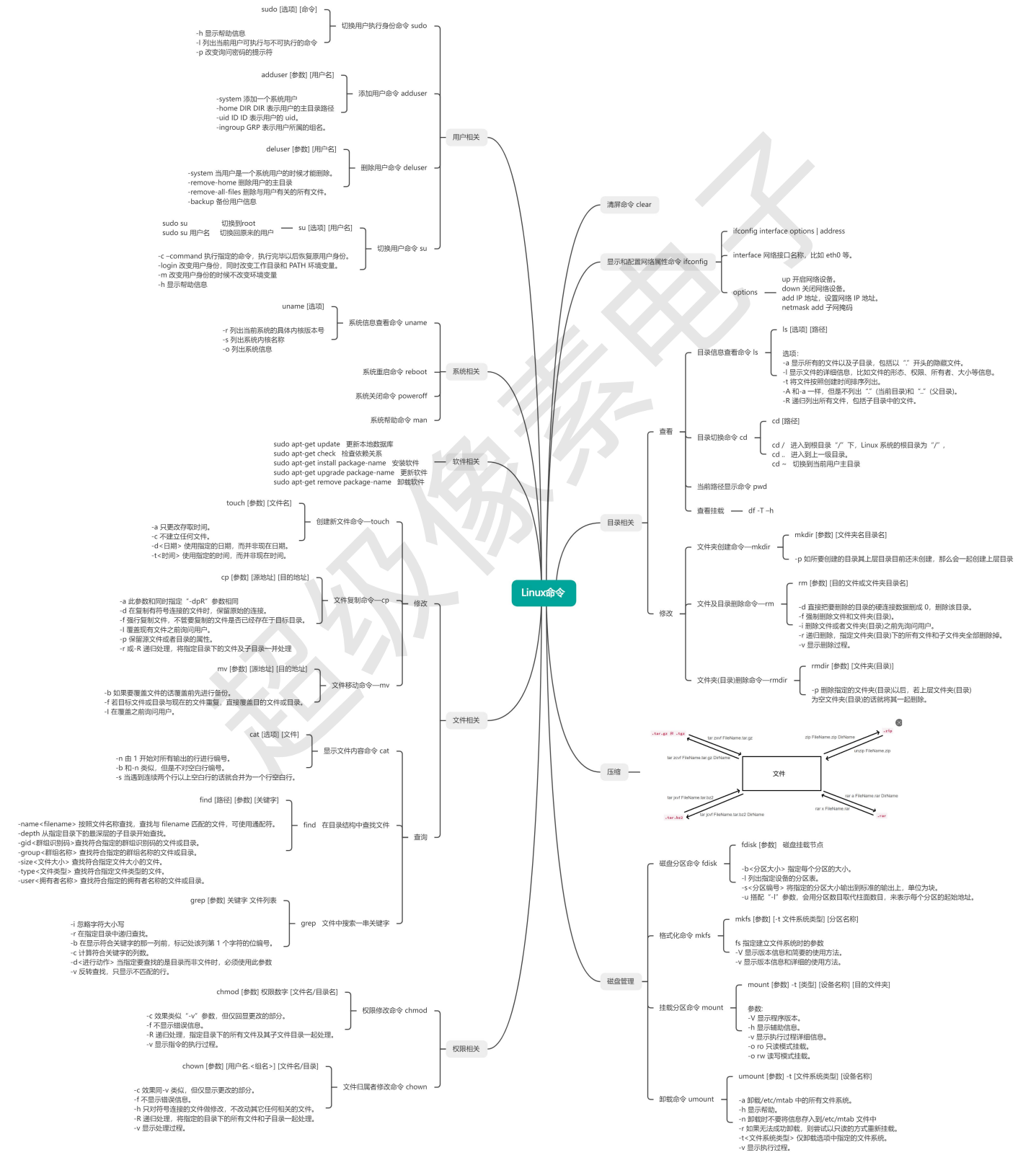
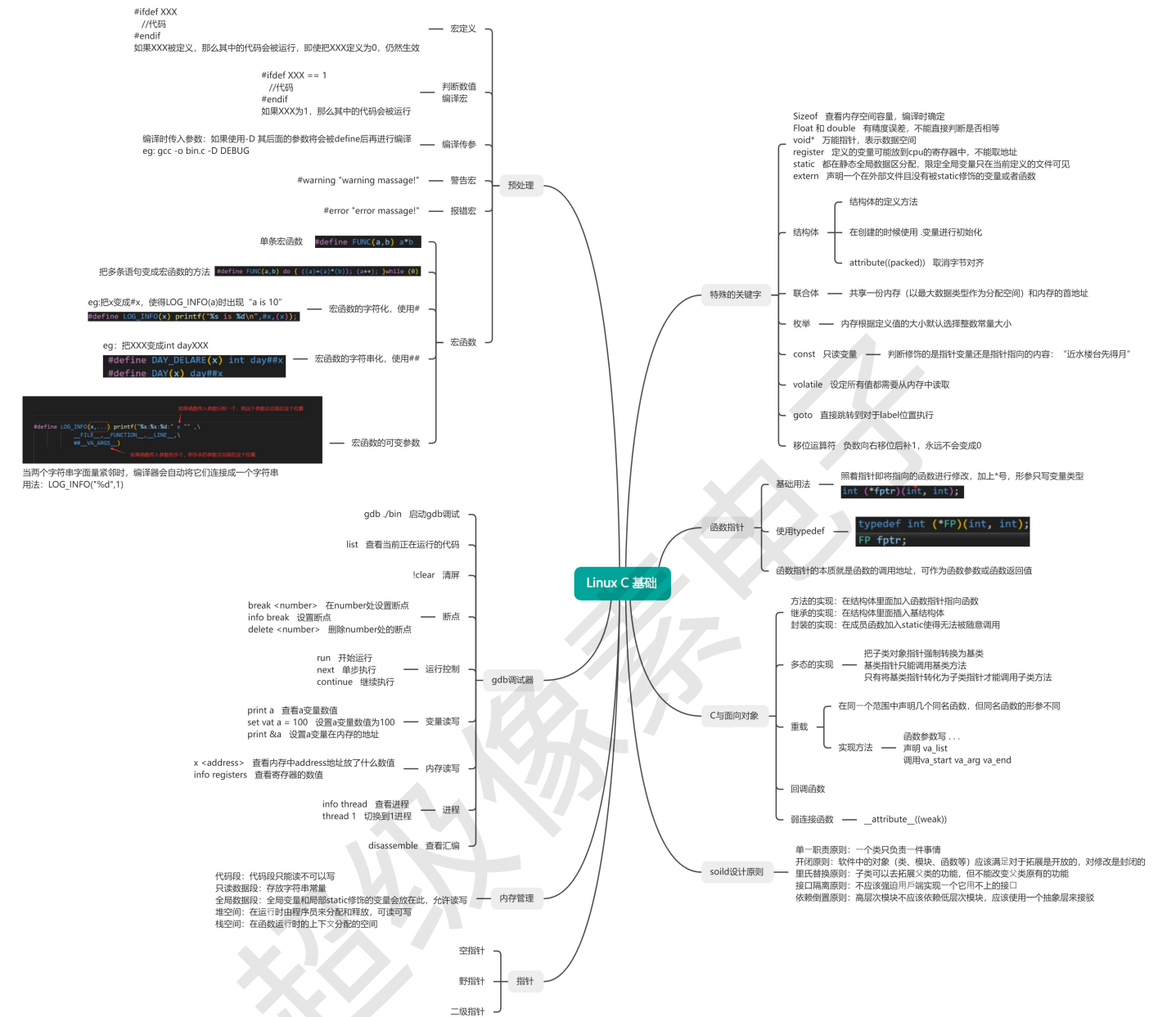


