LIBS $UUID\_LIBS $ZTSD\_LIBS -luuid -lz"

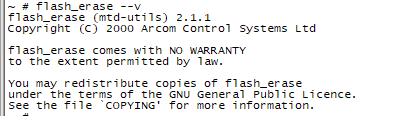
export CFLAGS="-O2 -g $ZLIB\_CFLAGS $LZO\_CFLAGS $UUID\_CFLAGS $ZTSD\_CFLAGS"

./configure --host=arm-linux CC=arm-linux-gnueabihf-gcc --prefix=/usr/code/mtd\_utils/mtd\_install --without-crypto

将编译完成后的固件通过

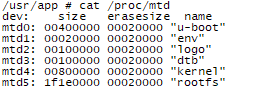
tar -vcjf mtd.tar.bz2 \*

压缩后上传到嵌入式开发板中，解压后在加到PATH中，后续可以使用flash\_erase相关指令更新固件。



### 单步新设备树到开发板中

1. 使用cat /proc/mtd查看分区情况



1. 更新设备树到nandflash中

flash\_erase /dev/mtd3 0 0

nandwrite -p /dev/mtd3 /home/root/imx6ull-14x14-nand-4.3-480x272-c.dtb

nandwrite -s 0x20000 -p /dev/mtd3 /home/root/imx6ull-14x14-nand-4.3-800x480-c.dtb

sync

如此，便可以完成设备树的更新.

### 单步更新uboot到开发板中

flash\_erase /dev/mtd0 0 0

kobs-ng init -x -v --chip\_0\_device\_path=/dev/mtd0 u-boot-imx6ull-14x14-ddr256-nand.imx

sync

### 单步内核到开发板中

flash\_erase /dev/mtd4 0 0

nandwrite -p /dev/mtd4 /home/root/zImage

sync

## 基础总结

至此，关于项目方案实现的第一步就已经完成，在本节中，体验了编译uboot，内核和文件系统，并进行了烧录，但如果对比本节的内容和参考文档的相关说明，就会发现我省略了很多内容，举几个例子，**官方的Uboot是如何变成原子的修改后的Uboot，如何通过make menuconfig配置符合需求应用**，还有对Makefile以及Shell命令也只是浅尝辄止，这些部分对嵌入式重要吗，事实上很重要，未来的很多时候都要和这些知识打交道，但当你当刚踏入嵌入式学习的时候，去深入钻研，一方面没有概念，难以建立清晰的脉络认知，另一方面这部分是基石，知识繁杂且耗时，学习就很难快速给予反馈，这其实是嵌入式学习起来最大的障碍，所以初步以运用为主，体验下编译流程即可，重心还是快速进入应用和驱动开发，等真正需要时(如将模块编入内核)，就需要去深入掌握，但那时肯定有了足够的嵌入式Linux开发经验，也更容易去理解，事倍功半，这也是我把这类知识放在后面去掌握的原因。

# LED驱动开发

在上一章中，构建了能够在开发板平台的完整系统，也学会了如何通过单步更新来满足后续功能操作的需求，从过程上因为直接使用原子修改好的uboot, 设备树和Linux内核，算是跳关了，不过马上就可以真正开始完整项目的第一实现了，那么对于一个完整项目的实现过程，最重要的是什么？其实不是对外设的掌握，也不是协议，或者应用层的逻辑实现，而是如何把**完整项目分解成独立的小任务，然后按照模块化的实现并组合的能力。**基于在第一章列出的项目模型，大致可以看到涉及的外设有led, beep, rs232, 六轴传感器(SPI), 光环境传感器(I2C), RTC等，并在这基础上构建基于UART的局域网通讯管理框架，最后实现上位机，在测试完成通讯后，便完整的构建了整个项目，对于实际开发中，这样并没有问题，无论是先驱动到框架，还是先框架，再将驱动按模块添加上去只是实现方式而已，并没有问题，不过从我的实际经验以及配合学习的效率来看，按照驱动模块开发 – 框架实现 – 驱动模块添加(...) –上层软件实现的迭代方式开发可以更加效率且完整的理解项目流程，按照这个经验，具体的任务可以分解如下:

