# 输入系统应用编程

输入事件的设备信息结构体:

<Linux-4.9.88\include\uapi\linux\input.h>

struct input\_event {

struct timeval time;

\_\_u16 type;

\_\_u16 code;

\_\_s32 value;

};

<Linux-4.9.88\include\uapi\linux\time.h>

struct timeval {

\_\_kernel\_time\_t tv\_sec; /\* seconds \*/

\_\_kernel\_suseconds\_t tv\_usec; /\* microseconds \*/

};

同步事件: 表示数据上报完毕, type, code, value三项都是0.

查看设备节点: ls /dev/input/\* -l

确定设备信息：

[root@imx6ull:~]# ls /dev/input/\* -l

crw-rw---- 1 root input 13, 64 Jan 1 00:00 /dev/input/event0

crw-rw---- 1 root input 13, 65 Jan 1 00:00 /dev/input/event1

crw-rw---- 1 root input 13, 63 Jan 1 00:00 /dev/input/mice

/dev/input/by-path:

total 0

lrwxrwxrwx 1 root root 9 Jan 1 00:00 platform-20cc000.snvs:snvs-powerkey-event -> ../event0

[root@imx6ull:~]# cat /proc/bus/input/devices

I: Bus=0019 Vendor=0000 Product=0000 Version=0000

N: Name="20cc000.snvs:snvs-powerkey"

P: Phys=snvs-pwrkey/input0

S: Sysfs=/devices/soc0/soc/2000000.aips-bus/20cc000.snvs/20cc000.snvs:snvs-powerkey/input/input0

U: Uniq=

H: Handlers=kbd **event0** evbug

B: PROP=0

B: EV=3

B: KEY=100000 0 0 0

I: Bus=0018 Vendor=dead Product=beef Version=28bb

N: Name=**"goodix-ts"**

P: Phys=input/ts

S: Sysfs=/devices/virtual/input/input1

U: Uniq=

H: Handlers=**event1** evbug

B: PROP=2

B: EV=b

B: KEY=1c00 0 0 0 0 0 0 0 0 0 0

B: ABS=6e18000 0

[cat /proc/bus/input/devices] 的设备ID由以下结构体来描述:

<Linux-4.9.88\include\uapi\linux\input.h>

struct input\_id {

\_\_u16 bustype;

\_\_u16 vendor;

\_\_u16 product;

\_\_u16 version;

};

查看支持哪些事件: <Linux-4.9.88\include\uapi\linux\input-event-codes.h>

查看触摸屏的输入数据: hexdump /dev/input/event1

0000610 0580 0000 8a7d 0006 0003 0036 01b5 0000

序号 秒 微秒 type code value

APP访问硬件的4种方式:

**①查询**, open传入O\_NONBLOCK

**②休眠-唤醒**, open不传入O\_NONBLOCK, 当read时, 有数据则返回, 无数据则休眠.

**③POLL**, 传入超时时间, 参考APUE P408. select, 参考APUE P405.

#include <poll.h>

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

返回值: 准备就绪的描述符数目; 若超时, 返回0; 若出错, 返回-1.

struct pollfd {

int fd; /\* file descriptor \*/

short events; /\* requested events \*/

short revents; /\* returned events \*/

};

--------------------------------------------

#include <sys/select.h>

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

#include <sys/time.h>

struct timeval {=

long tv\_sec; /\* seconds \*/

long tv\_usec; /\* microseconds \*/

};

返回值: 准备就绪的描述符数目; 若超时, 返回0; 若出错, 返回-1.

nfds是**最大文件描述符编号值加1**.

关键字restrict: 它只可以用于限定和约束指针, 并表明指针是访问一个数据对象的唯一且初始的方式.

**④异步通知**, APP可以忙自己的事, 当驱动程序有数据时会主动给APP发信号.

<Linux-4.9.88\include\uapi\asm-generic\signal.h>

#define SIGIO 29 // 表示有IO事件要处理

#include <signal.h>

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

signal用于注册信号处理函数.

signum: 哪个信号; handler: 信号处理函数.

* 编写信号处理函数.
* 使用signal()注册信号处理函数.
* APP打开设备驱动程序的设备节点, fd = open(argv[1], O\_RDWR).
* APP将自己的进程ID告诉驱动程序, fcntl(fd, F\_SET\_OWN, getpid()).

#include <sys/types.h>

#include <unistd.h>

pid\_t getpid(void);

* APP设置flag, 告诉驱动程序如何收信号, FASYNC启用异步通知 flags = fcntl(fc, F\_GETFL), fcntl(fd, F\_SETFL, flags | FASYNC).

**一个BUG**: 代码<11\_input/05\_input\_read\_fasync.c>如果printf最后不加\n则系统打印会出异常, 百问网提供的代码也一样.

while(1){

printf("cnt = %d\n", cnt++); // \n必须有 否则打印异常

sleep(2);

}

原因: printf()碰到\n才会将信息刷到标准输出上.

# TSLIB

tslib官网: <http://www.tslib.org/>

TSLIB\_TSDEVICE: 该环境变量指定触摸屏设备.

TSLIB\_CONFFILE: 该环境变量指定触摸屏配置文件, 该配置文件指定读入哪些module.

INPUT\_PROC\_DIRECT: 触摸屏设备的属性.

调用ts\_open(), 构造出一个tsdev结构体, 调用ts\_config(), 读取配置文件指定的模块, 每一个模块都插入链表头部. 所以配置文件中最后一个指定的模块将会最先得到执行, 但是模块的执行是按照链表顺序递归调用的, 所以配置文件中的第一个指定的模块将会是最后执行, 但也是最先被执行完. module插入tsdev.list, moudle\_raw插入tsdev.list\_raw, 一般只有一个module\_raw.

一个配置文件的例子: </etc/ts.conf>

module\_raw input

module pthres pmin=1

module dejitter delta=100

module linear

## IMX6ULL交叉编译tslib:

①tar xJf tslib-1.21.tar.xz

②cd tslib-1.21/

③./configure --host=arm-linux-gnueabihf --prefix=/ # freetype的安装是使用prefix指定tmp目录

④make

⑤make install DESTDIR=$PWD/tmp

⑥$echo 'main(){}' | arm-linux-gnueabihf-gcc -E -v -

头文件目录选择: /home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../arm-linux-gnueabihf/libc/usr/include

库文件目录选择: /home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin/../arm-linux-gnueabihf/libc/usr/lib/

⑦cd tmp/

⑧cp include/\* <所选择的头文件目录>

⑨cp -d lib/\*so\* <所选择的库文件目录>

## 测试tslib:

将整个tmp文件夹拷贝到板子上:

(1)cp -rf tmp/ ~/nfs\_rootfs/

(2)在板子/mnt目录执行cp tmp/lib/\*so\* -d /lib

(3)在板子/mnt目录执行cp tmp/bin/\* /bin

(4)在板子/mnt目录执行cp tmp/etc/ts.conf -d /etc

关闭IMX6ULL板子上的QT GUI

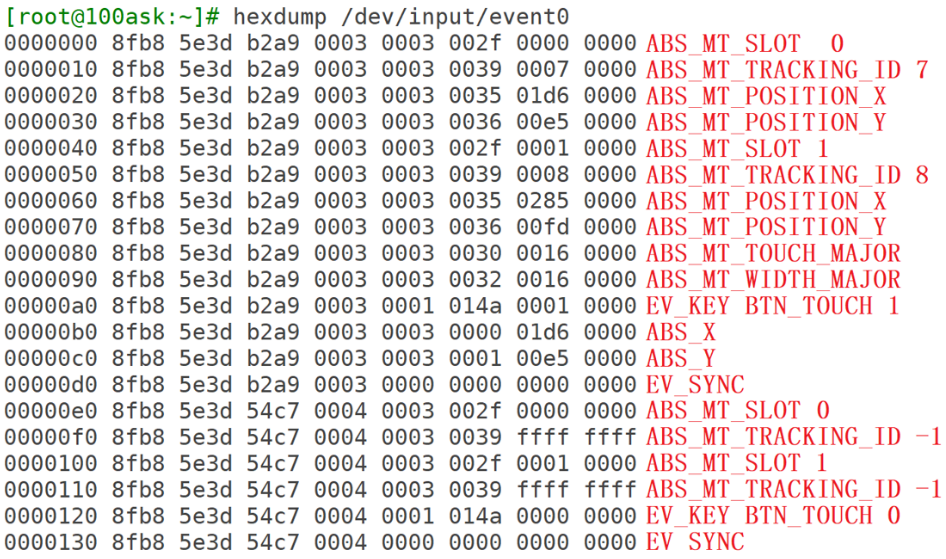
(1)mv /etc/init.d/S07hmi /root/

(2)reboot

运行测试程序(不需要输入./): ts\_test\_mt

## 自己编写tslib的测试程序

点击触摸屏, 返回以下数据:



驱动程序使用slot和tracking\_id来标识一个触点, 当tracking\_id等于-1时, 表示这个触点松开了.

程序的编写参考<tslib-1.21/tests/ts\_test\_mt.c>

tslib的重要头文件: <tslib-1.21\src\tslib.h>

struct ts\_sample\_mt {

/\* ABS\_MT\_\* event codes. linux/include/uapi/linux/input-event-codes.h

\* has the definitions.

\*/

int x;

int y;

……

};

TSAPI struct tsdev \*ts\_setup(const char \*dev\_name, int nonblock);

TSAPI int ts\_read\_mt(struct tsdev \*, struct ts\_sample\_mt \*\*, int slots, int nr);

TSAPI int ts\_close(struct tsdev \*);

<Linux-4.9.88\include\uapi\linux\input.h>

struct input\_absinfo {

\_\_s32 value;

\_\_s32 minimum;

\_\_s32 maximum;

\_\_s32 fuzz;

\_\_s32 flat;

\_\_s32 resolution;

};

ts\_setup(NULL, 0); // 0表示以阻塞方式打开

TSAPI int ts\_fd(struct tsdev \*);

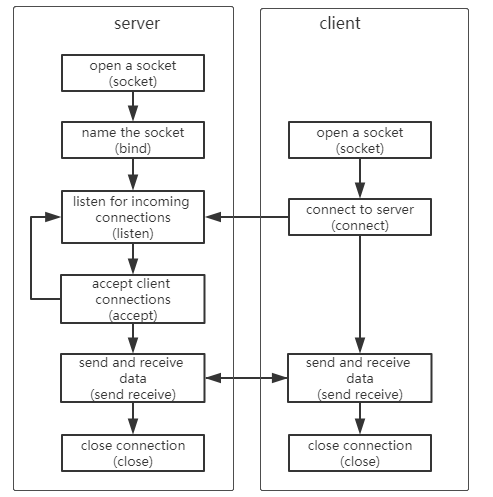
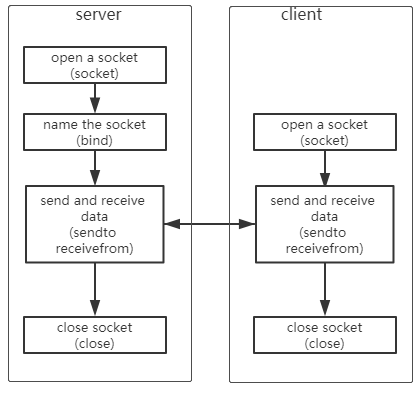
struct input\_absinfo slot;

ioctl(ts\_fd(ts), EVIOCGABS(ABS\_MT\_SLOT), &slot); max\_slots = slot.maximum + 1 - slot.minimum; // 获取最大触点数量

编译(记得加库): arm-linux-gnueabihf-gcc -o mt\_cal\_distance mt\_cal\_distance.c **-lts**

# 网络通信

服务器通过源端口来区分一个IP下的两个连接. http服务:80, ssh服务:22.

TCP UDP

## 编程接口

#include <sys/types.h>

int socket(int domain, int type, int protocol); // APUE P474

返回值: 若成功, 返回文件(套接字)描述符; 若出错, 返回-1.

domain: 地址族, 常使用AF\_INET或AF\_INET6.

type: 套接字类型, UDP使用SOCK\_DGRAM, TCP使用SOCK\_SREAM.

protocol: 通常为0, 表示为给定的域和套接字使用默认协议.

#include <sys/socket.h>

struct sockaddr {

sa\_family\_t sa\_family;

char sa\_data[14];

}

考虑兼容性, 更多使用以下结构体:

struct sockaddr\_in{

unsigned short sin\_family;

unsigned short sin\_port;

struct in\_addr sin\_addr;

unsigned char sin\_zero[8];

}

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen); // PUE P486

作用: 将套接字与地址关联.

返回值: 成功返回0, 出错返回-1.

对于因特网域, 如果指定IP地位为**INADDR\_ANY**, 套接字端点可以被绑定到所有的系统网络接口上. 这意味着可以接收这个系统所安装的任何一个网卡的数据包.

int listen(int sockfd, int backlog);

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags, const struct sockaddr \*dest\_addr, socklen\_t addrlen);

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags, struct sockaddr \*src\_addr, socklen\_t \*addrlen);

int close(int fd);

注意: UDP的客户端也需要调用connect(), 但这一步不是真的建立连接, 只是设置套接字的地址信息.

# 多线程编程

编译时要加上多线程库pthread: **-lpthread**

#include <pthread.h>

pthread\_t pthread\_self(void);

作用: 获得调用线程的ID.

## 线程API

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*arg);

作用: 创建新线程.

