# 前言

自从2015年正式入行以来，因为工作需求参与过交换机，停车场管理系统，指纹芯片，电源管理协议芯片以及医疗设备的开发，有基于裸机循环的前后台系统，有基于uC/OS-II或FreeRTOS的嵌入式实时系统，也有基于嵌入式Linux的运动控制平台，这些工作经历让我积累很多知识，不过虽然涉及面很广，也并非没有核心技术，但是一直没有系统的去整理, 回想起踏入嵌入式行业以来的经历，正是对单片机，网络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，进行系统的总结也突破瓶颈的最后办法，这也是本系列的由来。

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**，Cmake语法和Makefile编写, **uboot驱动和应用开发**，**Linux内核裁剪和驱动开发**，**设备树维护和修改**，**Linux系统接口访问和应用开发**，**C/C++项目应用和QT界面开发，**在掌握了前面所有知识后，对于完整的产品，也只对应项目的构建部分，在产品的开发过程中，还需要面临电力电子，网络，音视频图像处理，运动控制，传感器等方向的专业技术知识。这些技术知识因为面向行业的不同，往往需要深耕数年才能卓有建树，另外对于系统/软件版本引发的编译，调试问题，复杂应用和场景下的内存泄漏，异常处理机制，多线程/进程应用中的同步问题以及涉及行业的CMOS驱动，LCD，WIFI，zigbee，蓝牙，nb-iot等的调试开发经验，往往只有动手实践，遇到问题去理解总结才能掌握，这就对从业者有了更高的要求。

嵌入式Linux是一门应用开发技术，练习加总结是积累技术经验的唯一方法。这行业需要耐心，不断努力才能走下去，在开发中遇到问题要保持良好的心态，既要积极思考，也要善于使用搜索引擎，论坛以及其它从业朋友的分享中找到解决问题的办法，但解决问题只是目的之一，如何从这些方法中总结经验，并用于以后的开发实践中，也是整个过程中的重要部分，这部分对于初学者尤其重要。

对于大部分的入门者来说，都是按照如下的流程，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，然后在学习涉及多线程和多进程，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，但是从产品实践的角度来说，这些技术都还是Linux基础的范畴(基础不代表简单)，而不是具体的产品方案; 事实上，对于刚入门的开发者来说，如何从学习思维转变为工程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(可能不符合真正的产品模型)，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，而且也讲述工作中积累的开发经验，如何完成项目的思考，也方便了解嵌入式软件开发的工作是什么，这部分也是目前嵌入式开发资料中比较欠缺的经验分享。

对于本系列中可能存在的问题，如需要反馈，

可通过**QQ: 1107473010**或者对应QQ邮箱联系。

也可以通过我个人知乎<https://www.zhihu.com/people/wuzhidexingfu> 查看分享的嵌入式相关回答和文章。

# 目录

[1 前言 1](#_Toc12555)

[2 目录 2](#_Toc10944)

[3 项目产品架构 4](#_Toc7066)

[3.1 系统架构 4](#_Toc32655)

[3.2 硬件说明 5](#_Toc32467)

[3.3 代码路径 5](#_Toc21388)

[3.4 功能说明 5](#_Toc8710)

[3.5 任务分解 5](#_Toc17430)

[3.6 内核驱动模块初探 5](#_Toc6942)

[3.6.1 参考资料 5](#_Toc27984)

[3.6.2 必选接口 6](#_Toc1033)

[3.6.3 可选接口 6](#_Toc7777)

[3.6.4 模块间的调用 8](#_Toc24702)

[3.6.5 编译错误和解决办法 9](#_Toc8358)

[4 Linux系统命令和交叉编译 10](#_Toc24291)

[4.1 Linux常用命令整理 10](#_Toc12673)

[4.2 交叉编译环境构建 11](#_Toc13656)

[4.3 uboot编译和测试 13](#_Toc2993)

[4.4 Linux内核编译 14](#_Toc9273)

[4.5 基于Busybox的文件系统编译 15](#_Toc1110)

[4.6 mgtool文件下载更新 17](#_Toc12693)

[4.7 单步更新的方法说明 18](#_Toc8452)

[4.7.1 nandflash的单步更新 18](#_Toc3431)

[4.7.2 emmc的单步更新 21](#_Toc30619)

[4.8 总结 21](#_Toc19609)

[5 LED驱动开发 22](#_Toc28095)

[5.1 参考资料 22](#_Toc22807)

[5.2 LED硬件接口和DTS设备树 23](#_Toc23842)

[5.2.1 硬件原理图 23](#_Toc10105)

[5.2.2 设备树实现 23](#_Toc2811)

[5.2.3 对设备树的操作 24](#_Toc1811)

[5.3 Linux内核加载和删除接口 26](#_Toc5217)

[5.4 设备创建和释放 26](#_Toc18280)

[5.5 设备访问的接口 29](#_Toc15977)

[5.6 Makefile编译和模块加载 31](#_Toc1623)

[5.7 测试代码实现 31](#_Toc20526)

[5.8 总结 32](#_Toc21619)

[6 设备树的说明 33](#_Toc3637)

[6.1 设备树综述 33](#_Toc2411)

[6.2 设备树语法 34](#_Toc27079)

[6.2.1 #include语法 34](#_Toc12399)

[6.2.2 节点描述 34](#_Toc19505)

[6.2.3 节点属性 36](#_Toc29935)

[6.3 设备树在驱动中应用 40](#_Toc25877)

[6.3.1 内核设备树访问函数 40](#_Toc31279)

[6.4 总结 42](#_Toc23599)

[7 Uart通讯和协议实现 43](#_Toc7287)

[7.1 Uart通讯实现 43](#_Toc2610)

[7.2 通讯协议的制定 46](#_Toc20730)

[7.2.1 协议制定 46](#_Toc14766)

[7.2.2 数据处理 49](#_Toc16794)

[7.3 总结 53](#_Toc14376)

[8 QT界面开发和通讯实现 54](#_Toc27707)

[8.1 参考资料 54](#_Toc186)

[8.2 QT界面布局实现 54](#_Toc2374)

[8.3 数据处理逻辑 56](#_Toc21756)

[8.4 总结 61](#_Toc23419)

[9 I2C驱动和设备访问 62](#_Toc32269)

[9.1 硬件接口和协议 62](#_Toc23382)

[9.1.1 协议说明 62](#_Toc32491)

[9.1.2 硬件接口 63](#_Toc653)

[9.2 I2C总线框架 63](#_Toc2682)

[9.2.1 硬件拓扑和设备树 63](#_Toc24679)

[10 嵌入式多机通讯说明 65](#_Toc25331)

[11 附录 66](#_Toc9736)

[11.1 如何自学嵌入式Linux 66](#_Toc13708)

[11.1.1 嵌入式工作详解 66](#_Toc4583)

[11.1.2 嵌入式学习计划 67](#_Toc23701)

[11.2 嵌入式Linux问题总结 69](#_Toc7163)

[11.2.1 系统问题 69](#_Toc874)

[11.2.2 编译或执行失败问题 70](#_Toc4206)

[11.3 嵌入式系统开发中的中断机制 71](#_Toc17114)

[11.3.1 单片机中断机制 71](#_Toc29915)

[11.3.2 嵌入式Linux设备中的中断机制 74](#_Toc16893)

[11.3.3 总结 79](#_Toc13394)

[11.4 从USB协议解析去理解协议学习方法 79](#_Toc1684)

[11.4.1 USB协议解析 79](#_Toc12480)

[11.5 在芯片公司的嵌入式工作心得 79](#_Toc14677)

[11.6 从Linux Shell到Makefile语法 81](#_Toc3190)

# 项目产品架构

一个涉及嵌入式应用产品从无到有的设计生产是复杂的过程，包含以下步骤：

1. 用户需求分析和规格文档
2. 可行性分析，开发计划和框架设计
3. 硬件设备选型和软件技术选型
4. 硬件设计和demo方案开发
5. 桌面端软件设计，嵌入式Firmware设计，机械结构设计
6. 测试文档实现和验证计划安排，QA测试

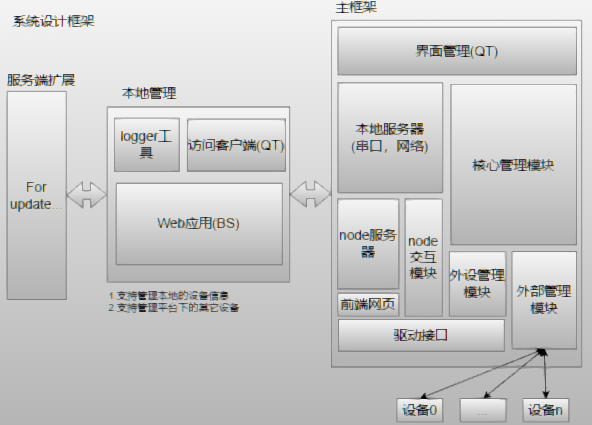
项目的开发都需要从用户的需求出发，结合市场经验和研发能力，分析可行性后，建立产品应用框架，经过硬件，Firmware，软件，云端服务等部分的开发，最后经过系统验证后量产交付。

根据应用框架，整个设计又可以根据功能的组件的差异，分解为具体的模块任务，最后组合起来，再测试给，就实现了完整的应用。

对于本文档希望实现的项目，因为时间和本人能力的限制，将不是具体的产品，而是主要展示应用技术的支持多硬件和多设备，用于验证所学并积累经验的带本地和云端控制的嵌入式平台，将涉及Linux，uboot, linux驱动，应用层C++开发，QT界面开发等。

## 系统架构

如何将项目分解成具体的任务分配当然是重要的知识，这部分的完成是需要对市场需求，SOC性能(工作频率，容量，外设参数, 功耗等)，硬件架构等技术支撑的，这也是系统工程师的主要工作，不过这里就不在深究，先把定义项目中本地管理中的内容分离出来，就得到了如下的精简框架, 这也是后续几个章节主要实现的应用模型。



基于这个模型，可以进一步拆分具体的工作。

对于嵌入式Linux平台开发是项目的核心，由以下部分构成。

1. 核心管理模块，包含外设管理，支持BS架构的TCP服务器和串口客户端，并提供接口与node服务器交互。
2. 界面管理，基于QT实现的界面, 用于支持本地的界面管理。
3. Node服务器+前端网页，基于node实现的web服务器
4. 扩展的外部管理模块，用于管理多个外部设备

## 硬件说明

硬件主要包含嵌入式Linux平台，作为下位机设备的主控，目前使用平台为正点原子的I.MX6U-ALPHA开发平台，涉及硬件如下表所示。

| 外部硬件 | 物理接口 |
| --- | --- |
| LED灯 | GPIO |
| 蜂鸣器 | GPIO |
| 用户按键 | GPIO |
| 光环境传感器 | I2C |
| 六轴传感器 | SPI |
| 串口 | RS232 |
| CAN口 | CAN |
| 网口 | RJ45 |
| TF卡 | SD/TF |
| USB摄像头 | USB |

上位机和云端服务器都使用PC平台，考虑到跨平台的应用的需求，使用QT作为应用界面开发.

## 代码路径

https://github.com/zc110747/remote\_manage

## 功能说明

从系统来说，产品包含下位机Firmware, 支持下位机访问和控制的Console客户端和Web界面，同是也提供远程访问的服务器，支持服务端功能的扩展。整个系统虽然看起来并不复杂，但包含嵌入式应用开发(Uboot，Linux内核，文件系统编译移植，驱动开发， 应用开发），桌面端界面开发技术，Web前端技术，这也是我进一步总结提高的技术，也是我坚持本项目开发的原因。

## 任务分解

1. 工作平台构建，包含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. 下位机驱动的开发调试，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，后期该数据可能用于网页界面的状态显示或者下位机QT界面的控制, 需要考虑到协议数据的复用)。

## 内核驱动模块初探

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，不过既然是嵌入式相关的学习，练习当然十分重要，适应Linux平台下的开发和编译习惯也是重要的能力之一。对于嵌入式的驱动开发， 其实就是配置和操作硬件，并提供一套API接口用于内核访问的过程。对于实现的驱动模块，可以通过在内核编译时直接加入，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，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来加载和移除驱动模块，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初步探究驱动模块吧。

### 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

### 必选接口

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"

### 可选接口

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，后续使用module\_param时需要引入时要在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.\*

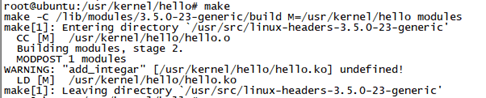
.PHONY:modules modules\_install clean

else

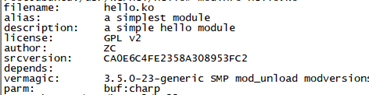
obj-m :=hello.o

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

### 模块间的调用

上面基本覆盖驱动开发遇到的大部分模块接口，但某些时候可能需要一些公共内核模块，提供接口给大部分模块使用，这就涉及到模块间调用。

对于模块间调用的实现，对于需要调用其它模块内函数的模块，需要包含如下2步：

1. 在代码实现中添加extern int add\_integar(int a, int b)。
2. 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)。 -- 校验机制，新版本内核不执行该操作会导致安全报错。

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)

{

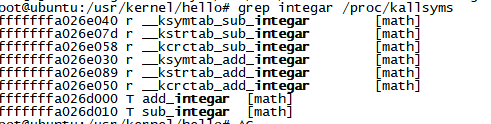
return a-b;

}

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。

### 编译错误和解决办法

这里记录在内核模块编译过程中常见问题和解决办法，详细如下:

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'

1. Makefile的内容，如果编译多个文件obj-m :=hello.o test.o
2. Makefile中，指令必须以Tab对齐，否则编译会异常。
3. 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。
4. 模块间调用出错

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系统命令和交叉编译

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shell和make相关的语法仍然是不可或缺的，这部分也穿插于整个项目的实现过程，是需要在实践中积累总结并掌握的。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Linux平台的开发者来说仍然占据重要的地位，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等操作，可是在熟悉后，发现命令行用起来十分的爽快，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，下面正式开始本章的讲解吧。

## 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 –a 显示当前的所有网络信息

ifconfig eth0 up 启动以太网口0

ifconfig eth1 up 启动以太网口0

**mkdir**  文件目录创建

mkdir -p xxx(filepath) 创建路径，可递归创建

**apt-get** 系统文件更新相关

apt-get install xxx(filename) 安装指定文件

apt-get update 更新当前的下载列表

apt-get upgrade 更新当前的文件

**alias** 重新定义指令

alias ll='ls -alF' 重新定义ll指令

**ls**

**常用指令**

ls /dev/\* 查询当前的设备

ls /dev/sd\* 查询当前的是的sd卡设备

ls /sys/firmware/devicetree/base 查询当前设备树文件

ls /proc/slabinfo 查看内存占用情况

**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，可配合ps使用

pid=$(ps -ex|grep xxx|awk '{print $1}'); kill -9 $pid 关闭指定名称的进程

tar 解压和压缩指令

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** 直接显示文件

cat /proc/devices 查询当前的设备总线

scp sshd作为客户端的发送指令

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

例如: scp -r uart\_proto [root@192.168.1.251:/usr/app](mailto:root@192.168.1.251:/usr/app)