1. 完成LED驱动，能够正常控制LED的点亮和关闭
2. 完成RS232的驱动，能够实现串口的通讯
3. 定义一套上位机、下位机之间的通讯协议(也可以使用主流工业协议如Modbus), 并在上位机和下位机编码实现通讯协议的组包和解包
4. 实现一套界面化的上位机工具，带有调试功能和控制功能

后续在通过迭代的方式继续完善各部分即可实现整个系统的升级，最终实现一个可用的产品项目，这也是本系列的最终目的，基于此策略，第一步就要实现LED的驱动，并完成LED的点亮和关闭的测试代码，虽然本身Linux内核自带LED对应GPIO的相关接口，并配置为heartbeat模式，建议直接在设备树中配置为off。在最初实现LED驱动的时候，因为对设备树不熟练，我也是使用ioremap实现物理地址到实际地址的转换，再操作控制LED，不过在使用readl和writel访问GPIO，因为都是对一组GPIO的访问，和其它驱动是会有冲突的(后面测试遇到过), 所以我还是放弃这种方式，直接选择设备树的方式来进行编写，这样Linux4.0内核主推的驱动编写方式，我也十分建议使用这种方式进行驱动模块的实现。从具体的功能来说，对于嵌入式Linux的驱动开发，可以归类于三个部分：

1. 对于硬件实际物理寄存器的配置和操作(这部分和单片机类似)
2. 封装的用于操作底层物理设备的设备树实现和接口访问
3. 将驱动添加到Linux内核的接口实现

而我么的实际操作也是通过这三部分进行的，这既是嵌入式Linux驱动开发的核心实现，从简单的GPIO，RTC，到复杂的SPI， I2C， LCD， 其本质上都要符合这个模型的实现。

## 参考资料

　　1. 开发板原理图 《IMX6UL\_ALPHA\_V2.0(底板原理图)》 《IMX6ULL\_CORE\_V1.4(核心板原理图)》

　　2. 正点原子《Linux驱动开发指南说明》 LED相关章节

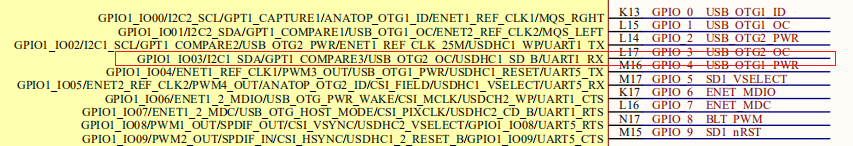
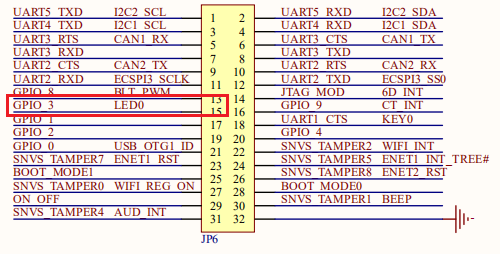
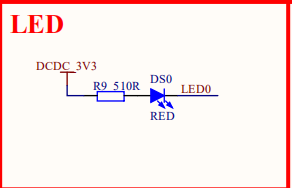
　　3. 宋宝华 《Linux设备驱动开发详解：基于最新的Linux 4.0内核》 第六章 字符驱动设备

　　4. Devicetree Specification Release v0.2

## LED硬件接口和DTS设备树

### 硬件原理图

　首先当然要确定原理图，下图来自底板和核心板原理图。



从上面的硬件可以得知，我们使用的是GPIO1\_IO3来进行LED的相关操作，如果是单片机来说，我们的大致操作大概是这样的：

1. 使能模块时钟
2. 配置模块或者相关模块的寄存器，使模块复用到需要的功能
3. 提供对外访问的接口

如果使用ioremap访问，那么具体实现和以上类似，不过本章中使用设备树和GPIO子系统实现驱动，其实设备树的添加也有一套固定的流程，大致如下:

1. 在板级添加相关的设备
2. 在iomuxc设备分支下添加GPIO相关的初始化
3. 基于设备树的初始化实现

### 设备树实现

这里先不讲DTS的语法，模拟其它模块实现LED的板级添加并不困难，找到设备树中的leds，在下面添加

    /\*led add by zc\*/

    led {

        compatible = "gpio-led";

        pinctrl-names = "default";

        pinctrl-0 = <&pinctrl\_gpio\_leds>;

        led-gpio = <&gpio1 3 GPIO\_ACTIVE\_LOW>;

        status = "okay";