参数: thread用于保存新线程的ID, 第二和第四个参数一般为NULL, 第三个参数是线程的实体函数, 该函数形式为void \*fun(void \*arg);.

返回值: 若成功,返回0; 若失败返回错误码.

void pthread\_exit(void \*retval);

作用: 线程主动退出, 在退出时, 将一个void \*类型的的数据带给主线程;

int pthread\_cancel(pthread\_t thread);

作用: 线程被动退出.

int pthread\_join(pthread\_t thread, void \*\*retval);

作用: 以阻塞的方式回收线程资源.

#define \_GNU\_SOURCE

int pthread\_tryjoin\_np(pthread\_t thread, void \*\*retval);

int pthread\_timedjoin\_np(pthread\_t thread, void \*\*retval, const struct timespec \*abstime);

作用: 以非阻塞方式回收线程资源.

返回值: 成功返回0, 出错返回错误码.

## 互斥量API

pthread\_mutex\_t mutex;

int pthread\_mutex\_init(phtread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*restrict attr);

作用: 初始化互斥量, 通常第二, 第三个参数都为NULL.

返回值: 成功返回0.

调用宏快速初始化互斥量: pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

作用: 以阻塞的方式, 互斥量加锁与解锁.

返回值: 成功返回0.

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

作用: 以非阻塞的方式, 互斥量加锁; 销毁互斥量.

返回值: 成功返回0.

## 信号量API

#include <semaphore.h>

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

int sem\_wait(sem\_t \*sem); // 以阻塞方式获取信号量, 进行sem--

int sem\_trywait(sem\_t \*sem); // 以非阻塞方式获取信号量, 进行sem--

int sem\_post(sem\_t \*sem); // 释放信号量, 进行sem++

int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout);

int sem\_destroy(sem\_t \*sem); // 销毁信号量

## 条件变量API

#include <pthread.h>

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*cond\_attr); // 初始化条件变量

int pthread\_cond\_destroy(pthread\_cond\_t \*cond); // 销毁条件变量

返回值: 成功返回0;

int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);

作用: 等待条件变量, 要结合互斥变量一起使用; 通常使用方法如下:

pthread\_mutex\_lock(&mutex);

pthread\_cond\_wait(&cond, &mutex);

/\*

\*\* 操作临界资源

\*/

pthread\_mutex\_unlock(&mutex);

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

作用: 通知条件变量, 只会唤醒一个等待cond条件变量的线程.

# LINUX设备驱动

## HELLO驱动

参考D:\document\100ask\_imx6ull\_2020.02.29\_v2.0\Linux-4.9.88\drivers\char\misc.c

fs.h file\_operations

copy\_from\_user

copy\_to\_user

用户层面调用int open(const char \*pathname, int flags, mode\_t mode), 会操作file结构体

file结构体有const struct file\_operations \*f\_op; 这个是驱动程序负责提供

用户层的opne函数最终会将file结构体指针以及i结点传给驱动层的open函数.



提供file\_operations结构体本质上是编写驱动程序的open, read, write, close等函数, 它们的原型如下:

int (\*open) (struct inode \*, struct file \*);

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);

int (\*release) (struct inode \*, struct file \*);

入口函数 static int \_\_init xxx\_init(void), 三件事:

注册驱动register\_chrdev

创建类class\_create

创建设备device\_create

出口函数 static void \_\_exit xxx\_exit(void), 三件事:

销毁设备device\_destroy

销毁类class\_destroy

注销驱动unregister\_chrdev

修饰入口函数和出口函数: MODULE\_INIT(xxx\_init), MODULE\_EXIT(xxx\_exit)

要遵守协议: MODULE\_LICENSE(“GPL”);

MKDEV(major, minor): 可用于确定要被创建的设备的主设备号和次设备号.

安装驱动: insmod <drvname>, 安装驱动程序会导致xxx\_init()被调用

卸载驱动: rmmod <drvname>, 卸载驱动程序会导致xxx\_exit()被调用

查看是否有装载成功: cat /proc/dev/ 或 lsmod

查看内核打印信息: dmsg

打开内核的打印信息: echo "7 4 1 7" > /proc/sys/kernel/printk

关闭内核的打印信息: echo 0 > /proc/sys/kernel/printk

## LED驱动框架

用户层: ./ledrtest /dev/led<n> <on | off>

应用层: open(“/dev/led<n>”, …), write(fd, &val, 1);

驱动层:

led\_open: 根据子设备号确定哪个LED, 使能GPIO模块, 配置引脚方向为OUPUT.

led\_write: 根据子设备号确定哪个LED, 根据val设置引脚的电平.

硬件层: board\_a.c board\_b.c, 提供board\_led\_init(), board\_led\_ctrl()函数. 抽象出led\_operations结构体.

## 具体的LED驱动

io端口重映射：

D:\document\100ask\_imx6ull\_2020.02.29\_v2.0\Linux-4.9.88\arch\arc\include\asm\io.h

#include <asm/io.h>



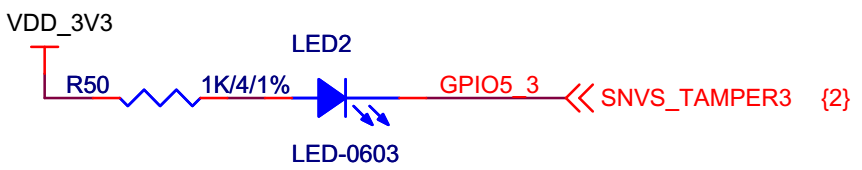


应用层: led\_drv\_test.c

驱动层: led\_drv.c

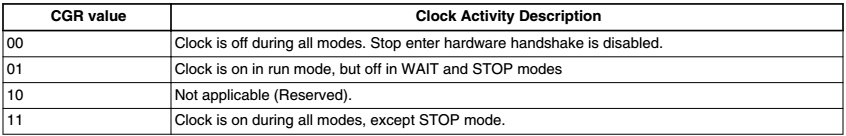
硬件层: board\_100ask\_imx6ull.c, 提供硬件的底层操作init(), ctrl().

im6ull板子上的LED2在GPIO5\_3引脚上， 该引脚的名称是SNVS\_TAMPER3

第一步， 通过时钟控制模块CCMG的CCGR1寄存器的[31 : 30]来使能GPIO5, 从下图可知应该设置成2’b11.

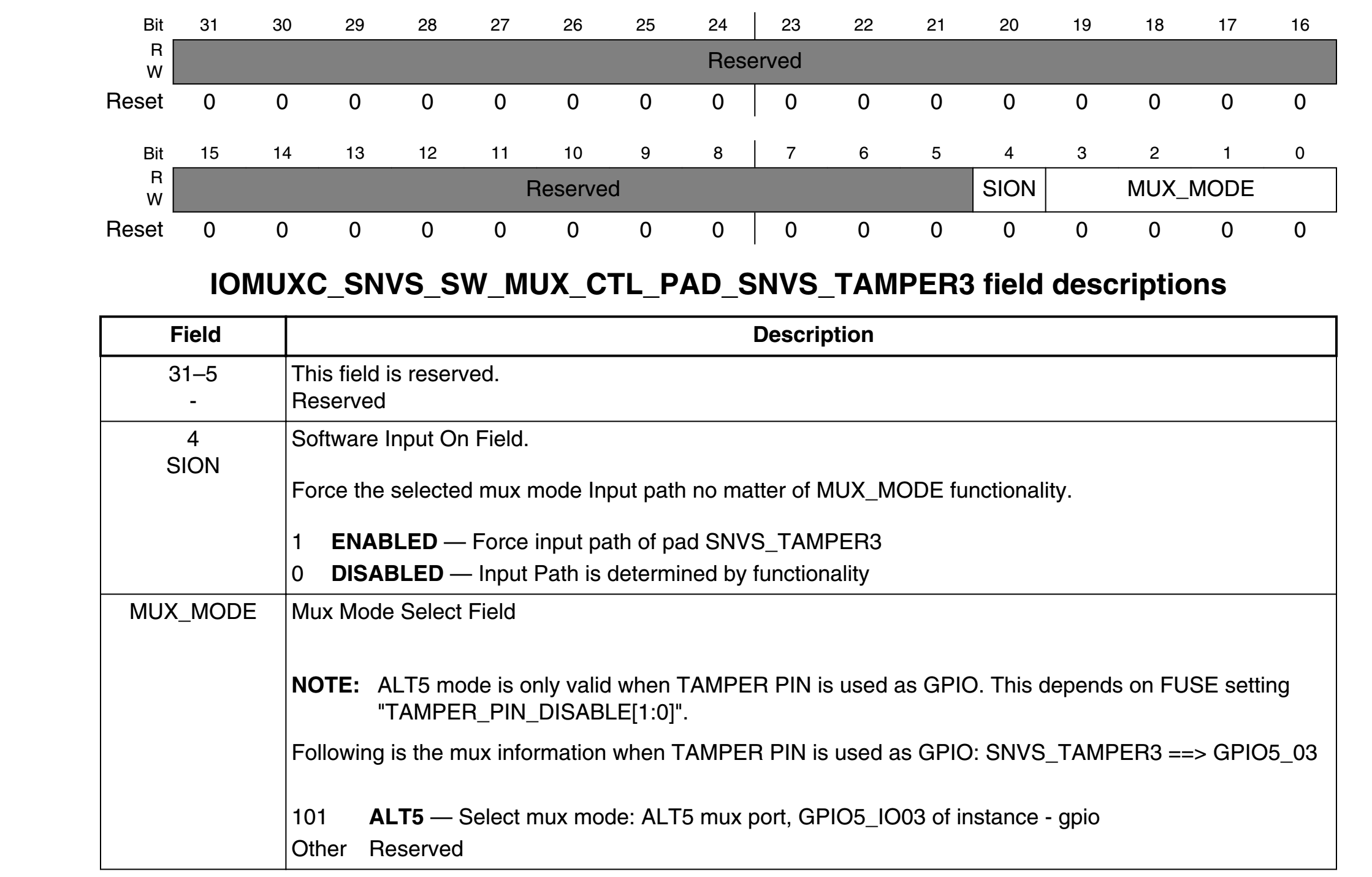
CCM\_CCGR1的地址: 20C\_4000h base + 6Ch offset = 20C\_406Ch



第二步, 设置GPIO5\_IO03引脚为GPIO模式. 该引脚的名称是SNVS\_TAMPER3, 所以要设置IOMUXC\_SNVS\_SW\_**MUX**\_CTL\_PAD\_SNVS\_TAMPER3.

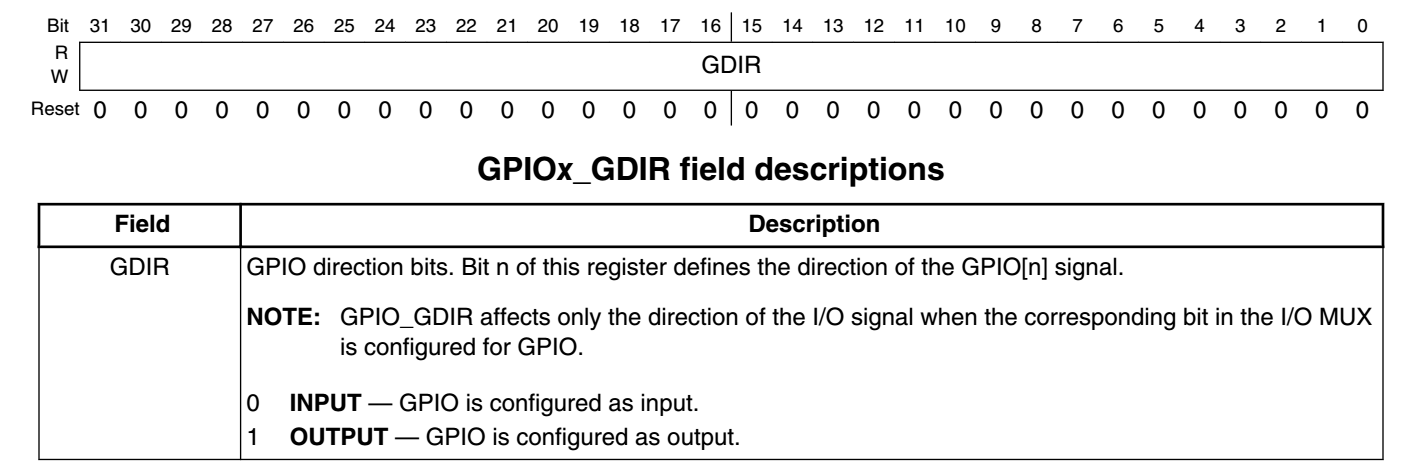
根据下图, 将[3 : 0]设置成4’b0101即可.

地址: 229\_0000h base + 14h offset = 229\_0014h.



第三步, 将GPIO5\_IO03引脚的控制方向设置为输出. 将GPIO方向寄存器GPIO5\_GDIR[3]设置为1’b1.

