

1 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，没有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瓶颈。发现了问题，自然就要去解决，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，C 语言开发，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，在掌握了前面所有知识后，对于完整的产品，也只是对应项目的基础构建部分，对于具体的产品，可能还需要电力电子，网络，音视频图像，声音处理等专业方向的技术知识，这些技术知识不仅是学习的难点，对于系统/软件版本引发的编译，调试问题，复杂应用中的异常处理问题，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，蓝牙等的调试开发经验，单纯用文字描述很难清晰，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，多练多总结积累技术经验的唯一方法。这行业需要耐心和不断的努力，在开发中遇到问题要保持良好的心态，积极思考，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。

2 目录

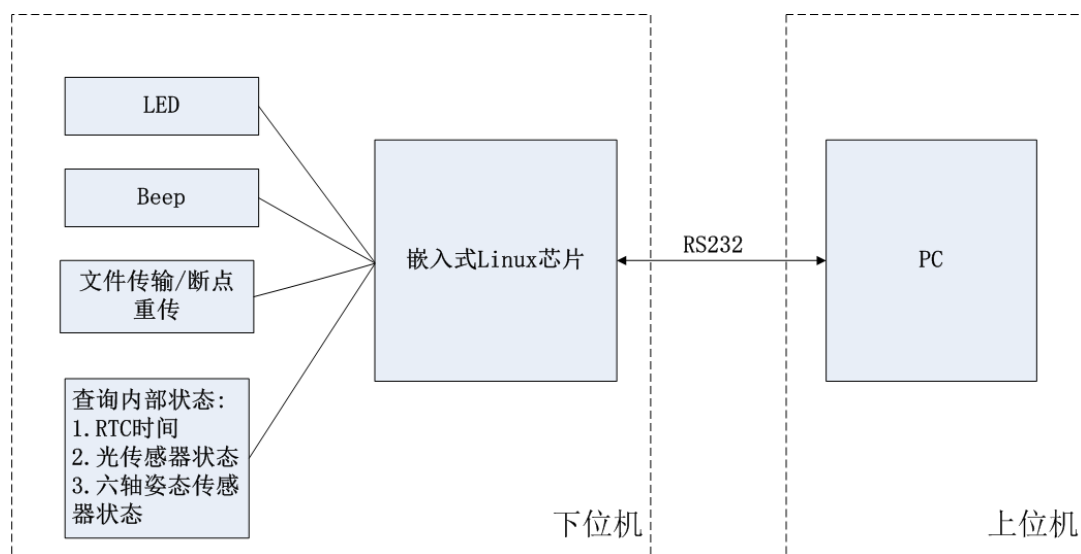
1	前言	1
2	目录	2
3	项目架构初探	4
3.1	系统架构	4
3.2	硬件说明	4
3.3	代码路径	4
3.4	功能说明	4
3.5	任务分解	5
3.6	内核驱动模块初探	5
3.6.1	参考资料	5
3.6.2	必选接口	5
3.6.3	可选接口	5
3.6.4	模块间的调用	7
3.6.5	编译错误和解决办法	9
4	Linux 系统命令和交叉编译	10
4.1	Linux 常用命令整理	10
4.2	交叉编译环境构建	11
4.3	uboot 编译和测试	13
4.4	Linux 内核编译	14
4.5	基于 Busybox 的文件系统编译	15
4.6	mgtool 文件下载更新	17
4.7	单步更新的方法说明	18
4.7.1	nandflash 的单步更新	18
4.7.2	emmc 的单步更新	21
4.8	总结	21
5	LED 驱动开发	22
5.1	参考资料	22
5.2	LED 硬件接口和 DTS 设备树	23
5.2.1	硬件原理图	23
5.2.2	设备树实现	23
5.2.3	对设备树的操作	24
5.3	Linux 内核加载和删除接口	26
5.4	设备创建和释放	26
5.5	设备访问的接口	29
5.6	Makefile 编译和模块加载	31
5.7	测试代码实现	31
5.8	总结	32
6	设备树的说明	33
6.1	设备树综述	33
6.2	设备树语法	34
6.2.1	#include 语法	34

6.2.2	节点描述	34
6.2.3	节点属性	36
6.3	设备树在驱动中应用	40
6.3.1	内核设备树访问函数	40
6.4	总结	42
7	Uart 通讯和协议实现	43
7.1	Uart 通讯实现	43
7.2	通讯协议的制定	46
7.2.1	协议制定	46
7.2.2	数据处理	49
7.3	总结	53
8	附录	54
8.1	如何自学嵌入式 Linux	62
8.1.1	嵌入式工作详解	62
8.1.2	嵌入式学习计划	63
8.2	嵌入式 Linux 问题总结	65
8.2.1	系统问题	65
8.2.2	编译或执行失败问题	66