- 1.韦东山嵌入式 Linux 教程 <https://space.bilibili.com/275908810>
- 2.正点原子 Linux 教程 <https://space.bilibili.com/394620890>
- 3.Murphy4code 嵌入式 C 语言教程 <https://space.bilibili.com/2035825636>

Linux 基础指令



Linux C 基础 (Murphy)



gcc 编译

gcc 用法

-g 生成调试信息

```
murphy@ubuntu:~/workspace/lesson$ gcc -o bin 1.c -g
```

-v 输出详细信息

```
murphy@ubuntu:~/workspace/lesson$ gcc -o bin 1.c -v
```

用 cc1 把.c 变为.s

```
/usr/lib/gcc/x86_64-linux-gnu/9/cc1 -quiet -v -imultilib x86_64-linux-gnu 1.c -quiet -dumpbase 1.c -mtune=generic -march=x86-64 -auxbase 1 -version -fasynchronous-unwind-tables -fstack-protector-strong -Wformat -Wformat-security -fstack-clash-protection -fcf-protection -o /tmp/cccXLE8b.s
```


生成 obj 文件

```
as -v --64 -o /tmp/cc0TPQue.o /tmp/cccX1E8b.s
```

链接 obj 文件

```
/usr/lib/gcc/x86_64-linux-gnu/9/collect2
```

指定.h 文件路径

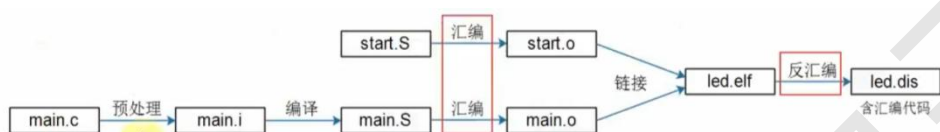
```
murphy@ubuntu:/home/workspace/lesson$ gcc -o bin 1.c -I ./inc
```

普通的编译过程

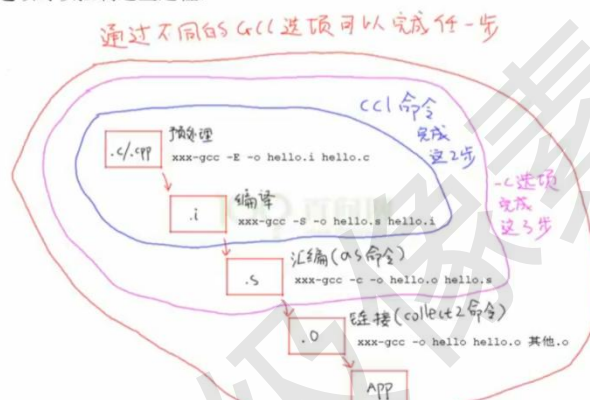
预处理：把头文件找到，把点 c 文件里面的头文件找到把宏展开

编译：变成汇编

汇编：变成机器码



通过不同的 gcc 选项可以控制这个过程：



预处理

```
murphy@ubuntu:/home/workspace/lesson$ gcc -o 1.i 1.c -E
```

此时宏定义已经被替换

编译

```
murphy@ubuntu:/home/workspace/lesson$ gcc -o 1.s 1.c -S
```

此时代码变为汇编

汇编

```
murphy@ubuntu:/home/workspace/lesson$ gcc -o 1.o 1.c -c
```

此时代码变成机器码

链接

和直接编译成可执行文件的命令相同，是 `gcc -o`，生成了 `elf` 文件，`keil` 里会把 `elf` 变为 `hex` 文件

静态链接库

```
book@100ask:~/nfs_rootfs/all_code1/source/02_options/02_multi_files$ gcc -c -o main.o main.c
book@100ask:~/nfs_rootfs/all_code1/source/02_options/02_multi_files$ ar crs mylib.a sub.o
book@100ask:~/nfs_rootfs/all_code1/source/02_options/02_multi_files$ gcc -o test3 main.o mylib.a
book@100ask:~/nfs_rootfs/all_code1/source/02_options/02_multi_files$ ls
main.c main.o Makefile mylib.a mylib.so* sub.c sub.h sub.o test* test2* test3*
book@100ask:~/nfs_rootfs/all_code1/source/02_options/02_multi_files$ ./test3
Main fun!
Sub fun!
```

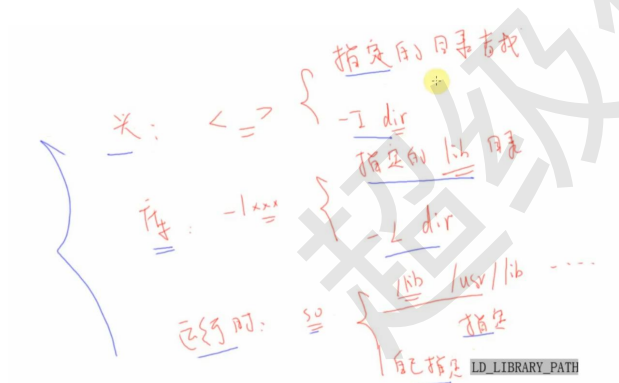
动态链接库

```
book@100ask:~/nfs_rootfs/all_code1/source/02_options/02_multi_files$ gcc -o test2 main.o mylib.so
book@100ask:~/nfs_rootfs/all_code1/source/02_options/02_multi_files$ gcc -o test2 main.o mylib.so
book@100ask:~/nfs_rootfs/all_code1/source/02_options/02_multi_files$ ls
main.c main.o Makefile mylib.so sub.c sub.h sub.o test2
book@100ask:~/nfs_rootfs/all_code1/source/02_options/02_multi_files$ ./test2
./test2: error while loading shared libraries: mylib.so: cannot open shared object file: No such file or directory
```

需要添加环境变量才能运行

```
book@book-virtual-machine:~/02_options/02_multi_files$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./
```

更改选项



gdb 调试器

调试前，`gcc` 编译的指令需要加上 `-g`

```
user@LAPTOP-1D04QTQ2:~/test07$ gcc -o bin test7.c -g
```