**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

**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

**fdisk –l**  查询当前的文件设备

**mount 设备名称 系统路径**  用于挂载设备到文件系统中来访问

mount –t vfat /dev/mmcblk0p1  /mnt/sdb1 将设备mmcblk0p1挂载到目录/mnt/sdb1下

unmout /mnt/sdb1 释放挂载的设备

## 交叉编译环境构建

在上章我们确定了硬件平台，后续我们就要在这个平台上构建我们应用的基础，具体过程如下:交叉环境编译环境的构建，Uboot的编译和下载，嵌入式Linux内核的编译和下载，文件系统的编译和下载。这部分的内容在产品应用其实也是重要的，如何选择合适版本的编译器并添加到系统中，uboot的开发和裁剪，配置满足应用需求的内核，设备树的构建和维护，文件系统的加载，这些都是整个产品开发中需要去实现的功能。但是在本需求中，这部分的流程目前不影响整个项目的开发，从工程思维来说我先不要在这部分花太多时间，这并不是它们不重要，而是不应该在最初的时候花费太大的精力去理解这些细节，某些时候使用官方或者开发板厂商提供的资源包，快速开发才是是比较合理的方法。当应用开发一段时间，在各方面有着一定基础之后，遇到问题在反过来去理解和掌握，化整为零，即可以满足收获感，也能够学以致用，事半功倍。本系列平台将会使用正点原子提供的修改后的内核和系统文件，仅会根据开发的需求，裁剪和修改设备树及内核相关内容。

理解了这些，下面就开始正式的交叉编译环境构建。

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

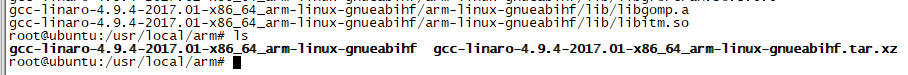
修改路径的权限, 上传之后如图所示



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

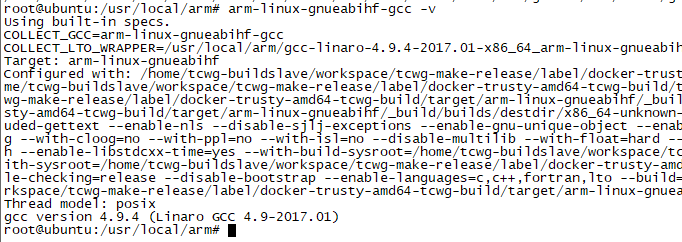
echo $PATH

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

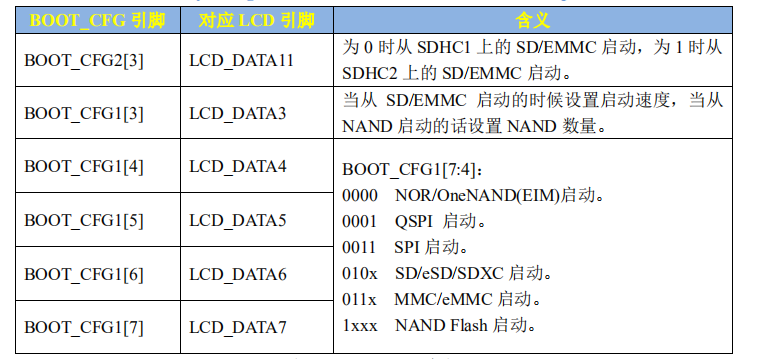


至此，交叉环境的编译搭建完成，因为是以应用为目的，所以后续都以开发板提供的工具为准，当然如果希望选择其它版本的arm-linux-gnueabihf-gcc的编译器，也可以直接去编译器的官网下载，具体路径如下: https://releases.linaro.org/components/toolchain/binaries/。

## 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，这部分是否重要，当然也是必须要掌握的，但是从完整应用的角度，首先最重要的是去实现需求，初期不深究这三部分其实并不影响实际项目的开发。如果偏离应用需求去钻研，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，会很容易就因为失去目标，没有反馈而不知道如何学习下去，这不是入门者的问题，即使资深的嵌入式工程师，也会同样面临同样的问题。不先以嵌入式系统平台为目标，在完整的项目开发中穿插了解，这种方法是合理且高效的。因此这里就以正点原子提供的uboot和linux系统为准，后续根据需求在来掌握上述知识，

当然在掌握这部分知识之前，需要了解开发板的启动方式，这部分后面会经常用到，具体如下:



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(DDR3/512M, emmc/8G)规格为例, 使用的配置文件名称为**mx6ull\_14x14\_evk\_emmc\_defconfig**, 则使用如下指令进行编译。

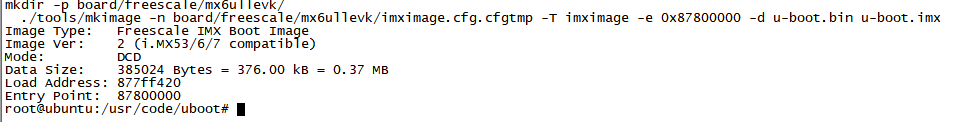
make distclean

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- mx6ull\_14x14\_evk\_emmc\_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



这里的u-boot.bin和u-boot.imx就是后续需要用到的下载文件。

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

CONFIG\_SOC\_IMX6ULL下添加对应的设备树，如下



使用:wq保存文件即可。

1. 使用如下指令进行编译

make distclean

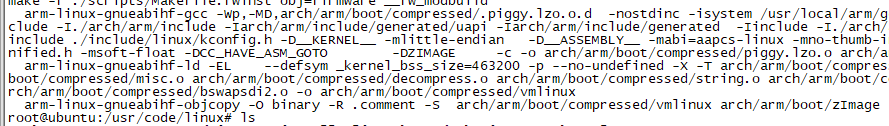
make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- imx\_v7\_defconfig

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：



编译后文件路径如下:

arch/arm/boot/zImage

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

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开发来说，可能运行的是基于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

1. 执行命令

make menuconfig

在弹出的界面中进入Busybox Settings 选项

1. 在Build Options -> Cross Compiler prefix中添加编译器**arm-linux-gnueabihf-**
2. 在Installation Options -> BusyBox install prefix中将./\_install修改为../system(目的是将输出路径修改。
3. 执行

make&&make install

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,执行

mkdir dev etc lib mnt proc sys tmp var

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

1. 编译完成后，将编译工具下的lib库复制到编译完成后的lib路径下。

主要路径包含

安装目录/arm-linux-gnueabihf/lib/

安装目录/arm-linux-gnueabihf/libc/lib

cp \*so\* \*.a /usr/code/rootfs/lib –d

并对

安装目录/arm-linux-gnueabihf/libc/lib下的ld-linux-armhf.so.3重新执行复制操作

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

LANG=en\_US.UTF-8

mount -a

mkdir /dev/pts

mount -t devpts devpts /dev/pts

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

mkdir /var/empty

mkdir /mnt/root

mount -t vfat /dev/mmcblk1p1 /mnt/root

ifconfig eth0 up

ifconfig eth1 up

ifconfig eth0 192.0.18.250 netmask 255.255.255.0

route add default gw 192.0.18.1

ifconfig eth0 mtu 1200

/usr/local/bin/sshd

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，用于下载的最小文件系统。

## mgtool文件下载更新

工作模式:1,3,6,7高

下载模式:2高

首先mgtool是NXP官方提供用于下载的工具，这并非通用的技术，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，下面说明具体的流程。

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-emmc-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外部设备中中

**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

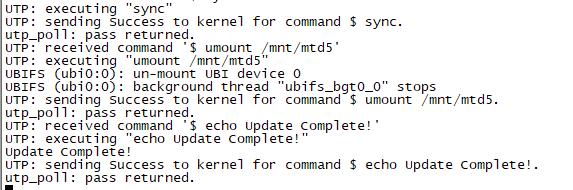
uboot路径下替换：



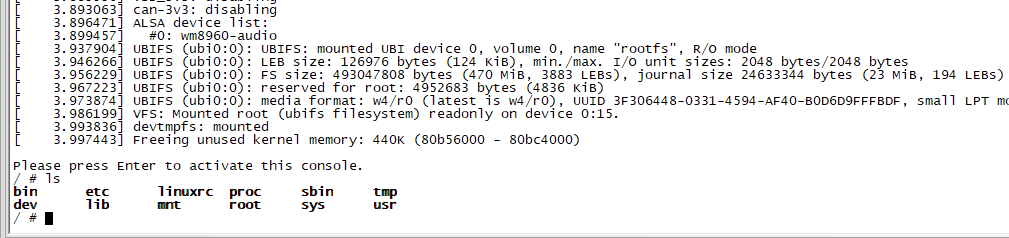
filesystem下直接用上章的rootfs替换



1. 替换后使用Mfgtool2-eMMC-ddr256-eMMC.vbs烧录即可, 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1， 3， 6， 7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

### nandflash的单步更新

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"

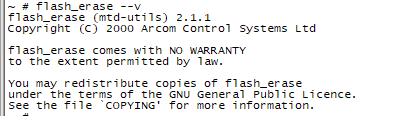
export CFLAGS="-O2 -g $ZLIB\_CFLAGS $LZO\_CFLAGS $UUID\_CFLAGS $ZTSD\_CFLAGS"

./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。



#### 更新uboot

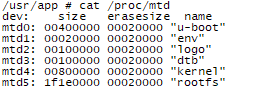
flash\_erase /dev/mtd0 0 0

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

sync

#### 更新设备树

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.

#### 更新内核

flash\_erase /dev/mtd4 0 0

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

sync

### emmc的单步更新

#### 更新uboot

echo 0 > /sys/block/mmcblk1boot0/force\_ro

dd if=/tmp/u-boot.imx of=/dev/mmcblk1boot0 bs=1024 seek=1 conv=fsync

echo 1 > /sys/block/mmcblk1boot0/force\_ro

## 总结

至此，整个项目运行的硬件平台就实现了(后续有可能替换为支持QT的系统)，在本节中，我们进行编译工具的环境构建，编译了uboot，内核和文件系统，并进行了代码烧录和初步代码的测试，不过跟实际开发中，整个流程忽略了很多内容，举几个例子，**官方的Uboot是如何变成修改后的uboot，如何通过make menuconfig裁剪出需要的内核**，还有对Makefile以及shell命令部分也只是浅尝辄止，这些部分对嵌入式事实上很重要，未来的很多时候都要和这些知识打交道，但对于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，学习不简单，也很难出成果有正面反馈，这也是我学习嵌入式遇到的最大障碍。正是这种经历让我认识到要把重心先放在如何去了解嵌入式应用和驱动开发，在结合实际的需求，在需要时在去深入理解掌握这些嵌入式Linux中的基础知识，目前实践起来效果也不错，这也给了我按照此方法学习提供的信心。

# LED驱动开发

构建了开发板平台的基础软件后，下一步就是将**任务进一步分解成独立的模块，在组合完成具体的任务。**

基于之前列出的任务模型，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，便完整的实现任务需求，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无非是实现方式的问题，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能够的更有效的验证，按照这个经验，任务的具体实现步骤如下:

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
5. 在这基础上扩展底层驱动，同时协议和上位机工具增加相应的模块或接口来处理显示，通过迭代完整的实现整个应用。

整个项目经过模块化的组合和分模块迭代，最终实现一个可用的产品项目，这也是比较通用的产品开发办法。

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，因为本身Linux内核自带LED对应GPIO的相关接口，并配置为heartbeat模式，因此建议在内核中关闭该功能，具体为:

Device\_Driver->LED\_SUPPORT->LED Trigeer support->LED Heartbeat Trigger

在make menuconfig中关闭上述应用，如此就可以进行本章节的测试。

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，我也十分建议直接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

而我本节的实际开发也是可以分解为三部分进行的，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。

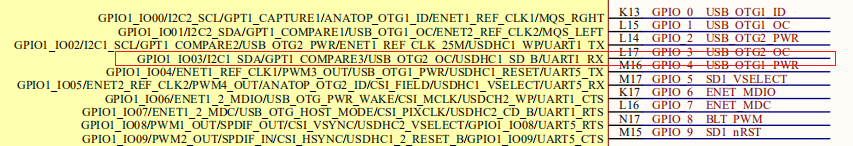
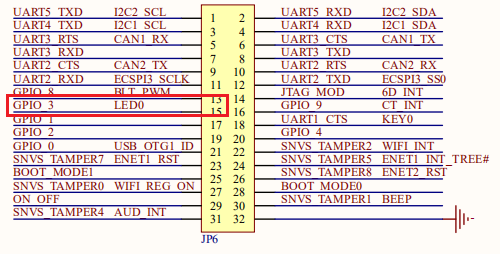
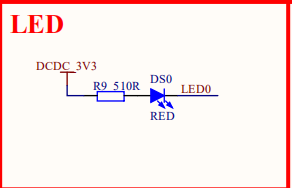
## 参考资料

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. 基于设备树访问接口的驱动实现

### 设备树实现

DTS语法并不困难，但描述起来也需要很多知识要讲，而且上来去深入理解DTS并不简单，所以这里先暂且不讲DTS的语法，后面更熟练些再讲解，另外模仿其它节点下的模块实现LED的板级添加并不困难，具体如下:

    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，这里不详细讲述，后面有计划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

对与嵌入式Linux上层应用来说，是使用open这一类接口是用来访问文件的，而且在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;

//打开LED设备

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]);

}

//将控制数据写入LED设备中

write(fd, &val, 1);

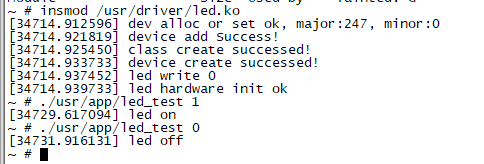
close(fd);

}

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

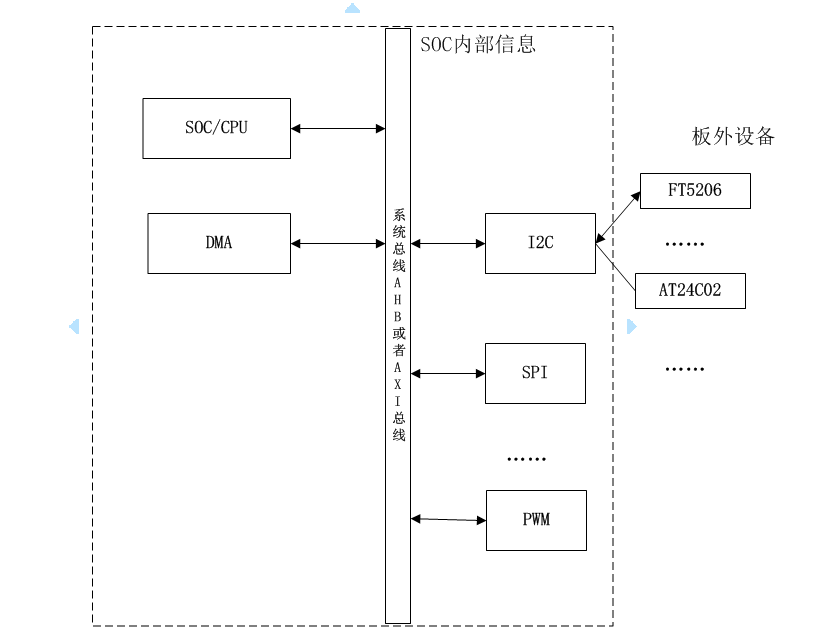
至此，关于LED的驱动开发基本讲解完成，为了能够将知识整理并能够讲解出来，需要去查询书籍，查看内核代码，分析实现的原理，花费了比实现驱动更多的时间，但是这是值得的，通过对驱动流程的整理，对驱动和设备树实现有了更清晰的认知，这是有重要意义的, 不过这也是因为首次接触驱动部分的内容，所以把整个驱动部分进行了比较详细的讲解，后续的驱动将以模块和内核为主，不会过多涉及基础的Makefile和处理接口的实现。