    };

如此，便完成了板级的添加，这部分建议深入去了解掌握下，后面在驱动部分会经常和设备树打交道，这里不详细讲述，后面专门会对设备树进行深入讲解。在上述设备树实现的基础上，在iomuxc下添加配置信息，如下:

        pinctrl\_gpio\_leds: gpio-leds {

            fsl,pins = <

                MX6UL\_PAD\_GPIO1\_IO03\_\_GPIO1\_IO03    0x17059

            >;

        };

如此，便完成了设备树的修改，其中

MX6UL\_PAD\_GPIO1\_IO03\_\_GPIO1\_IO03 对应的宏定义在imx6ui-pinfunc.h中

#define MX6UL\_PAD\_GPIO1\_IO03\_\_GPIO1\_IO03   0x0068 0x02F4 0x0000 0x5 0x0

分别代表

mux\_reg, config\_reg, input\_reg, mux\_mode, input\_val, 后面参数为config\_reg的值的值。

### 对设备树的操作

LED基于设备树的初始化如下：

static int led\_gpio\_init(void)

{

    int ret;

    /\*1.获取设备节点 TREE\_NODE\_NAME(node:led)\*/

    led\_driver\_info.nd = of\_find\_node\_by\_path(TREE\_NODE\_NAME);

    if(led\_driver\_info.nd == NULL){

        printk(KERN\_INFO"led node no find\n");

        return -EINVAL;

    }

    /\*2.获取设备树中的gpio属性编号  TREE\_GPIO\_NAME (compatible:led-gpio)\*/

    led\_driver\_info.led\_gpio = of\_get\_named\_gpio(led\_driver\_info.nd, TREE\_GPIO\_NAME, 0);

    if(led\_driver\_info.led\_gpio < 0){

        printk(KERN\_INFO"led-gpio no find\n");

        return -EINVAL;

    }

    /\*3.设置beep对应GPIO输出\*/

    ret = gpio\_direction\_output(led\_driver\_info.led\_gpio, 1);

    if(ret<0){

        printk(KERN\_INFO"led gpio config error\n");

        return -EINVAL;

    }

    led\_switch(LED\_OFF);

    printk(KERN\_INFO"led tree hardware init ok\r\n");

    return 0;

}

对于LED的硬件操作，实现则如下：

static void led\_switch(u8 status)

{

    switch(status)

    {

        case LED\_OFF:

            printk(KERN\_INFO"led off\r\n");

            gpio\_set\_value(led\_driver\_info.led\_gpio, 1);

            led\_driver\_info.led\_status = 0;

            break;

        case LED\_ON:

            printk(KERN\_INFO"led on\r\n");

            gpio\_set\_value(led\_driver\_info.led\_gpio, 0);

            led\_driver\_info.led\_status = 1;

            break;

        default:

            printk(KERN\_INFO"Invalid LED Set");

            break;

    }

}

其中

gpio\_direction\_output

gpio\_set\_value

通过这两个接口，即可实现对于外部LED设备的访问。

## Linux内核加载和删除接口

在完成对LED底层硬件的封装后，下一步就是添加内核模块加载的接口，这部分并不复杂，参考之前接触的模块相关的知识，具体实现如下:

static int \_\_init led\_module\_init(void)

{

//加载后执行的动作

//......

}

static void \_\_exit led\_module\_exit(void)

{

   //删除时执行的动作

//......

}

module\_init(led\_module\_init);

module\_exit(led\_module\_exit);

MODULE\_AUTHOR("zc");                            //模块作者

MODULE\_LICENSE("GPL v2");                     //模块许可协议

MODULE\_DESCRIPTION("led driver");         //模块许描述

MODULE\_ALIAS("led\_driver");                    //模块别名

在完成上述流程后，一个最基本的模块框架即搭建完毕，下一步就是在框架的基础上，在Linux系统中完成对硬件的配置，并添加到设备总线上。

## 设备创建和释放

对于上层应用了解的开发者来说，可以很清楚open这一类接口是用来访问文件的，而在Linux中，字符型设备和块设备就体现了"一切都是文件"的思想，通过VFS(virtual Filesytem)将上层接口操作/dev/\*下的设备文件，最后访问到驱动内部注册的实际操作硬件的接口。那么如何让上层应用能够找到内核提供的接口，并能够管理内核模块，这就需要实现将设备添加的内核，以及设备释放的实现。