启动方法（加上 `-q` 不显示简介）

```
user@LAPTOP-1D04QTQ2:~/test07$ gdb ./bin -q
```

查看当前正在运行的代码（可以加上行号看第几行）

```
(gdb) list
1      #include <stdio.h>
2      int main()
3      {
4          int b;
5          b = 20;
6          printf("hello wprld\n");
7          return 0;
8      }
```

清屏

```
(gdb) !clear
```

设置断点（第几行）

```
(gdb) break 6
Breakpoint 1 at 0x115c: file test7.c, line 6.
```

查看断点

```
(gdb) info break
Num      Type      Disp Enb Address          What
1        breakpoint keep y   0x000055555555192 in main at test7.c:8
breakpoint already hit 1 time
```

删除断点（加上断点序号）

```
(gdb) delete 1
```

开始运行

```
(gdb) run
Starting program: /home/user/test07/bin
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at test7.c:6
6          printf("hello wprld\n");
```

单步执行

```
(gdb) next
hello wprld
7          return 0;
```

继续运行

```
(gdb) continue
Continuing.
[Inferior 1 (process 11758) exited normally]
```

查看变量数值

```
(gdb) print b
$2 = 20
```

更改变量数值

```
(gdb) set var b = 100
```

查看变量在内存的地址

```
(gdb) print &b
$1 = (int *) 0x7fffffff63c
```

查看内存中对应地址放了什么数值

```
(gdb) x 0x7fffffff63c
0x7fffffff63c: 0x0000001e
(gdb) x /d 0x7fffffff63c
0x7fffffff63c: 30
```

查看寄存器的数值

```
(gdb) info registers
rax      0xc          12
rbx      0x0          0
rcx      0x755557c1887 146727252732087
```

查看进程和切换进程

```
(gdb) info thread
Id      Target Id      Frame
* 1     Thread 0x7ffff7d8a740 (LWP 16827) "bin" main () at test7.c:8
(gdb) thread 1
[Switching to thread 1 (Thread 0x7ffff7d8a740 (LWP 16827))]
#0      main () at test7.c:8
8          printf("b=%d", b);
```

查看汇编

```
(gdb) disassemble
Dump of assembler code for function main:
0x000055555555169 <+0>:    endbr64
0x00005555555516d <+4>:    push    %rbp
0x00005555555516e <+5>:    mov     %rsp,%rbp
0x000055555555171 <+8>:    sub     $0x10,%rsp
```

预编译

条件编译宏

```
#ifdef XXX
    //代码
#endif
```

如果 XXX 被定义，那么其中的代码会被运行，即使把 XXX 定义为 0，仍然生效

```
#include <stdio.h>
#define DEBUG
int password = 0x37847110;

int main(void)
{
#ifdef DEBUG
    printf("password is %x\n",password);
#endif
    return 0;
}
```

编译传参

编译时传入参数：如果使用 -D 其后面的参数将会被 define 后再进行编译

```
murphy@ubuntu:~/workspace/lesson$ gcc -o bin 1.c -D DEBUG
```

判断数值编译宏

对宏进行判断从而实现条件编译的方法

```
#if DEBUG == 1
    printf("password is %x\n",password);
#endif
```

警告宏

警告宏：如果没有定义，就警告

```
#ifndef TOUCH
#warning "the touch is not add!"
#endif
```

报错宏

报错宏：如果没有定义，就报错

```
#ifndef TOUCH
#error "the touch is not add!"
#endif
```

宏函数

简单写法

```
1 #include <stdio.h>
2 #define FUNC(a,b) a*b
3 int main(){
4     int num;
5     num=FUNC(1+3,4); //1+3*4
6     printf("%d",num);
7     return 0;
8 }
```

把多条语句变成宏函数的方法

```
#define FUNC(a,b) do { ((a)=(a)*(b)); (a++); }while (0)
```

宏函数的字符化, 使用# eg:把 x 变成#x, 使得 LOG_INFO(a)时出现 “a is 10”

```
#define LOG_INFO(x) printf("%s is %d\n",#x,(x));
```

宏函数的字符串化, 使用## eg: 把 XXX 变成 int dayXXX

```
#define DAY_DECLARE(x) int day##x
```

```
#define DAY(x) day##x
```

宏函数的可变参数

```
#define LOG_INFO(x,...) printf("%s:%s:%d:" x "" , \
    __FILE__, __FUNCTION__, __LINE__, \
    ## __VA_ARGS__)
```

如果函数传入参数只有一个, 则这个参数会出现在这个位置

如果函数传入参数有多个, 则多余的参数会出现在这个位置

```
LOG_INFO("DAY21 = %d \n", DAY(21));
//变成
printf("%s:%s:%d:" "DAY21 = %d \n" "" , "1.c",__FUNCTION__,21, day21);

LOG_INFO("abcd\n");
//变成
printf("%s:%s:%d:" "abcd\n" "" , "1.c",__FUNCTION__,22);
```

特殊的关键字

函数指针

基础用法

```
1 #include <stdio.h>
2 void hello()
3 {
4     printf("hello\n");
5 }
6 int main()
7 {
8     void (*fptr)();
9     // 定义函数指针形式: 返回类型 (*名字)(参数)
10
11     fptr = hello;
12     (*fptr)();
13     return 0;
14 }
```

```
1 #include <stdio.h>
2 int sum(int x, int y)
3 {
4     return x + y;
5 }
6 int main()
7 {
8     int ans;
9     int (*fptr)(int, int);
10
11     fptr = sum;
12     ans = (*fptr)(10, 5);
13     printf("ans=%d", ans);
14     return 0;
15 }
```

使用 typedef


```

1  #include <stdio.h>
2  typedef int (*FP)(int, int);
3  int sum(int x, int y)
4  {
5      return x + y;
6  }
7  int main()
8  {
9      FP fptr;
10     fptr = sum;
11
12     int ans;
13     ans = (*fptr)(10, 5);
14     printf("ans=%d", ans);
15     return 0;
16 }

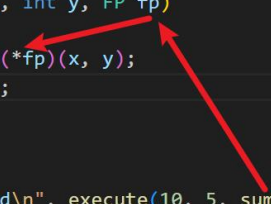
```

函数指针作为函数参数

```

1  #include <stdio.h>
2  typedef int (*FP)(int, int);
3  int sum(int x, int y)
4  {
5      return x + y;
6  }
7  int sub(int x, int y)
8  {
9      return x - y;
10 }
11 int execute(int x, int y, FP fp)
12 {
13     int result = (*fp)(x, y);
14     return result;
15 }
16 int main()
17 {
18     printf("a+b=%d\n", execute(10, 5, sum));
19     printf("a-b=%d\n", execute(10, 5, sub));
20     return 0;
21 }