# 设备树的说明

对于整个嵌入式Linux驱动开发中，设备树语法和构建是其中比较核心的部分，是需要比较系统的去学习掌握的。本文参考设备树说明文档<Devicetree Specification Release v0.2>, 在结合日常驱动开发中积累的经验，总结完成的一篇设备树语法的说明。对于驱动的编写来说，设备树语法的了解自然必不可少，但大多数情况下我们模仿厂商的实现，在结合芯片手册就可以增加自己需要的功能，不过如何来添加设备节点，保证添加的有效性，这时候就需要掌握理和解设备树语法。随着设备树逐渐成为嵌入式驱动开发中的主流，并逐渐取代寄存器的访问方式，设备树对于驱动开发越来越重要，**另外如果在本章了解中对于设备树中有疑惑不理解的部分，这很正常，可以先大概浏览下，做到心中有着概念，在带着疑问去学习后续涉及驱动的内容，当你困惑的时候，可以回来系统的理解设备树的语法(不理解不要纠结)，不要在缺少积累的时候去钻牛角尖，这也是嵌入式学习的重要经验。**

## 设备树综述

对SOC构造和嵌入式硬件有了解的话，芯片一般由内核(Cortex-A), 以及通过系统总线(AHB, AXI等)挂载的GPIO，I2C，SPI，PWM，Ethernet等外设模块构成。而对于具体的外设模块，如I2C外设，又支持访问多个器件来满足不同功能的需求。而设备树，就是基于系统总线作为主干，将芯片SOC和各类外设以及外部器件用数据结构的形式组合起来描述硬件结构的文件，是对硬件模型的抽象，总结来说，设备树就是对硬件结构的抽象描述。



上面就是比较常见的基于SOC构建的嵌入式应用系统，包含芯片和外围的设备，虽然总线可能不指一条，外设模块的设备连接情况也会更加复杂，但都没有树结构模型，而设备树也是按照如此模型进行设计的，理解这一点，也可以更深刻的知道设备树的实现思路。

在对嵌入式整个硬件框架有一定了解后，下面还要理解几个名词。

DTS 设备树的源码文件，用于描述设备硬件的具体抽象实现

DTSI 和C语言的头文件类似，基于#inclue语法包含，也是描述设备树的源码文件，另外DTS文件同样可以被包含

DTB 基于DTS源码编译的二进制文件，用于内核调用解析设备树信息的文件。

DTC 用于编译DTS到DTB的工具，由内核编译时使用make dtbs编译设备树二进制文件过程中生成。

基于以上信息，我们理解DTS/DTSI是基于DTS语法实现的设备描述文件，DTB则是用于内核解析，需要下载的文件即可。

## 设备树语法

在上一小节，对设备树的基本概念和定义的来源有了初步认知，接下来更重要的就是DTS语法了，这里就以实际的例子来说明。

### #include语法

DTS中#include语法和C语言中类似，支持将包含的文件内容直接放置在#include位置从而访问到其它文件的数据，如imx6ull的官方设备树文件内的如下实现:

#include <dt-bindings/input/input.h>

#include "imx6ull.dtsi"

另外，也可以用来包含dts文件，如下

#include "imx6ull-14x14-evk.dts"

### 节点描述

对于设备树来说，都是有由根节点开始，在添加不同的设备节点描述的，以比较简单的LED驱动中对应的设备树为例:

/{ //根节点

//......

led { //节点名(子节点) <name>

         compatible = "gpio-led"; //节点属性

         pinctrl-names = "default";

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

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

         status = "okay";

     };

     gpio\_keys: gpio\_keys@0 { //节点名(子节点) <label>:<name>[@<unit\_address>]         compatible = "gpio-keys"; //节点属性

         pinctrl-names = "default";

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

         #address-cells = <1>;

        #size-cells = <0>;

         autorepeat;

         key1@1 { //节点名(子节点) <name>[@<unit\_address>]

             label = "USER-KEY1"; //节点属性 key-value键值对

             linux,code = <114>;

             gpios = <&gpio1 18 GPIO\_ACTIVE\_LOW>;

             gpio-key,wakeup;

         };

    };

}

基于上述的实现，设备树具有如下的特性:

1. 设备树文件都由根节点开始，每个设备只有一个根节点(如果包含多个文件都存在根节点，则根节点会合并)，其它所有设备都作为子节点存在，由节点名和节点属性构成。
2. 节点属性都是由key-value的键值对来描述，并以 ; 结束
3. 节点间可以嵌套形成父子关系，这样可以方便描述设备间的关系
4. 节点名支持<name>[@<unit\_address>]的格式，其中后面的unit\_address可选，一般为设备地址，这是为了用于保证节点是唯一的标识，当然用其它数字也可以。

同时，节点名也支持**<label>:<name>[@<unit\_address>]**的格式, 这里的**label**就是节点的标签别名，我们可以**&<label>**来直接访问节点。如对于gpio\_keys: gpio\_keys@0 可以通过&gpio\_keys来访问gpio\_keys@0，后面我们就将用到这个说明。

1. 在设备树中查找节点需要完整的节点路径，对于项目来说，直接修改官方的dts文件是不推荐的，如果自己建立路径，又过于复杂，因此设备树提供通过标注引用<label>的方式，允许我们在其它文件中修改已存在的节点，或者添加新的节点，对于节点的合并原理，包含以下原则:
2. 不同的属性信息进行合并
3. 相同的属性信息进行覆写

基于这种原则，我们可以通过如下的代码，在已有节点添加更新新的数据，如使用如下代码在gpio\_keys: gpio\_keys@0中增加节点。

&gpio\_key{  
 key2@2{

label=”usr-key2”;

//.....

}

}

上面就是节点相关的信息，下面就开始深入节点内部，讲述节点内部如何基于属性来定义设备的说明。

1. 在驱动中可以同/<node-1>/<node-2>/.../<node-n>的方式访问到指定设备节点

如上面的访问key1@1节点即为

/gpio\_keys@0/key1@1

方式即可访问到指定key1@1节点

### 节点属性

在上一章我们理解了设备树的节点间关系，并讲述了如何添加节点或修改已经存在节点的方法，进一步我们就要抽丝剥茧，讲述属性的说明。

在我们上一章节中，讲到属性是key-value的键值对，这就分两部分讲解设备树的说明。

#### 常见value类型

其中value中常见的几种数据形式如下:

1. 空类型

ranges；

空类型，仅需要键值，用来表示真假类型，

1. 字符串<string>

compatible = "simple-bus";

这里”simple-blus”就是属性中对应的字符串值。

1. 字符串表<stringlist>

compatible = "fsl,sec-v4.0-mon", "syscon", "simple-mfd";

值也可以为字符串列表，中间用，号隔开，这样既可以支持多个字符串的匹配

1. 无符号整数<u32>/<u64>

无符号的证书型变量

offset = <0x38>;

1. 可编码数组<prop-encoded-array>

支持编码的多无符号整数的数组，如reg可以通过#address-cells指定地址单元的数目，#size-cells指定长度单元的数目

reg = <0x020ac000 0x4000>;

可以通过&<label>的方式，即可引用其它节点的数据，用于后续的处理

clocks = <&clks IMX6UL\_CLK\_PLL3\_USB\_OTG>;

#### 常用key属性

1. compatible

<stringlist>字符串列表类型

compatible属性是值是由特定编程模型的一个或多个字符串组成，用于将驱动和设备连接起来，我们在驱动中也是通过compatible来选择设备树中指定的硬件，是非常重要的属性。compatible的格式一般为：

“[<manufacturer>,]<model>”

如

compatible = "arm,cortex-a7";

compatible = "fsl,imx6ul-pxp-v4l2", "fsl,imx6sx-pxp-v4l2", "fsl,imx6sl-pxp-v4l2";

compatible = "gpio-led";

在该模型中，

Manufacturer表示厂商，可选

Model表示指定型号，一般和模块对应的驱动名称一致(当然不一致也不影响实际功能)

在驱动中使用platform\_driver中of\_match\_table里即可使用.compatible用来匹配该对应设备节点，另外匹配时严格的字符串匹配的，所以驱动内的匹配值和设备树中的value要保持一致。

spidev: icm20608@0 {

    compatible = "alientek,icm20608";

    spi-max-frequency = <8000000>;

    reg = <0>;

};

/\* 设备树匹配列表 \*/

static const struct of\_device\_id icm20608\_of\_match[] = {

    { .compatible = "alientek,icm20608" },

    { /\* Sentinel \*/ }

};

/\* SPI驱动结构体 \*/

static struct spi\_driver icm20608\_driver = {

    .probe = icm20608\_probe,

    .remove = icm20608\_remove,

    .driver = {

            .owner = THIS\_MODULE,

            .name = SPI\_ICM\_NAME,

            .of\_match\_table = icm20608\_of\_match,

           },

};

参考上述结构，即可看到通过of\_math\_table指定设备树匹配列表，找到指定的节点去访问。

1. model

<string>字符串类型

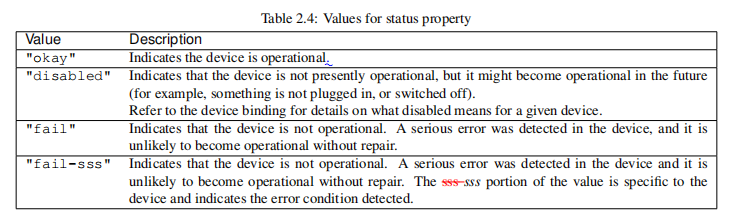
指定设备商信息和模块的具体信息，也有用于模块功能说明，和compatible类似，但仅支持单个字符串模式。

model = "Freescale i.MX6 ULL 14x14 EVK Board";

1. status

<string>字符串类型

指示设备的运行状态，目前支持的状态列表如下:



1. #address-cell和#size-cell

<u32>无符号整型

#address-cells和#size-cells可以用在任何拥有子节点的设备中，用于描述子节点中”reg”对应属性内部值的信息，其中

#address-cells 用来描述字节点中”reg”对应属性中描述**地址列表**中cell数目

#size-cells 用来描述字节点中”reg”对应属性中描述**长度列表**中cell数目

#address-cells和#size-cells属性不是从devicetree的祖先继承的。它们需要明确定义，如果未定义，对于设备树则默认按照地址cell为2个，长度cell为1个去解析reg的值。

soc {

#address-cells = <1>;

#size-cells = <1>;

serial

{

compatible = "ns16550";

reg = <0x4600 0x100>;

clock-frequency = <0>;

interrupts = <0xA 0x8>;

interrupt-parent = <&ipic>;

};

};

如这里reg就要被解析为address-1位，值为0x4600, size-1位，值为0x100。

1. reg

<pro-encode-array> 可编码数组类型

由任意长度的地址和长度构成，描述设备在父设备地址空间中的总线范围，通过#address-cells和#size-cells变量去解析，另外如果#size-cells的长度位0，则reg中后面关于长度的部分应该去除，reg的举例如下:

 #address-cells = <1>; //指定address的范围长度

#size-cells = <0>; //指定size的范围长度

ethphy0: ethernet-phy@0 {

compatible = "ethernet-phy-ieee802.3-c22";

 reg = <0>; //实际reg对应的寄存器地址和范围

};

1. virtual-reg

<u32>

指定设备节点中的reg属性中第一个物理地址的虚拟地址，提供虚拟地址到物理地址的映射关系。

1. ranges

<empty>或者<child-bus-address, parent-bus-address, length>类型

ranges 非空时是一个地址映射/转换表，ranges 属性每个项目由子地址、父地址和地址空间长度

这三部分组成。

**child-bus-address**：子总线地址空间的物理地址

**parent-bus-address：**父总线地址空间的物理地址

**length：**子地址空间的长度

如果 ranges 属性值为空值，说明不需要进行地址转换。

1. name和device\_type

<string>字符串类型

分别表示节点名称或者设备树类型属性，这两个属性在新版本中已经被废弃，这里就不再讨论，可以通过《Devicetree Specifification Release v0.2》的2.3章节查看。

除上述的标准属性外，设备树也支持其它属性如

clock-frequency 指定模块的时钟频率

label 指定可读的标签，用于开发者查看的属性

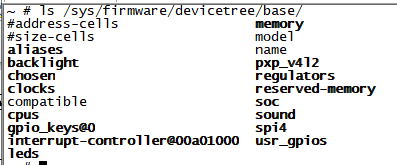
current-speed: 串口的波特率

## 设备树在驱动中应用

上面描述的基本都是设备树语法的部分，不过对于驱动来说，如何从设备树中提取有效的设备信息，从而在驱动中脱离对硬件寄存器的直接访问，如何把设备树用于嵌入式驱动开发中，这部分内容也相当重要，对于嵌入式Linux设备，语法树是在/sys/firmware/devicetree下，可使用

ls /sys/firmware/devicetree/base/

来查看当前根节点下的设备树文件，如下：



对于驱动来说，可以使用内核提供访问设备树的函数或驱动框架中用于匹配节点的接口来访问设备树，下面以GPIO和SPI为例阐述这两种不同的模式。

### 内核设备树访问函数

内核访问设备树的函数主要包含获取节点的函数和获取节点内部属性的函数，这些函数都定义在内核include/linux/of.h中

//根据节点路径获取设备节点信息

struct device\_node \*of\_find\_node\_by\_path(const char \*path)

struct device\_node \*of\_find\_node\_opts\_by\_path(const char \*path, const char \*\*opts)

//根据设备属性获取设备节点信息

struct device\_node \*of\_find\_node\_by\_name(struct device\_node \*from, const char \*name)

struct device\_node \*of\_find\_node\_by\_type(struct device\_node \*from, const char \*type)

struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from, const char \*type, const char \*compat)

//根据匹配的of\_device\_id表格获取node节点(在框架中常用的匹配方式)

static inline struct device\_node \*of\_find\_matching\_node\_and\_match(

    struct device\_node \*from,

    const struct of\_device\_id \*matches,

    const struct of\_device\_id \*\*match)

根据这些信息，我们就可以实现如下代码，找到设备树内的指定节点，代码如下:

    /\*根据路径获取设备节点, 和节点内部属性信息\*/

    nd = of\_find\_node\_by\_path("/usr\_gpios/beep");

    if(nd == NULL){

        printk(KERN\_INFO"node find by path failed!\n");

        return -EINVAL;

    }

    else{

        printk(KERN\_INFO"node find by path success!\n");

}

  /\*根据compatible属性查询节点\*/

    nd = of\_find\_compatible\_node(NULL, NULL, "gpio-key");

    if(nd == NULL){

        printk(KERN\_INFO"node find by compatible failed!\n");

        return -EINVAL;

    }

    else{

        printk(KERN\_INFO"node find by compatible success!\n");

}

     /\*根据匹配表格获取节点，并获取属性\*/

     static const struct of\_device\_id key\_of\_match[] = {

            { .compatible = "gpio-key" },

            { /\* Sentinel \*/ }

        };

        nd = of\_find\_matching\_node\_and\_match(NULL, key\_of\_match, NULL);

        if(nd == NULL){

            printk(KERN\_INFO"node find by of\_device\_id failed!\n");

            return -EINVAL;

        }

        else{

            printk(KERN\_INFO"node find by of\_device\_id success!\n");

        }

在获取设备节点后，我们可以通过内核提供的接口对节点内的key-value键值对进一步读取，具体接口如下:

//提取通用属性的接口

struct property \*of\_find\_property(const struct device\_node \*np, const char \*name, int \*lenp);

int of\_property\_read\_u32\_index(const struct device\_node \*np, const char \*propname, u32 index, u32 \*out\_value);

int of\_property\_read\_string(struct device\_node \*np, const char \*propname, const char \*\*out\_string);

//用于

在了解这些代码后，就可以实现如下代码访问设备树内的参数属性，具体如下

        proper = of\_find\_property(nd, "name", NULL);

        if(proper != NULL)

            printk(KERN\_INFO"%s:%s\n", proper->name, (char \*)proper->value);

        ret = of\_property\_read\_string(nd, "status", &pStr);

        if(pStr != NULL)

            printk(KERN\_INFO"status:%s\n", pStr);