3 项目架构初探

3.1 系统架构

任何项目的开发都应该是从框架构建开始的，根据用户的需求，将功能组件分解为具体的模块任务，最后组合起来，进行完整的应用测试。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，这也是项目经理的主要工作，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。



3.2 硬件说明

正点原子的 I.MX6U-ALPHA 开发平台，512MB(DDR3)+8GB(emmc)核心板。

涉及硬件 RS232, GPIO, I2C, SPI

3.3 代码路径

https://github.com/zc110747/remote_manage

3.4 功能说明

结合上面的框图需求，需要分为上位机和下位机两个部分的实现，具体需求如下：

1. 上位机软件支持界面化管理(确定使用 QT 开发)，带串口通讯，双机通讯需要制定协议(可使用自定义协议或者 Modbus)。
2. 下位机支持常用的外设，设置和读取 GPIO，并能够获得 RTC，各类传感器，ADC 等的参数信息。

3. 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)
4. 支持后期的扩展

3.5 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)。

3.6 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。

3.6.1 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

3.6.2 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"

3.6.3 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_SYMBOL(func)/EXPORT_SYMBOL_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。

```
root@ubuntu:/usr/kernel/hello# make
make -C /lib/modules/3.5.0-23-generic/build M=/usr/kernel/hello modules
make[1]: Entering directory '/usr/src/linux-headers-3.5.0-23-generic'
CC [M] /usr/kernel/hello/hello.o
Building modules, stage 2.
MODPOST 1 modules
WARNING: "add_integar" [/usr/kernel/hello/hello.ko] undefined!
LD [M] /usr/kernel/hello/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-3.5.0-23-generic'
```

之后执行指令 **modinfo hello.ko** 即可查看当前的模块信息。

```
filename:      hello.ko
alias:         a simplest module
description:   a simple hello module
license:       GPL v2
author:        ZC
srcversion:    CA0E6C4FE2358A308953FC2
depends:
vermagic:      3.5.0-23-generic SMP mod_unload modversion:
parm:          buf:charp
```

如果无法查看信息, 可通过 **dmesg** 查看加载信息。

```
[13735.374660] hello world enter, driver, 11
[13740.781315] hello world_exit
```

3.6.4 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：

```

root@ubuntu:/usr/kernel/hello# grep integar /proc/kallsyms
ffffffffffa026e040 r __ksymtab_sub_integar [math]
ffffffffffa026e07d r __kstrtab_sub_integar [math]
ffffffffffa026e058 r __kcrctab_sub_integar [math]
ffffffffffa026e030 r __ksymtab_add_integar [math]
ffffffffffa026e089 r __kstrtab_add_integar [math]
ffffffffffa026e050 r __kcrctab_add_integar [math]
ffffffffffa026d000 T add_integar [math]
ffffffffffa026d010 T sub_integar [math]

```

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。

3.6.5 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'
```

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. 模块间调用出错

除 `EXPORT_SYMBOL` 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)
```

4 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。

4.1 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	
insmod/rmmod/modprobe/lsmmod	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	
unmount /mnt/sdb1 释放挂载的设备	

4.2 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-gnueabi**hf.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
```

3. 将 **gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi**hf.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

```
root@ubuntu:/usr/local/arm# ls
gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi
```

4. 使用指令

```
tar -xvf gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi
```

解压到当前路径，如下：

```
gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi/arm-linux-gnueabi/lib/libgomp.a
gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi/arm-linux-gnueabi/lib/libitm.so
root@ubuntu:/usr/local/arm# ls
gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi
```

5. 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-gnueabi/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-gnueabi/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

6. 最后通过指令，即可查看是否安装成功