```



函数指针作为函数返回值

```

1  #include <stdio.h>
2  typedef int (*FP)(int, int);
3  int sum(int x, int y)
4  {
5      return x + y;
6  }
7  int sub(int x, int y)
8  {
9      return x - y;
10 }
11 FP get(char c)
12 {
13     if (c == '+')
14         return sum;
15     else
16         return sub;
17 }
18 int main()
19 {
20     FP fp;
21     fp = get('+'); printf("a+b=%d\n", (*fp)(10, 5));
22     fp = get('-'); printf("a-b=%d\n", (*fp)(10, 5));
23     return 0;
24 }

```



C 与面向对象

方法，继承与封装

方法的实现：在结构体里面加入函数指针指向函数

继承的实现：在结构体里面插入基结构体

封装的实现：在成员函数加入 `static` 使得无法被随意调用

多态

多态的关键操作：

c

↓ 下载

📄 复制

▶ 运行

🗑

```
struct person *p = (struct person *)&usa_p;
```

步骤	操作	内存影响
1	取子类地址 &usa_p	获得 usa_person的起始地址
2	强制转换为 person*	编译器将这段内存"解释"为 person结构
3	赋值给 p	p只能"看到"基类部分的成员

多态能够实现的原因：

(1) C标准保证

根据C99标准 (§6.7.2.1) :

结构体指针可以安全转换为指向其第一个成员的指针，反之亦然。

(2) 实际内存示例

假设 usa_p在内存中的布局:

复制

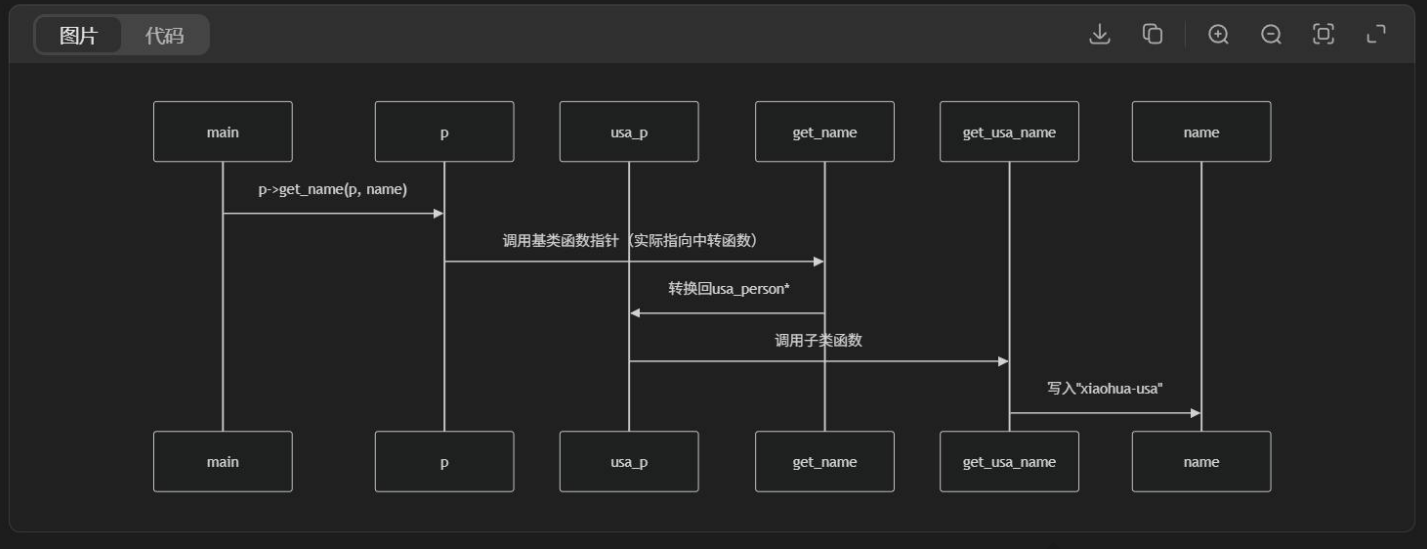
地址	内容
0x1000	name[24] <- person* p 从这里开始"看"
0x1018	age
0x101C	get_name <- 基类函数指针
0x1020	/ get_name <- 子类函数指针（被忽略）

• p->get_name只能访问到 0x101c处的指针

• 子类独有的 get_name在 0x1020，需要通过 usa_person类型访问

多态的转换过程：

当调用 `p->get_name(p, name)` 时:



重载

可变参数函数: 使用 `va_list`

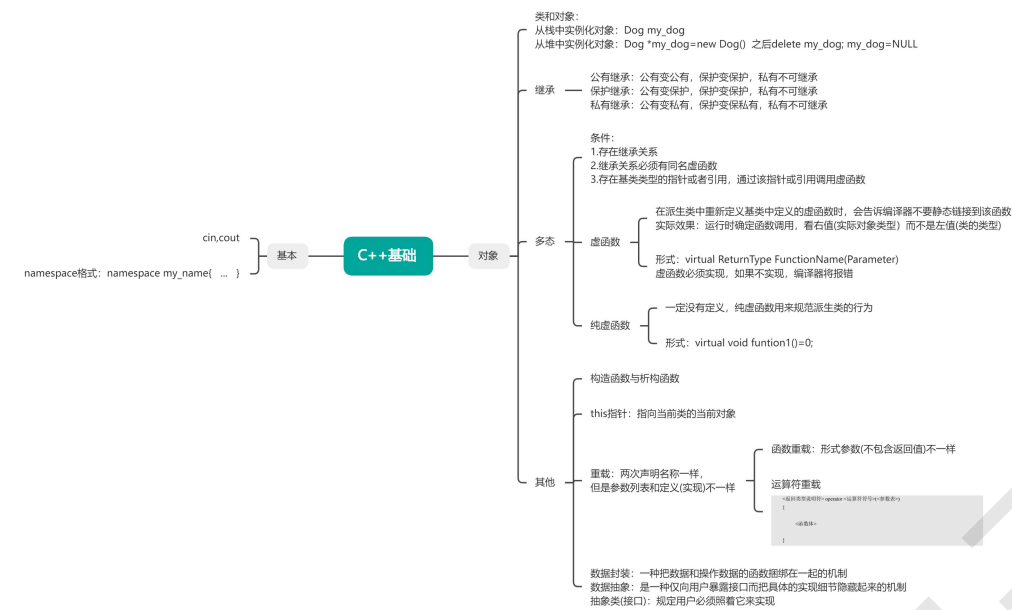
```
1 #include <stdio.h>
2 #include <stdarg.h>
3
4 // 这里约定n为后面参数的个数 (一种规则)
5 static void va_func(int n, ...)
6 {
7     va_list ptr;
8     va_start(ptr, n);
9     while (n-->0)
10     {
11         printf("%d\n", va_arg(ptr, int));
12     }
13     va_end(ptr);
14     printf("hello world\n");
15 }
16
17 int main(void)
18 {
19     va_func(1, 100);
20     va_func(2, 200, 300);
21     return 0;
22 }
```