而在部分框架中，也对上述接口进一步封装，如platform\_device\_driver中需要提供的of\_device\_id就是更进一步的调用接口，通过

static const struct of\_device\_id key\_of\_match[] = {

{ .compatible = "usr-gpios" },

{ /\* Sentinel \*/ }

};

结构，也能实现对设备树的匹配，这在很多驱动框架中都是十分常用的，需要在实践中总结理解。

## 总结

至此，对设备树的基本语法进行了基本的讲解，当然这里面还有很多不完善的地方，如对中断控制器和中断相关的语法目前尚未说明，没有讲述驱动中如何调用设备树的部分，另外很多部分的理解受水平限制有遗漏或者错误的地方，如果有发现，希望能够反馈，这里先说声谢谢了。这些知识在实际产品的驱动开发中，理解了这些还是不够的，日常打交道更多的是芯片厂商或者方案商定义的具有特定功能的自定义属性键值对。不过理解了设备树语法的原理，反过来去理解这些自定义属性，也会比较清晰明了，原理仍然是相通的。这篇文章只能算是对设备树语法的入门指引，如果希望深入去掌握嵌入式驱动开发，还是配合着实际产品的硬件框架，在实际任务的维护或者修改设备树，再结合参考资料中提到的文档和本文的说明，带着目的去学习，才是高效且快速的方式。另外，如果感觉本文对你有帮助，记得点赞！这也能给我更大的动力花时间去总结这些经验。

# Uart通讯和协议实现

在理解了嵌入式驱动概念和设备树后，本节就开始应用的实现，对于下位机来说具体实现包含三部分内容，Uart驱动，Uart应用以及基于Uart的完整通讯协议的实现。

对于Uart驱动，因为使用默认Linux提供驱动框架，且我们使用到的Uart已经集成在内核中，所以初期不建议去深入了解这部分，Uart框架已经属于在嵌入式驱动中十分复杂的模块，初期理解起来困难，后续等对内核框架有更深入的理解后在反过来学习，至此，实现的内容就简化成两部分，Uart通讯的实现，并基于通讯构建协议的实现。

## Uart通讯实现

对于Uart通讯来说，主要内容包含以下两点：

1. Uart硬件模块的配置，支持波特率，数据位，奇偶校验位，停止位，开关和读写接口的实现。
2. 基于Uart的串口简单通讯实现。

对于Uart硬件的硬件配置，按照后续设计的需求，需要满足波特率，数据位，奇偶校验位，停止位的可配，基于这些需求，就需要两个接口**tcgetattr， tcsetattr**来配置Uart属性，具体代码如下:

static int set\_opt(int nFd, int nBaud, int nDataBits, std::string cParity, int nStopBits)

{

    struct termios newtio;

    struct termios oldtio;

tcgetattr(nFd, &oldtio);

bzero(&newtio, sizeof(newtio));

//接收模式

    newtio.c\_cflag |= (CLOCAL|CREAD);

newtio.c\_cflag &= ~CSIZE;

//设置数据位

    switch(nDataBits){

        case 7:

            newtio.c\_cflag |= CS7;

            break;

        case 8:

            newtio.c\_cflag |= CS8;

            break;

        default:

            break;

    }

//设置奇偶校验位

    switch(cParity[0]){

        case 'O':

        case 'o':

            newtio.c\_cflag |= PARENB;

            newtio.c\_cflag |= PARODD;

            newtio.c\_iflag |= (INPCK | ISTRIP);

            break;

        case 'E':

        case 'e':

            newtio.c\_iflag |= (INPCK | ISTRIP);

            newtio.c\_cflag |= PARENB;

            newtio.c\_cflag &= ~PARODD;

            break;

        case 'N':

        case 'n':

            newtio.c\_cflag &= ~PARENB;

            break;

    }

//设置波特率

    switch(nBaud){

        case 9600:

            cfsetispeed(&newtio, B9600);

            cfsetospeed(&newtio, B9600);

            break;

        case 115200:

            cfsetispeed(&newtio, B115200);

            cfsetospeed(&newtio, B115200);

            break;

        default:

            cfsetispeed(&newtio, B9600);

            cfsetospeed(&newtio, B9600);

            break;

    }

    //设置停止位

    if(nStopBits == 1){

        newtio.c\_cflag &=  ~CSTOPB;

    }

    else if (nStopBits == 2){

        newtio.c\_cflag |=  CSTOPB;

}

//最小等待数和最小等待时间，表示不等待

    newtio.c\_cc[VTIME]  = 0;

    newtio.c\_cc[VMIN] = 0;

    tcflush(nFd, TCIFLUSH);

    if((tcsetattr(nFd, TCSANOW,&newtio))!=0){

        return -1;

    }

    return 0;

}

基于上述代码，即可以实现对串口的配置，下面实现基于串口的简单通讯测试代码。

int main(int argc, char \*argv[])

{

    int fd;

    ssize\_t nLen;

    /\*开启串口模块\*/

    fd = open(RS232\_DRIVER\_NAME, O\_RDWR|O\_NOCTTY|O\_NDELAY);

if(fd >0){

        if(set\_opt(fd, opt) != 0){

            do {

                /\*从串口缓冲区中读出数据，并写入到发送缓冲区\*/

                nLen = read(fd, nCacheBuffer, UART\_BUFFER\_SIZE);

                if(nLen > 0) {

                    write(fd, nCacheBuffer, nLen);

                }

            } while (1);

        }

    }

    close(fd);

    return 0;

}

通过open, read, write和close接口，即实现了基本的串口通讯应用，后续我们即可在这套应用的基础上构建协议通讯。

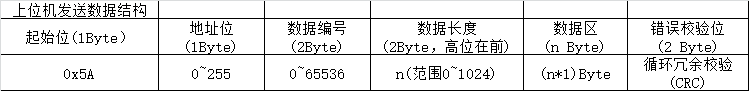
## 通讯协议的制定

通讯协议可以理解为约束多设备通讯的一套规则，像Modbus，TCP/IP, BLE都是在嵌入式开发中常用的协议。不过协议落实到实现后，就可以理解为对数据的结构化处理，对于嵌入式Linux来说，协议的功能也是如此，使用现成的协议如Modbus构建应用当然可以，但构建一套私有的自定义协议，并基于此构建应用，也是嵌入式开发中的常用方式，而且通过私有协议的制定和实现，也更容易理解协议的本质，这也是我选择私有协议的主要原因。协议的制定并不困难，定义数据结构，用于确保数据的完整性，可靠性即可，如果更深入些，就要考虑数据的安全性，基于这些需求，就可以进行协议的制定了。

### 协议制定

协议的制定在大致的数据发送和返回数据结构上大致如下:

1. 上位机发送指令包含**起始位**, **地址位**(用于多机通讯), **数据长度**(指示内部后面的数据长度), **数据**, **CRC校验位**等基础结构，在这基础上增加了**数据编号位**，它是2字节的随机数，在处理完成后可以用于上位机验证返回的数据是否为同一数据包



确认了通讯的结构后，下位机代码就可以实现了，其中接收数据代码如下:

int protocol\_info::check\_receive\_data(int fd){

int nread;

int CrcRecv, CrcCacl;

struct req\_frame \*frame\_ptr;

/\*从设备中读取数据\*/

nread = this->device\_read(fd, &this->rx\_ptr[this->rx\_size],

(this->max\_buf\_size-this->rx\_size));

if(nread > 0){

this->rx\_size += nread;

frame\_ptr = (struct req\_frame \*)this->rx\_ptr;

/\*接收到头不符合预期\*/

if(frame\_ptr->head != PROTOCOL\_REQ\_HEAD) {

USR\_DEBUG("No Valid Head\n");

this->rx\_size = 0;

return RT\_FAIL;

}

/\*已经接收到长度数据\*/

else if(this->rx\_size > 5){

int nLen;

/\*设备ID检测\*/

if(frame\_ptr->id != PROTOCOL\_DEVICE\_ID){

this->rx\_size = 0;

USR\_DEBUG("Valid ID\n");

return RT\_FAIL;

}

/\*获取接收数据的总长度\*/

this->rx\_data\_size = LENGTH\_CONVERT(frame\_ptr->length);

/\*crc冗余校验\*/

nLen = this->rx\_data\_size+FRAME\_HEAD\_SIZE+CRC\_SIZE;

if(this->rx\_size >= nLen){

/\*计算head后到CRC尾之前的所有数据的CRC值\*/

CrcRecv = (this->rx\_ptr[nLen-2]<<8) + this->rx\_ptr[nLen-1];

CrcCacl = this->crc\_calculate(&this->rx\_ptr[1], nLen-CRC\_SIZE-1);

if(CrcRecv == CrcCacl){

this->packet\_id = LENGTH\_CONVERT(frame\_ptr->packet\_id);

return RT\_OK;

}

else{

this->rx\_size = 0;

USR\_DEBUG("CRC Check ERROR!. rx\_data:%d, r:%d, c:%d\n", this->rx\_data\_size, CrcRecv, CrcCacl);

return RT\_FAIL;

}

}

}

}

return RT\_EMPTY;

}

1. 下位机返回指令也是**起始位**，**地址位**，**ACK应答状态**，**数据长度**，**数据**和**CRC校验位**，同样也包含编号用于上位机的校验。



确认了通讯的结构后，下位机代码就可以实现了，其中生成发送指令代码的结构如下:

int protocol\_info::create\_send\_buf(uint8\_t ack, uint16\_t size, uint8\_t \*pdata)

{

uint8\_t out\_size, index;

uint16\_t crc\_calc;

out\_size = 0;

this->tx\_ptr[out\_size++] = PROTOCOL\_ACK\_HEAD;

this->tx\_ptr[out\_size++] = PROTOCOL\_DEVICE\_ID;

this->tx\_ptr[out\_size++] = (uint8\_t)(this->packet\_id>>8);

this->tx\_ptr[out\_size++] = (uint8\_t)(this->packet\_id&0xff);

this->tx\_ptr[out\_size++] = ack;

this->tx\_ptr[out\_size++] = (uint8\_t)(size>>8);

this->tx\_ptr[out\_size++] = (uint8\_t)(size&0xff);

if(size != 0 && pdata != NULL)

{

for(index=0; index<size; index++)

{

this->tx\_ptr[out\_size++] = \*(pdata+index);

}

}

crc\_calc = this->crc\_calculate(&this->tx\_ptr[1], out\_size-1);

this->tx\_ptr[out\_size++] = (uint8\_t)(crc\_calc>>8);

this->tx\_ptr[out\_size++] = (uint8\_t)(crc\_calc&0xff);

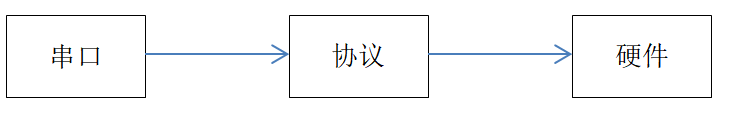
return out\_size;

}

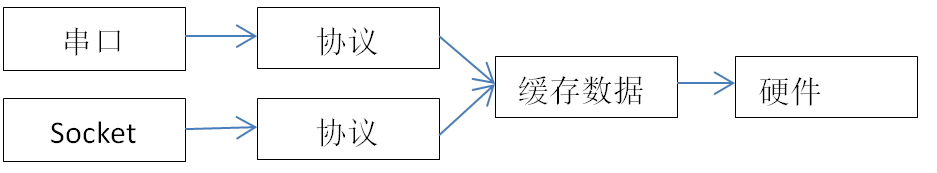
因为是嵌入式Linux开发，因此推荐使用C++, 封装可以让代码结构更加清晰，从代码的实现可以看到实现包含:硬件的数据接收，起始位检测，数据编号的获取，以及后续数据的接收和数据的CRC校验，至于发送数据，则主要是创建发送数据的接口，这部分即为通讯相关的结构数据实现，通过协议的发送和接收结构的剥离，此时我们就获得了需要处理的实际数据，下面进一步进行数据的处理。

### 数据处理

　在之前的协议设计中，指令是包含在上述数据结构中的，到具体执行的地方直接操作硬件，对于串口操作LED，流程如下:



　　在整个流程中，协议和串口，以及硬件绑定，这在多任务处理时，对于硬件的同步处理就比较困难，而且硬件的处理也是十分耗时的，特别是对于很多时候也影响通讯的效率，记得在操作系统的学习中，有特别经典的一句话，解耦的通常方法就是增加中间层，在本项目也是如此，在协议和硬件中增加缓冲数据层，这样同步问题都在缓冲数据层处理，就避免了对硬件的资源抢占动作，修改后结构如下:



　　为了实现这个结构，就增加对于缓存数据的处理，其中缓存数据的处理结构如下:

class app\_reg

{

public:

app\_reg(void);

~app\_reg();

int hardware\_refresh(void); /\*硬件的实际更新\*/

uint16\_t get\_multiple\_val(uint16\_t reg\_index, uint16\_t size, uint8\_t \*pstart); /\*获取寄存器的值\*/

void set\_multiple\_val(uint16\_t reg\_index, uint16\_t size, uint8\_t \*pstart); /\*设置寄存器的值\*/

int diff\_modify\_reg(uint16\_t reg\_index, uint16\_t size, uint8\_t \*pstart, uint8\_t \*psrc);

private:

uint8\_t reg[REG\_NUM];

pthread\_mutex\_t reg\_mutex; /\*数据读取都要执行该锁\*/

};

其中hardware\_refresh就是实际对硬件的操作，其它协议通关get和set即可修改缓存数据，在协议中操作修改内部缓存数据就可以了，剩余硬件相关处理就由缓存数据管理，其中协议中的执行如下:

int protocol\_info::execute\_command(int fd){

uint8\_t cmd;

uint16\_t reg\_index, size;

uint8\_t \*cache\_ptr;

app\_reg \*app\_reg\_ptr;

cmd = this->rx\_data\_ptr[0];

reg\_index = this->rx\_data\_ptr[1]<<8 | this->rx\_data\_ptr[2];

size = this->rx\_data\_ptr[3]<<8 | this->rx\_data\_ptr[4];

cache\_ptr = (uint8\_t \*)malloc(this->max\_buf\_size);

this->tx\_size = 0;

app\_reg\_ptr = get\_app\_reg();

switch (cmd){

case CMD\_REG\_READ:

app\_reg\_ptr->get\_multiple\_val(reg\_index, size, cache\_ptr);

this->tx\_size = this->create\_send\_buf(ACK\_OK, size, cache\_ptr);

break;

case CMD\_REG\_WRITE:

memcpy(cache\_ptr, &this->rx\_data\_ptr[5], size);

app\_reg\_ptr->set\_multiple\_val(reg\_index, size, cache\_ptr);

this->tx\_size = this->create\_send\_buf(ACK\_OK, 0, NULL);

break;

case CMD\_UPLOAD\_CMD:

break;

case CMD\_UPLOAD\_DATA:

break;

default:

break;

}

free(cache\_ptr);

/\*发送数据，并清空接收数据\*/

this->rx\_size = 0;

this->device\_write(fd, this->tx\_ptr, this->tx\_size);

return RT\_OK;

}

对于硬件的处理则由数据层管理，结构如下:

int app\_reg::hardware\_refresh(void){

uint8\_t \*reg\_ptr;

uint8\_t \*reg\_cache\_ptr;

uint8\_t is\_reg\_modify;

uint16\_t reg\_set\_status;

reg\_ptr = (uint8\_t \*)malloc(REG\_CONFIG\_NUM);

reg\_cache\_ptr = (uint8\_t \*)malloc(REG\_CONFIG\_NUM);

is\_reg\_modify = 0;

if(reg\_ptr != NULL && reg\_cache\_ptr != NULL)