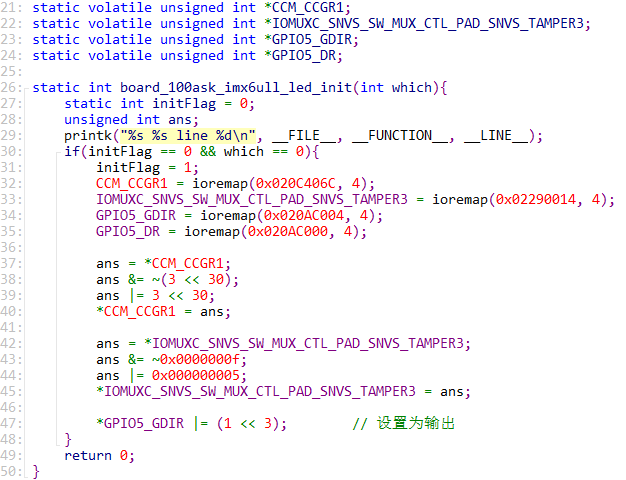
地址: 20A\_C004, 参考表格<GPIO memory map>

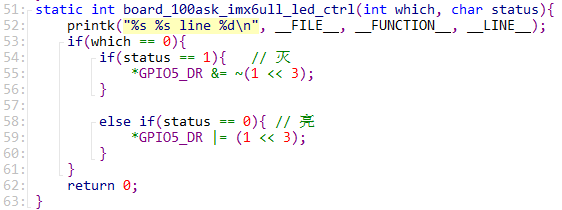


第四步, 写GPIO5\_DR寄存器, 控制LED亮灭. 将GPIO5\_DR[3]设置为1’b0或1’b1.

地址: 20A\_C000, 参考表格<GPIO memory map>

核心代码展示:





## 面向对象分层分离的方法:

应用层: led\_drv\_test.c

驱动层: led\_drv.c

硬件层: chip\_demo\_gpio.c led\_opr.h | board\_A\_led.c led\_resource.h

硬件操作 硬件资源

这里的硬件层将前面的硬件层代码board\_100ask\_imx6ull.c(具体单板的硬件)拆分成硬件(具体芯片)操作和硬件(具体单板)资源两部分.

## 总线设备驱动模型

struct platform\_device, platform\_device\_register, 设备链表, 放入链表后去驱动链表找匹配项, 比较成功则调用drv->probe()

struct platform\_driver, platform\_driver\_register, 驱动链表, 放入链表后去设备链表找匹配项, 比较成功则调用drv->probe()

device和driver用相同的name来匹配.

xxx\_device.ko调用入口函数来注册资源.

xxx\_driver.ko调用入口函数来注册驱动.

驱动层: 做传统的事情, 注册字符设备

probe()函数要做的事情: 1. 记录资源; 2. 调用device\_create().

返回设备中某类资源的第几个:

struct resource \*platform\_get\_resource(struct platform\_device dev, unsigned int type, unsigned int num);

返回设备所用的第几个中断:

int platform\_get\_irq(struct platform\_device \*dev, unsigned int num);

struct bus\_type platform\_bus\_type里面有match函数, 该函数用来找匹配项, register()函数最终导致match()函数调用

定义IO资源的文件: Linux-4.9.88\include\linux\**ioport.h**

chip\_demo\_gpio.c要用到上层leddrv.c提供的led\_device\_create()函数和led\_device\_destroy()函数.

[D:\document\100ask\_imx6ull\_2020.02.29\_v2.0\shareddir\mycode\driver\02\_led\_drv\04\_led\_drv\_template\_bus\_dev\_drv](file:///D:\document\100ask_imx6ull_2020.02.29_v2.0\shareddir\mycode\driver\02_led_drv\04_led_drv_template_bus_dev_drv)

|  |  |
| --- | --- |
| **命令** | **执行函数顺序** |
| insmod led\_drv.ko | led\_drv\_init |
| chip\_demo\_gpio.ko | chip\_demo\_gpio\_drv\_init  register\_led\_operations |
| insmod board\_A\_led.ko | led\_dev\_init  chip\_demo\_gpio\_probe  led\_class\_create\_device \*2 |
| ./led\_test /dev/imx\_led0 on | led\_drv\_open  board\_demo\_led\_init  led\_drv\_write  board\_demo\_led\_ctrl |
| rmmod board\_A\_led.ko | led\_dev\_exit  chip\_demo\_gpio\_remove  led\_class\_destroy\_device  led\_class\_destroy\_device |
| rmmod chip\_demo\_gpio.ko | chip\_demo\_gpio\_drv\_exit |
| rmmod led\_drv.ko | led\_drv\_exit |

## 设备树

dts 🡪 dtb 🡪 内核, 设备树用来指定资源.

查看设备树: ls /sys/firmware/devicetree

该目录下是以目录结构呈现的dtb文件, 根节点对应base目录, 每个节点对应一个目录, 每一个属性对应一个文件.

单板启动时, uboot先运行, uboot将内核和设备树加载入内存, 并把设备树在内存的地址告诉内核.

设备树文件布局:

/dts-v1/; // 表示版本

[memory reservations] // 格式: /memreserve/ <addr> <size>

/{

[property definitons]

[child nodes]

};

设备树结点的表示方法:

[label:]node-name[@unit-address]{

[properties definitions]

[child nodes]

};

label的作用是为了更方便的引用node

属性有3种形式:

1. 用**双引号**括起来的是字符串

2. 用**尖括号**括起来的是无符号32位数值

3. 用**中括号**括起来的是无符号16进制8位数值

编译设备树(在内核源码目录中执行): make dtbs V=1

dtb中每一个节点都被转换为device\_node结构体

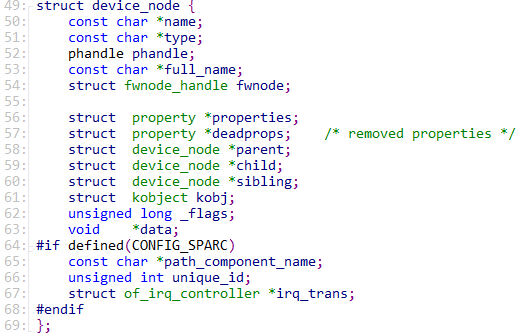
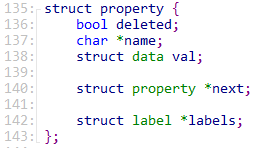
根节点被保存在全局变量**of\_root**中, 从of\_root可以访问到任意节点.

Linux-4.9.88\drivers\of\base.c

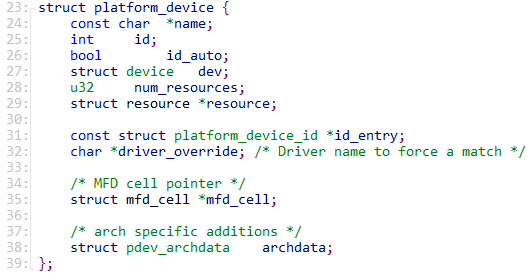
Linux-4.9.88\include\linux\of.h

Linux-4.9.88\scripts\dtc\dtc.h

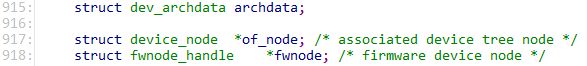


Linux-4.9.88\include\linux\platform\_device.h



Linux-4.9.88\include\linux\device.h



会被转换为platform\_device的节点:

A. 根节点下含有compatible的子节点

B. 含有特定compatible属性的节点的子节点: “simple-bus”, “simple-mfd”, “isa”, “arm,amba-bus”.

C. 总线I2C, SPI下的**子节点**不会被转换为platform\_device.

(/i2c节点会被转换为platform\_device, 它表示I2C控制器)

(/i2c/at24c02不会被转换为platform\_device, 它一般由父节点的platform\_driver创建一个i2c\_client)

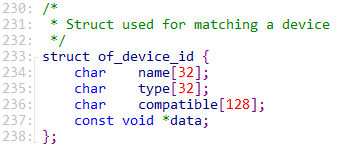
转换为platform\_device的过程:

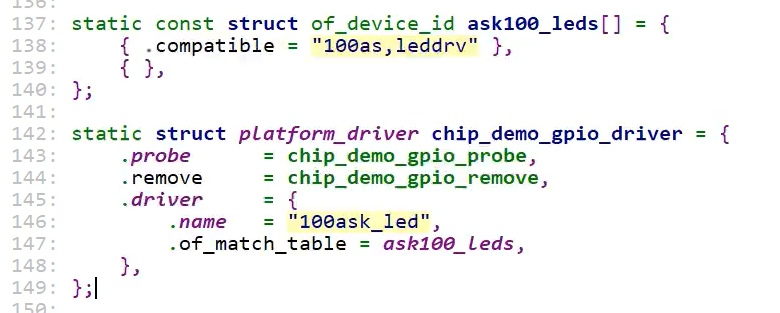
A. platfrom\_device中含有resource数组, 它来自device\_node的reg, interrupts属性. reg属性转换为MEM资源, interrupts属性转换为IRQ资源.

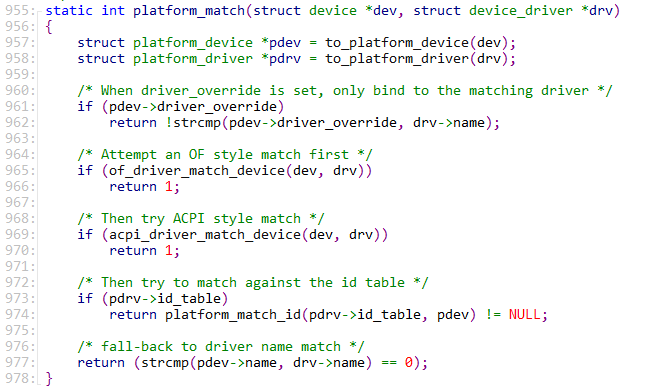
B. platform\_device.dev.of\_node指向device\_node, 可通过它获得其它属性

比较设备树信息:

Linux-4.9.88\drivers\base\platform.c

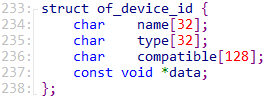






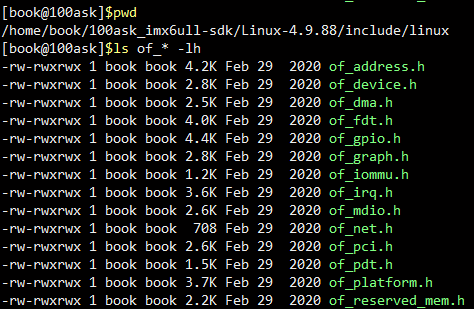
Linux-4.9.88\include\linux\mod\_devicetable.h

**struct of\_device\_id**



没有转换为platform\_device的节点也可以直接访问设备树:

of开头的文件表示**open\_firmware**, 即开放固件, 这些头文件在/Linux-4.9.88/include/linux/.

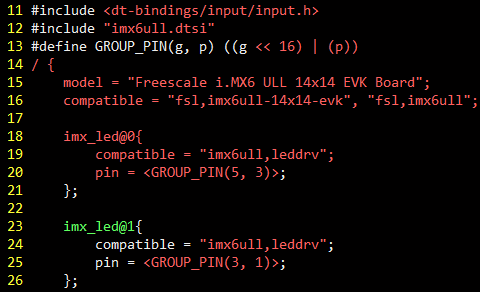
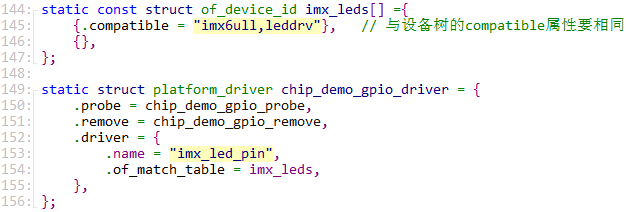


设备树参考文档: Linux-4.9.88/Documentation/devicetree/bindings

编程:

改造chip\_demo\_gpio\_probe, chip\_demo\_gpio\_probe

不再需要board\_A\_led.c

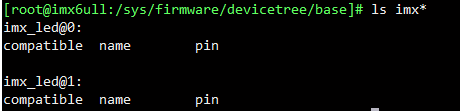
 \



在linux内核顶层目录下编译设备设: make dtbs

然后将编译好的.dtbs文件拷贝到单板的/boot/目录下, 并且**reboot**单板.

查看自己定义的设备树节点: cd /sys/firmware/devicetree/base/



## 按键驱动

阻塞, 非阻塞, 休眠, poll, 异步通知

drv\_open要:

配置GPIO为输入引脚

注册按键中断服务程序

中断服务程序:

保存按键信息

发送信号kill\_fasync(SIGIO)

驱动程序的drv\_fasync由系统间接调用, 启动调用的方法时是设置FASYNC标识: fcntl(fd, F\_SETFL, Oflags | FASYNC);

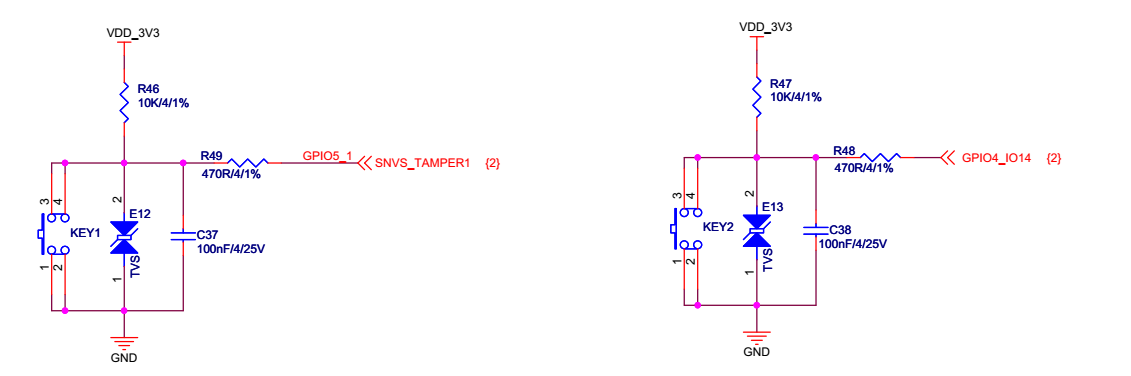
drv\_fasync只做一件事: 记录进程的ID.

drv\_read:

若由按键数据则直接返回

若无数据则睡眠

100ask\_imx6ull的两个用户按键:



KEY1 – GPIO5\_IO01 – SNVS\_TAMPER1

KEY2 – - GPIO4\_IO14

## pinctrl-gpio子系统

参考文档(在内核): Documentation\devicetree\bindings\pinctrl\pinctrl-bindings.txt

pin controller: 复用引脚, 配置引脚

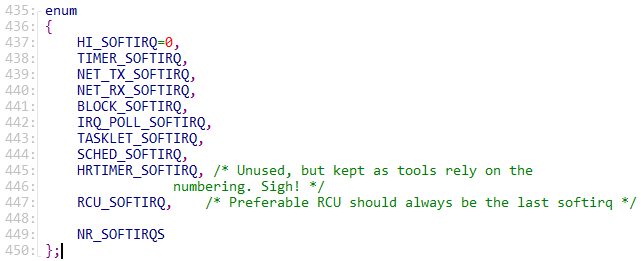
client devicw: 声明要使用哪些引脚, 哪些功能, 如何配置.