回调函数

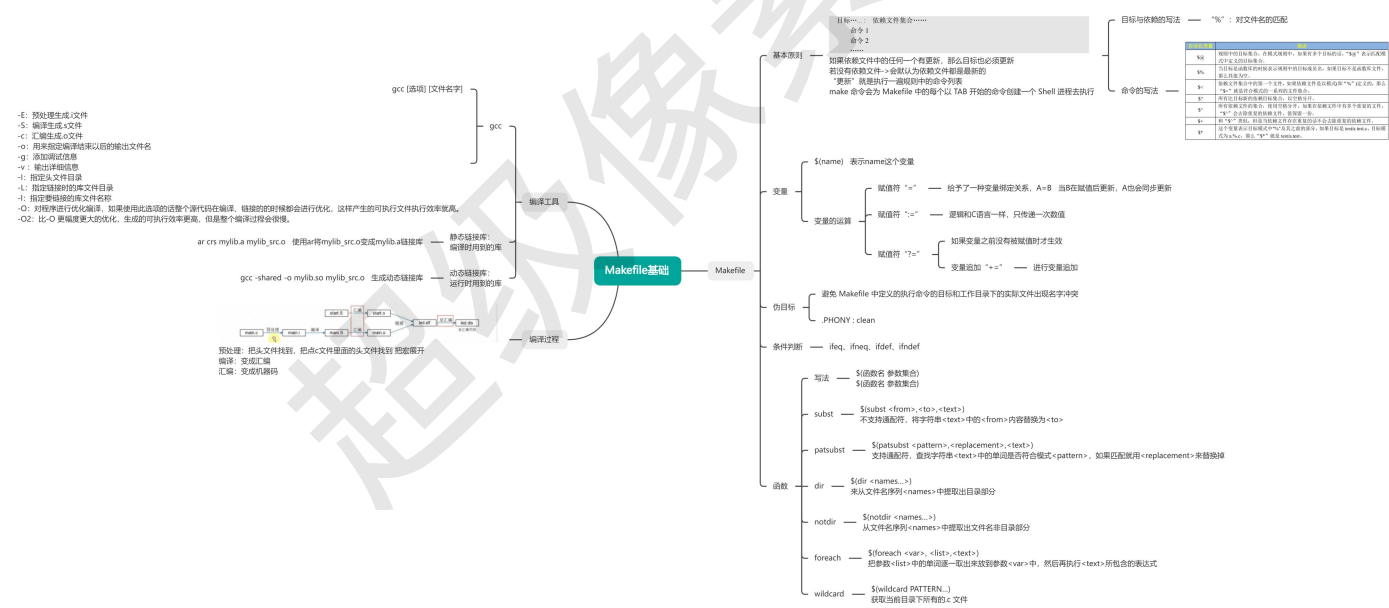
弱连接函数: 当其他函数文件定义了同名函数的时候, 原本的默认函数会被新的替代

```
void __attribute__((weak)) config(void)
{
    printf("config default\n");
}
```

C++基础



Makefile 基础 (韦东山)



基本规则

目标：依赖1 依赖2 ...
[TAB] 命令

当"目标文件"不存在，
或
某个依赖文件比目标文件"新"
则：执行"命令"

Eg:

```
7 test: main.o sub.o
8     gcc -o test main.o sub.o
9 main.o : main.c
10     gcc -c -o main.o main.c
11 sub.o : sub.c
12     gcc -c -o sub.o sub.c
```

便捷写法

a. 通配符: %.o
\$@ 表示目标
\$< 表示第1个依赖文件
\$^ 表示所有依赖文件

Eg:

```
1 test: main.o sub.o add.o # ^:所有的依赖
2     gcc -o test $^
3
4 %.o : %.c # %: 通配符
5     gcc -c -o $@ $<
```

内置函数

```
clean:
    rm *.o test
.PHONY:clean
```

```
user@LAPTOP-1D04QTQ2:~/test02$ make clean
rm *.o test
```

变量

简单变量(即时变量) :

A := xxx # A的值即刻确定, 在定义时即确定
B = xxx # B的值使用到时才确定

Eg:

```
A := $(C)
B = $(C)
C = abc
all:
    @echo $(A)
    @echo $(B)
    @echo $(C)
```

```
user@LAPTOP-1D04QTQ2:~/test02$ make
```

```
abc
abc
```

?= # 延时变量, 如果是第1次定义才起效, 如果在前面该变量已定义则忽略这句
+= # 附加, 它是即时变量还是延时变量取决于前面的定义

Eg:

```
A := $(C)
B = $(C)
C = abc
all:
    @echo $(A)
    @echo $(B)
    @echo $(C)
C+=123
```

```
user@LAPTOP-1D04QTQ2:~/test02$ make
```

```
abc 123
abc 123
```

Eg2:

```
17 A := $(C)
18 B = $(C)
19 C = abc
20 D = 123
21 D?= hhh
22 all:
23     @echo $(A)
24     @echo $(B)
25     @echo $(C)
26     @echo $(D)
27 C+=123
```

```
user@LAPTOP-1D04QTQ2:~/test02$ make
```

```
abc 123
abc 123
123
```

```
17 A := $(C)
18 B = $(C)
19 C = abc
20 # D = 123
21 D?= hhh
22 all:
23     @echo $(A)
24     @echo $(B)
25     @echo $(C)
26     @echo $(D)
27 C+=123
```

```
user@LAPTOP-1D04QTQ2:~/test02$ make
```

```
abc 123
abc 123
hhh
```

函数

循环

```
A = a b c
B=$(foreach f,$(A),$(f).o)
✓ all:
    @echo B = $(B)
```

```
user@LAPTOP-1D04QTQ2:~/test02$ make
```

```
B = a.o b.o c.o
```

筛选

```
$(filter pattern...,text)      # 在text中取出符合patten格式的值
$(filter-out pattern...,text)  # 在text中取出不符合patten格式的值
```

```
C= a b c d/
D=$(filter %/ ,$(C))
E=$(filter-out %/ ,$(C))
all:
    @echo C = $(C)
    @echo D = $(D)
    @echo E = $(E)

user@LAPTOP-1D04QTQ2:~/test02$ make
C = a b c d/
D = d/
E = a b c
```

文件查找

```
$(wildcard pattern)          # pattern定义了文件名的格式,
                             # wildcard取出其中存在的文件 ...
```

```
files=$(wildcard *.c)
all:
    @echo files = $(files)
```

```
user@LAPTOP-1D04QTQ2:~/test02$ make
files = add.c main.c sub.c
```