{

/\*读取所有的寄存值并复制到缓存中\*/

this->get\_multiple\_val(0, REG\_CONFIG\_NUM, reg\_ptr);

memcpy(reg\_cache\_ptr, reg\_ptr, REG\_CONFIG\_NUM);

/\*有设置消息\*/

reg\_set\_status = reg\_ptr[1] <<8 | reg\_ptr[0];

if(reg\_set\_status&0x01)

{

/\*LED设置处理\*/

if(reg\_set\_status&(1<<1))

{

led\_convert(reg\_ptr[2]&0x01);

}

/\*修改beep\*/

if(reg\_set\_status&(1<<2))

{

beep\_convert((reg\_ptr[2]>>1)&0x01);

}

reg\_ptr[0] = 0;

reg\_ptr[1] = 0;

is\_reg\_modify = 1;

}

/\*更新寄存器状态\*/

if(is\_reg\_modify == 1){

if(this->diff\_modify\_reg(0, REG\_CONFIG\_NUM, reg\_ptr, reg\_cache\_ptr) == RT\_OK){

is\_reg\_modify = 0;

}

else{

free(reg\_ptr);

free(reg\_cache\_ptr);

USR\_DEBUG("modify by other interface\n");

return RT\_FAIL;

}

}

free(reg\_ptr);

free(reg\_cache\_ptr);

}

else{

USR\_DEBUG("malloc error\n");

}

return RT\_OK;

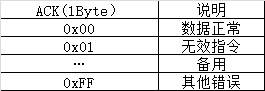
}

这里就将对硬件的处理，就准换成了对内部缓存数据的处理，缓存数据由专用的线程管理，执行对硬件的操作。

定义发送的实际指令如下

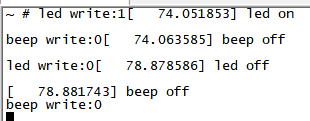


返回应答数据格式则为



至此，我们就完成了协议层的操作，这时我们就可以通过串口操作硬件，且提供了多线程兼容的支持，在实现上述协议接口后，在结合上一章节的串口驱动和串口操作，就可以实现完整的功能。

再结合上面的Uart应用的实现，既可以已实现通过二进制数据控制硬件的操作，结果如下:



## 总结

在本章节中，我们了解了串口的应用，并基于串口的基础上构建了一套支持数据完整性和有效性的私有协议实现，同时考虑到可扩展性，进行了对底层硬件操作和协议本身的解耦，支持多种接口如(TCP, UDP或者CAN)等通过本协议完成对硬件的访问，不过对于完整的项目来说，这还仅仅是其中很小的一部分，后续需要更多的模块完善，继续加油进步吧。

# QT界面开发和通讯实现

在上一章我们实现了下位机的协议制定，并通过串口通讯工具完成了对设备内外设(LED)的状态修改，下面就要进行上位机软件的实现了(事实上这部分不属于嵌入式Linux的内容，所以只在本章节讲述下上位机实现的流程和思路,后续维护更新不在进行详细说明，不过下位机界面实现肯定还会涉及这些技术),上位机的界面方案一般指在Windows平台的软件界面开发，如UWP，WINFORM/C#, WPF/C#， QT/C++等，如果说我的个人倾向的话，当然更喜欢的WINFORM/C#技术，一方面C#相对于C++更简单，不会因为复杂的模板和继承机制，导致出问题的报错比代码都长，另一方面网上的资料也多，遇到问题很容易找到解决办法，在我之前实现的应用中，也都是使用WINFORM技术，再加上对于QT/C++根本没有了解过，算是第一次接触(之前接触的都是无界面应用或者使用的Android/Java)，不过对于嵌入式Linux来说，QT/C++是也是十分需要掌握的，既然都要学习，那么上位机选择QT/C++先来熟悉语法和基础，完成上位机QT界面和通讯协议的实现，这也是这篇文章耽误一段时间的原因，在QT还没有熟悉之前，参考例程写应用还可以,清晰的讲清楚还是很困难的，在应用接近大半个月后，也算有些心得，可以进行后续的进度了，下面开始本节的实现吧。

## 参考资料

1. [开源QT例程项目](https://github.com/feiyangqingyun/QWidgetDemo" \t "_blank)
2. 《QT5开发和实例》 -- 参考这本书不是因为写的有深度，而是因为里面全是例程，适合初学者了解
3. 《C++ Primer Plus》

## QT界面布局实现

基于从Winform的界面开发经验，QT界面的布局也是类似，参考上面的例程项目，主要涉及的的窗体有:

QFrame:基本控件的基类，用于将功能类似的结构整理在一起

QLabel:标签控件，用于显示文字说明

QPushButton:按键控件，执行按键动作

QTextEdit:编辑文本框控件，用于输入或者显示文本

QComboBox:选择框控件，支持下拉菜单的选择

QLineEdit:行编辑框,用于行输入和显示文本

在掌握基础的基本的布局编辑框后，就可以使用设计栏左边的控件框中，拖出如下的编辑框。



在构建完成上述编辑框后，下面就要实现界面内容的填充，主要包含页面布局的显示，下拉框的完善，代码如下:

//添加COM口

QStringList comList;

for (int i = 1; i <= 20; i++) {

comList << QString("COM%1").arg(i);

}

ui->combo\_box\_com->addItems(comList);

//波特率选项

QStringList BaudList;

BaudList <<"9600"<<"38400"<<"76800"<<"115200"<<"230400";

ui->combo\_box\_baud->addItems(BaudList);

ui->combo\_box\_baud->setCurrentIndex(3);

//数据位选项

QStringList dataBitsList;

dataBitsList <<"6" << "7" << "8"<<"9";

ui->combo\_box\_data->addItems(dataBitsList);

ui->combo\_box\_data->setCurrentIndex(2);

//停止位选项

QStringList StopBitsList;

StopBitsList<<"1"<<"2";

ui->combo\_box\_stop->addItems(StopBitsList);

ui->combo\_box\_stop->setCurrentIndex(0);

//校验位

QStringList ParityList;

ParityList<<"N"<<"Odd"<<"Even";

ui->combo\_box\_parity->addItems(ParityList);

ui->combo\_box\_parity->setCurrentIndex(0);

//设置协议类型

QStringList SocketTypeList;

SocketTypeList<<"TCP"<<"UDP";

ui->combo\_box\_socket\_type->addItems(SocketTypeList);

ui->combo\_box\_parity->setCurrentIndex(0);

// //正则限制部分输入需要为数据

// QRegExp regx("[0-9]+$");

// QValidator \*validator\_time = new QRegExpValidator(regx, ui->line\_edit\_time);

// ui->line\_edit\_time->setValidator( validator\_time );

// QValidator \*validator\_id = new QRegExpValidator(regx, ui->line\_edit\_dev\_id);

// ui->line\_edit\_dev\_id->setValidator( validator\_id );

//默认按键配置不可操作

init\_btn\_disable(ui);

ui->btn\_uart\_close->setDisabled(true);

ui->btn\_socket\_close->setDisabled(true);

至此，我们就完成了布局相关的代码。

## 数据处理逻辑

对于无界面的软件或者方案实现，我们主要关注的是**数据在整个逻辑模型之间的流通，转移和处理**，对于有界面的软件实现，其实这套逻辑也是存在的。除了涉及界面的处理，其它部分其实也是这套逻辑，不过是将部分数据的源头来自于界面的动作，并且将最后的输出结果从命令行转移到界面的窗口中，如果理解了这一点，就会发现其实带界面的应用实现并没有太困难，这也是我接触QT/C++很短时间就能将Winform和下位机经验快速转换的原因。对于这个项目来说，主要实现的背后数据逻辑包含以下三个方面:

1. 按键动作的信号和界面的输入信息处理
2. 硬件通讯相关的串口知识，socket通讯以及涉及的TCP和UDP协议传输
3. 协议相关的硬件实现和数据处理
4. 处理结果的界面输出显示

其中按键部分的动作和界面输出显示都是QT界面背后的逻辑，包含信号槽的绑定和界面变量的操作方法，如下所示

//获取设备ID信息

pMainUartProtocolThreadInfo->SetId(ui->line\_edit\_dev\_id->text().toShort());

//界面显示的操作

if(ui->text\_edit\_test->document()->lineCount() > 20)

{

qDebug()<<"lines do";

ui->text\_edit\_test->setText(s);

}

else

{

ui->text\_edit\_test->append(s);

}

这部分是涉及QT的基础知识，主要都是积累的技巧，难度不高，建议参考《QT5开发和实例》实例去学习。

硬件串口知识和Socket知识就是应用实现需要的其它能力，包含对QextSerialPort和Socket接口的应用，此外为了满足多接口应用同时操作的需求，需要实现多线程的编程，其中串口的应用初始化配置主要包含的有

flush:清空缓存区

setBaudRate:设置波特率

setDataBits:设置数据位

setParity:设置奇偶校验位

setStopBits:设置停止位

setFlowControl:设置流量控制

setTimeout:设置接收和发送超时时间

pMainUartProtocolThreadInfo->m\_pSerialPortCom = new QextSerialPort(ui->combo\_box\_com->currentText(), QextSerialPort::Polling);

pMainUartProtocolThreadInfo->m\_bComStatus = pMainUartProtocolThreadInfo->m\_pSerialPortCom->open(QIODevice::ReadWrite);

if(pMainUartProtocolThreadInfo->m\_bComStatus)

{

//清除缓存区

pMainUartProtocolThreadInfo->m\_pSerialPortCom ->flush();

//设置波特率

pMainUartProtocolThreadInfo->m\_pSerialPortCom ->setBaudRate((BaudRateType)ui->combo\_box\_baud->currentText().toInt());

//设置数据位

pMainUartProtocolThreadInfo->m\_pSerialPortCom->setDataBits((DataBitsType)ui->combo\_box\_data->currentText().toInt());

//设置校验位

pMainUartProtocolThreadInfo->m\_pSerialPortCom->setParity((ParityType)ui->combo\_box\_parity->currentText().toInt());

//设置停止位

pMainUartProtocolThreadInfo->m\_pSerialPortCom->setStopBits((StopBitsType)ui->combo\_box\_stop->currentText().toInt());

pMainUartProtocolThreadInfo->m\_pSerialPortCom->setFlowControl(FLOW\_OFF);

pMainUartProtocolThreadInfo->m\_pSerialPortCom ->setTimeout(10);

init\_btn\_enable(ui);

pMainUartProtocolThreadInfo->SetId(ui->line\_edit\_dev\_id->text().toShort());

ui->btn\_uart\_close->setEnabled(true);

ui->btn\_uart\_open->setDisabled(true);

ui->btn\_socket\_open->setDisabled(true);

ui->btn\_socket\_close->setDisabled(true);

ui->combo\_box\_com->setDisabled(true);

ui->combo\_box\_baud->setDisabled(true);

ui->combo\_box\_data->setDisabled(true);

ui->combo\_box\_stop->setDisabled(true);

ui->combo\_box\_parity->setDisabled(true);

append\_text\_edit\_test(QString::fromLocal8Bit("serial open success!"));

protocol\_flag = PROTOCOL\_UART;

}

else

{

pMainUartProtocolThreadInfo->m\_pSerialPortCom->deleteLater();

pMainUartProtocolThreadInfo->m\_bComStatus = false;

append\_text\_edit\_test(QString::fromLocal8Bit("serial open failed!"));

}

串口的通讯读写接口主要包含

Write:数据发送接口

Read:数据读取接口

//设备写数据

int CUartProtocolThreadInfo::DeviceWrite(uint8\_t \*pStart, uint16\_t nSize)

{

m\_pSerialPortCom->write((char \*)pStart, nSize);

return nSize;

}

//设备读数据

int CUartProtocolThreadInfo::DeviceRead(uint8\_t \*pStart, uint16\_t nMaxSize)

{

return m\_pSerialPortCom->read((char \*)pStart, nMaxSize);

}

socket通讯的的初始化配置包含

abort:中断当前的所有连接

　　connectToHost:指定连接到指定的IP地址和端口

　　waitForConnect:等待服务器的连接

　　waitForBytesWritten:等待数据发送完成

　　waitForReadyRead:等待数据可以接收

　　此外，还包含和Socket通讯相关的

　　信号:connect <-> 槽函数:slotConnected

　　信号:diconnect <-> 槽函数:slotDisConnected

　　信号:readyRead <-> 槽函数:dataReceived

　　具体代码实现如下:

void CTcpSocketThreadInfo::run()

{

bool is\_connect;

int nLen;

int nStatus;

m\_pTcpSocket = new QTcpSocket();

m\_pServerIp = new QHostAddress();

connect(m\_pTcpSocket, SIGNAL(connected()), this, SLOT(slotConnected()));

connect(m\_pTcpSocket, SIGNAL(disconnected()), this, SLOT(slotDisconnected()));

connect(m\_pTcpSocket, SIGNAL(readyRead()), this, SLOT(dataReceived()));

for(;;)

{

if(m\_nIsStop)

return;

nStatus = m\_pQueue->QueuePend(&SendBufferInfo);

if(nStatus == QUEUE\_INFO\_OK)

{

m\_pTcpSocket->abort();

m\_pTcpSocket->connectToHost(\*m\_pServerIp, m\_nPort);

nLen = this->CreateSendBuffer(this->GetId(), SendBufferInfo.m\_nSize,

SendBufferInfo.m\_pBuffer, SendBufferInfo.m\_IsWriteThrough);

is\_connect = m\_pTcpSocket->waitForConnected(300);

if(is\_connect)

{

emit send\_edit\_test(QString("socket client ok"));

this->DeviceWrite(tx\_buffer, nLen);

//通知主线程更新窗口

emit send\_edit\_test(byteArrayToHexString("Sendbuf:", tx\_buffer, nLen, "\n"));

//等待发送和接收完成

m\_pTcpSocket->waitForBytesWritten();

m\_pTcpSocket->waitForReadyRead();

}

else

{

emit send\_edit\_test(QString("socket client fail\n"));

}

qDebug()<<"thread queue test OK\n";

}

}

}

完成上述接口应用的实现，后续的逻辑就是涉及协议实现的部分，这部分的实现与协议相关的章节实现一致，具体如下，包含

CreateSendBuffer:生成发送数据

DeviceRead:数据接收

DeviceWrite:数据发送

CheckReceiveData:接收数据，并校验

ExecuteCommand:执行指令的处理

如此就完整实现了整个数据逻辑的框架，完成了从按键数据发送触发，协议数据发送和接收处理，接收界面显示的完整流程，最后实现如图所示的功能:



## 总结

至此，我们对于QT上位机界面的基本应用框架已经实现完毕，后续就是在该平台的基础上构建新的接口实现，满足不同应用的需求，因为本身这个系列是学习嵌入式Linux开发的，虽然后续肯定会在这份代码的基础上去完善上位机的应用，但对于上位机的说明目前就到此为止了(毕竟本系列的实现并非嵌入式开发)，当然这并不是说对QT的讲述结束了，因为后续下位机的开发仍然会涉及QT的开发。在整个上位机应用的开发中，我对曾经学到的类的继承和派生，模板，lambda表达式都有了进一步的实践和应用，加深了相关的理解，触类旁通对于软件开发来说某些情况下也是正常的，所以说不要局限自己的视野，多学多练才是成功的唯一之道。