#include <linux/gpio/consumer.h> // descriptor-based

#include <linux/gpio.h> // legacy

## 中断

软件中断: Linux-4.9.88\include\linux\interrupt.h



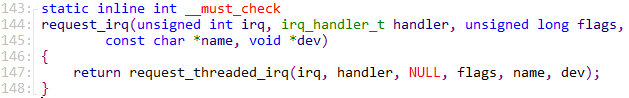
设置软件中断处理函数:

extern void open\_softirq(int nr, void (\*action)(struct softirq\_action \*));

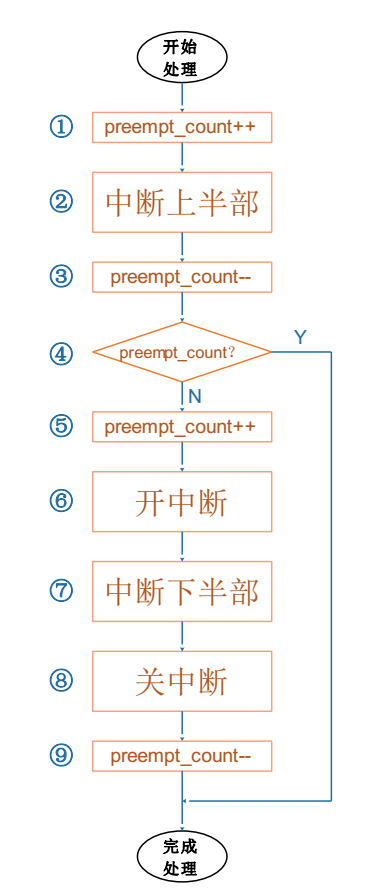
触发软件中断:

extern void raise\_softirq(unsigned int nr);

注册硬件中断: request\_irq()



中断下半部使用软件中断来实现.



tasklet: 处理耗时不长的事情, 使用软中断实现, 中断类型为TASKLET\_SOFTIRQ

同一个中断, 上半部多次执行, 下半部只执行一次.

不同的中断, 上半部分开执行, 下半部合并执行.

tasklet执行结束前, APP是不能执行的.

work\_queue(工作队列): 中断事件耗时, 那就不用软件中断来做, 而是使用内核线程. 在中断上半部唤醒内核线程, 内核线程和APP竞争执行.

这个内核线程是kworker, 它去工作队列取出一个个work, 执行里面的函数.

D:\document\100ask\_imx6ull\_2020.02.29\_v2.0\Linux-4.9.88\include\trace\events\workqueue.h

查看kworker线程: ps -A | grep kworker

a> 创建work:

**static DECLARE\_WORK(work结构体, work函数);**

b> 如何提交work结构体:

**schedule\_work(&work结构体)**

该函数将work结构体提交给系统没默认的work队列system\_wq.

c> 执行work: 提交work结构体的时候会唤醒kworker线程, 该线程取出队列中的work, 执行里面的函数.

d> 谁提交: 中断上半部调用schedule\_work函数.

新技术: 使用thread\_irq来处理中断(request\_irq()就是调用这个函数的)

**int request\_threaded\_irq(unsigned int irq, // 哪个中断**

**irq\_handler\_t handler, // 上半部函数, 可以为空**

**irq\_handler\_t thread\_fn, // 在线程里执行的函数(request\_irq为NULL)**

**unsigned long irqflags, // 中断标识, 设置发生中断的条件**

**const char \*devname, // 中断名称**

**void \*dev\_id // 传给中断例程的参数**

**);**

wait\_event\_interruptiable(wq, condition)

等待@condition为真, 或接收到一个信号.

驱动程序poll

驱动层的poll函数只做两件事:

1. 把线程放入wq, 但是不休眠

2. 返回event状态

## 常用头文件与符号

#include <linux/module.h>

#include <linux/platform\_device.h>

#include <linux/device.h>

#include <linux/slab.h>

#include <linux/of\_gpio.h>

#include <linux/interrupt.h>

#include <linux/fs.h>

file\_operations结构体

# 鸿蒙移植

## 使用SAMBA连接ubuntu

使用SAMBA连接ubuntu, 192.168.44.129是虚拟机的IP地址:

(1)win + r,

(2)\\192.168.44.129,

连接后, 可以在windows的窗口操作ubuntu中的文件, 路径: 网络\192.168.44.129.

增加写权限, 使得windows可以写ubuntu中的目录: sudo chmod -R go+rwx /home/book

将share\_directory映射到本地磁盘硬盘:

(1)对share\_directory右键, 选择[映射网络驱动器(M)…]

(2)驱动器选择[Y:]

于是鸿蒙源码就在这个目录: Y:\openharmony

创建Source Insight的工程目录:

(1)在Y:\openharmony目录下创建目录si, 工程文件都放在si文件夹.

(2)在source insight中新建一个工程, 将driver, kernel/liteos-a, vendor/nxp这三个目录添加到si的目录树中.

(3)project->synchronice解析文件.

注意: 要将.s文件类型添加到搜索范围.

Kconfig是用来生成配置文本文件.

make menuconfig 则会解析Kconfig文件, 然后生成.config文件.

grep “FS\_NFS” \* -nr | grep Kconfig

>> fs/nfs/Kconfig:1:config FS\_NFS

所做的所有配置都保存在内核根目录下的.config文件, 会加上前缀**LOSCFG\_**.

makefile里面设置了export CONFIG\_=LOSCFG\_

MODULE\_NAME = $(notdir $(shell pwd))

LOCAL\_SRCS := $(wildcard os\_adapt/\*.c)

LOCAL\_SRCS += $(wildcard $(LITEOSTHIRDPARTY)/FatFs/source/\*.c)

LOCAL\_INCLUDE指定头文件路径:

LOCAL\_INCLUDE := -I $(LITEOSTHIRDPARTY)/FatFs/source -I $(LITEOSTOPDIR)/fs/fat/virpart/include

LOCAL\_FLAGS指定编译选项: LOCAL\_FLAGS := $(LOCAL\_INCLUDE) $(LITEOS\_GCOV\_OPTS)

子目录下的Makefile包含了内核顶层目录下的config.mk:

include $(LITEOSTOPDIR)/config.mk

最后一行: include $(MODULE)

MODULE是核心: MODULE=$(MK\_PATH)/module.mk, 实质上就是kernel/liteos\_a/tools/build/mk/module.mk

MODULE文件的依赖路径:

all: $(LIBA) $(LIBSO)

LIBA := $(OUT)/lib/libfat.a

$(LIBA) : $(LOCAL\_OBJ)

$(HIDE) $(OBJ\_MKDIR)

ifeq($(LOCAL\_MODULE),)

$(HIDE) $(AR) $(ARFLAGS) $@ $(LOCAL\_OBJS) # [$@就是libfat.a](mailto:$@就是libfat.a)

如何得到$(LOCAL\_OBJS):

LOCAL\_OBJS := $(patsubst %.c,$(OBJOUT)/%.o,$(LOCAL\_CSRCS))

los\_config.mk文件, 在LITEOS\_BASELIB增加库, 在LIB\_SUBDIRS增加目录.

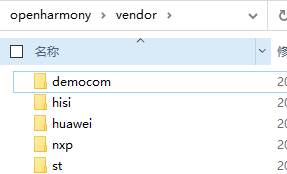
-include有文件就包含, 没有文件就不包含.

all: $(OUT) $(BUILD) $(LITEOS\_TARGET) $(APPS)

## 添加一个单板

假定是使用democom公司的demochip芯片.

先在vendor目录下添加单板相关的文件夹



方法: 复制st文件夹, 改名为democom

在配置界面添加单板

cd kernel/liteos\_a/

pwd: **/home/book/openharmony/kernel/liteos\_a**

如何选择编译什么单板:

**make menuconfig, [Platform -> Board -> Help]**

Symbol: PLATFORM\_STM32MP157

Defined at ../../kernel/liteos\_a/platform/Kconfig:24

添加demochip的选项.

修改makefile

grep “LOSCFG\_PLATFORM\_STM32MP157” \* -nr

涉及三个配置文件:(1)Makefile; (2)platform/Makefile; (3)platform/bsp.mk

**Makefile:**

+69

ifeq ($(LOSCFG\_PLATFORM\_DEMOCHIP), y)

FSTYPE = vfat

ROOTFS\_SIZE = 0xA00000

endif

+119

else ifeq ($(LOSCFG\_PLATFORM\_DEMOCHIP), y)

BOARD\_INCLUDE\_DIR := $(LITEOSTOPDIR)/../../vendor/democom/demochip/board

+168 暂时不管, 只是将liteos转换为bin文件

<顶层Makefile修改完毕>

**platform/Makefile:**

+43

在判断条件里加上$(LOSCFG\_PLATFORM\_DEMOCHIP)

**platform/bsp.mk:**

+66 中断相关的配置

else ifeq ($(LOSCFG\_PLATFORM\_DEMOCHIP), y)

HWI\_TYPE := arm/interrupt/gic

TIMER\_TYPE := arm/timer/arm\_generic

HRTIMER\_TYPE := demochip/hrtimer

+98 与单板相关的头文件

else ifeq ($(LOSCFG\_PLATFORM\_DEMOCHIP),y)

PLATFORM\_INCLUDE += -I $(LITEOSTOPDIR)/../../vendor/democom/demochip/board/include

在内核顶层目录下执行: **make clean**

输出大量错误信息:

make[1]: /home/book/openharmony/kernel/liteos\_a/tools/build/mk/get\_compiler\_path.sh: Command not found

make[1]: /bin/arm-linux-gnueabihf-gcc: Command not found

**grep "get\_compiler\_path.sh" \* -nr**

tools/build/mk/**los\_config.mk**:561:LITEOS\_COMPILER\_PATH := $(shell $(LITEOSTOPDIR)/tools/build/mk/get\_compiler\_path.sh $(CROSS\_COMPILE) $(LITEOSTOPDIR))

**vim tools/build/mk/los\_config.mk +561**

537 ifeq ($(LOSCFG\_COMPILER\_CLANG\_LLVM), y)

538 ifeq ($(LITEOS\_COMPILER\_PATH),)

539 LITEOS\_COMPILER\_PATH := $(shell $(LITEOSTOPDIR)/tools/build/mk/get\_llvm\_compiler\_path.sh $(CROSS\_COMPILE) $(LITEOSTOPDIR))

……

560 ifeq ($(LITEOS\_COMPILER\_PATH),)

561 LITEOS\_COMPILER\_PATH := $(shell $(LITEOSTOPDIR)/tools/build/mk/get\_compiler\_path.sh $(CROSS\_COMPILE) $(LITEOSTOPDIR))

**vim .config**

**/LOSCFG\_COMPILER\_CLANG\_LLVM**

10 # LOSCFG\_COMPILER\_CLANG\_LLVM is not set

**grep "COMPILER\_CLANG\_LLVM" \* -nr | grep Kconfig**

Kconfig:39: default COMPILER\_CLANG\_LLVM

Kconfig:47:config COMPILER\_CLANG\_LLVM

**vim Kconfig +47**

发现是编译器缺少了依赖项, 在49行添加**|| PLATFORM\_DEMOCHIP**

通过**make menuconfig**选择编译器, 然后**make clean**.