对于设备的创建需要四步:

1. 申请字符设备号(可以自己选择主设备号和从设备号, 也可以通过alloc申请设备号)
2. 配置设备信息，将设备接口和设备号关联
3. 创建设备类
4. 创建设备

static int \_\_init led\_module\_init(void)

{

    int result;

    led\_driver\_info.major = DEFAULT\_MAJOR;

    led\_driver\_info.minor = DEFAULT\_MINOR;

    /\*硬件初始化 – 参考设备树的实现\*/

    result = led\_gpio\_init();

    if(result != 0)

    {

        printk(KERN\_INFO"led gpio init failed\n0");

        return result;

    }

    /\*在总线上创建设备\*/

    /\*1.申请字符设备号\*/

    if(led\_driver\_info.major){

        led\_driver\_info.dev\_id = MKDEV(led\_driver\_info.major, led\_driver\_info.minor);

        result = register\_chrdev\_region(led\_driver\_info.dev\_id, DEVICE\_LED\_CNT, DEVICE\_LED\_NAME);

    }

    else{

         result = alloc\_chrdev\_region(&led\_driver\_info.dev\_id, 0, DEVICE\_LED\_CNT, DEVICE\_LED\_NAME);

         led\_driver\_info.major = MAJOR(led\_driver\_info.dev\_id);

         led\_driver\_info.minor = MINOR(led\_driver\_info.dev\_id);

    }

    if(result < 0){

         printk(KERN\_INFO"dev alloc or set failed\r\n");

         return result;

    }

    else{

        printk(KERN\_INFO"dev alloc or set ok, major:%d, minor:%d\r\n", led\_driver\_info.major,  led\_driver\_info.minor);

    }

    /\*2. 配置设备信息，将设备接口和设备号进行关联\*/

    cdev\_init(&led\_driver\_info.cdev, &led\_fops);

    led\_driver\_info.cdev.owner = THIS\_MODULE;

    result = cdev\_add(&led\_driver\_info.cdev, led\_driver\_info.dev\_id, DEVICE\_LED\_CNT);

    if(result != 0){

         unregister\_chrdev\_region(led\_driver\_info.dev\_id, DEVICE\_LED\_CNT);

         printk(KERN\_INFO"cdev add failed\r\n");

         return result;

    }else{

         printk(KERN\_INFO"device add Success!\r\n");

    }

    /\* 3.创建设备类  DEVICE\_LED\_NAME(led)\*/

    led\_driver\_info.class = class\_create(THIS\_MODULE, DEVICE\_LED\_NAME);

    if (IS\_ERR(led\_driver\_info.class)) {

         printk(KERN\_INFO"class create failed!\r\n");

         unregister\_chrdev\_region(led\_driver\_info.dev\_id, DEVICE\_LED\_CNT);

         cdev\_del(&led\_driver\_info.cdev);

         return PTR\_ERR(led\_driver\_info.class);

    }

    else{

         printk(KERN\_INFO"class create successed!\r\n");

    }

    /\* 4、创建设备(等同mknod) \*/

    led\_driver\_info.device = device\_create(led\_driver\_info.class, NULL, led\_driver\_info.dev\_id, NULL, DEVICE\_LED\_NAME);

    if (IS\_ERR(led\_driver\_info.device)) {

         printk(KERN\_INFO"device create failed!\r\n");

               unregister\_chrdev\_region(led\_driver\_info.dev\_id, DEVICE\_LED\_CNT);

               cdev\_del(&led\_driver\_info.cdev);

         class\_destroy(led\_driver\_info.class);

         return PTR\_ERR(led\_driver\_info.device);

    }

    else{

        printk(KERN\_INFO"device create successed!\r\n");

    }

    return 0;

}

同理，按照上面的流程，实现释放时的处理，如下：

static void \_\_exit led\_module\_exit(void)

{

    /\* 注销字符设备驱动 \*/

    device\_destroy(led\_driver\_info.class, led\_driver\_info.dev\_id);

    class\_destroy(led\_driver\_info.class);

    cdev\_del(&led\_driver\_info.cdev);

    unregister\_chrdev\_region(led\_driver\_info.dev\_id, DEVICE\_LED\_CNT);

