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，仍然停留在初级的水平，没有过系统深入的去总结整理，随着工作年限的递增，越来越感受到这种浮躁感带来的技术面瓶颈。既然发现了问题，自然就要去解决，回想起我踏入嵌入式行业来的经历，正是对 STM32 芯片以及网络部分的学习总结笔记支撑我走到如今的地步，那么沉淀下来，从嵌入式 Linux 入门开始整理，层层深入，对嵌入式 Linux 进行系统的总结也是最符合我目前现状的解决办法，这也是我下定决心放弃日常娱乐，开始本系列的由来。

嵌入式 Linux 的掌握学习是很复杂的过程，从最基础的 Linux 安装，shell 指令的学习和应用，交叉编译环境搭建，C 语言开发，Uboot 启动开发，Linux 内核接口访问和驱动开发，Linux 内核裁剪，Linux 系统接口和应用开发，在掌握了前面所有知识后，对于一个完整的产品，也只是完成了整个项目基础构建，这些知识不仅对于学习是难点，对于已经掌握的人来用文字描述清楚，特别是系统/软件版本引发的编译，调试问题，复杂应用中的指针处理问题，多线程/进程应用中的同步问题，这部分开发经验仅仅用文字描述是很难讲解清楚的，往往只有自己去动手实践，才能够理解，这就对从业者或者入门者有了更高的要求。

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

对于大部分的学习者来说，可能按照如下的流程，从 uboot，驱动，内核开始，先学习外设模块，在理解如何注册字符型设备，然后按照从易到难的顺序在掌握中断和时钟，文件系统，块设备，I2C 驱动，LCD 驱动，摄像头驱动，网络设备驱动，设备树，然后在学习涉及上层的 QT 界面，远程访问的网络 socket(B/S， C/S 框架)，以及应用端的 Android 平台开发，多线程，多进程同步等知识，这也是大部分开发板或者教程的学习方案，可从我经验来看，如果按照上面的流程是可以覆盖嵌入式 Linux 的主要工作需求的(部分知识是溢出的)。但是从产品实践的角度来说，这些事实都还是基础技术的范畴(基础不代表简单)，而不是具体实现产品的方案; 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(可能不符合现有的产品模型)，所有学习都围绕着此产品来开发。这个系列将不仅仅讲述学习嵌入式，而且也讲述我根据工作积累的开发经验，讲述如何完成项目中的思考，也方便了解嵌入式软件开发的工作是什么，这部分也是目前嵌入式方向比较欠缺的经验分享。

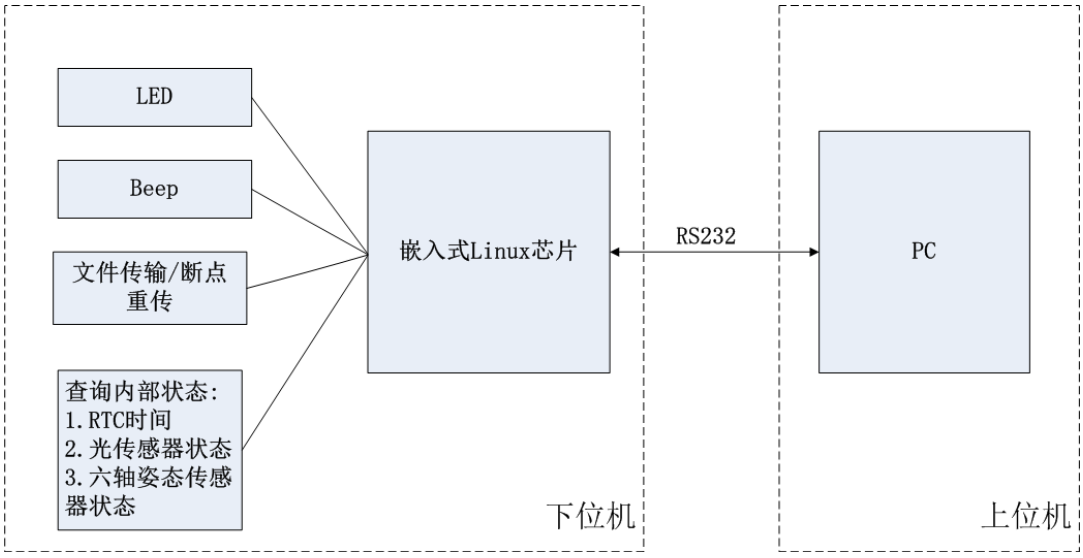
对于本系列中可能存在的问题，如需要反馈，可通过 QQ: 1107473010 或者对应 QQ 邮箱联系。