替换

```
$(patsubst pattern,replacement,$(var))  # 从列表中取出每一个值
                                         # 如果符合pattern
                                         # 则替换为replacement
```

```
files=$(wildcard *.c)
files2 = $(patsubst %.c,%.d,$(files))
all:
    @echo files = $(files)
    @echo files = $(files2)
```

```
user@LAPTOP-1D04QTQ2:~/test02$ make
files = add.c main.c sub.c
files = add.d main.d sub.d
```

实例

a. 改进：支持头文件依赖
<http://blog.csdn.net/qql452008/article/details/50855810>

```
gcc -M c.c // 打印出依赖
gcc -M -MF c.d c.c // 把依赖写入文件c.d
gcc -c -o c.o c.c -MD -MF c.d // 编译c.o, 把依赖写入文件c.d
```

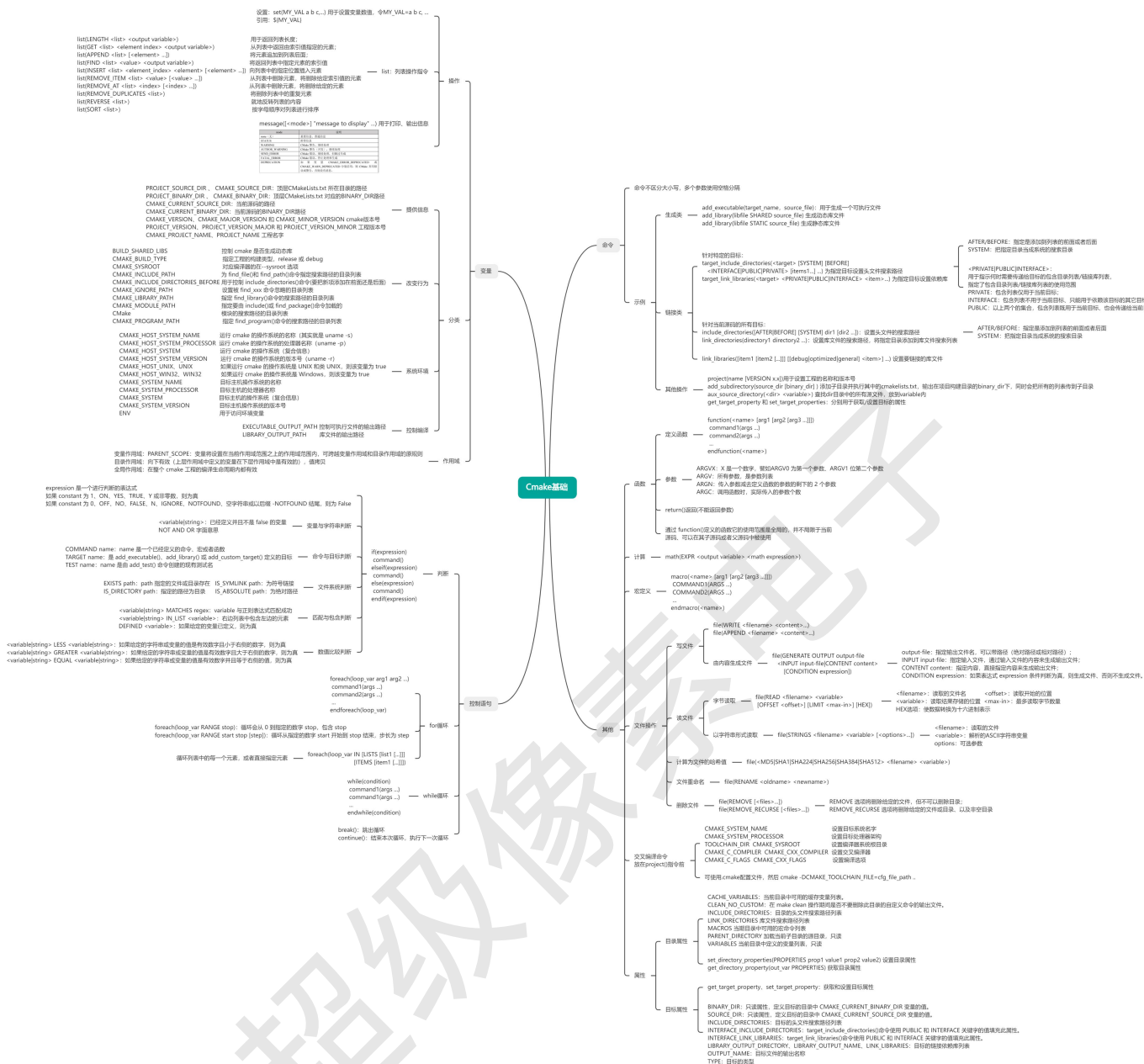
其他

把所有警告变成报错

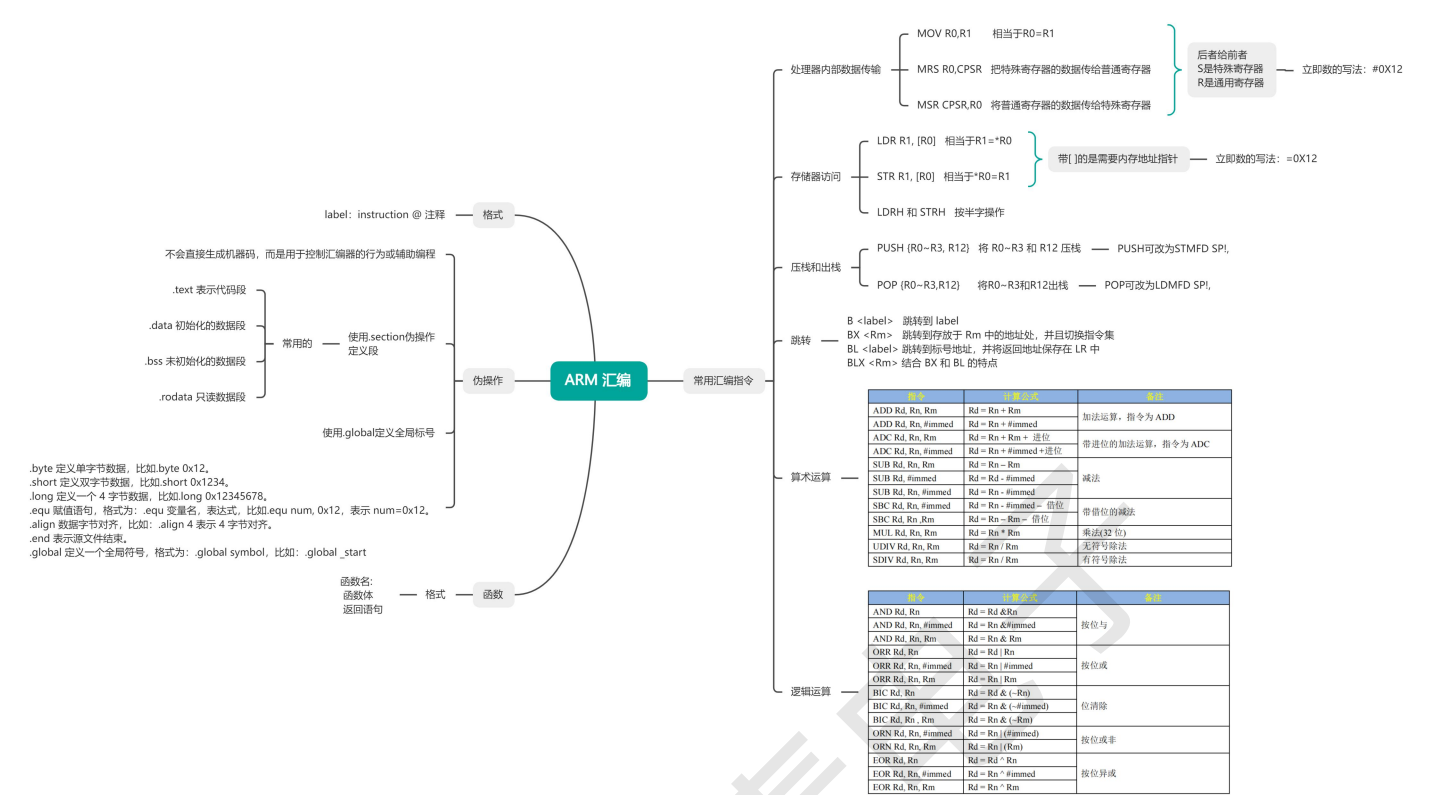
```
CFLAGS = -Werror
```

```
%.o : %.c
gcc $(CFLAGS) -c -o $@ $< -MD -MF $.d
```