# I2C驱动和设备访问

讲过前面的流程，已经通过QT实现了桌面端界面化的访问和配置开发板中的数据，在本节中，将通过I2C驱动的实现和应用，来扩展上位机对底层设备的访问。本文将不仅仅实现I2C的设备驱动，也将系统的阐述内核中为了实现I2C通讯协议，用于抽象，管理I2C资源的I2C总线框架，这也是Linux设备模型的精髓，对于驱动开发，我们将会经常和这部分打交道，另外因为I2C驱动相比较于TTY，CMOS，以太网驱动会简单些，这里就以I2C总线框作为起始，逐渐拨开内核中设备模型的知识迷雾。

## 硬件接口和协议

### 协议说明

在理解Linux中的I2C协议之前，需要先对I2C协议进行简单的回顾，如果有单片机开发经验，对于这部分并不陌生，I2C为两线的半双工通讯协议，对于SOC来说一般配置为开漏模式，由外部上拉电阻来确保高电平的输出，具体通讯的结构见图 9‑1说明。

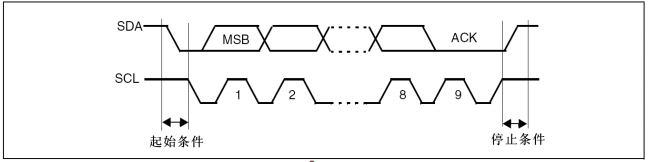


图 9‑1 I2C通讯时序图

每次通讯以SDA拉低作为start，在每个8bit数据发送完成后, 第9个发送clk时钟作为从机响应ack的检测，SDA线需要为低电平来表示**从机正确的响应ack**，重复数据发送流程，最终以stop信号作为一次通讯的截止，下面I2C访问一款OLED设备的示波器截图，可以清晰的反映该特征，详细见图 9‑2说明。

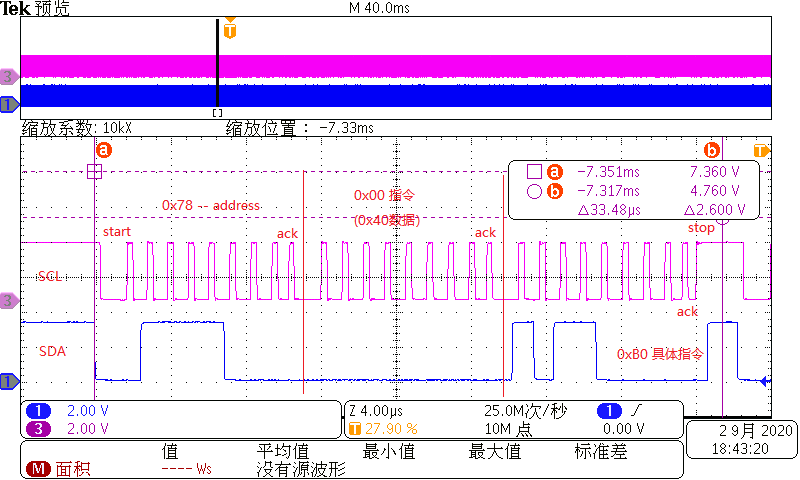


图 9‑2 I2C通讯实际波形图

从这图我们也可以看出，I2C通讯的知识点包含:

1. I2C硬件是双线的半双工协议，分为时钟线和数据线两部分
2. I2C通讯从start开始，以stop结束，每个数据都以ack结尾
3. I2C支持多字节收发通讯

当然这里主要波形是写时序，结构如下

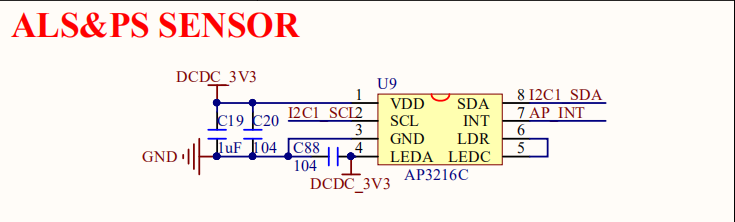
起始位 | 设备地址(7位)+写指令(0) | 寄存器地址 | 数据 | ...... | 数据 | 停止位

对于读指令，则稍微复杂些，需要先指定读取数据的地址，在进行读取操作，具体如下

起始位 | 设备地址(7位)+写指令(0) | 寄存器地址 | 设备地址(7位)+读指令(1) | 数据 | ...... | 数据 | 停止位

### 硬件接口

上述就是常见通用I2C发送和读取数据，和下位机通讯的主要方式，而本节所访问的外部设备ap3216c，也满足该协议架构，其中硬件连接方式如图所示:



C:\Users\zhangchao\AppData\Roaming\Tencent\Users\1107473010\QQ\WinTemp\RichOle\(U@0RB$ZO0DX1Y6L}1QE(6W.png

如上图所示，硬件的连接如下，ap3216c与I2C1相连，对应的硬件GPIO口为GPIO1\_IO28和GPIO1\_IO29, 基于上述信息，可以进行设备树相关的修改，不过后续我们结合实际的框架，讲述从设备树，驱动到测试示例的实现。

## I2C总线框架

### 硬件拓扑和设备树

虽然在前面我们已经接触过TTY设备框架，不过因为代码比较复杂，就直接采用内核里的实现，直接进行应用的开发，作为知识难点，自然要攻破，I2C总线框架的实现相对简单，比较适合去了解设备框架模型，下面开始正式的学习，结合上章的知识，可以整理出I2C的硬件结构如下所示。



图 9‑3 SOC的硬件框架

上图为典型的I2C的硬件架构，

# 嵌入式多机通讯说明

对于目前复杂的嵌入式环境，

# 附录

## 如何自学嵌入式Linux

来源详见我的知乎: <https://www.zhihu.com/question/395435388/answer/1279941719>

我的经历可能就是那种最传统的嵌入式学习路线，从51开始，然后熟悉STM32，然后慢慢过渡转向嵌入式Linux，在一年前，对于嵌入式Linux我也只能说会用，项目中参与过其中一小部分的功能和模块的开发，无法系统的去描述整个项目的运转，不过经过这段时间的补足，也系统的看了Linux驱动和应用层相关的书籍，并进行了整理实践，也算有些心得体会，后面的叙述只是描述我个人的学习的路线，不一定适合所有人，但取长补短，如果能给你些启发就足够了。

我之前写过一篇文章叫**嵌入式背后的思想-数据的流转**，事实上对于嵌入式Linux开发所需要的知识，也正是处理数据流转时所需要的技术, 对于嵌入式Linux项目的开发，主要包含以下的知识点:

1. 数据输入/输出的接口 -- 嵌入式Linux驱动开发
2. 数据运行的平台和支持环境 -- uboot开发，Linux内核裁剪，文件系统移植
3. 数据处理和转换 -- 嵌入式应用层开发

如果继续细分下来，就要包含以下工作:

### 嵌入式工作详解

#### 嵌入式Linux驱动开发

1. 对于外部设备的硬件实际操作和调试(这部分和单片机时相通的)，包含SPI，I2C，ETH，HDMI， CMOS等
2. Linux实现驱动模块的API接口(包含基础的Module\_xxx的函数，还有添加类，设备的接口函数以及引申的虚拟总线接口）
3. 为解决驱动代码冗余的设备树相关知识和解析设备树的代码实现

#### uboot, 内核与文件系统

1.uboot主要涉及嵌入式裸机的应用开发，非原厂人员基本不会修改，顶多修改下logo，或者加些简单的控制, 基本不会去在uboot中实现复杂应用。

2.Linux内核裁剪，我所接触的就是Menuconfig内的配置删减功能，以及将驱动模块添加到内核的实现，更复杂的我没有涉及过，不发表看法

3.文件系统移植，主流的是移植支持QT或者安卓的环境到平台中，当然无界面的应用使用最小系统也能满足需求, 已经在此基础上移植支持的基础lib，启动文件，指令文件和目录等，更复杂的我也没有涉及

#### 嵌入式应用层开发

嵌入式的应用层是具体实现功能的部分，主要包含

1.使用open，write，read，ioctl等对底层硬件信息的操作(包含接收，发送)

2.基于系统接口的对于命令行信息，进程和线程，socket通讯接口(TCP/UDP), 同步机制(互斥锁，自旋锁，管道，消息队列等), 文件I/O等基础Linux API处理

3.基于上述接口实现的上层应用实现，如基于TCP的网络应用HTTP或MQTT，视频应用RTMP，以及为了功能需求而引入的线程或进程机制，而为了满足复杂应用的需求，线程或进程间同步机制也被使用，此外可能会引入Sqlite用于数据的管理，引入openssl用于数据的加密，这些都是应用层开发常用的技术手段，嵌入式Linux比单片机的最大优点就是有很多拿来即用的方案，十分方便应用层的功能开发。

4.为了满足工业界面化需求引入的界面实现方案，目前嵌入式的主流就两大类QT/C++和Android/Java(为了满足Jave对底层的操作，引入了JNI的实现)

#### 工作内容总结

了解到这里，你大概对嵌入式Linux的项目整体有了认知，下面标些重点:

**1. 嵌入式Linux驱动中驱动层相关的API接口，DTS语法这些都是从业者必须掌握的技术，如果你想从事驱动层的开发，这部分是必须去理解掌握的**，那么从业中一般人和资深者最大的区别是什么，对于硬件的调试经验，如何I2C或SPI数据不通，如何快速区分是软件还是硬件问题(器件问题还是电路问题)，并有可行的方法去解决，这才是最重要的技巧, 涉及硬件的调试经验是你买块开发板去学习最难掌握的事，这种就是纯项目经验。

**2. uboot开发，Linux内核裁剪，文件系统移植**，这部分我很难给出经验，一方面这方面在工作中占比不高(从我的开发经验来说)，另一方面这部分其实都是按照教程在走流程(包括实际开发中，很多也是按照官方的方案走流程)，非原厂的很少去理解这部分内部的实现，不过我认为这是合理的，这三块都是这个世界上最优秀的那一批工程师多年积累的，如果一般人都能快速吃透，那个人一定是天才，我是做不到，这部分的源码很多是精华，但我不推荐入门者去啃这部分，能够编译构建个满足需求的环境就够了，如果真想去学习，等经验丰富想提升在去花时间会更快捷且有用。

**3. 嵌入式应用层开发,** 对于项目来说，主要关注的其实就是驱动和应用两块，其中应用又是具体的实现部分，往往也是需要掌握的核心(事实上，很多公司购买的测试方案板基本都包含了所有的需要的外设驱动和实现, 顶多改下pin脚或者将接口更换下，或者外部器件更换，修改些寄存器的配置，非原厂和方案商，驱动开发在工作中占比很少), 我上面提到了应用基本在现实的产品都有运用，而且应用层目前已经开始向桌面或移动端的应用靠拢，**除了对底层硬件的直接操作外**，其运用的语言包含**C(Linux API), C++, Python, nodejs, Java以应对图形界面，网络web，算法等的多方面需求**，事实上，这里很多实现真的在桌面端的Linux中进行执行完全没有问题，而且跑的更快，调试也更方便，我自己实现的很多代码，都是在WSL或者虚拟机里运行，涉及硬件的话包装成数据包测试的，都是验证完功能后在交叉编译到嵌入式平台测试，效果也基本一样。

### 嵌入式学习计划

讲完了这些，应该基本对嵌入式Linux到底做什么，什么是重点又一定认识了，下面真正开始分享我的学习方法了。

1. 熟悉Linux平台的常用指令，顺便熟悉vim的用法

我列举些常用的sudo，ls，vim/vi, ifconfig, clear, chmod, mkdir, cp, tar, cat等

2. 熟悉交叉编译的相关知识，包含基础Makefile的理解，入门不要花费大量时间去掌握Makefile的语法，这里推荐文档**<跟我一起写Makefile.pdf>**, 初期理解前三章，后面根据经验总结在去同步掌握。

这两部分必须掌握，至少是熟悉，只有这样才不会托后面的进度。

下面开始选择主要学习的方向，驱动层还是应用层，和大多数培训机构或者开发板的厂商按照驱动动辄几十章来说，我更倾向于应用层的开发，同时兼顾对底层驱动的开发调试，这当然和我自身的从业经历有关，我更偏向于工作上用的到的技术，当然这并不是底层不重要，当给你一块完整经过测试的开发板时，很多底层硬件的调试经验也就剩下驱动软件的开发(很可能都不会自己去实现一遍，只是同一套代码编译在跑一遍)，很多为什么这样设计的思想也就理解不了，在这种情况下，重复花大量时间是不明智的，当然无论最后选择驱动层还是应用层，入门都不建议在uboot，内核或者文件系统花大量时间，能够搭建满足需求的稳定平台就够了，初期不要去深究，不是因为这部分不重要，而是因为当你对嵌入式Linux都一知半解的时候，去啃其中最复杂的部分不是事倍功半吗，等开发经验丰富，自然会总结类似的设计思想，在回过头过来去学习，那时也能有自己学习方法，事半功倍。

3.对于应用层，怎么学习就有的说了，既有python，node这样环境的移植，又有sqlite数据库，openssl加密算法，opencv图像处理库，mqtt这种通讯应用的移植，还有建立在这些移植的库和应用基础上的功能应用需求，界面的开发，因为涉及行业的不同，往往需要的技术也不同，不过需要的也基本就是上面提到的知识，这些知识基本上和桌面端开发并没有太大的区别(事实上很多代码使用桌面端编译工具重新编译下，就可以直接执行，所以没有开发板完全可以学习), 这里主要提一点，关于Linux API是其中比较需要系统去学习的，我也在整理这方面的知识和demo方案，分享如下:

<https://zhuanlan.zhihu.com/p/149344526>

另外这里可以推荐本书《Unix环境高级编程》和 《UNIX网络编程 卷2：进程间通信（第2版.pdf》--这本书基本涉及的API都有，不过中文翻译的看起来没有英文版的舒服。

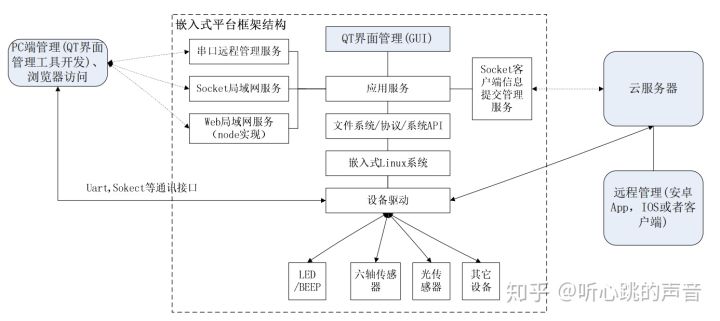
另外对于如何学习这部分内容，我个人认为我一年前所说的方法仍然是合适的:

<https://www.zhihu.com/question/322341076/answer/682578055>

不过一年前我关注的是用需求学习嵌入式可以更加快速的学习嵌入式，但是现在我更关注的是通过整个应用的实现，可以系统的掌握嵌入式整个产品的开发流程，这也是我在这行业走了几年后获取的最重要的经验，很多时候嵌入式产品并没有那么复杂，而学习嵌入式Linux同样。对与过来人来说，Linux操作系统和uboot那块是精华，这本身没错，我目前偶尔也在去深入学习了解那部分的代码，但对于入门者来说，特别是为了踏入行业，找份工作的入门学习者，这部分是和实际产品开发脱钩的，或者其中很小的一部分，如果初入门就扎入茫茫的Linux内核代码和驱动应用中，不仅体验十分差，对于找到工作的帮助也没有那么大。

我遇到很多提升非常快的人，都是工作中为解决实际问题而去针对性学习和提升的，而我说的学习方法也同样如此，把学习当作工作目的去对待，当你转变思想后，自然就会去思考如何做，会去检索查资料，会去设计软件实现，也就自然去琢磨如何学才能够实现，会在遇到问题时去从整体方面去思考解决，这不仅是嵌入式Linux的学习思路，也是工作中去实现需求，解决问题的思路。