2 目录

3 项目架构初探

3.1 系统架构

任何项目的发展都是先构建整体的框架，在分解为具体的模块任务，最后组合起来，进行综合的应用调试的，如何将项目分解成具体的任务分配当然是重要的知识，但这部分工作的完成是需要多种技术支撑的，既然本章只是初探，这里就不在深究，把完整项目中本地管理分解出来，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模型。



3.2 硬件说明

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

3.3 代码路径

详细代码见:https://github.com/zc110747/remote_manage

3.4 功能说明

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

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

3.能够查询内部的一些数据，除显示已经列出状态外，支持后期扩展查询其他状态

3.5 任务分解

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

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

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

3.6 内核模块初探

本节作为整个系列的起点，重点当然是上面的项目规划和任务分解，不过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。下面代码可以在 PC 端 Ubuntu 系统上验证，事实上 PC 端的 Ubuntu 可以验证很多实现，如加**载驱动和设备，实现 QT 界面，进行网络通讯的应用端测试**，所以一定不要忽略这个优势，本小节的代码都是在 PC 端测试完成，用于体验内核模块开发的特征。作为内核模块，可以通关 Kernel 编译时加入到内核中，也可以通过 insmod/rmmod 动态的加载到系统中，为了满足 Linux 系统的访问，内核模块就需要实现接口用于 Linux 访问，开发者只要按照规则用 C 语言实现这些需要的接口，在按照一定的规则编译后，就可以使用 lsmod/rmmod 来加载和移除驱动模块，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 内核的相关接口。

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 必须模块

模块加载函数：

```
module_init(func)
```

模块卸载函数:

```
module_exit(func)
```

模块许可声明：

```
MODULE_LICENSE("xxx") 支持的许可有:
"GPL", "GPL V2", "GPL and additional right", "Dual BSCD/GPL", "DUAL MPL/GPL", "Proprietary"
```

3.6.3 可选模块

模块参数 -- 模块加载时传递变量 module_param(name, charp, S_IRUGO);

模块导出符号 --用于将符号导出，用于其它内核模块使用。

```
EXPORT_SYSMBOL(func)/EXPORT_SYSMBOL_GPL(func)
```

注意:Linux 内核 2.6 增加了函数校验机制，后续使用 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。

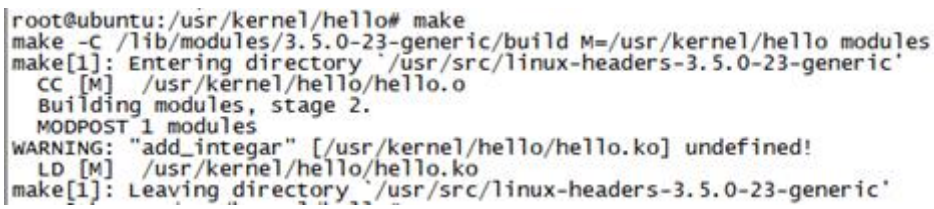
在了解上述模块的基础上，就可以实现如下的模块代码：

```
//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
```

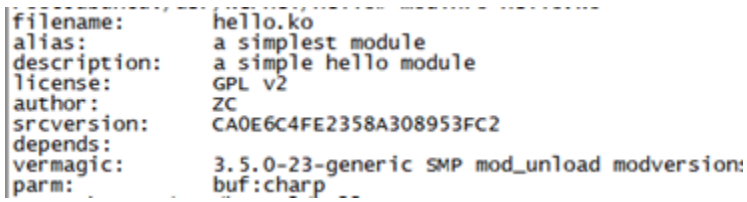
使用 Makefile 文件如下：

```
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
```

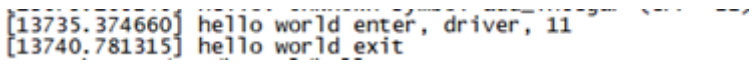
保存后，使用 make 指令既可以编译，如果遇到编译错误，请先查看文章最后的备注，未包含问题请搜索或者留言，编译结果如图所示。



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。



3.6.4 内核模块的跨模块调用

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

对于跨核模块调用的实现，对于调用的模块，主要包含 2 步：

在代码实现中添加 extern int add_integar(int a, int b);