Cmake 基础



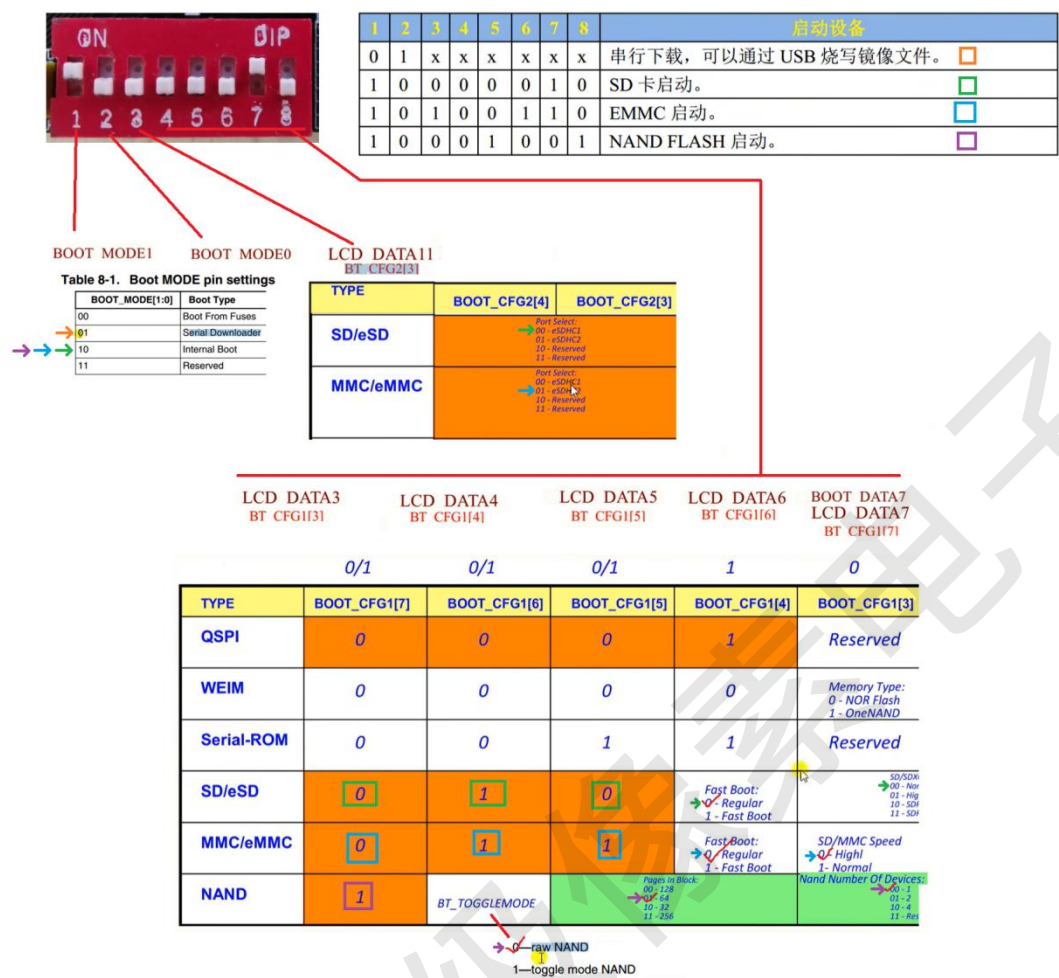
ARM 汇编基础



超级像素

开发板如何启动

启动方式



1、启动方式的选择

LED 灯实验，是从 SD 卡读取 bin 文件并启动，说明 6UL 支持从 SD 卡启动。6ULL 支持多种启动方式。

6ULL 是怎么支持从多种外置 flash 启动程序的。

1、启动方式选择

BOOT_MODE0 和 BOOT_MODE1，这两个是两个 IO 来控制的。选择从 USB 启动还是内部 BOOT 启动。如果要烧写系统到开发板中可以选择从 USB 下载，下载到 SD 卡，EMMC、NAND 等外置存储中。烧写完成设置从内部 BOOT 启动，然后从相应的外置存储中启动。

2、选择启动设备

2、启动设备的选择

前提是，你设置 MODE1 和 MODE0 是从内部 BOOT 启动的，也就是 MODE1=1，MODE0=0。

支持哪些设备：

NOR flash, oneNAND、NAND Flash、QSPI flash、SD/EMMC、EEPROM。我们最常用的就是 NAND、SD、EMMC 甚至 QSPI Flash。

如何选择启动设备。

通过 BOOT_CFG 选择,有 BOOT_CFG1,2,4, 每个 8 位。BOOT_CFG 是由 LCD_DATA0~23 来设置的。在 ALPHA 开发板上，大部分默认都 47K 下拉电阻接地。BOOT_CFG4 的 8 根线全部接地。BOOT_CFG2 全部接地，除了 BOOT_CFG2[3]，此位用来选择 SD 卡启动接口。BOOT_CFG1, 0, 1, 2 都是顶死的。3, 4, 5, 6, 7 是可以设置的。