```
arm-linux-gnueabi-gcc -v
```

```

root@ubuntu:/usr/local/arm# arm-linux-gnueabi-gcc -v
using built-in specs.
COLLECT_GCC=arm-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabi
Target: arm-linux-gnueabi
Configured with: /home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/arm-linux-gnueabi/_build/sty-amd64-tcwg-build/target/arm-linux-gnueabi/_build/builds/destdir/x86_64-unknown-
uded-gettext --enable-nls --disable-sjlj-exceptions --enable-gnu-unique-object --enab
g --with-cloog=no --with-ppl=no --with-isl=no --disable-multilib --with-float=hard --
h --enable-libstdc++-time=yes --with-build-sysroot=/home/tcwg-buildslave/workspace/tc
ith-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd
le-checking=release --disable-bootstrap --enable-languages=c,c++,fortran,lto --build=
rkspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/arm-linux-gnuea
Thread model: posix
gcc version 4.9.4 (Linaro GCC 4.9-2017.01)
root@ubuntu:/usr/local/arm#

```

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-gnueabi-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/>。

4.3 uboot 编译和测试

从学习的角度来说，uboot，内核，文件系统在嵌入式 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：

BOOT_CFG 引脚	对应 LCD 引脚	含义
BOOT_CFG2[3]	LCD_DATA11	为 0 时从 SDHC1 上的 SD/EMMC 启动，为 1 时从 SDHC2 上的 SD/EMMC 启动。
BOOT_CFG1[3]	LCD_DATA3	当从 SD/EMMC 启动的时候设置启动速度，当从 NAND 启动的话设置 NAND 数量。
BOOT_CFG1[4]	LCD_DATA4	BOOT_CFG1[7:4]: 0000 NOR/OneNAND(EIM)启动。 0001 QSPI 启动。 0011 SPI 启动。 010x SD/eSD/SDXC 启动。 011x MMC/eMMC 启动。 1xxx NAND Flash 启动。
BOOT_CFG1[5]	LCD_DATA5	
BOOT_CFG1[6]	LCD_DATA6	
BOOT_CFG1[7]	LCD_DATA7	

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：

```

root@ubuntu:/usr/code/uboot# ls
api      cmd      configs  drivers  fs        kconfig  MAINTAINERS  net      scripts  test      u-boot.bin  uboot-imx-2016.03-2.1.0-g9bd38ef-v1.0.tar.bz2  u-boot-nodtb.bin
arch     common  disk     dts      include  lib       MAKEALL      post     snapshot commit  tools      u-boot.cfg  u-boot.lds  u-boot.srec
board   config.mk  doc     examples  kbuild   Licenses  Makefile     README  System.map  u-boot     u-boot.imx  u-boot.map  u-boot.sym

```

3. 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 ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- distclean
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- 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-gnueabi- -j12
```

如此就完成了编译

```
mkdir -p board/freescale/mx6ullevk/
./tools/mkimage -n board/freescale/mx6ullevk/imximage.cfg.cftmp -T imximage -e 0x87800000 -d u-boot.bin u-boot.imx
Image Type:   Freescale IMX Boot Image
Image Ver:    2 (i.MX53/6/7 compatible)
Mode:        DCD
Data Size:    385024 Bytes = 376.00 kB = 0.37 MB
Load Address: 877ff420
Entry Point:  87800000
root@ubuntu:/usr/code/uboot#
```

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。

4.4 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:

3. 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

```
400 dtb=$(CONFIG_SOC_IMX6ULL) += \
401     imx6ull-14x14-ddr3-arm2.dtb \
402     imx6ull-14x14-ddr3-arm2-adc.dtb \
403     imx6ull-14x14-ddr3-arm2-cs42888.dtb \
404     imx6ull-14x14-ddr3-arm2-ecspi.dtb \
405     imx6ull-14x14-ddr3-arm2-emmc.dtb \
406     imx6ull-14x14-ddr3-arm2-epdc.dtb \
407     imx6ull-14x14-ddr3-arm2-flexcan2.dtb \
408     imx6ull-14x14-ddr3-arm2-gpmi-weim.dtb \
409     imx6ull-14x14-ddr3-arm2-ldif.dtb \
410     imx6ull-14x14-ddr3-arm2-ldo.dtb \
411     imx6ull-14x14-ddr3-arm2-qspi.dtb \
412     imx6ull-14x14-ddr3-arm2-qspi-all.dtb \
413     imx6ull-14x14-ddr3-arm2-tsc.dtb \
414     imx6ull-14x14-ddr3-arm2-uart2.dtb \
415     imx6ull-14x14-ddr3-arm2-usb.dtb \
416     imx6ull-14x14-ddr3-arm2-wm8958.dtb \
417     imx6ull-14x14-evk.dtb \
418     imx6ull-14x14-evk-btwifi.dtb \
419     imx6ull-14x14-evk-emmc.dtb \
420     imx6ull-14x14-evk-gpmi-weim.dtb \
421     imx6ull-14x14-evk-usb-cert1.dtb \
422     imx6ull-14x14-nand-4.3-480x272-c.dtb \
423     imx6ull-14x14-nand-4.3-800x480-c.dtb \
424     imx6ull-14x14-emmc-4.3-480x272-c.dtb \
425     imx6ull-14x14-emmc-4.3-800x480-c.dtb \
```

使用:wq 保存文件即可。

4. 使用如下指令进行编译

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- distclean
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- imx_v7_defconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- 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-gnueabi- all -j16
```

如此即完成了编译，编译结果如下:

```
make -f ./scripts/makefile.twrnsc obj=twrware _twrmodboot
arm-linux-gnueabi-gcc -wp,-MD,arch/arm/boot/compressed/.piggy.lzo.o.d -nostdinc -isystem /usr/local/arm/g
include -I./arch/arm/include -Iarch/arm/include/generated/uapi -Iarch/arm/include/generated -Iinclude -I./arch/
include ./include/linux/kconfig.h -D__KERNEL__ -mlittle-endian -D__ASSEMBLY__ -mabi=aapcs-linux -mno-thumb-i
nified.h -msoft-float -DCC_HAVE_ASM_GOTO -DZIMAGE -c -o arch/arm/boot/compressed/piggy.lzo.o arch/a
arm-linux-gnueabi-ld -EL --defsym _kernel_bss_size=463200 -p --no-undefined -X -T arch/arm/boot/compres
boot/compressed/misc.o arch/arm/boot/compressed/decompress.o arch/arm/boot/compressed/string.o arch/arm/boot/c
rch/arm/boot/compressed/bswapdi2.o -o arch/arm/boot/compressed/vmlinux
arm-linux-gnueabi-objcopy -O binary -R .comment -S arch/arm/boot/compressed/vmlinux arch/arm/boot/zImage
root@ubuntu:/usr/code/linux# ls
```

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。

4.5 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
```

3. 执行命令

```
make menuconfig
```

在弹出的界面中进入 Busybox Settings 选项

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

4. 执行

```
make&&make install
```

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。

```
cp *so* *.a /usr/code/rootfs/lib -d
```

同时修改 ld-linux-armhf.so.3 为原始的文件，删除链接

5. 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
```

```
root@ubuntu:/usr/code/rootfs# cd nfs/  
root@ubuntu:/usr/code/rootfs/nfs# ls  
bin dev etc lib linuxrc mnt proc root sbin sys tmp usr
```

6. 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。

4.6 mgtool 文件下载更新

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-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外部设备中



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

3. 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：

 imx6ull-14x14-emmc-4.3-480x272-c.dtb	2020/8/11 19:36	DTB 文件	38 KB
 imx6ull-14x14-emmc-4.3-800x480-c.dtb	2020/8/11 19:36	DTB 文件	38 KB
 u-boot.imx	2020/8/9 15:22	IMX 文件	375 KB
 zImage	2020/8/9 14:40	文件	6,629 KB

filesystem 下直接用上章的 rootfs 替换

 rootfs.tar.bz2	2020/8/9 16:26	bz2 Archive	28,084 KB
 rootfs-openedv.tar	2020/1/16 2:17	tar Archive	468,080 KB

4. 替换后使用 Mfgtool2-eMMC-ddr512-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）：

```

UTP: executing "sync"
UTP: sending Success to kernel for command $ sync.
utp_poll: pass returned.
UTP: received command '$ umount /mnt/mtd5'
UTP: executing "umount /mnt/mtd5"
UBIFS (ubi0:0): un-mount UBI device 0
UBIFS (ubi0:0): background thread "ubifs_bgt0_0" stops
UTP: sending Success to kernel for command $ umount /mnt/mtd5.
utp_poll: pass returned.
UTP: received command '$ echo update complete!'
UTP: executing "echo update complete!"
Update complete!
UTP: sending Success to kernel for command $ echo update complete!.
utp_poll: pass returned.

```

5. 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。

```

3.893063] can-3v3: disabling
3.896471] ALSA device list:
3.899457] #0: wm8960-audio
3.937904] UBIFS (ubi0:0): UBIFS: mounted UBI device 0, volume 0, name "rootfs", R/O mode
3.946266] UBIFS (ubi0:0): LEB size: 126976 bytes (124 KiB), min./max. I/O unit sizes: 2048 bytes/2048 bytes
3.956229] UBIFS (ubi0:0): FS size: 493047808 bytes (470 MiB, 3883 LEBs), journal size 24633344 bytes (23 MiB, 194 LEBs)
3.967223] UBIFS (ubi0:0): reserved for root: 4952683 bytes (4836 KiB)
3.973874] UBIFS (ubi0:0): media format: w4/r0 (latest is w4/r0), UUID 3F306448-0331-4594-AF40-80D6D9FFFBDF, small LPT mc
3.986199] VFS: Mounted root (ubifs filesystem) readonly on device 0:15.
3.993836] devtmpfs: mounted
3.997443] Freeing unused kernel memory: 440K (80b56000 - 80bc4000)

