# 前言

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

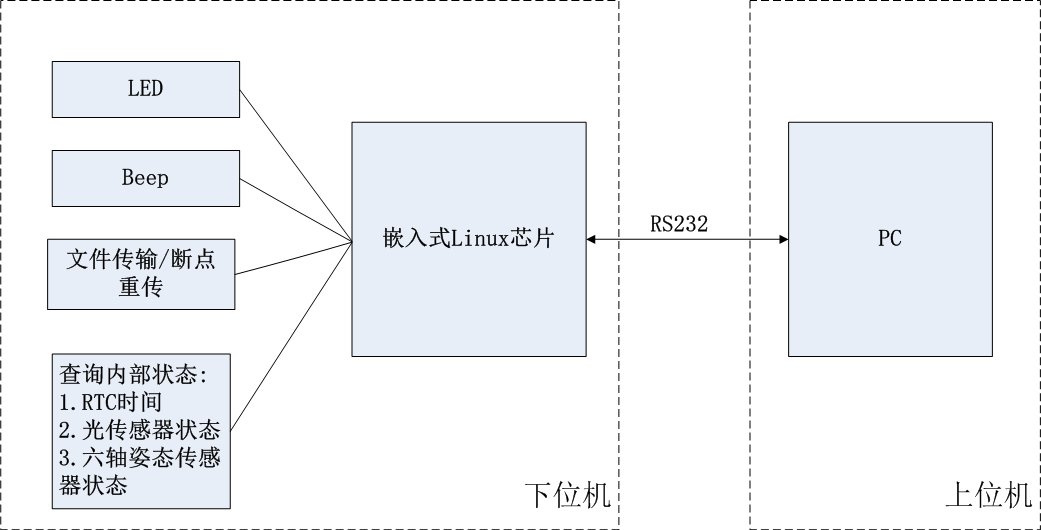
嵌入式Linux的掌握学习是很复杂的过程，从最基础的**Linux安装**，**shell指令**的学习和应用，**交叉编译环境搭建**，**C语言开发**，**Uboot启动维护**，**Linux内核接口**，**Linux应用系统接口**，在掌握了前面所有知识后，对于一个完整的产品，也只是完成了整个项目基础构建，这些知识不仅对于学习是难点，对于已经掌握的人来用文字描述清楚，特别是系统/软件版本引发的编译，调试问题，复杂应用的指针处理问题，这部分开发经验是文字很难描述的，往往只有自己去动手实践，才能够理解，这种复杂度对于入门者门槛是相当高的。

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

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

# 局域网管理设备项目

## 系统架构



## 硬件说明

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

## 代码路径

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

## 软件说明

　　1.上位机软件支持串口通讯，双机通讯需要制定协议(可使用自定义协议或者Modbus)，支持界面化管理(目前定义使用QT开发, 与后续的完善计划有关)  
　　2.支持文件传输，文件传输支持断点重传(传输后文件位于指定文件夹，初步定义为/usr/download)  
　　3.能够查询内部的一些数据，除显示已经列出状态外，支持后期扩展查询其他状态

## 任务分解

　　1. uboot，内核和文件系统的编译，下载和调试，并集成ssh方便传输应用文件调试  
　　2. 分模块完成驱动的开发调试，不过为了方便测试及后期集成，需要同步完成串口驱动，串口通讯协议定义及上位机的软件框架  
　　3. 后期的综合性功能调试和应用开发(如协议扩展问题，状态查询到界面显示，考虑到协议数据的复用, 后期该数据可能用于网页界面的状态显示或者QT界面的控制)

## 参考资料

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

## 内核模块初探

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

### 必须模块

模块加载函数：

module\_init(func)

模块卸载函数:

module\_exit(func)

模块许可声明：

MODULE\_LICENSE("xxx") 支持的许可有:

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

### 可选模块

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

　　模块导出符号 --用于将符号导出，用于其它内核模块使用。

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

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

　　模块作者 -- MODULE\_AUTHOR("xxx")

　　模块描述 -- MODULE\_DESCRIPTION("xxx")

　　模块版本 -- MODULE\_VERSION("xxx")

　　模块别名 -- MODULE\_ALIAS("xxx")

模块设备表 -- MODULE\_DEVICE\_TABLE

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

在了解上述模块的基础上，就可以实现如下的模块代码：

//hello.ko

#include <linux/init.h>

#include <linux/module.h>

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

static char \*buf = "driver";

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

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

{

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

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

return 0;

}

module\_init(hello\_init); //模块加载函数

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

{

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

}

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

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

MODULE\_LICENSE("GPL v2"); //模块许可协议

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

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

使用Makefile文件如下：

ifeq ($(KERNELRELEASE),)

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

PWD := $(shell pwd)

modules:

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

modules\_install:

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

clean:

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

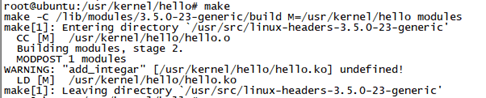
.PHONY:modules modules\_install clean

else

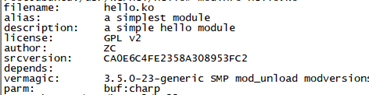
obj-m :=hello.o

endif

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



之后执行指令modinfo hello.ko即可查看当前的模块信息。



如果无法查看信息，可通过dmesg查看加载信息。

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

### 内核模块的跨模块调用

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

对于跨核模块调用的实现，对于调用的模块，主要包含2步：

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

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

对于被链接的模块，代码实现如下:

//math.ko

#include <linux/init.h>

#include <linux/module.h>

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

{

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

return 0;

}

module\_init(math\_init);

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

{

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

}

module\_exit(math\_exit);

int add\_integar(int a, int b)

{

return a+b;

}

EXPORT\_SYMBOL(add\_integar);

int sub\_integar(int a, int b)

{

return a-b;

}

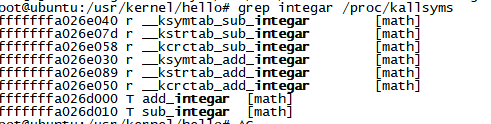
EXPORT\_SYMBOL(sub\_integar);

MODULE\_LICENSE("GPL V2");

编译Makefile同上，需要将obj-m :=hello.o修改为obj-m :=math.o

执行make编译完成该文件，并通过insmod加载完模块后，可通过

grep integar /proc/kallsyms 查看加载在内核中的符号，状态如下：



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

## 备注

1.内核编译名称必须为Makefile，否则编译会出错

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

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

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

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

3.Makefile中，指令必须以Tab对齐，否则编译会异常。

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

5.内核跨文件访问接口

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

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

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

否则编译时报警告

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

安装模块时出错

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

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