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

对于被链接的模块，代码实现如下:

```
//math.ko
```

```
#include <linux/init.h>
#include <linux/module.h>

static int __init math_init(void)
{
	printk(KERN_WARNING "math enter\n");
	return 0;
}
module_init(math_init);

static void __exit math_exit(void)
{
	printk(KERN_WARNING "math exit\n");
}
module_exit(math_exit);

int add_integar(int a, int b)
{
		return a+b;
}
EXPORT_SYMBOL(add_integar);

int sub_integar(int a, int b)
{
		return a-b;
}
EXPORT_SYMBOL(sub_integar);

MODULE_LICENSE("GPL V2");
```

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

```
oot@ubuntu:/usr/kernel/hello# grep integar /proc/kallsyms
ffffffffa026e040 r __ksymtab_sub_integar	[math]
ffffffffa026e07d r __kstrtab_sub_integar	[math]
ffffffffa026e058 r __kcrctab_sub_integar	[math]
ffffffffa026e030 r __ksymtab_add_integar	[math]
ffffffffa026e089 r __kstrtab_add_integar	[math]
ffffffffa026e050 r __kcrctab_add_integar	[math]
ffffffffa026d000 T add_integar	[math]
ffffffffa026d010 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。

3.7 备注

1.**内核编译名称必须为 Makefile，否则编译会出错**

```
make[2]: *** No rule to make target `/usr/kernel/hello/Makefile'.  Stop.
make[1]: *** [_module_/usr/kernel/hello] Error 2
make[1]: Leaving directory `/usr/src/linux-headers-3.5.0-23-generic'
```

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

3.Makefile 中，指令必须以 Tab 对齐，否则编译会异常。

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

5.内核跨文件访问接口

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

```
0x13db98c9      sub_integar     /usr/kernel/math/math     EXPORT_SYMBOL
0xe1626dee      add_integar     /usr/kernel/math/math     EXPORT_SYMBOL
否则编译时报警告
WARNING: "add_integar" [/usr/kernel/hello/hello.ko] undefined!
安装模块时出错
[ 9091.025357] hello: no symbol version for add_integar
[ 9091.025360] hello: Unknown symbol add_integar (err -22)
```

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，发现命令行用起来也十分的爽快，对于命令行的学习基本没有捷径，无论是鸟哥的私房菜，还是专门讲述 Shell 语法的书籍，最后归结到原点，还是要多加练习，并整理总结，这是从菜鸟到高手的必经之路，下面正式开始这部分的讲解吧。

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 eth0 up
ifconfig eth1 up
mkdir -p filepath 创建路径，可递归创建
apt-get install filename 安装指定文件
alias ll='ls -alF' 重新定义指令
ls /dev/* 查询当前的设备
ls /dev/sd* 查询当前的是的 sd 卡设备
ps -a | grep ssh* 查询当前执行的后台应用， grep 可以限制名称
kill -9 id 关闭指定 ID 的后台应用
tar -xvf filename 解压到当前文件夹，后面可指定目录
tar -vcjf filename.tar.bz2 * 压缩目录下文件和文件夹到指定的文件
cat /proc/devices 查询当前的设备总线
scp -r file_name system_usr@ip_addr:/filepath
例如: scp -r uart_proto root@192.168.1.251:/usr/app
通过 ssh 快速上传文件到指定地址
insmod/rmmod/modprobe/lsmod 加载/删除/带关联加载/显示内核模块
modinfo xx.ko 列出模块的信息
mknode /dev/... c main_id slave_id
例如: mknod /dev/led c 1 0
根据主从设备号创建设备节点
ls /proc/slabinfo 查看内存占用情况
free -m 查询内存的占用
ln -s 原始路径 链接路径 **生成文件链接，用于其它方式的访问**

4.2 参考资料

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

4.3 交叉编译环境构建

在上一章我们已经根据项目需求确定了后续的实现目标。

熟悉硬件开发平台，完成交叉环境编译环境的构建，然后进行嵌入式 Linux 系统执行需要的编译，下载和执行过程，这部分的内容当然是十分复杂的，如何选择合适版本的编译器添加到系统中，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，这些都是整个嵌入式生涯中需要去了解并掌握的知识。但是在本项目中，这部分的流程其实不影响整个项目的开发，从工程思维来说我们是不能够在这部分花太多时间，并不是它们不重要，而是对于产品来说，这部分是成体系且完善的东西，不应该在最初的时候花费太大的精力去理解细节，某些时候使用官方或者开发板厂商提供的资源包，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。

按照上面的思维，本系列编译将会使用正点原子提供的修改后的内核和系统文件，仅后续会根据开发的需求，裁剪和修改设备树或者内核的内容，另外某些驱动章节参考原子的驱动文档会讲解的更详细些，这里因为篇幅原因只讲解流程中用到的知识，如果后续设计需要，则会根据应用需求来裁剪，

在大致理解上述资料后，就可以开始正式的交叉编译环境构建了：

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

并根据 Ubuntu 系统位数的不同选择指定的编译器，如我安装的系统是 64 位，选择 gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf.tar.xz 文件

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