Please press Enter to activate this console.
/ # ls
bin      etc      linuxrc  proc    /sbin    tmp
dev      lib      mnt     root     sys     usr
/ #

```

4.7 单步更新的方法说明

4.7.1 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-utils`, 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:

4.7.1.1 获取安装的资源包

wget <http://ftp.infradead.org/pub/mtd-utils/mtd-utils-2.1.1.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 <https://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git>

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

4.7.1.2 交叉编译资源包

4.7.1.2.1 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-gnueabi-hf-gcc ./configure --prefix=/usr/code/mtd_utils/lib/zlib --static  
make && make install
```

4.7.1.2.2 交叉编译 lzo

```
mkdir -p lib/lzo  
tar -xvf lzo-2.10.tar.gz  
cd lzo-2.10  
./configure CC=arm-linux-gnueabi-hf-gcc --host=arm-linux --prefix=/usr/code/mtd_utils/lib/lzo --enable-static  
make && make install
```

4.7.1.2.3 交叉编译 e2fsprogs

```
mkdir -p lib/e2fsprogs  
cd e2fsprogs/  
./configure CC=arm-linux-gnueabi-hf-gcc --host=arm-linux prefix=/usr/code/mtd_utils/lib/e2fsprogs  
make && make install
```

4.7.1.2.4 交叉编译 zstd

```
mkdir -p lib/zstd  
cd zstd/  
export CC=arm-linux-gnueabi-hf-gcc CXX=arm-linux-gnueabi-hf-g++ LD=arm-linux-gnueabi-hf-ld  
RANLIB=arm-linux-gnueabi-hf-ranlib AR=arm-linux-gnueabi-hf-ar CFLAGS=-fPIC CXXFLAGS=-fPIC LDFLAGS=-fPIC GYP_DEFINES="$GYP_DEFINES target_arch=armv7"  
make && make install  
cp -r lib/* ../lib/zstd
```

4.7.1.2.5 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_CFLA
GS"
```

```
./configure --host=arm-linux CC=arm-linux-gnueabi-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。

```
~ # flash_erase --v
flash_erase (mtd-utils) 2.1.1
Copyright (C) 2000 Arcom Control Systems Ltd

flash_erase comes with NO WARRANTY
to the extent permitted by law.

You may redistribute copies of flash_erase
under the terms of the GNU General Public Licence.
See the file 'COPYING' for more information.
```

4.7.1.3 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
```

4.7.1.4 更新设备树

1. 使用 **cat /proc/mtd** 查看分区情况

```
/usr/app # cat /proc/mtd
dev:   size  erasesize  name
mtd0: 00400000 00020000 "u-boot"
mtd1: 00020000 00020000 "env"
mtd2: 00100000 00020000 "logo"
mtd3: 00100000 00020000 "dtb"
mtd4: 00800000 00020000 "kernel"
mtd5: 1f1e0000 00020000 "rootfs"
```

2. 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。

4.7.1.5 更新内核

```
flash_erase /dev/mtd4 0 0
nandwrite -p /dev/mtd4 /home/root/zImage
sync
```

4.7.2 emmc 的单步更新

4.7.2.1 更新 uboot