    /\*硬件资源释放\*/

    led\_gpio\_release();

}

## 设备访问的接口

对于上层应用来说，访问的是open，read，write，close，ioctl的接口，对于底层来说，也增加相应的接口访问对应的接口。就是cdev\_add关联的设备接口和设备id，具体如下：

int led\_open(struct inode \*inode, struct file \*filp)

{

    filp->private\_data = &led\_driver\_info;

    return 0;

}

int led\_release(struct inode \*inode, struct file \*filp)

{

    return 0;

}

ssize\_t led\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*f\_pos)

{

    int result;

    u8 databuf[2];

    //LED开关和引脚电平相反

    databuf[0] = led\_driver\_info.led\_status;

    result = copy\_to\_user(buf, databuf, 1);

    if(result < 0) {

        printk(KERN\_INFO"kernel read failed!\r\n");

        return -EFAULT;

    }

    return 1;

}

ssize\_t led\_write(struct file \*filp, const char \_\_user \*buf, size\_t count,  loff\_t \*f\_pos)

{

    int result;

    u8 databuf[2];

    result = copy\_from\_user(databuf, buf, count);

    if(result < 0) {

        printk(KERN\_INFO"kernel write failed!\r\n");

        return -EFAULT;

    }

    /\*利用数据操作LED\*/

    led\_switch(databuf[0]);

    return 0;

}

long led\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)

{

    switch(cmd){

        case 0:

            led\_switch(0);

            break;

        case 1:

            led\_switch(1);

            break;

        default:

            printk(KERN\_INFO"Invalid Cmd!\r\n");

            return -ENOTTY;

    }

    return 0;

}

/\* 设备操作函数 \*/

static struct file\_operations led\_fops = {

    .owner = THIS\_MODULE,

    .open = led\_open,

    .read = led\_read,

    .write = led\_write,

    .unlocked\_ioctl = led\_ioctl,

    .release = led\_release,

};

如此便完成了上层接口需要访问的底层接口，至此，对于驱动加载的全部模块实现完毕，后续虽然有其它方法的驱动实现，但核心内容仍然在此框架下，经验是相通的。

## Makefile编译和模块加载

修改2.7章节的Makefile文件，如下：

KERNELDIR := /usr/code/linux

CURRENT\_PATH := $(shell pwd)

obj-m := kernal\_led.o

build: kernel\_modules

kernel\_modules:

    $(MAKE) -C $(KERNELDIR) M=$(CURRENT\_PATH) modules

clean:

    $(MAKE) -C $(KERNELDIR) M=$(CURRENT\_PATH) clean

执行make指令，编译完成后，通过ssh，sdcard或nfs的方式，将模块传输到开发板上，使用指令

insmod kernal\_mod.ko

即可完成模块的加载。

## 测试代码实现

测试代码就是对上层接口的访问，具体如下：

int main(int argc, const char \*argv[])

{

unsigned char val = 1;

int fd;

fd = open("/dev/led", O\_RDWR | O\_NDELAY);

if(fd == -1)

{

printf("/dev/led open error");

return -1;

}

if(argc > 1){

val = atoi(argv[1]);

}

write(fd, &val, 1);

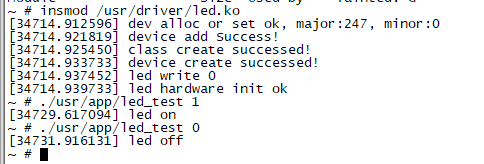
close(fd);

}

使用指令

arm-linux-gnueabihf-gcc xxx.c –o xxx

即可编译实现测试代码，将编译好的固件同样传输到开发板中，即可完成测试，结果如下:



## 总结

至此，关于LED的驱动开发基本讲解完成，虽然开发参考了部分例程用了不到1个小时，但完成这篇文档用了3个小时，为了能够将知识整理并能够讲解出来，是需要去查询书籍，查看内核代码，分析实现的原理，但是这部分是值得的，通过对完整流程的整理，我对完整的驱动有了更清晰的认知，很多曾经在开发和学习中不能够理解的问题也被一一突破，这是具有重要意义的。至此我们完成了整个项目开发的第一步，千里之行始于足下，后面也将一一突破难关，完成整个项目的实现。