再来一次编译: **make -j 16**, 输出错误信息:

make[1]: Entering directory '/home/book/openharmony/kernel/liteos\_a/shell'

full/src/base/shcmd.c:47:10: fatal error: 'hisoc/uart.h' file not found

full/src/base/show.c:43:10: fatal error: 'hisoc/uart.h' file not found

full/src/base/shmsg.c:50:10: fatal error: 'hisoc/uart.h' file not found

full/src/cmds/dmesg.c:68:10: fatal error: 'hisoc/uart.h' file not found

使用vim编辑这四个文件, 添加条件编译:

**vim shell/full/src/base/shcmd.c +47**

**vim shell/full/src/base/show.c +43**

**vim shell/full/src/base/shmsg.c +50**

**vim shell/full/src/cmds/dmesg.c +68**

添加以下内容:

#elif defined LOSCFG\_PLATFORM\_DEMOCHIP

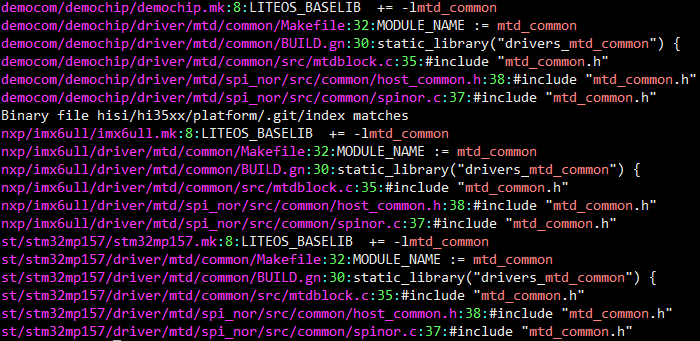
#include "uart.h"

再次**make -j 16**, 输出错误: ld.lld: error: unable to find library -l**mtd\_common**

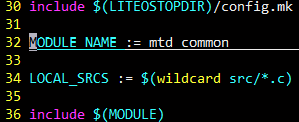
接下来解决链接错误.

**cd ../../vendor**

**grep "mtd\_common" \* -nr**



**vim democom/demochip/driver/mtd/common/Makefile +32**



34行是要将src目录下的.c文件编译成mtd\_common模块.

**vim democom/demochip/demochip.mk +8**



指定了makefile所在的目录, 编译时就会进入这个目录, 使用该目录下的Makefile, 生成mtd\_common文件.

问题: 内核顶层的Makefile怎么才能看到democom/demochip/demochip.mk?

**cd ../kernel/liteos\_a**

**grep "demochip.mk" \* -nr 无输出**

**grep "stm32mp157.mk" \* -nr**

tools/build/mk/**los\_config.mk**:171:include $(LITEOSTOPDIR)/../../vendor/st/stm32mp157/stm32mp157.mk

**vim tools/build/mk/los\_config.mk +171**



添加代码:



openharmony/vendor/democom/demochip目录下的.mk文件要改名成: demochip.mk

去到openharmony/vendor/democom/demochip, 执行

**sudo chmod a+w demochip.mk**

**vim demochip.mk**

**:1,$s/STM32MP157\_BASE\_DIR/DEMOCHIP\_BASE\_DIR/g**

并把第一行的vendor/st/stm32mp157改成/vendor/democom/demochip, 其它内容先不替换.

回到内核顶层目录

**make -j 16, 输出错误:**

make[1]: Entering directory '/home/book/openharmony/vendor/democom/demochip/board'

board.c:20:40: error: use of undeclared identifier 'DDR\_RAMFS\_REAL\_SIZE'

if (add\_mtd\_partition("spinor", 0, DDR\_RAMFS\_REAL\_SIZE, 0))

**cd ../../vendor/democom/demochip/board/**

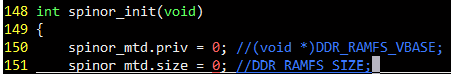
**chmod a+w board.c**

**vim board.c +20**

先把相关代码注释, 暂时不用.

**sudo chmod a+w /home/book/openharmony/vendor/democom/demochip/driver/mtd/spi\_nor/src/common/spinor.c**

**vim /home/book/openharmony/vendor/democom/demochip/driver/mtd/spi\_nor/src/common/spinor.c +150**



**sudo chmod a+w /home/book/openharmony/vendor/democom/demochip/driver/stm32mp157-fb/stm32mp157\_lcd.c**

**vim /home/book/openharmony/vendor/democom/demochip/driver/stm32mp157-fb/stm32mp157\_lcd.c +76**



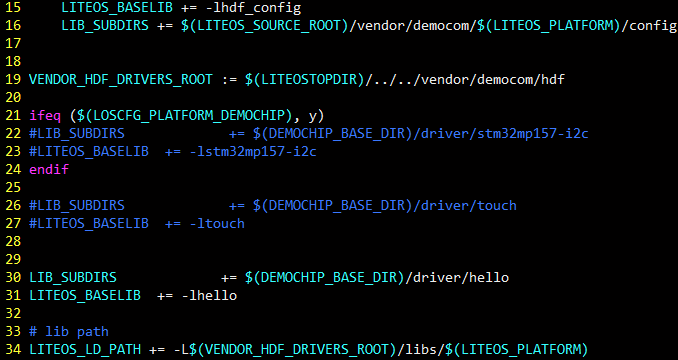
**make -j 16**

ld.lld: error: undefined symbol: HdfGetBuildInConfigData

**sudo chmod a+w /home/book/openharmony/vendor/democom/hdf/hdf\_vendor.mk**

**vim /home/book/openharmony/vendor/democom/hdf/hdf\_vendor.mk**

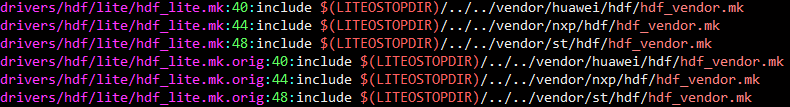
作相应的修改, 适配DEMOCHIP板卡, 22, 23行stm32mp157还没有修改成demochip.



问题: hdf\_vendor.mk如何才能被顶层的Makefile看到?

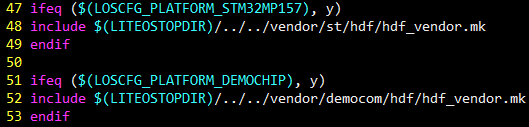
**cd /home/book/openharmony**

**grep "hdf\_vendor.mk" \* -nr**



**vim drivers/hdf/lite/hdf\_lite.mk +48**

添加51~53行代码



**cd kernel/liteos\_a/**

**make -j 16**

**编译成功!**

下一步是修改源文件.

备份一下.config文件: **cp .config tools/build/config/debug/demochip\_clang.config**

## 内存映射

内核的启动流程:

启动: 汇编代码, CPU设置, 代码重定位, 地址映射, 调用main.

main函数: 调用OsMain函数, 最终启动用户进程.

OsMain函数: 操作系统层面的初始化, 比如异常初始化, 任务初始化, IPC初始化.

SystemInit函数: 应用程序初始化, 挂载根文件系统, 启动第一个用户进程.

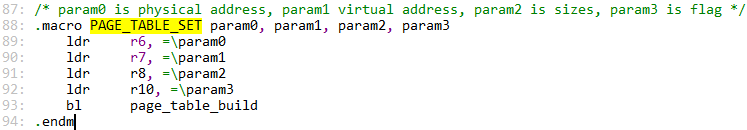
写通(write through): CPU将数据写入cache, 同时立即写入硬件.

写回(write back): CPU将数据写入cache, 当cache满之后, 再将数据写入内存.

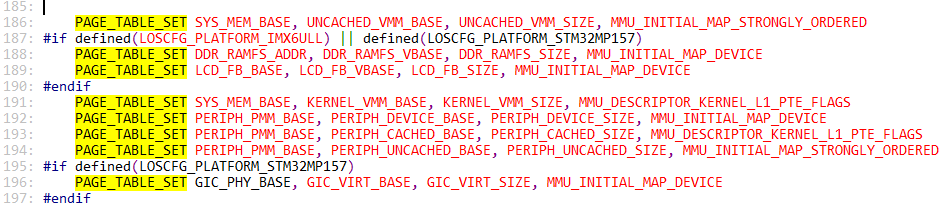
写buf: CPU直接将数据写入buffer, 有buffer来将数据刷到内存, 回导致写合并.

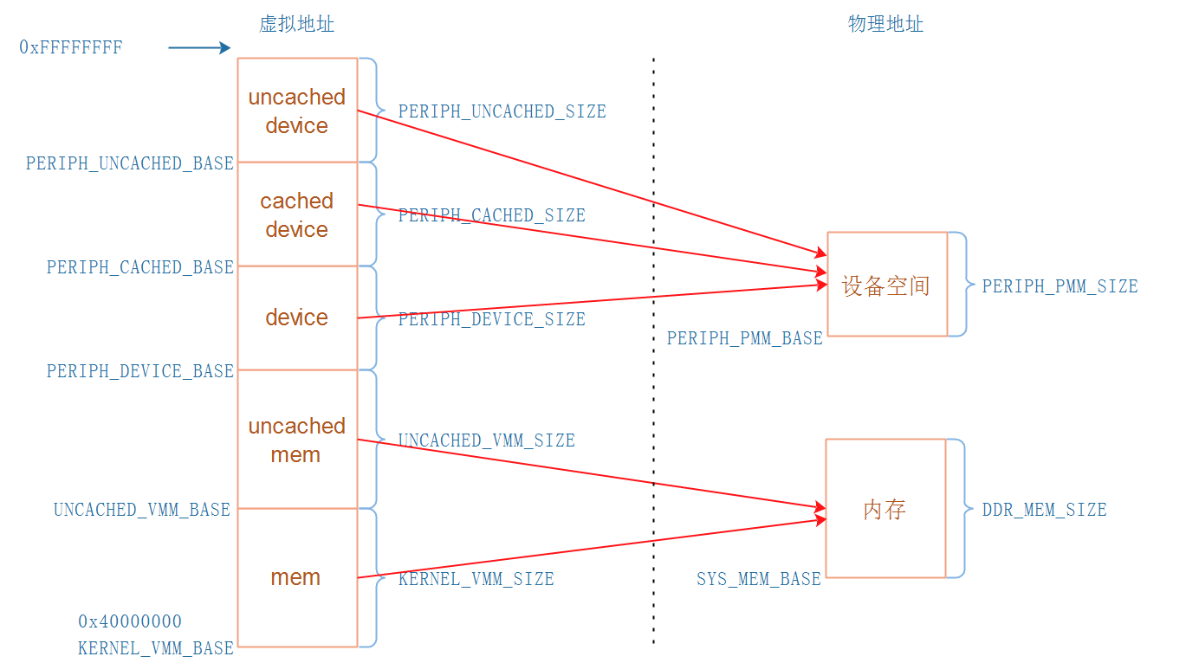
是否使用buffer, 是否使用cache, 组合起来有4种方式, 可查看linux内核的pgtable-2level.h文件.

鸿蒙地址映射在reset\_vector\_up.s



PAGE\_TABLE\_SET(物理地址, 虚拟地址, flag), 将虚拟地址按照flag指定的方式映射到物理地址, flag将决定CPU如何访问内存.





liteos\_a的地址空间如何分配: Y:\openharmony\kernel\liteos\_a\kernel\base\include\los\_vm\_zone.h



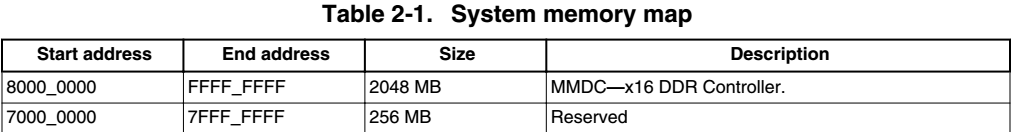
内控空间: 0x4000000 ~ 0xfffffff

用户空间: 0x0000000 ~ 03fffffff

鸿蒙系统对于地址映射的一个缺点: 将所有外设空间当作一个整体, 这很容易导致可用的虚拟地址不够用, 因为很多芯片的外设空间都是保留的. 在映射时应该根据不同的外设分段映射.

修改imx6ull内存映射的代码>

参考文档: IMX6ULLRM.pdf: Chapter 2 Memory Maps.



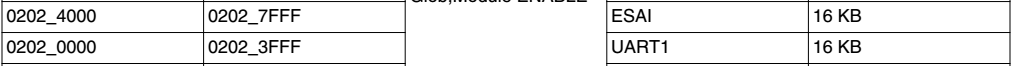
imx6ull的内存起始地址为0x80000000, 实际物理大小为512MB.

**vim ../../vendor/democom/demochip/board/include/board.h +27**



本次移植先搞定串口和中断, 从表可以看到GIC的基地址为0x00A00000, 大小32KB, 串口基地址为0x02020000, 大小为16KB.





在实际做映射时, 取GIC的基地址, 大小设置大一点, 只要足够覆盖到UART1就行, 当然可以稍微再大一点, 以后添加新外设就不需要修改映射空间大小.

映射空间大小必须是1M字节对齐.

**vim ../../vendor/democom/demochip/board/include/board.h +37**

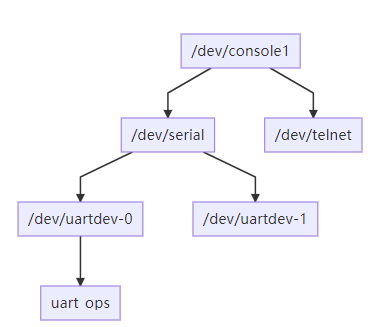


## 串口移植

内核打印程序的函数调用关系:

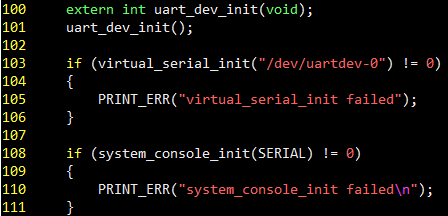
PRINT\_RELEASE -> LOS\_LkPrint -> g\_osLkHook -> OsLkDefaultFunc -> OsVprintf -> UartPuts -> UartPutsReg -> UartPutStr -> UartPutcReg.