```
echo 0 > /sys/block/mmcblk1boot0/force_ro
echo 1 > /sys/block/mmcblk1boot0/force_ro
```

4.8 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，在需要时再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。

5 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。

5.1 参考资料

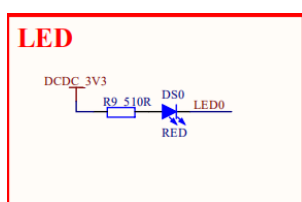
1. 开发板原理图 《IMX6UL_ALPHA_V2.0(底板原理图)》 《IMX6ULL_CORE_V1.4(核心板原理图)》
2. 正点原子《Linux 驱动开发指南说明》 LED 相关章节

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

5.2 LED 硬件接口和 DTS 设备树

5.2.1 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。



UART5_TXD	I2C_SCL	1	2	UART5_RXD	I2C_SDA
UART4_TXD	I2C1_SCL	3	4	UART4_RXD	I2C1_SDA
UART3_RTS	CAN1_RX	5	6	UART3_CTS	CAN1_TX
UART3_RXD		7	8	UART3_TXD	
UART2_CTS	CAN2_TX	9	10	UART2_RTS	CAN2_RX
UART2_RXD	ECSP3_SCLK	11	12	UART2_TXD	ECSP3_SS0
GPIO_8	BLT_PWM	13	14	JTAG_MOD	6D_INT
GPIO_3	LED0	15	16	GPIO_9	CT_INT
GPIO_1		17	18	UART1_CTS	KEY0
GPIO_2		19	20	GPIO_4	
GPIO_0	USB_OTG1_ID	21	22	SNVS_TAMPER2	WIFI_INT
SNVS_TAMPER7	ENET1_RST	23	24	SNVS_TAMPER5	ENET1_INT
BOOT_MODE1		25	26	SNVS_TAMPER3	ENET2_RST
SNVS_TAMPER0	WIFI_REG_ON	27	28	BOOT_MODE0	
ON_OFF		29	30	SNVS_TAMPER1	BEEP
SNVS_TAMPER4	AUD_INT	31	32		

GPIO1_IO00/I2C2_SCL/GPT1_CAPTURE1/ANATOP_OTG1_ID/ENET1_REF_CLK1/MQS_RGHT	K13	GPIO 0	USB_OTG1_ID
GPIO1_IO01/I2C2_SDA/GPT1_COMPARE1/USB_OTG1_OC/ENET2_REF_CLK2/MQS_LEFT	L15	GPIO 1	USB_OTG1_OC
GPIO1_IO02/I2C1_SCL/GPT1_COMPARE2/USB_OTG2_PWR/ENET1_REF_CLK_25M/USDHC1_WP/UART1_TX	L14	GPIO 2	USB_OTG2_PWR
GPIO1_IO03/I2C1_SDA/GPT1_COMPARE3/USB_OTG2_OC/USDHC1_SD_B/UART1_RX	L17	GPIO 3	USB_OTG2_OC
GPIO1_IO04/ENET1_REF_CLK1/PWM3_OUT/USB_OTG1_PWR/USDHC1_RESET/UART5_TX	M16	GPIO 4	USB_OTG1_PWR
GPIO1_IO05/ENET2_REF_CLK2/PWM4_OUT/ANATOP_OTG2_ID/CSI_FIELD/USDHC1_VSELECT/UART5_RX	M17	GPIO 5	SD1_VSELECT
GPIO1_IO06/ENET1_2_MDIO/USB_OTG_PWR_WAKE/CSI_MCLK/USDHC2_WP/UART1_CTS	K17	GPIO 6	ENET_MDIO
GPIO1_IO07/ENET1_2_MDC/USB_OTG_HOST_MODE/CSI_PIXCLK/USDHC2_CD_B/UART1_RTS	L16	GPIO 7	ENET_MDC
GPIO1_IO08/PWM1_OUT/SPDIF_OUT/CSI_VSYNC/USDHC2_VSELECT/GPIO1_IO08/UART5_RTS	N17	GPIO 8	BLT_PWM
GPIO1_IO09/PWM2_OUT/SPDIF_IN/CSI_HSYNC/USDHC1_2_RESET_B/GPIO1_IO09/UART5_CTS	M15	GPIO 9	SD1_nRST

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

5.2.2 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。

5.2.3 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。

5.3 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。

5.4 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 Filesystem)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，这就需要通过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_L
ED_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 succeeded!\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, N
ULL, 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 succeeded!\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();
}

```

5.5 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。

5.6 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。

5.7 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-gnueabi-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：

```

~ # insmod /usr/driver/led.ko
[34714.912596] dev alloc or set ok, major:247, minor:0
[34714.921819] device add Success!
[34714.925450] class create succeeded!
[34714.933733] device create succeeded!
[34714.937452] led write 0
[34714.939733] led hardware init ok
~ # ./usr/app/led_test 1
[34729.617094] led on
~ # ./usr/app/led_test 0
[34731.916131] led off
~ # █

```

5.8 总结

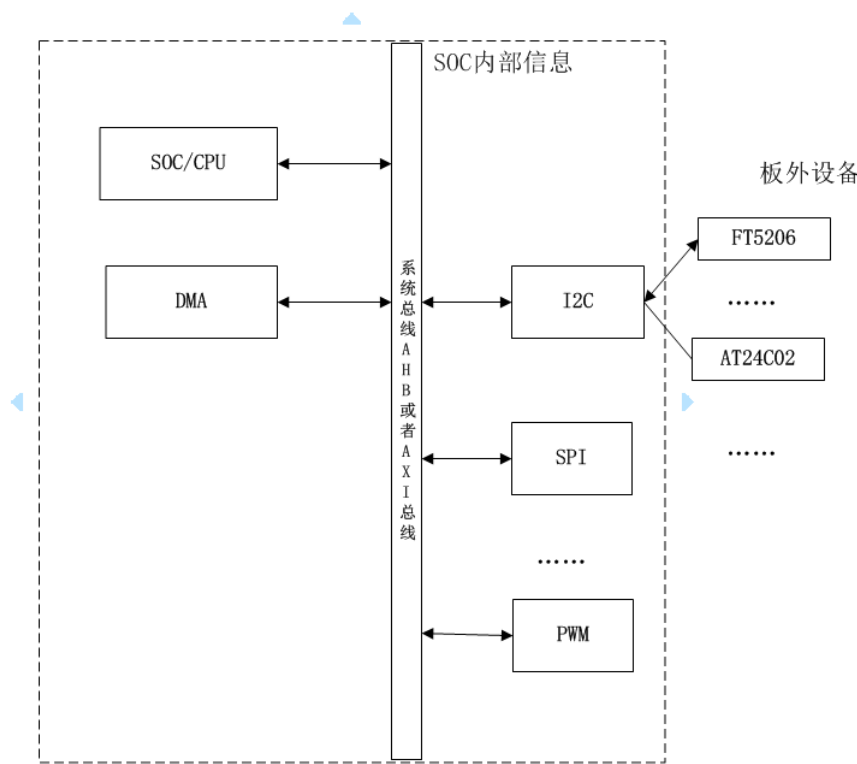
至此，关于 LED 的驱动开发基本讲解完成，虽然开发参考了原子例程用了不到 1 个小时，但完成这篇文档用了 3 个小时，为了能够将知识整理并能够讲解出来，是需要去查询书籍，查看内核代码，分析实现的原理，但是这部分是值得的，通过对完整流程的整理，我对完整的驱动实现和设备树都有了更清晰的认知，很多曾经在开发和学习中不能够理解的问题也被突破，这是具有重要意义的，因为这也是首次接触驱动部分的内容，所以把整个驱动部分进行了比较详细的讲解，后续的