另外，如果你只想从事嵌入式Linux开发，单片机不学也影响不大，他们共通点也就只有对硬件底层的实际操作是一致的，但这部分技术选择Linux学习还是单片机学习在我看来并没有太大区别，不过如果你不熟悉软硬件联调，用单片机过渡下也可以，最后我分享一个我自己正在实现的嵌入式方案，你可以考虑实现其中的一部分，自然就知道我所说的方法的含义。



另外这两篇文章也是我对自己经验的总结，希望对你有用:

<https://zhuanlan.zhihu.com/p/146850253>

<https://zhuanlan.zhihu.com/p/142524275>

## 嵌入式Linux问题总结

本节中将列出嵌入式Linux编译，下载，执行中的遇到问题，以及我解决问题所使用的方法，如果对问题理解知道原因的话，也会列出相应的原因。

### 系统问题

1. 系统找不到编译工具(如gcc，arm-linux-gnueabihf-gcc等)，或者在用户权限下能够正常找到，在root权限下无法找到

原因: 未将编译工具的路径添加到PATH中，或者添加的编译路径出错，可使用echo $PATH查看当前的全局路径

解决办法:

在普通模式下添加PATH对应路径为/etc/profile中的export PATH="xxx:<添加gcc路径>",

在管理员模式下PATH对应路径为/etc/environment中的PATH="xxx: <添加gcc路径>"

可通过source /etc/<file>手动更新系统全局变量

1. apt-get update报错: **Could not resolve host**

原因: 网络出错或者DNS服务器异常

解决办法: 网络出错解决网络问题，DNS服务器异常的话解决办法如下

sudo vim /etc/resolv.conf

内部添加如下DNS服务器, wq保存。

nameserver 8.8.8.8

nameserver 8.8.4.4

#WSL下使用如下指令重启DNS服务

sudo /etc/init.d/networking restart

#ubuntu下使用如下指令重启DNS服务

sudo /etc/init.d/network-manager restart

之后apt-get update即可正常工作。

1. 驱动加载显示 xxx: disagrees about version of symbol module\_layout

原因:缺少驱动加载的相关信息

解决办法:

mkdir –p /lib/modules/<linux-version> #如4.1.15

depmod

后续即可正常加载

1. 在执行apt-get找不到资源包如**Unable to locate package openssh-service**

原因:apt-get对应的源路径未完全更新

执行:

apt-get update

apt-get upgrade

即可

1. 嵌入式Linux系统启动时找不到proc路径下文件，命令执行异常

解决办法: 在启动rcS文件中加入

mount –a

mount -t proc proc /proc

### 编译或执行失败问题

本内容主要分享编译失败的原因和各种问题的解决办法。

1. **内核编译时找不到Makefile**

错误声明:

No rule to make target `/usr/kernel/hello/Makefile'. Stop.

解决办法:

当前路径下不存在Makefile，或者使用小写的makefile，会导致以上错误。

1. **编译时/bin/sh: 1: lzop: not found问题**

原因:系统缺少lzop相关的包

解决办法: sudo apt-get install lzop

1. **mq\_open: Function not implemented执行报错**

原因: 内核中不支持mq消息队列功能

解决办法:

make menuconfig

  General setup  --->

[ ] POSIX Message Queues

修改为选中选中状态。

**4．modprobe报错modprobe: can't change directory to '4.1.15': No such file or directory**

原因:缺少4.1.15路径

使用mkdir /lib/modules/4.1.15创建

modprobe找不到模块

原因:在/lib/modules/4.1.15下缺少模块的依赖关系

将加载的模块复制到/lib/modules/4.1.15下，执行depmod指令，后续即可使用modprobe加载。

## 嵌入式系统开发中的中断机制

原文: https://zhuanlan.zhihu.com/p/196452953

在嵌入式系统开发中，中断是十分重要的知识点，在大部分单片机构建的应用产品中，基本都是以前后台方式(大循环加中断)的方式来实现功能，在主循环中处理应用，并在中断中处理外部操作，以及对响应时间有要求的应用，如用于时间相关处理的定时器中断，对按键响应的外部中断，用于通讯的收发和异常处理的串口中断，SPI中断等。另外，对于大部分RTOS来说，Cortex-M系统中的systick中断和PendSV中断，又是实现基于队列和任务调度算法的RTOS的核心；对于嵌入式Linux应用来说，中断也是处理CPU的突发事件的主要方法，包含对于溢出，除零等异常情况的通知，多核之间通讯，外部设备的请求处理等。因此理解中断的背后执行逻辑，对于单片机和嵌入式Linux的开发都有重要意义，虽然中断的目的对于单片机和嵌入式Linux设备基本一致，但因为Linux系统与裸机开发，以及两种芯片设计上的差异，Cortex-A系列和M系列的中断设计是有很大不同的，所以下面就分两部分讲解嵌入式开发中的中断机制。

### 单片机中断机制

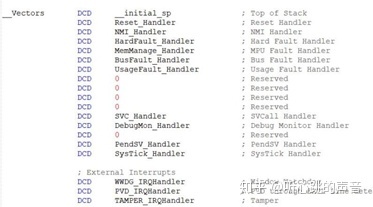
在单片机开发中，对于中断的表示方法也因为内核的不同有很大的差异，如51使用中断号来表示指定中断，而ARM Cortex-M内核中则使用中断向量表的方式配合内核中的NVIC控制器来实现中断的处理，不过考虑到目前的主流单片机方案，因此以典型的Cortex-M3内核说明单片机中的中断控制机制, 该系列的其它内核中的中断流程也基本一致。

#### 中断向量表

对于Cortex-M3内核，支持最大编号包含0~255的中断类型，其中**0~15为系统异常**，主要处理系统执行中产生的复位，错误，主动触发的SVC，异常等, **编号16~255则是由芯片设计厂商自定义设计，用于满足芯片功能需求的中断(芯片厂商可以自由定制, 不超过最大编号且不重复即可)**, 这两部分共同组成了单片机的向量表，参考<Cortex-M3权威指南中的说明>,向量表的格式如下:



反映在软件实现就是在startup\_xxx.s启动文件中定义的中断向量表，具体结构如下：



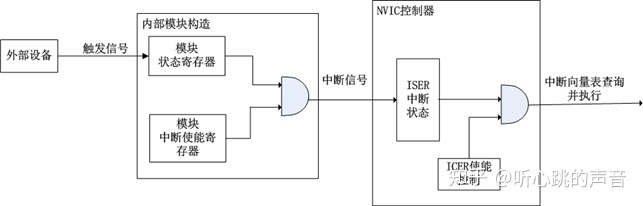
其中External Interrupts就是由厂商定义的中断类型，另外**中断号为0的位置为空，设计上就用来存储堆栈指针**。

在芯片在上电的过程中就是执行复位机制 根据SCB->VTOR查询向量表，找到**Reset\_Handler入口**，并加载\_\_initial\_sp到堆栈指针R13中，后续就可以正常的工作了。

在上述结构中，**系统中断是在内核定义时确定的，外部中断在芯片设计时被确定**，将中断编号和指定外设的中断触发信号绑定，就构建了完整的中断向量表。

#### 中断执行流程

对于指定的外设，如何在中断触发时找到指定的中断入口函数，这就涉及到模块内部的中断机制以及NVIC的功能设计，下面就是典型的单片机中的中断执行流程：



1. 外设或者模块本身根据实际的硬件情况产生触发信号，置位模块内相应状态位，如外部中断的触发，串口数据接收或发送完成等事件，转换为模块内部的状态的变化，并置位相应状态位，总结流程就是触发信号->模块内部状态变化->修改状态标志位。
2. 模块根据内部状态变化，在结合中断使能寄存器相关的配置，将状态变化通过中断信号提交到中断向量控制器(也就是NVIC)中。
3. NVIC控制器根据中断信号定义的编号信息，置位ISER中的相应中断Pending位，在配合ICER的使能状态，如果当前触发的为执行中的最高优先级中断，则查询中断向量表并执行。

上述就是Cortex-M系列中断的主要执行流程，根据这样的流程，我们在软件中的中断执行应用也包含模块中断的使能和内核中断的使能，如对于UART模块中断配置中的

USART\_ITConfig(USART1, USART\_IT\_RXNE, ENABLE);

NVIC\_EnableIRQ(USART1\_IRQn);

就是执行上述流程中的模块内部和NVIC内部的中断使能工作，了解了这些，就可以基本知道从外设提供信号到中断触发的全部流程。

不过对于实际的应用，还有个说明是内核如何找到指定的中断函数， NVIC会根据中断信号，置位相应的中断Pending标志位，对于Cortex-M内核，这些标志位的和向量表中的编号是一一对应的，如Cortex-M3内核中为8\*32bit的寄存器即256bit，每一个bit对应一个中断向量表中的编号位置。

https://pic4.zhimg.com/80/v2-d95978c4a8b99247f9371ec84a54898b_720w.png

通过上面我们了解到，可以通过SCB->VTOR寄存器的值找到中断向量表的首地址，而中断的编号和其在向量表中的位置是一一对应的，以中断向量表的首地址为Start，中断编号为num为例，则中断的入口地址为

Saddr = Start + num\*4;

同时内核将修改后的地址赋值给PC指针，即可实现软件代码的执行路径改变，这方面的知识在软件中并不常用，只有在带bootloader的升级应用或允许设备在SDRAM或SRAM这种地址运行时，需要了解并进行修改，才能保证代码中断的正常执行。

#### 中断优先级

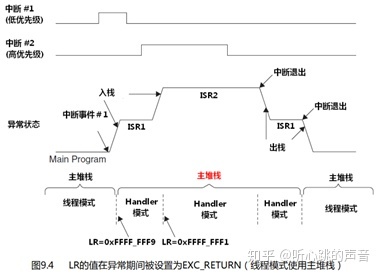
中断中的另一个重要知识就是中断优先级和中断抢占，在上述我们讲述的主要是中断对于主循环的打断和流程改变，但实际应用中，有多个中断同时执行的情况，这就需要通过优先级来满足不同中断的执行需求，一般来说，通讯的不及时处理会导致FIFO溢出，接收超时等问题，导致通讯失败，一些特殊检测的设备，如过压，过流，保护的响应，如果处理不及时，会导致电路和芯片的损毁，这些应用的中断需要立即响应，中断优先级就是为了满足中断执行的响应快慢要求不同而设计的，在内核中，通过

1. SCB->SHP设置系统中断的优先级
2. NVIC->IP设置厂商自定义中断的优先级
3. SCB->AIRCR(应用程序中断及复位控制寄存器)设置寄存器的分组

在说明优先级相关的知识下面，要先了解几个概念。

优先级等级:决定中断执行顺序的等级，M3规定优先级值越小，优先级越高

中断嵌套:当一个中断打断另一个中断的执行流程时，优先执行时，被称为中断嵌套。



上图就是典型的发生中断嵌套的过程，当触发中断事件后，流程如下:

1. 中断#1判断为触发，进行入栈操作，同时系统模式从线程切换到Handler模式，开始执行中断服务函数ISR1
2. 当优先级更高的中断#2判断为触发，则打断中断#1的执行，再次执行入栈操作，系统模式不变，执行中断服务函数ISR2
3. ISR2执行完毕后，进行出栈操作，执行ISR1的后续部分，系统模式不变
4. ISR1执行完毕后，进行出栈操作，后续进行被打断的主循环中，继续执行，同时系统模式从Handler切换为线程模式，整个中断嵌套的流程执行完毕。

**抢占优先级:**优先级的寄存器高位bit[7:n]，声明抢占优先级高的中断可以打断低优先级中断，进行中断嵌套。

**亚优先级:**优先级的寄存器高位bit[n-1:m], 声明亚优先级高的中断，能够优先执行，但不能打断其它中断，需要等低优先级中断结束后才能执行。

M3内核规定最大有8bit的优先级等级(具体支持级数由芯片公司设计时确认)，其中至少有1位为亚优先级，按照这样设计，内核规定最大的强占优先级级为128级。下面以3bit可用的优先等级为例:



其中低5位不可用，但仍可以被分配给亚优先级，如此就有2^3=8个等级的抢占优先级。

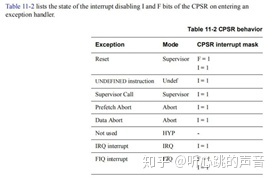
中断流程，中断向量表，中断嵌套，优先级，这些就是单片机中需要了解和掌握的中断相关的知识，这部分其实在产品开发中十分重要，如进行以太网，Uart等硬件交互的时候，确保底层FIFO不会溢出，对于定时器，如何保证时间的精确有效而不被其它应用干扰，如何规划避免风险，这都需要对中断机制和优先级部分有着深入的考虑，所以这部分知识是十分值得了解掌握的。

### 嵌入式Linux设备中的中断机制

在上述流程，我们大致对于单片机中的中断有了比较清晰的认识，也可以理解在启动文件startup\_xxx.s中中断向量表的含义，以及中断触发后找到中断入口的具体流程。不过对于嵌入式Linux系统来说，虽然有一定的参考作用，单从原理和应用来说差别还是蛮大的。

#### 1.2.1 中断机制

下面是Cortex-A7中的向量中断表，不过和M3相比，仅支持7个异常中断(一个未使用)。



详细表格如下:

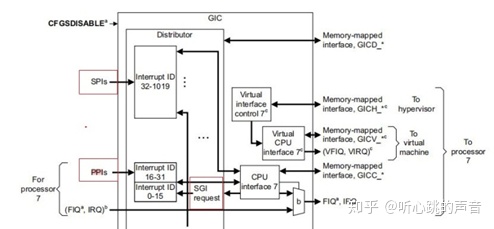
|  |  |  |
| --- | --- | --- |
| 0x00 | 复位中断(Reset) | 特权(Supervisor) |
| 0x04 | 未定义指令中断(Undefined Instruction) | 未定义指令(undef) |
| 0x08 | 软中断(Software Interrupt, SWI) | 特权(Supervisor) |
| 0x0C | 指令预取中止中断(Prefetch Abort) | 中止(Abort) |
| 0x10 | 数据访问中止中断(Data Abort) | 中止(Abort) |
| 0x14 | 未使用(No Used) | / |
| 0x18 | IRQ中断(IRQ Interrupt) | 外部中断(IRQ) |
| 0x1C | FIQ中断(FIQ Interrupt) | 快速中断(FIQ) |

上图就是Cortex-A7芯片支持的中断的全部类型，其中0x00~0x10为系统中断，当芯片执行出现异常，或者由SWI指令主动触发时，就会执行这些中断，一般由Linux内核管理，这里可以了解下，其中关键的有以下几点。

**复位中断:**CPU上电复位后会进入该中断，一般会执行系统硬件的初始化工作，如初始化堆栈指针，配置硬件接口访问外部的DDR等。

**软中断:**由SWI指令触发的中断，一般Linux系统调用会通过SWI指令触发软中断，从而进入内核空间。

基于对单片机的了解，芯片都是支持多个外设的，特别是对于Cortex-A系统芯片，往往更加复杂，那么仅依靠7个中断如何支持外部中断的需求？这就要提到IRQ和FIQ了, 对于Cortex-A系列芯片来说，任意一个外部中断如IO\_Interrupt, Legacy\_Interrupt等，都会触发IRQ或FIQ中断，进入对应的中断函数，，并在此函数中通过软件读取寄存器的值来判断具体发生了什么中断，管理外部信号到中断触发的结构被称为GIC(Generic Interrupt Controller)，而中断又根据源头的不同，分为