控制台的实现分为4层:



**vim ../../vendor/democom/demochip/board/board.c +90**

在board.c的SystemInit():



uart\_dev\_init()用于初始化/dev/uartdev-0

virtual\_serial\_init(“/dev/uartdev-0”)将/dev/derial与/dev/uartdev-0关联

system\_console\_init(SERIAL)将/dev/console1与/dev/serial关联

每个设备文件都有对应的file\_operations\_vfs结构体.

**inode**是关键.

**vim openharmony/kernel/liteos\_a/kernel/common/console.c +1303**

system\_console\_init(deviceName) 🡪

OsConsoleCreate(consoleID, deviceName) 🡪

OsConsoleDevInit(consoleCB, deviceName) 🡪

vfs\_normalize\_path(NULL, deviceName, &fullpath);

inode\_find(&desc); inode = desc.node;

consoleCB->devInode = inode;

传入的参数deviceName就是”/dev/derial”,最终这个inode就对应/dev/derial

virtual\_serial\_init("/dev/uartdev-0") 🡪

ret = vfs\_normalize\_path(NULL, deviceName, &fullpath);

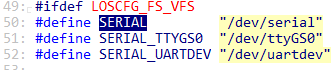
ret = inode\_find(&desc);

ret = inode\_find(&desc);

inode = desc.node;

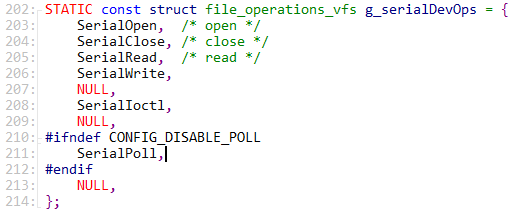
(VOID)register\_driver(SERIAL, &g\_serialDevOps, DEFFILEMODE, &g\_serialFilep);

Y:\openharmony\kernel\liteos\_a\kernel\common\virtual\_serial.h

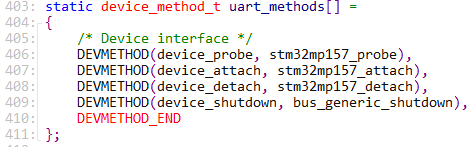
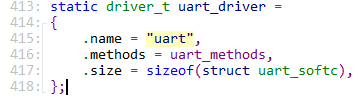


g\_serialDevOps就是一个file\_operations\_vfs结构体, 里面实现各种操作:

Y:\openharmony\kernel\liteos\_a\kernel\common\virtual\_serial.c

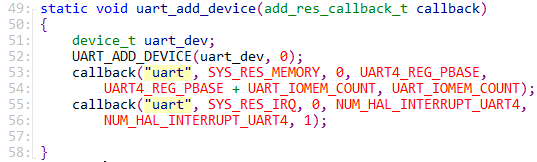


当内核发现有同名的”uart”驱动和设备, 就会调用probe函数, probe函数返回成功, 紧接着调用attach函数.

callback实际上就是添加一个uart设备

typedef void (\*add\_res\_callback\_t)(const char \*devclass\_name, int type, int unit, rman\_res\_t start, rman\_res\_t end, rman\_res\_t count);



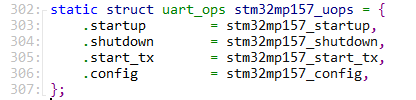
stm32mp157\_attach(device\_t self) 🡪

res = bus\_alloc\_resource\_any(self, SYS\_RES\_MEMORY, &unit, 0); // 获得内存资源

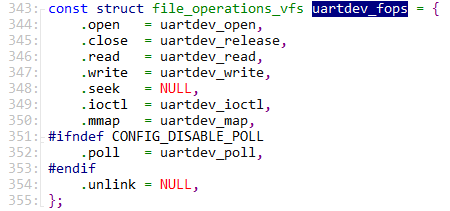
port->phys\_base = (unsigned long)(uintptr\_t)ioremap(res->start, res->count); // 虚拟地址映射

res = bus\_alloc\_resource\_any(self, SYS\_RES\_IRQ, &unit, 0); // 获得中断资源

udd->ops = &stm32mp157\_uops;



register\_driver(dev\_name, &uartdev\_fops, 0666, udd);



uartdev\_fops结构只是**框架**, 要访问uart硬件, 需要提供**stm32mp157\_uops**结构体.

app调用open(“/dev/uartdev-0”), 会找到uartdev\_fops.

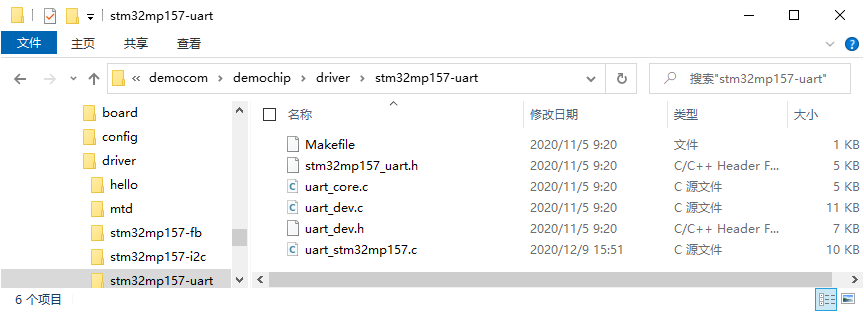
内核的注册有一个私有数据udd, udd有成员stm32mp157\_uops.

app调用open函数, 将会导致uartdev\_fops.open被调用, 也就是uartdev\_open被调用, uartdev\_open最终会调用stm32mp157\_uops的startup函数.

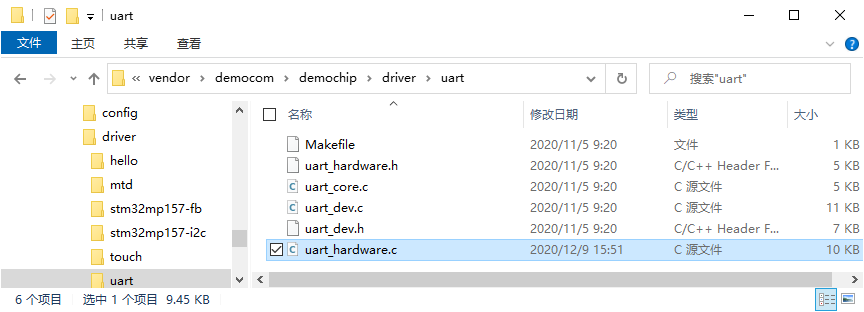
而app调用write函数, 将会最终导致stm32mp157\_uops的start\_tx函数被调用.



先对文件夹和文件改名, 改名以前:



改名以后:

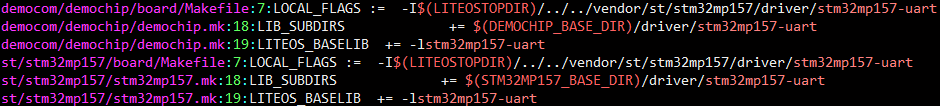


要修改Makefile:



**cd vendor/**

**grep "stm32mp157-uart" \* -nr**



**vim democom/demochip/board/Makefile +7**

**vim democom/demochip/demochip.mk +18**

**vim democom/demochip/demochip.mk +19**

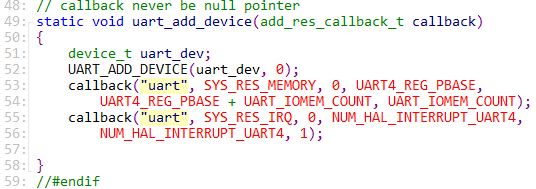
将”stm32mp157-uart”修改成”uart”

从device\_t入手, 修改Y:\openharmony\vendor\democom\demochip\board\bsd\_board.c

uart\_add\_device函数会对串口添加一个设备:

typedef void (\*add\_res\_callback\_t)(const char \*devclass\_name, int type, int unit, rman\_res\_t start, rman\_res\_t end, rman\_res\_t count);

第二个参数type代表资源的类型, 这将在static int demochip\_attach(device\_t self)中被使用.



Y:\openharmony\vendor\democom\demochip\board\include\asm\platform.h

#define UART4\_REG\_PBASE 0x40010000 /\* stm32mp157 uart4 \*/

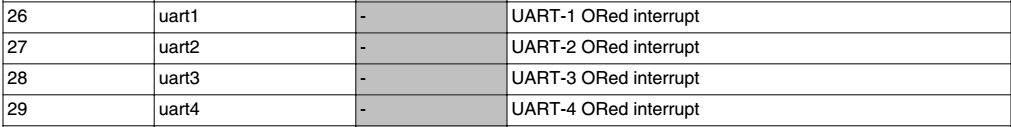
imx6ull不使用串口4, 使用串口1, 地址为0x02020000

故改为:

**#define UART1\_REG\_PBASE 0x02020000 /\* imx6ull uart1 \*/**

修改串口中断, 查看Table 3-1. ARM Cortex A7 domain interrupt summary

前32个中断是CPU的私有中断, 该表列出来是后面的非私有中断, 在表中uart1的中断号是26, 实际中断号为26+32=58.

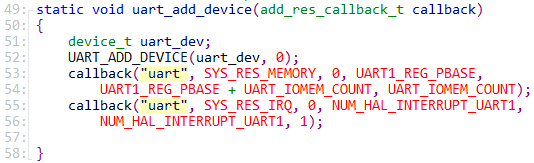


Y:\openharmony\vendor\democom\demochip\board\include\asm\hal\_platform\_ints.h



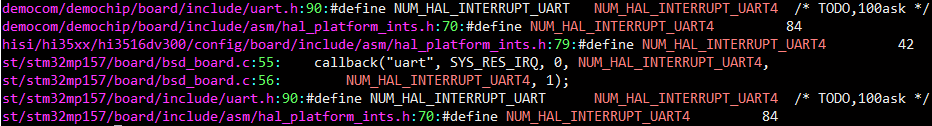
改为: **#define NUM\_HAL\_INTERRUPT\_UART1 58**

uart\_add\_device函数修改完以后:



cd /home/book/openharmony/vendor/

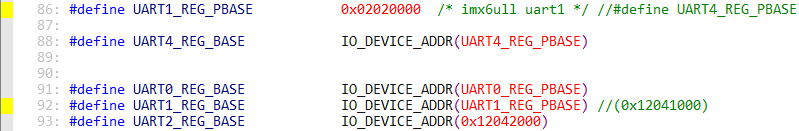
grep "NUM\_HAL\_INTERRUPT\_UART4" \* -nr



可以看到, \openharmony\vendor\democom\demochip\board\include\uart.h这个文件将串口的中断定义成了串口4:

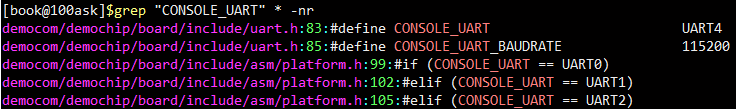


首先修改Y:\openharmony\vendor\democom\demochip\board\include\asm\platform.h



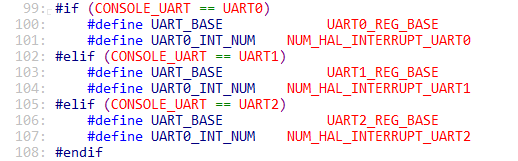
然后修改Y:\openharmony\vendor\democom\demochip\board\include\uart.h



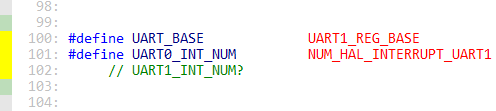


Y:\openharmony\vendor\democom\demochip\board\include\asm\platform.h

定义了两个宏: 串口的基地址和串口的中断号.



删除多余的部分, 并修改:



接下来修改driver部分, 这两个文件

Y:\openharmony\vendor\democom\demochip\driver\uart\uart\_hardware.c

Y:\openharmony\vendor\democom\demochip\driver\uart\uart\_hardware.h

使用

Y:\openharmony\vendor\nxp\imx6ull\driver\imx6ull-uart\uart\_imx6ull.c

Y:\openharmony\vendor\nxp\imx6ull\driver\imx6ull-uart\uart\_imx6ull.h

代替, 注意修改头文件包含:



并将.c文件的”imx6ull”换成”demochip”, 在source insight中使用ctrl + h.