```
mkdir -p /usr/local/arm
```

3. 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

```
root@ubuntu:/usr/local/arm# ls
gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf.tar.xz
root@ubuntu:/usr/local/arm#
```

4. 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：

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

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-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

6. 最后通过指令，即可查看是否安装成功

```
arm-linux-gnueabihf-gcc  -v
```

```
root@ubuntu:/usr/local/arm# arm-linux-gnueabihf-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gnueabihf-gcc
COLLECT_LTO_WRAPPER=/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabih
Target: arm-linux-gnueabihf
Configured with: /home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trust
me/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/t
wg-make-release/label/docker-trusty-amd64-tcwg-build/target/arm-linux-gnueabihf/_buil
sty-amd64-tcwg-build/target/arm-linux-gnueabihf/_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-libstdcxx-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，交叉环境的编译即搭建完成，因为以应用为目的，所以后续都以开发板提供的工具为准，如果想了解编译器和下面的安装包，建议详细去看《Linux 驱动开发指南说明》的指定章节。

4.4 Uboot 编译和测试

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

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(DDR3/256M，NAND/512M)规格为例，使用的默认配置文件名称为 **mx6ull_14x14_ddr256_nand_defconfig，** 使用如下指令进行编译。

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- mx6ull_14x14_ddr256_nand_defconfig
```

执行结果如下，即用于编译的.config 的文件,最后执行编译指令:

```
make V=1 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j12
```

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

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

即可实现下载，结果如下：

```
root@ubuntu:/usr/code/uboot# ./imxdownload u-boot.bin /dev/sdb
I.MX6UL bin download software
Edit by:zuozhongkai
Date:2018/8/9
Version:V1.0
file u-boot.bin size = 471748Bytes
Delete old load.imx
Create New load.imx
Download load.imx to /dev/sdb  ......
927+1 records in
927+1 records out
474820 bytes (475 kB, 464 KiB) copied, 7.57574 s, 62.7 kB/s
```

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

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

```
U-Boot 2016.03-g9bd38ef (Oct 22 2019 - 18:15:59 +0800)

CPU:   Freescale i.MX6ULL rev1.1 69 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 41C
Reset cause: POR
Board: MX6ULL 14x14 EVK
I2C:   ready
DRAM:  256 MiB
NAND:  512 MiB
MMC:   FSL_SDHC: 0
*** Warning - bad CRC, using default environment

Display: ATK-LCD-4.3-800x480 (800x480)
Video: 800x480x24
```

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。

4.5 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
```

在 400 多行添加对应的设备树，如我的开发板对应：

```
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-lcdif.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-certi.dtb \
422         imx6ull-14x14-nand-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-gnueabihf-  distclean
make  ARCH=arm  CROSS_COMPILE=arm-linux-gnueabihf-  imx_v7_defconfig
make  ARCH=arm  CROSS_COMPILE=arm-linux-gnueabihf-  menuconfig
```

执行结果如，即用于编译的.config 的文件,最后执行编译指令:

```
make  V=1  ARCH=arm  CROSS_COMPILE=arm-linux-gnueabihf-  all  -j16
```

如此即完成了编译，编译结果如下：

```
  arm-linux-gnueabihf-gcc -Wp,-MD,arch/arm/boot/compressed/.piggy.lzo.o.d  -nostdinc -isystem /usr/local/arm/g
clude -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-gnueabihf-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/bswapsdi2.o -o arch/arm/boot/compressed/vmlinux
  arm-linux-gnueabihf-objcopy -O binary -R .comment -S  arch/arm/boot/compressed/vmlinux arch/arm/boot/zImage
root@ubuntu:/usr/code/linux# ls
```