6 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。

6.1 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 `#include` 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。

6.2 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。

6.2.1 #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"
```

6.2.2 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 **clock@ gpio_keys@0**，后面我们就将用到这个说明。

5. 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：

- c. 不同的属性信息进行合并
- d. 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。

6. 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

6.2.3 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。

6.2.3.1 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，

2. 字符串<string>

```
compatible = "simple-bus";
```

这里"simple-bus"就是属性中对应的字符串值。

3. 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

4. 无符号整数<u32>/<u64>

无符号的证书型变量

```
offset = <0x38>;
```

5. 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>;
```

6.2.3.2 常用 key 属性

1. compatible

<stringlist>字符串列表类型

compatible 属性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。

2. 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";
```

3. 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：

Table 2.4: Values for status property

Value	Description
"okay"	Indicates the device is operational.
"disabled"	Indicates that the device is not presently operational, but it might become operational in the future (for example, something is not plugged in, or switched off). Refer to the device binding for details on what disabled means for a given device.
"fail"	Indicates that the device is not operational. A serious error was detected in the device, and it is unlikely to become operational without repair.
"fail-sss"	Indicates that the device is not operational. A serious error was detected in the device and it is unlikely to become operational without repair. The <code>sss-sss</code> portion of the value is specific to the device and indicates the error condition detected.

4. #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` 用来描述子节点中“reg”对应属性中描述地址列表中 cell 数目

`#size-cells` 用来描述子节点中“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。

5. 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
};
```

6. 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。

7. 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。

8. 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 Specification 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

6.3 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：

```
~ # ls /sys/firmware/devicetree/base/
#address-cells      memory
#size-cells         model
aliases            name
backlight          pxp_v4l2
chosen             regulators
clocks             reserved-memory
compatible          soc
cpus               sound
gpio_keys@0        spi4
interrupt-controller@00a01000  usr_gpios
leds
```

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。

6.3.1 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。

6.4 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。

7 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。

7.1 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。

7.2 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。

7.2.1 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

上位机发送数据结构					
起始位 (1Byte)	地址位 (1Byte)	数据编号 (2Byte)	数据长度 (2Byte, 高位在前)	数据区 (n Byte)	错误校验位 (2 Byte)
0x5A	0~255	0~65536	n(范围0~1024)	(n*1)Byte	循环冗余校验 (CRC)

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, CrcCalc;
    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;
}

```

2. 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。

下位机返回数据结构						
起始位(1Byte)	地址位 (1Byte)	数据编号 (2Byte)	ACK(1Byte)	数据长度 (2Byte)	数据区 (n Byte)	错误校验位 (2 Byte)
0x5B	0~255	0~65536	应答状态	ⁿ (范围0~1024)	(n*1)Byte	循环冗余校验 (CRC)

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;
}
```