函数static int demochip\_attach(device\_t self)调用

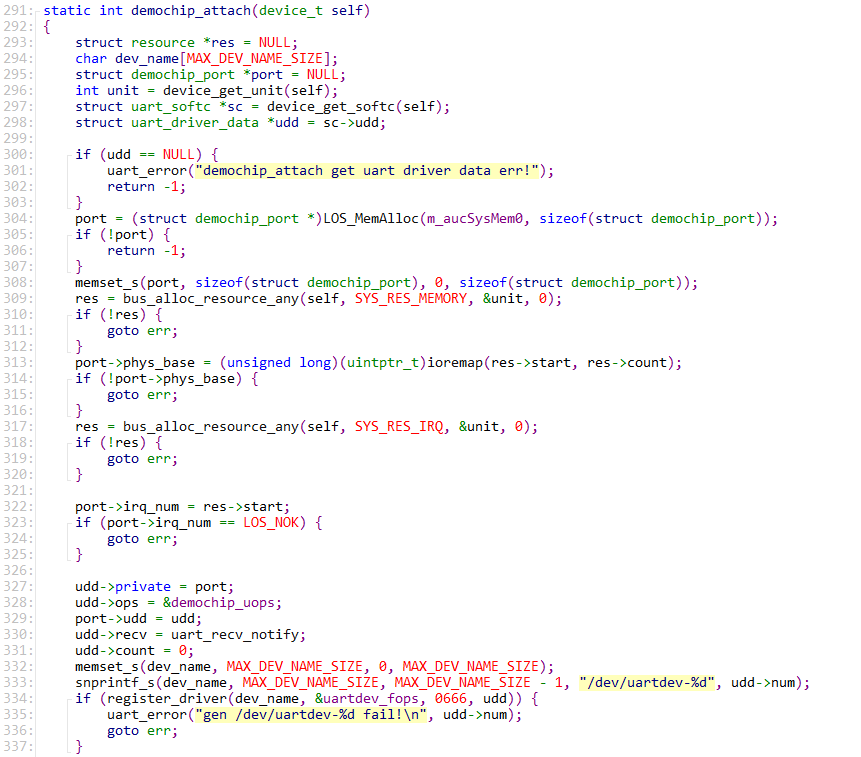
res = bus\_alloc\_resource\_any(self, SYS\_RES\_MEMORY, &unit, 0);

port->phys\_base = (unsigned long)(uintptr\_t)ioremap(res->start, res->count);

bus\_alloc\_resource\_any(self, SYS\_RES\_IRQ, &unit, 0);

分别获取了串口资源和中断资源.

Y:\openharmony\vendor\democom\demochip\driver\uart\uart\_hardware.c



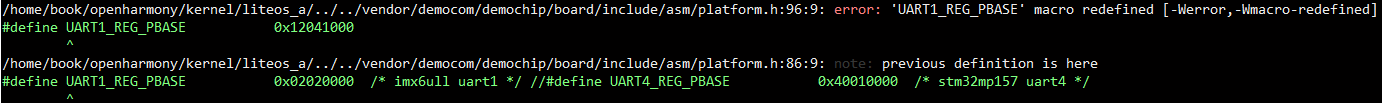
编译

cd /home/book/openharmony/kernel/liteos\_a/

cp tools/build/config/debug/demochip\_clang.config .config

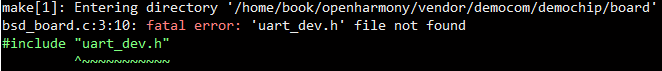
make clean

make -j 16



将/home/book/openharmony/vendor/democom/demochip/board/include/asm/platform.h的96行注释掉.

再次make -j 16.



**sudo chmod a+w /home/book/openharmony/vendor/democom/demochip/board/Makefile**

**vim /home/book/openharmony/vendor/democom/demochip/board/Makefile**

**将第7行修改:**

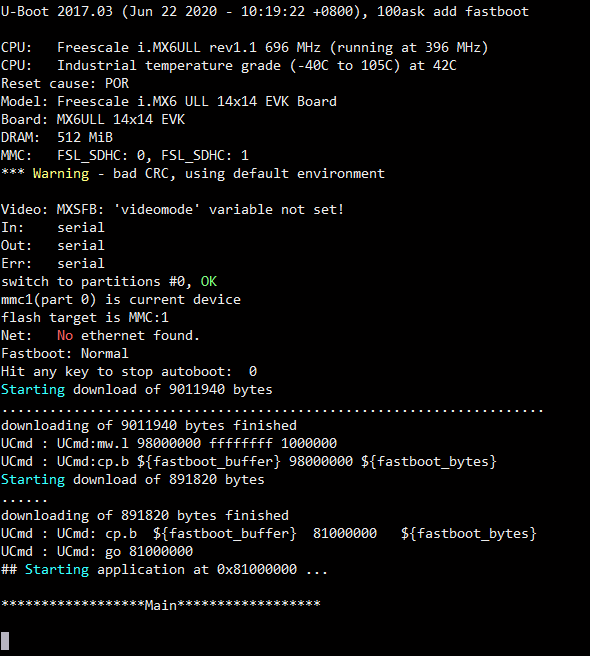


make -j 16

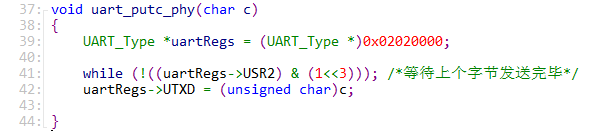


将liteos.bin拷贝到D:\document\doc\_and\_source\_for\_openharmony\IMX6ULL\开发板配套资料\软件\烧写工具\100ask\_imx6ull烧写工具v4\files, 将开发板设定为USB启动, 启动板子, 连接串口, 使用flash工具烧写.

开发板输出以下信息后便再无输出.

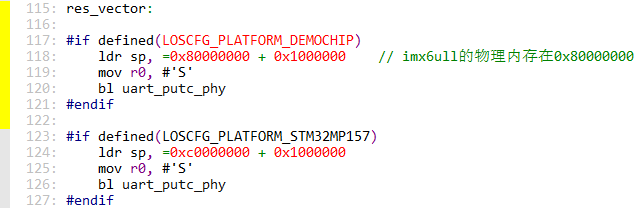


为了方便调试, 在Y:\openharmony\vendor\democom\demochip\driver\uart\uart\_hardware.c中添加以下代码.



添加代码:

Y:\openharmony\kernel\liteos\_a\arch\arm\arm\src\startup\reset\_vector\_up.S

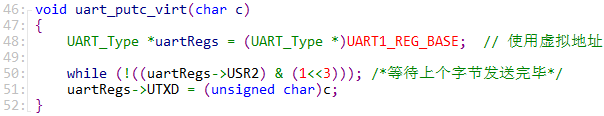


编译后下载运行, 打印出了字符’S’.

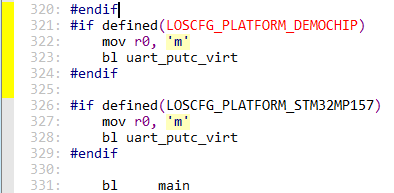


再次修改代码

Y:\openharmony\vendor\democom\demochip\driver\uart\uart\_hardware.c

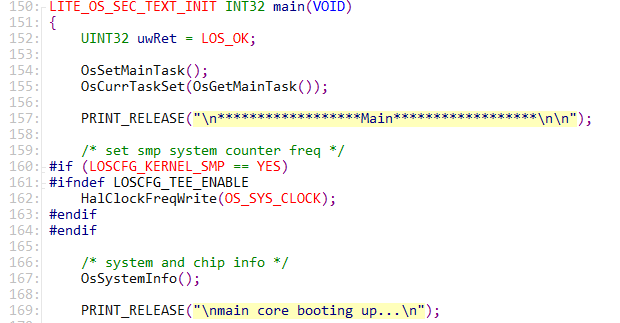


Y:\openharmony\kernel\liteos\_a\arch\arm\arm\src\startup\reset\_vector\_up.S

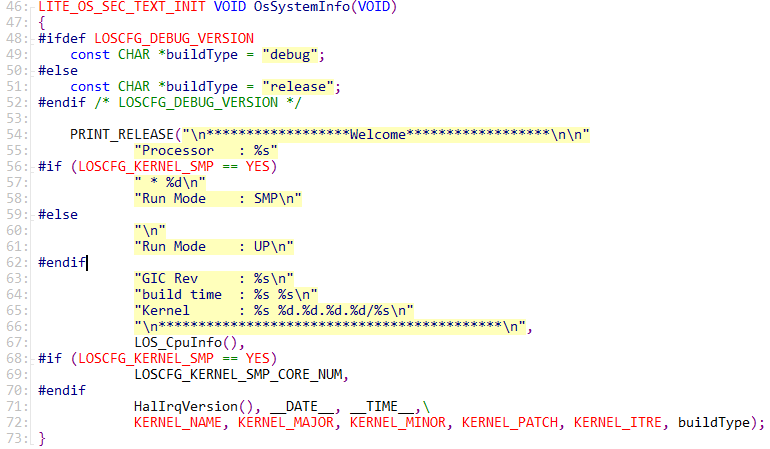


编译后下载运行, 打印出了字符’S’, ’m’.

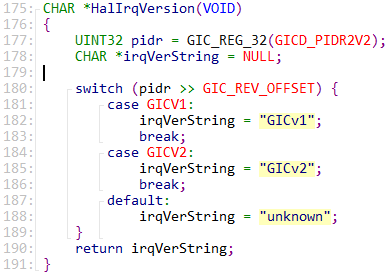




程序停在了OsSystemInfo();只是打印一些信息,



问题出在71行的HalIrqVersion().  
Y:\openharmony\kernel\liteos\_a\platform\hw\arm\interrupt\gic\gic\_v2.c



需要确定GIC的虚拟地址

Y:\openharmony\kernel\liteos\_a\platform\hw\include\gic\_common.h



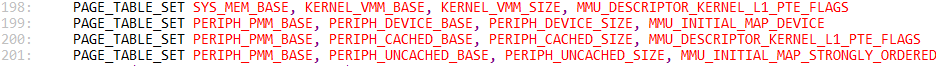
Y:\openharmony\vendor\democom\demochip\board\include\asm\platform.h



Y:\openharmony\kernel\liteos\_a\kernel\base\include\los\_vm\_zone.h



Y:\openharmony\kernel\liteos\_a\arch\arm\arm\src\startup\reset\_vector\_up.S



已经将PERIPH\_DEVICE\_BASE映射到PERIPH\_PMM\_BASE

修改代码:

Y:\openharmony\kernel\liteos\_a\kernel\base\include\los\_vm\_zone.h



Y:\openharmony\vendor\democom\demochip\board\include\asm\platform.h



修改了头文件, 要make clean一下

然后 make -j 16

编译烧写运行.

# 裸机编程

获取链接地址：

ldr r0, =\_start

获取加载地址：

adr r2, \_start

使用函数指针

void (\*funcptr)();

funcptr = xxfun;

实际上使用的是链接地址.

如何写出位置无关码:

只使用相对跳转指令: b bl

不使用绝对跳转指令: ldr pc, =main // 实质上是令pc为链接地址

不访问全局变量, 静态变量, 字符串, 数组

重定位完后, 使用绝对跳转指令跳转到xxx函数的链接地址

C语言使用链接脚本里面的变量

方法1: 使用外部变量, 使用时需要使用取址符&

extern unsigned int \_\_bss\_start;

extern unsigned int \_\_bss\_end;

unsigned int len;

len = (unsigned int)&\_\_bss\_start - (unsigned int)&\_\_bss\_end;

方法2: 声明为外部数组, 使用时不需要使用取址符&

extern char \_\_bss\_start[];

extern char \_\_bss\_end[];

unsigned int len;

len = \_\_bss\_start - \_\_bss\_end;

# 其它笔记

使用git bash连接服务器: ssh <user\_name>@<server\_ip>

rz, sz命令可以在windows和ubuntu之间传文件.

在linux中添加新用户: adduser <user\_name>

将新用户添加到sudo组中: usermod -G sudo <user\_name>

切换到新用户: su - <user\_name>

全局变量

extern char \*\*environ;

包含了环境表的地址.

size命令可以报告正文段, 数据段和bss段的长度: size <可执行文件>

查找文件: find -name <文件名>

查看上次执行结果的返回值: echo $?

连续执行命令: ; && ||. 其中;表示无条件执行.

$ ls /tmp/cmh && echo "exist" || echo "not exist"

ls: cannot access '/tmp/cmh': No such file or directory

not exist

$ ls /tmp/cmh || echo "exist" && echo "not exist"

ls: cannot access '/tmp/cmh': No such file or directory

exist

not exist

总结： 命令行[cmd1 && cmd2 || cmd3]的作用是, 若cmd成功执行($?=0), 则执行cmd2, 否则($?!=0)执行cmd3.

管道命令|: 命令行[cmd1 | cmd2 | cmd3]的作用是, 将cmd1的标准输出作为cmd2的标准输入, 将cmd2的标准输出作为cmd3的标准输入. 管道后面的命令必须具备接受标准输入的数据的能力. less, more, head, grep是管道命令, ls, cp, mv不是管道命令. 强行使标准错命令可以被管道命令处理的方法: 2>&1.

打补丁(在内核顶层目录下): patch -p1 < /home/book/openharmony\_100ask\_v1.2.patch

输出文件的560到56ZZ1行: sed -n "560,561p" <文件名>

`

实验室的服务器: IP:113.54.153.232 用户:cmh 密码:cmh

阿里云的服务器: IP:39.102.75.205 用户cmh 密码:123456a

临时配置交叉编译工具链的环境变量:

export ARCH=arm

export CROSS\_COMPILE=arm-linux-gnueabihf-

export PATH=$PATH:/home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin

book是用户名

使用repo可以恢复最初的源代码:

cd ~/openharmony

rm -rf \*

repo sync -j 8

ctrl+z ；把程序调度到后台执行

jobs 查看正在运行的任务

bg <num> 把程序调度到后台执行

fg <num> 把程序调度到前台执行

# VIM

使用vim查看二进制文件并以十六进制显示:

vim -b inst\_rom.o

:%!xxd -g 1

vim中的列操作

ctrl + v: 可视区块, 可以用矩形的方式选择数据.

小写v: 字符选择, 会将光标经过的地方反白选择. , 这样就选中了booptc.

大写V: 行选择, 会将光标经过的行反白选择.

选择区块后, 按y复制, 输入小写p则在光标所在列后面粘贴, 输入大写P则在光标所在列前面粘贴.