编译之后文件路径在

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

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

4.6 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
```

指令解压，解压后结果如下:

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

```
164  CROSS_COMPILE  ?=  /usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf
/bin/arm-linux-gnueabihf-
190  ARCH  ?=  arm
```

4. 使用指令进行将链接编译到指定的路径

```
make  install  CONFIG_PREFIX=/usr/code/rootfs/nfs/
```

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，参考文档 38.2.3 说明。

```
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
```

```
mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts

echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

创建文件 etc/fstab，内容

```
#<system> <mount point> <type> <options> <dump> <pass>
proc /proc proc defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
sysfs /sys sysfs defaults 0 0
```

创建 etc/inittab 文件，内容

```
#etc/inittab
::sysinit:/etc/init.d/rcS
console::askfirst:-/bin/sh
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
::shutdown:/sbin/swapoff -a
```

然后使用打包指令

```
tar -vcjf rootfs.tar.bz2 *
```

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

4.7 Mgtool 文件下载更新

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

完成上述所有流程，我们就获得了最基础的底层应用结构，包含:

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

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

文件系统 -- rootfs.tar.bz2

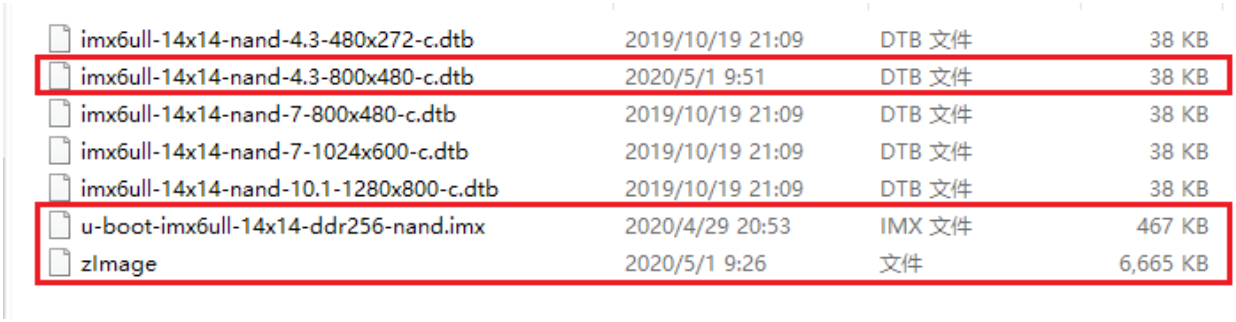
有了上述软件，就可以进行后续的代码下载，参考 39.5 章节说明：

1. MgTool 的工具使用原子提供，路径为 A 盘>5.开发工具>4.正点原子修改过的 MFG_TOOL 烧写工具>mfgTool，通关将拨码开关置到仅 2 为高，然后复位，使用 Mfgtool2-NAND-ddr256-NAND.vbs 指令先下载测试。
2. MgTool 的下载分为两部分，
   - ✧ 将 Profiles/Linux/OS Firmware/firmware 下的文件下载到 DRAM 中，在跳转执行系统
   - ✧ 与 DRAM 中运行的系统交互，将 Profiles/Linux/OS Firmware/files 内的文件通过 UTP 通讯使用指令将数据更新到 NandFlash 中

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

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：



   filesystem 下直接用 rootfs 替换，

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

```
241     <CMD state="Updater" type="push" body="send" file="files/modules/modules.tar.bz2" ifdev="MX6ULL">Sending Modules file</CMD>
242     <CMD state="Updater" type="push" body="$ tar jxf $FILE -C /mnt/mtd5/lib/modules/" ifdev="MX6ULL">tar Modules file</CMD>
243     <CMD state="Updater" type="push" body="$ sleep 1">delay</CMD>
```

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

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

```
[    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-B0D6D9FFFBDF, small LPT m
[    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
/ #
```

4.8 单步更新的方法说明

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

4.8.1 获取安装的资源包

wget ftp://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 git://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git

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

4.8.2 交叉编译资源包

4.8.2.1 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-gnueabihf-gcc ./configure --prefix=/usr/code/mtd_utils/lib/zlib --static
make && make install
```

4.8.2.2 交叉编译 lzo

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

4.8.2.3 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
```

4.8.2.4 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
```

4.8.2.5 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_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。

```
~ # 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.
```

4.8.3 单步新设备树到开发板中

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.

4.8.4 单步更新 uboot 到开发板中