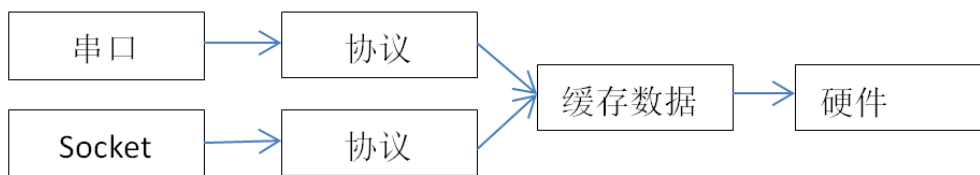
因为是嵌入式 Linux 开发，因此推荐使用 C++，封装可以让代码结构更加清晰，从代码的实现可以看到实现包含：硬件的数据接收，起始位检测，数据编号的获取，以及后续数据的接收和数据的 CRC 校验，至于发送数据，则主要是创建发送数据的接口，这部分即为通讯相关的结构数据实现，通过协议的发送和接收结构的剥离，此时我们就获得了需要处理的实际数据，下面进一步进行数据的处理。

7.2.2 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) == R
T_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

指令(1Byte)	功能	结构
0x01	读内部状态	cmd(1)+reg(2)+size(2)
0x02	写内部状态	cmd(1)+reg(2)+size(2)+data
0x03	上传指令	cmd(1)+size(2)+data
0x04	上传数据	cmd(1)+size(2)+data
其它	/	备用

返回应答数据格式则为

ACK(1Byte)	说明
0x00	数据正常
0x01	无效指令
...	备用
0xFF	其他错误

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：

```
~ # led write:1[ 74.051853] led on
beep write:0[ 74.063585] beep off
led write:0[ 78.878586] led off
[ 78.881743] beep off
beep write:0
■
```

7.3 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。

8 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。

8.1 参考资料

1. [开源 QT 例程项目](#)
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》

8.2 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->addItem(StopBitsList);
ui->combo_box_stop->setCurrentIndex(0);

//校验位
QStringList ParityList;
ParityList<<"N"<<"Odd"<<"Even";
ui->combo_box_parity->addItem(ParityList);
ui->combo_box_parity->setCurrentIndex(0);

//设置协议类型
QStringList SocketTypeList;
SocketTypeList<<"TCP"<<"UDP";
ui->combo_box_socket_type->addItem(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。

8.3 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_pSerialPort
Com->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:disconnect <-> 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：



8.4 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，后续就是在这个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)，即使进一步更新，也不会在本系列文档中添加，不过在应用开发中，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。

9 附录

9.1 如何自学嵌入式 Linux

来源: <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：

9.1.1 嵌入式工作详解

9.1.1.1 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

9.1.1.2 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，更复杂的我并没有涉及

9.1.1.3 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 Java 对底层的操作, 引入了 JNI 的实现)

9.1.1.4 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。

9.1.2 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。

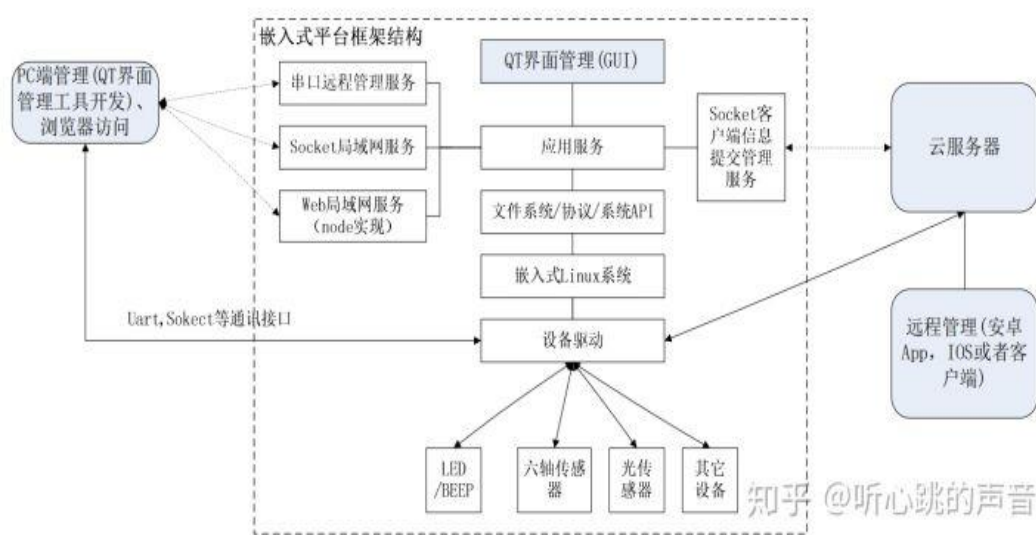
另外对于如何学习这部分内容，我个人认为我一年前所说的方法仍然是合适的：

<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，都是工作中为了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，希望对你有帮助：

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

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

9.2 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，如果问题理解知道原因的话，也会列出相应的原因。

9.2.1 系统问题

1. 系统找不到编译工具(如 gcc, arm-linux-gnueabi-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

2. 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。

3. 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

4. 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

5. 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
```

9.2.2 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。

2. 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

3. **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。