**SGI(Software Generated Interrupt):**软件触发中断，如用于多核之间主动通讯的中断，通过软件向GICD\_SGIR 中写入数据触发的中断，中断ID编号可选1~15

**PPI(private Perpherial Interrupt):**私有外设中断， 每个CPU核心的特有中断，中断编号可选为16~31

**SPI(Shared Perpherial interrupt):** 共享外设中断，这些中断可以指定到任意一个CPU内核，可通过接口irq\_set\_affinity接口指定中断最后响应的内核, 中断编号可选为32-1019。

另外对于Cortex-A7芯片来说，中断使能包含IRQ或FIQ总中断使能，以及ID0-ID1019可选的中断源使能两部分；优先级为32bit的数据，支持抢占优先级和子优先级，这部分和Cortex-M系列基本是一致的，可参考上节相应说明这里不在赘述。

#### 中断处理

在理解Cortex-A7内核处理机制之前，需要了解到芯片内核状态分类如下:

1. user mode:用户模式，用户空间AP执行所处于的模式。
2. superiver mode: 超级模式，或者SVC模式，大部分Linux内核执行代码处于该模式下。
3. IRQ mode: 中断模式，触发中断后，处理器进入的模式。

参考上面中断表格，还包含Abort mode来处理上面提到的Data Abort和prefetch Abort异常。下面以在用户进程执行下的IRQ中断来演示大部分中断的执行流程。

1. 和上面的单片机的流程类似，当有外部触发信号到达，并且所有中断相关的使能都打开的情况下，中断控制器GIC就会根据配置好的硬件信息，将IRQ(或FIQ)的中断触发信息告知指定的Core。
2. 处理器感知到该信号后到达时，对于进行irq模式前的系统状态值如cpsr寄存器值，PC指针进行保存(分别保存到SPSR和LR寄存器中)
3. 置位相应的中断状态标志位
4. 根据上面的表格计算中断向量的入口位置，将PC设置为该值并跳转，

总结下来，在中断发生时，内核的硬件处理包含置位中断信息，保存中断前关键状态，进入IRQ模式，然后在跳转到中断向量的入口，后续就由软件接口进行后续处理。

对于软件部分的处理，则包含irq模式，svc/usr模式处理和应用代码处理。

1. irq模式主要进行了r0，lr以及cpsr的保存，压栈处理(Irq模式下的堆栈设定为12字节)
2. 根据进入中断前的系统模式，维护中断处理表格，进行后续的中断处理，根据**中断前模式的不同**分别执行不同的入口函数\_irq\_usr(用户模式入口函数)和irq\_svc(superior模式入口函数)，同时将系统模式切换到SVC模式
3. 以用户模式为例,主要实现流程包含:保存用户现场, 执行中断向量irq\_hander，这是软件部分的核心处理，在其中实现将当前硬件中断系统的状态转换为定义好的软件IRQ Number, 然后调用IRQ Number的处理函数即可。
4. 执行完中断相关的处理函数，将进入中断时候保存的现场恢复到实际的ARM寄存器中，返回到中断触发时执行的流程，即实现了中断返回。

#### 中断应用

对于内核中，涉及到中断访问的接口主要由中断申请和释放的函数组成

static inline int \_\_must\_check

request\_irq(unsigned int irq, irq\_handler\_t handler, unsigned long flags,

const char \*name, void \*dev);

void free\_irq(unsigned int irq, void \*dev);

irq为要申请中断的中断号，由CPU分配的软件中断号

handler即为中断触发时执行的回调函数，一般handler实现如下

static irqreturn\_t key0\_handler(int irq, void \*dev\_id)

{

*//中断的具体实现*

*//...*

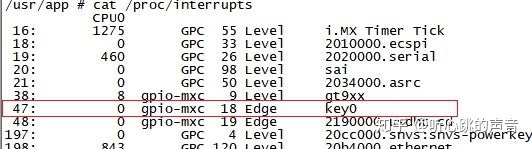
return IRQ\_RETVAL(IRQ\_HANDLED);

}

flags为指定中断实际触发的执行条件，可以是IRQF\_TRIGGER\_RISING， IRQF\_TRIGGER\_FALLING等。

name为中断指定的名称，在cat /proc/interrupts下显示，如key0就是通过申请的中断。

dev则为传递到中断执行的参数，一般用于IRQF\_SHARED中断执行时，区分不同的中断，大部分情况为设备结构体，dev会传递给中断处理函数中的第二位。

>

当然，也可以通过devm\_request\_irq来申请中断，可以在驱动卸载时不用主动调用free\_irq显示释放中断请求。

此外，也可以通过disable\_irq(非中断函数中)， disable\_irq\_nosync(中断函数中)和enable\_irq来管理中断的开关。

在单片机中，我们了解到中断的执行会打断其它应用的执行，所以中断的动作应该尽可能的短，如果具体的操作过长，会把代码分为两部分，其中时间相关比较紧要的在中断中执行，而非必要的则移动到主循环中执行，如UART通讯中，数据接收往往通过中断或者DMA获取，而协议解析和应用处理则在主循环执行，那么对于嵌入式Linux来说，在驱动中也采用类似的机制，把中断的应用拆成两部分执行，一般称为**顶半部和底半部机制**执行，其中顶半部即为上述的中断回调函数key0\_handler，其在内核模式下调用，因此会影响到系统的切换。所以一般来说仅进行简单的事件触发动作，具体的应用则在**底半部执行**，通过常见的**tasklet，工作队列，软中断或软件定时器机制**，避免在硬件中断流程中执行大量的代码，从而影响到Linux系统的执行和切换，这里tasklet为例：

//底半部应用部分执行

static void tasklet\_do\_func(unsigned long data)

{

printk(KERN\_INFO"key interrupt tasklet do:%ld!\r\n", data);

}

DECLARE\_TASKLET(tasklet\_func, tasklet\_do\_func, 0);

//顶半部中断向量执行

static irqreturn\_t key0\_handler(int irq, void \*dev\_id)

{

/\*触发事件\*/

tasklet\_schedule(&tasklet\_func);

return IRQ\_RETVAL(IRQ\_HANDLED);

}

如此，当执行按键时，即可正常触发中断显示如下：

理解到这，我们对嵌入式Linux中的中断触发和在内核的应用已经有了初步的了解，那么我们是如何获取外设对应的硬件中断号，这就需要涉及设备树中的中断信息节点，以imx6ull为例，在3.2 Cortex A7 interrupts章节中，定义了内部的中断ID，以按键对应的GPIO1\_18引脚中断为例，其中

基于上述结构，我们定义gpio1作为GPIO1上对应的总的中断控制入口(即中断控制器), 实现如下:

//定义中断控制器节点

gpio1: gpio@0209c000 {

//其它已经注释

interrupts = <GIC\_SPI 66 IRQ\_TYPE\_LEVEL\_HIGH>,

<GIC\_SPI 67 IRQ\_TYPE\_LEVEL\_HIGH>;

interrupt-controller;

#interrupt-cells = <2>;

};

//具体的中断节点

key {

interrupt-parent = <&gpio1>;

interrupts = <18 IRQ\_TYPE\_EDGE\_BOTH>;

};

在具体的代码实现中，可通过函数

unsigned int irq\_of\_parse\_and\_map(struct device\_node \*node, int index);

即可找到引脚对应的中断线号。

### 总结

至此，我们对嵌入式应用中的中断机制进行了解读，当然这些都是理论知识的说明，对于嵌入式应用中，如何结合实际情况，配置合适的中断优先级，并实现应用从而满足产品的需求才是最重要的部分。这部分经验是需要实践积累和总结的，不过理解了中断实现的背后机制，在实践中知其然也知其所以然，以理论配合应用来学习，也是嵌入式开发的最佳提升之道。

## 从USB协议解析去理解协议学习方法

在嵌入式的工作中，协议作为多设备协同的基础，是必须被掌握的，也是面试中经常考验的知识点。对于稍有经验的开发者来说，CAN、Modbus、Zigbee、BLE、USB、以太网TCP/IP和WIFI等协议可能都或多或少的接触过，不过大部分情况下我们使用的都是串口转指定协议模的块或者封装好的库来实现目标，只有在遇到问题或者特殊需求时，才会去关注协议本身的知识，这时基础的薄弱往往是很大的门槛。而且即使搜索可以很简单的找到大量相关的资料，但应用时仍然不知道如何解决，这是从入门后走向资深开发者的最大门槛，即使是我，在这条路上之前也并不顺畅，不过当跨过门槛后，我希望总结出一条比较合理的路，由浅到深的系统的去理解协议本身，而不是说这样写的代码就能做出这样的产品，相反，这样的知识也许对我来说意义并不大，虽然移植修改代码完整实现功能来说并不简单，但系统的掌握一种协议则困难太多了，话不多说，下面开始正文吧。

### USB协议解析

在学习USB协议之前，首先要理解协议的本质是什么。前面已经说了，协议是用来解决多设备协同工作的，那么两台互相不了解的设备如何知晓对方的信息，这就需要我们在设备的系统外进行一定的约束。

1. 约定数据的格式、命令和接口的电气特性，确保两台或多台设备能够互相识别信息
2. 约定一套上层的软件处理机制，确保两台或多台设备能够进行正确的数据交互

而协议就是综合了上述需求定义的文档，而嵌入式的工作则是根据文档实现具体的交互应用，这就是协议的本质和应用方法。对于MCU设计人员来说，就要根据协议中(1)的规定，确定硬件模块规格，进行相关设计，而对于嵌入式驱动或应用开发人员，则根据(2)的内容，通过软件实现协议本身的命令，格式处理和数据通讯，从而构建完整的应用，这部分也是作为嵌入式开发者需要关注和掌握的知识点，了解了这个知识，就可以开始USB协议的学习，通过学以致用，真的的理解上述信息。

## 在芯片公司的嵌入式工作心得

几年前，机缘巧合进入了芯片行业，之后兜转两家芯片公司，过程参与了图形处理芯片和电源管理芯片的开发，如今虽然已经跳出芯片行业，从事产品应用的开发，但这段过程也积累的很多心得体会，这里回顾下，对过去的经历总结，夯实基础，如果看到的人有所收获，那就更好不过了。

这里先总结下整个芯片流程开发中，嵌入式相关的主要工作：

准备阶段 -- 根据产品调研，确定芯片整体规格和模块规格，这里可以简单说下，对于整体规格，主要是运行频率，容量，外部接口(UART, SPI, I2C)，中断支持，特殊模块(如一些算法加速器支持), 这个数据主要来源有前代积累经验，对市场预期评估以及竞品分析，在结合应用场景和成本控制需求，裁剪到合理的范围，芯片的工艺，功率和成本考量，决定了一定要取舍。

研发阶段 -- 将准备阶段的规格书，转换成设计需求文档, IC设计会根据设计需求文档实现模块相应的设计文档(这个就是芯片参考手册的原型), 之后就要配合IC设计在FPGA上进行**数字原型验证**，IC工程师在开发中是可以数字仿真的，但耗时间十分长，如果有应用逻辑参与的测试，以及外部触发条件，组合逻辑，一般都是通过提供FPGA的固件来由嵌入式工程师测试，这里说下我总结的测试数字模块功能性的主要选项:

1. 模块的寄存器复位值，读/写, 时钟使能/复位对寄存器影响
2. 中断功能触发，标志位置位/清零
3. 单模块的功能性测试，这个就具体模块具体分析，如UART就要测试不同波特率，数据位，停止位的组合，有可能还支持不同引脚的映射，这里组合就更多了，一般都是实现循环逻辑，通过上位机命令切换配置，然后重复测试。对于Timer，就是定义不同的模式，看看触发时间精度，当然一般配合外部I/O在示波器上进行检查。
4. 在比较后期阶段，往往还会添加不同模块组合功能测试，如配合DMA，低功耗，不同执行空间在应用环境下。

这个阶段也是整个芯片过程中嵌入式工作最繁琐，也是最重要的阶段，

一是要查缺补漏，把芯片功能相关的尽可能覆盖全，当然，此阶段随着对模块的功能深入了解，对于准备阶段的需求也要有同步的更新纠错，有可能需求和设计不合理，虽然满足要求，但后期应用过于复杂，也有需求余量过于狭窄，后期对于可靠性带来风险，查缺补漏是整个设计阶段都要时刻警惕的问题。

二是只要测试出问题了，就需要配合IC设计分析原因和解决，解决完成后至少还要完整一轮测试(对于A的修改有可能影响B，甚至有笔误改错了都有)，因为这步骤基本循环很多轮，我一般遇到问题, 解决完成后，之前不重测，等此轮循环跑完后，在覆盖一轮测试，直到执行没问题。芯片整个研发流程，嵌入式的工作是芯片成品质量和后期应用的基石，很难保证没有错误，但尽可能的覆盖十分重要，不然后期会带来很多麻烦。

流片阶段 -- 在流片阶段，一般3个月以上空档期，这段时间主要的工作分为三类

1. 驱动库开发和完善，类似于ST的HAL和LL库等(我参与的两个都是单片机项目），一般在研发阶段，我都习惯按照驱动库的接口来实现测试代码(当然有一部分只能寄存器访问, 驱动库只是芯片功能的一部分), 那么在这个阶段就基本补充下缺少的接口，完善下错误检查等，相对轻松些，二阶段工作量大一些，但也避免了重复劳动。
2. 芯片参考手册， 之前我说过，在研发阶段，IC工程师会根据芯片需求文档，进行模块设计文档的实现，这一步就要把模块分章节组合起来，同时删减设计实现细节，修改为应用相关的配置和操作说明，最后就是成品的芯片参考手册。
3. 应用方案开发，其实在研发阶段末期，最后一般都会执行应用方案Demo，如我之前做的图形识别处理芯片，就实现过指纹解锁，二维码识别，摄像头屏幕显示这些Demo方案，在芯片流片前就测试通过，不过在此阶段，就不是类似于上述的demo方案，而是产品级的解决方案，像电源管理芯片，就是实现了支持多协议快充，分别为AC-DC的充电头以及DC-DC的充电宝的两套方案。

验证阶段 -- 芯片流片返回后，就是决定上述所有工作意义的阶段，不提芯片到研发之前的封测等厂商步骤，在拿到第一版芯片后，就需要在提前设计好的PCB板上完成和研发阶段的重复步骤，验证芯片的功能性，虽然就参与了两颗芯片设计，而且因为本身公司从业行业就经验丰富，参与的两个芯片都成功了，但是那个阶段开盲盒的心理压力还是很大的。当然这个阶段是有可能遇到测试阶段没有发现的bug问题，这时解决办法就比之前仅和IC分析解决就难的多，首先要评估对后续应用的影响，是否能够通过软件规避，如果能够，出相应技术说明给上级就可以，如果不能，且严重，就可能涉及芯片改版问题，要先分析原因，通过FPGA原型验证解决后，然后出报告书，还得给出流程上如何避免下次出现同样问题(这个是同事经验), 我参与时间没那么久，没有遇到，也算幸运。

开发阶段 -- 芯片验证完成，如果符合预期，后续就开始小批量生产，进行正常的商业流程，这时就是和正常嵌入式工作差不多了，应对不同客户的需求，进行方案开发，以及解决软件bug和问题质询，因为是原厂，所以面临的问题往往更接近底层，但也可以和设计端得到更多支持，总体来说不算困难。当然，按照现在的迭代速度，开发阶段一般也就半年左右，就要开始下一代芯片的预研，一年又一年，工作就是个轮回。

至此，对于几年在芯片公司的经历做了个总结，写完也有点唏嘘，时间不等人，我也快30了，头发也日渐稀少，希望几年后，我还能如此对嵌入式行业充满激情和热爱，共勉之。

## 从Linux Shell到Makefile语法

对于从