```
flash_erase  /dev/mtd0  0  0
kobs-ng  init  -x  -v  --chip_0_device_path=/dev/mtd0  u-boot-imx6ull-14x14-ddr256-nand.imx
sync
```

4.8.5 单步内核到开发板中

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

4.9 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，体验了编译 uboot，内核和文件系统，并进行了烧录，但如果对比本节的内容和参考文档的相关说明，就会发现我省略了很多内容，举几个例子，**官方的 Uboot 是如何变成原子的修改后的 Uboot，如何通过 make menuconfig 配置需要的内核**，还有对 Makefile 以及 Shell 命令部分也只是浅尝辄止，这些部分对嵌入式重要吗，事实上很重要，未来的很多时候都要和这些知识打交道，但当你当刚踏入嵌入式学习的时候，去深入钻研，一方面没有概念，难以建立清晰的脉络认知，另一方面这部分是基石，知识繁杂且耗时，学习不简单，也很难出成功有正面反馈，这也是我曾经学习嵌入式遇到的最大障碍。正是这种经验让我总结了，重心是快速进入应用和驱动开发，结合实际的需求，在需要时去深入理解掌握的方法，而且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

在上一章中，构建了能够在开发板平台的完整系统，也学会了如何通过单步更新来进行模块化更新。基础已经满足了，那么对于任务的开发，下一步要做什么？这里不卖关子了，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。**

基于在第 2 章列出的任务模型，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

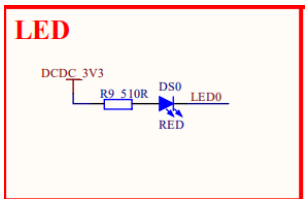
1. 开发板原理图 《IMX6UL_ALPHA_V2.0(底板原理图)》 《IMX6ULL_CORE_V1.4(核心板原理图)》
2. 正点原子《Linux 驱动开发指南说明》 LED 相关章节

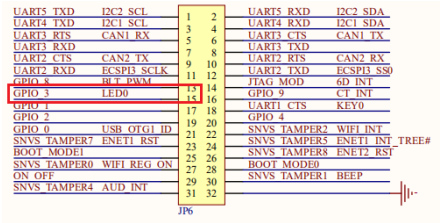
3. 宋宝华 《Linux 设备驱动开发详解：基于最新的 Linux 4.0 内核》 第六章 字符驱动设备
4. Devicetree Specification Release v0.2

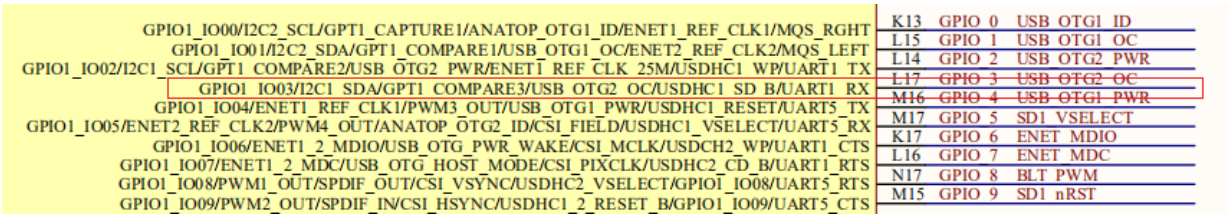
5.2 LED 硬件接口和 DTS 设备树

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。







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，另外模拟其它根节点下的模块实现 LED 的板级添加并不困难，在根节点/下面添加

```
/*led add by zc*/
led {
```

```
    compatible = "gpio-led";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_gpio_leds>;
    led-gpio = <&gpio1 3 GPIO_ACTIVE_LOW>;
    status = "okay";
  };
```

如此，便完成了板级的添加，这里不详细讲述，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。

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 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_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 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, 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 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();
}
```

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;
```

```
    fd = open("/dev/led", O_RDWR | O_NDELAY);
    if(fd == -1)
    {
        printf("/dev/led open error");
        return -1;
    }

    if(argc > 1){
        val = atoi(argv[1]);
    }
    write(fd, &val, 1);
    close(fd);
}
```

使用指令

```
arm-linux-gnueabihf-gcc xxx.c –o xxx
```

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

```
~ # 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 successed!
[34714.933733] device create successed!
[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，也希望看到文章的你也有类似的收获，这也是学习和分享的意义所在。

6 UART 应用测试开发