列输入: ctrl + v, 向下选择n列, 输入shift + i, 则在光标的所在列的左边插入, 输入shift + a则在光标所在列的右边插入, 然后输入所需内容, 按两次esc, 就可以完成跨越多行的列输入.

u: 撤销上一次操作

ctrl + r: 重复上一个操作.

vim多文件操作:

同时打开多个文件: vim file1 file2, -o: 水平显示, -O: 垂直显示

在新的垂直分屏打开: vs filename

在新的水平分屏打开: sv filename

sp + {filename}: 若省略文件名, 则使当前文件水平分割出两个窗口.

ctrl + w + 上下左右: 选择窗口.

vim关键词补全:

**ctrl + x -> ctrl + n**: 通过目前正在编辑的这个[文件的内容文字]作为关键词, 予以补齐.

**ctrl + x -> ctrl + f**: 以当前目录内的[文件名]作为关键词, 予以补齐.

**ctrl + x -> ctrl + o**: 以扩展名作为语法补充, 以vim内置的关键词, 予以补齐.

**:r! cmd**: cmd命令的结果插入光标下一行.

**:%! cmd**: cmd命令的结果覆盖当前文件内容.

cw: 删除从光标位置到当前词结尾处的文本.

使用vim的出场配置: vim -u NONE -N

使用临时的配置文件取代vimrc: vim -u <file>

增加缩进: >

减小缩进: >

自动缩进: =

重复查找上次f命令所查找的字符: ;

反方向查找上次f命令所查找的字符: ,

查找当前光标下的单词: \*

高亮匹配项: :set hls

对光标下的数字进行加减操作: {num}<C-a> {num}<C-x>

光标移动到最上方那一行的第一个字母: H

光标移动到最下方那一行的第一个字母: L

vim的自动位置标记

`` 当前文件中上跳转动作之前的位置

`. 上次修改的地方

`^ 上次插入的地方

`[ 上次修改或复制的起始位置

`] 上次修改或复制的结束位置

`< 上次高亮选区的起始位置

`> 上次高亮选区的结束位置

% 在一组开闭括号之间跳转, 可作用于() {} []

<C-i> 前进 <C-o>后退

参数对比

|  |  |  |
| --- | --- | --- |
|  | MSP430FR5994 | apollo3 blue |
| CPU核心 | MSP430 16BIT RSIC | ARM Cortex M4, 32BIT RISC |
| 功耗 | 正常为118uA/MHZ  睡眠模式下有45 350 500nA | 3.3V从flash或RAM取值执行时为6uA/MHz, 深度睡眠时电流1uA |
| 工作频率 | 最稳定8MHz, 最高16MHz | 一般48MHz, TurboSPOT模式下为96MHz |
| 非易失性存储NVM | 256KByte | none |
| RAM | 8KB | 384KB |
| ADC | 精度12bit, 共20通道 | 14bit精度, 15通道 |
| GPIO引脚 | 48~80个 | 37~50个 |
| I2C | 4 | 1~7个(与I2C共用) |
| SPI | 4 | 1~8个(与I2C共用) |
| UART | 4 | 2 |
| 定时器 | 16bit 16个 | 8个 32bit |
| 工作范围 | -40C~85C 1.8V~3.6V | -40~85 1.8V ~ 3.6V |
| 价格 | $3.406($1000+) | $3.03 |

apollo的工作模式

Cortex-M4内核的电源模式:

TurboSpot Mode: M4内核和所有存储器都以较高频率运行, 所有非调试时钟也在较高频率下运行, 外设在标称频率运行.

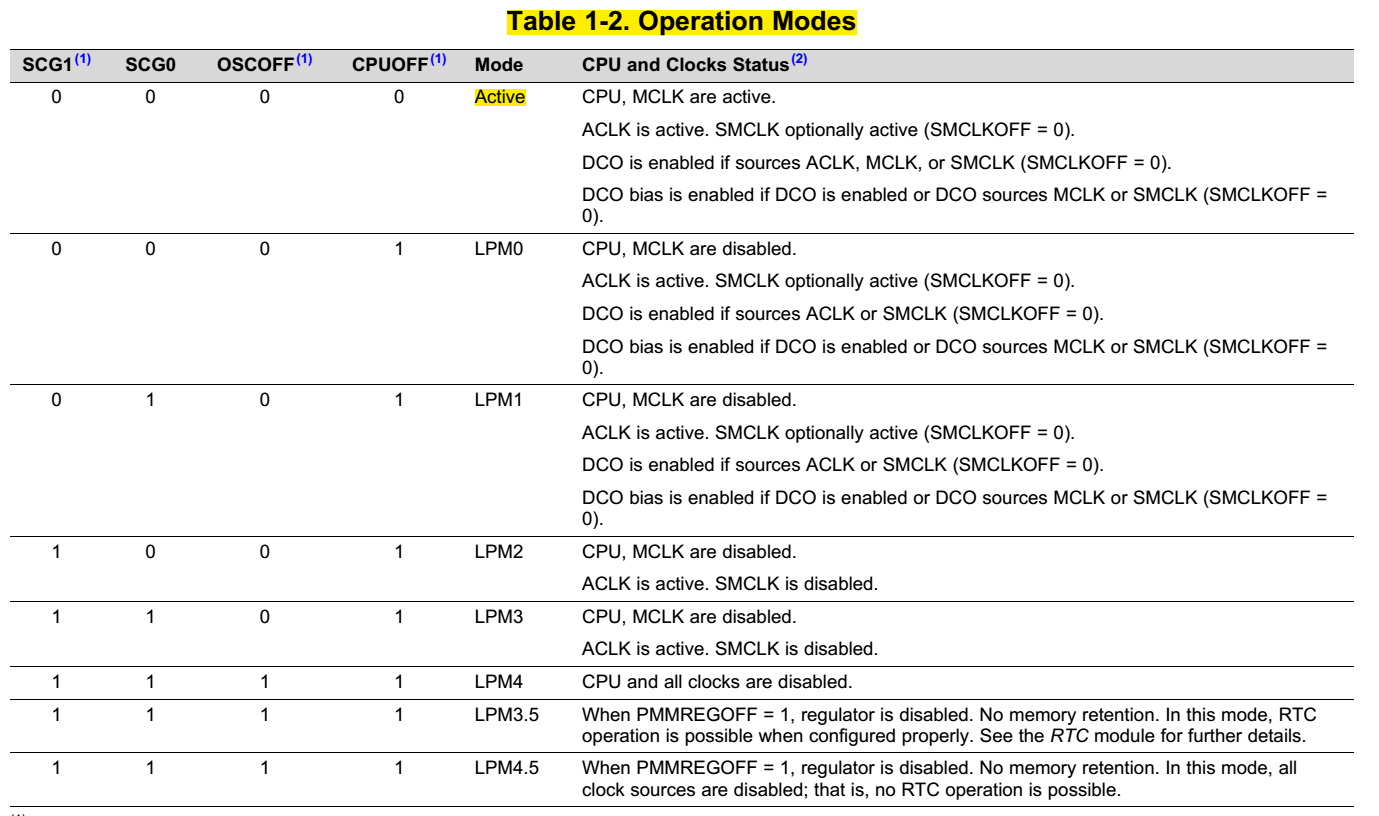
Active Mode: M4内核和所有外设都以正常状态运行.

Sleep Mode: M4内核通电, 时钟不运行, 等待唤醒事件激活.

Deep Sleep Mode: In the Deep Sleep Mode, the M4 enters SRPG mode where the main power is removed, but the flops retain their state. The clocks are not active, and the MCU clock sources for HCLK and FCLK can be deactivated. To facilitate the removal of the source supply and entry into SRPG mode, the M4 will handshake with the Wake-up Interrupt Controller and Power Management Unit and set up the possible wakeup conditions.(这段读得不是特别明白, SRPG手册没有解释)

搭配内核的电源模式, 系统可以工作在激活模式和5种不同的睡眠模式, 这些睡眠模式的主要区别在于工作的晶振和外设不同.

MSP430FR5994的工作模式:



# CSAPP

库打桩机制:

给定一个需要打桩的目标函数, 创建一个包装函数, 它的原型与目标函数完全一致. 使用打桩机制, 系统会先调用包装函数, 然后再调用目标函数, 再将目标函数的返回值传递给调用者. 使用打桩机制可以追踪对特殊库函数的调用次数, 验证和追踪它的输入和输出值.

编译时打桩:

**gcc -DCOMPILETIME -c mymalloc.c**

**gcc -I. -o intc int.c mymalloc.o**

链接时打桩:

--wrap f标识: 告诉编译器, 把对符号f的引用解析成**\_\_wrap\_f**, 还要把对符号\_\_read\_f的引用解析为f.

**gcc -DLINKTIME -c mymalloc.c**

**gcc -c int.c**

**gcc -W1,--wrap,malloc -W1,--wrap,free -o int1 int.o mymalloc.o**

-W1,option标识把option传递给链接器. option中每个逗号都要替换为一个空格.

运行时打桩:

LD\_PRELOAD设置成共享库路径名的列表. 解析未定义引用时, 动态连接器会先搜索LD\_PRELOAD库, 然后才搜索其它库.

构建包含包装函数的共享库: **gcc -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl**

编译主程序: **gcc -o intr int.c**

运行程序: **LD\_PRELOD=”./malloc.so” ./intr**

# Makefile

$@ 表示目标文件

$^ 表示所有依赖文件

$< 表示第一个依赖文件

$? 表示比目标还要新的依赖文件

**$(foreach var, list, text)**

对于list中的每个元素, 取值出来赋值给var, 然后把var改为text所描述的形式

例如

objs := a.o b.o

dep\_files := $(foreach f, $(ogjs), .$(f).d)

最终dep\_files := .a.o.d .b.o.d

**$(wildcard pattern)**

pattern所列出的文件是否存在, 把存在的文件都列出来.

例如

src\_files := $(wildcard \*.c)

列出当前目录下所有.c文件. 这里用\*号作通配符, 是因为在文件系统中搜索.

$(patsubst pat, rep, txt)

txt中符合pat格式的字用rep格式替换

subdir-y := c/ d/

subdir-y := $(patsubst %/, %, $(subdir-y))

结果为 subdir-y = c d

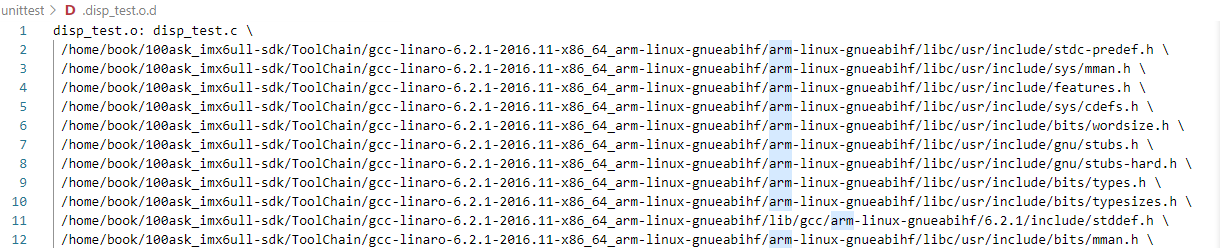
**@echo $(A)**

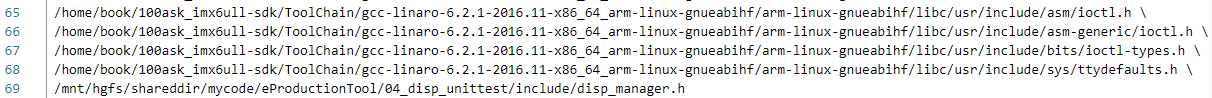
打印出变量A的内容, @表示不回显这一句命令

支持自动检测头文件的Makefile例子

|  |
| --- |
| objs = main.o sub.o  test : $(objs)  gcc –o test $^  dep\_files := $(foreach f, $(objs), .$(f).d)  dep\_files := $(wildcard $( dep\_files))  ifneq($(dep\_files),)  include $(dep\_files)  endif  %.o : %.c  gcc -Wp,-MD,.$@.d -c -o $@ $< # 生成依赖文件放在.xxx.o.d  clean:  rm \*.o test -f  distclean:  rm $(dep\_files) \*.o test -f |

一个依赖文件的例子, 前面都系统头文件, 最后一个是disp\_teat.c对应的头文件disp\_test.h





Makefile的一些选项:

-f 指定执行make文件, 不再使用默认的Makefile

-C 切换到其它目录执行make

延时变量:

A = xxx

A ?= xx, 如果曾经定义过, 此次赋值无效

延时变量就相当于C语言的指针, 当用到时才能确定该地址上的值.

立即变量:

D := xxx

马上确定变量的值, 就相当于C语言正常的赋值过程

变量的导出-export

如果想让某个变量的值在所有目录中都可见, 要把它导出来

在Makefile中使用shell命令

TOPDIR = $(shell pwd)

通用Makefile在Makefile.build中设置编译规则

(1) 怎么编译子目录

$(subdir-y):

make -C $@ -f $(TOPDIR)/Makefile.build

(2)怎么编译当前目录中的文件

%.o : %.c

$(CC) &(CFLAGS) $(EXTRA\_CFLAGS) $(CFLAGS\_$@) -Wp,-MD,$(dep\_file) -c -o $@ $<

(3)当前目录下的.o文件和子目录下的built-in.o文件要打包起来

built-in.o : $(cur\_objs) $(subdir\_objs)

$(LD) -r -o $@ $^