正点原子开发板 BOOT 电路设置。核心板 LCD_DATA0~23 基本 47K 下拉。

启动流程

1、Boot Rom 做的事情。

设置内核时钟为 396MHz。使能 MMU 和 Cache，使能 L1cache L2 cache MMU，就是为了加速启动。

从 BOOT_CFG 设置的外置存储中，读取 image，然后做相应的处理。

2、IVT 和 Boot Data 数据

Bin 文件前面要添加头部。可以得到，我们烧写到 SD 卡中的 load.imx 文件在 SD 卡中的起始地址是 0x400，也就是 1024。

头部大小为 3KB，加上偏移的 1KB，一共是 4KB，因此在 SD 卡中 bin 文件起始地址为 4096。

IVT 大小为 32B/4=8 条。

IVT+Boot Data 的数据，很多是我从 NXP 官方 u-boot.imx 文件里面提取出来的。

3、DCD 数据

Device configuration data, DCD 数据就是 配置 6ULL 内部寄存器的。

首先，将 CCRG0~CCRG6 全部写为 0xFFFFFFFF,表示打开所有外设时钟。然后就是 DDR 初始化参数。设置 DDR 控制器，也就是初始化 DDR。

4、其他的数据

检查数据命令、NOP 命令、解锁命令。这些其实也都属于 DCD。

裸机例程

交叉编译工具链名字的由来

部分	含义	说明
arm	目标CPU架构	编译生成的代码运行在ARM处理器上
linux	目标操作系统	代码运行在Linux系统环境
gnu	工具链标准	使用GNU的编译工具和库
eabi	嵌入式应用二进制接口	定义二进制文件格式和调用约定
hf	硬件浮点支持	使用硬件FPU加速浮点运算
gcc	GNU编译器集合	实际执行的编译器程序

裸机汇编 LED

硬件

·一、I.MX6ULL 芯片简介

NXP 出品的，528~900MHz 的 Cortex-A7 内核的 MPU。

·三、I.MX6U IO 表示形式

STM32: 管脚名字: PA0~15, PB0~15 PC、PD。。。
I.MX6ULL: 管脚名字: PAD_BOOT_MODE0, 管脚的复用功能:
IOMUXC_SNVS_SW_MUX_CTL_PAD_BOOT_MODE0
IOMUXC_SNVS_SW_PAD_CTL_PAD_BOOT_MODE0
对于 6ULL, 查看管脚复用的步骤:
①、打开参考手册
②、找到 32 章, IO 复用章节
③、查找对应的管脚。

·一、汇编 LED 原理分析

为什么要学习 Cortex-A 汇编:
①、需要用汇编初始化一些 SOC 外设。
②、使用汇编初始化 DDR, I.MX6U 不需要。
③、设置 sp 指针, 一般指向 DDR, 设置好 C 语言运行环境。

1、ALPHA 开发板 LED 灯硬件原理分析：

STM32 IO 初始化流程：

- ①、使能 GPIO 时钟。
- ②、设置 IO 复用，将其复用为 GPIO。
- ③、配置 GPIO 的电气属性。
- ④、使用 GPIO，输出高/低电平。

IMX6ULL IO 初始化：

- ①、是使能时钟，CCGR0~CCGR6 这 7 个寄存器控制着 6ULL 所有外设时钟的使能。为了简单，设置 CCGR0~CCGR6 这 7 个寄存器全部为 0xFFFFFFFF，相当于使能所有外设时钟。
- ②、IO 复用，将寄存器 IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 的 bit3~0 设置为 0101=5，这样 GPIO1_IO03 就复用为 GPIO。
- ③、寄存器 IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO03 是设置 GPIO1_IO03 的电气属性。包括压摆率、速度、驱动能力、开漏、上下拉等。GPIO1_GDR
- ④、配置 GPIO 功能，设置输入输出。设置 GPIO1_DR 寄存器 bit3 为 1，也就是设置为输出模式。设置 GPIO1_DR 寄存器的 bit3，为 1 表示输出高电平，为 0 表示输出低电平。

编译方法

四、编写驱动

1、编译程序

- ①、使用 `arm-linux-gnueabi-gcc`，将.c.s 文件变为.o。
- ②、将所有的.o 文件连接为 elf 格式的可执行文件。
- ③、将 elf 文件转为 bin 文件。
- ④、将 elf 文件转为汇编，反汇编。

链接：

链接就是将所有.o 文件链接在一起，并且链接到指定的地方。本实验链接的时候要指定链接起始地址。

链接：

链接就是将所有.o 文件链接在一起，并且链接到指定的地方。本实验链接的时候要指定链接起始地址。链接起始地址就是代码运行的起始地址。

对于 6ULL 来说，链接起始地址应该指向 RAM 地址。RAM 分为内部 RAM 和外部 RAM，也就是 DDR。6ULL 内部 RAM 地址范围 0X900000~0X91FFFF。也可以放到外部 DDR 中，对于 IMX6U-ALPHA 开发板，512MB 字节 DDR 版本的核心板，DDR 范围就是 0X80000000~0X9FFFFFFF。对于 256MB 的 DDR 来说，那就是 0X80000000~0X8FFFFFFF。

本系列视频，裸机代码的链接起始地址为 0X87800000。要使用 DDR，那么必须要初始化 DDR，对于 IMX 来说 bin 文件不能直接运行，需要添加一个头部，这个头部信息包含了 DDR 的初始化参数，IMX 系列 SOC 内部 boot rom 会从 SD 卡，EMMC 等外置存储中读取头部信息，然后初始化 DDR，并且将 bin 文件拷贝到指定的地方。

Bin 的运行地址一定要和链接起始地址一致。位置无关代码除外。

```
alientek@zdyz-imx6ull: ~/Desktop/01、例程源码/01、裸机例程/01_leds$ arm-linux-gnueabi-gcc -g -c led.s -o led.o
alientek@zdyz-imx6ull: ~/Desktop/01、例程源码/01、裸机例程/01_leds$ arm-linux-gnueabi-ld -Ttext 0x87800000 led.o -o led.elf
alientek@zdyz-imx6ull: ~/Desktop/01、例程源码/01、裸机例程/01_leds$ arm-linux-gnueabi-objcopy -O binary -g -S led.elf led.bin
```

六、烧写 bin 文件

STM32 烧写到内部 FLASH。

6ULL 支持 SD 卡、EMMC、NAND、nor、SPI flash 等等启动。裸机例程选择烧写到 SD 卡里面。

在 ubuntu 下向 SD 卡烧写裸机 bin 文件。烧写不是将 bin 文件拷贝到 SD 卡中，而是将 bin 文件烧写到 SD 卡绝对地址上。而且对于 IMX 而言，不能直接烧写 bin 文件，比如先在 bin 文件前面添加头部。完成这个工作，需要使用正点原子提供的 imxdownload 软件。

```
imxdownload 使用方法，确定要烧写的 SD 卡文件，我的是/dev/sdf。  
给予 imxdownload 可执行权限：  
Chmod 777 imxdownload  
烧写：
```

```
./imxdownload led.bin /dev/sdf
```

imxdownload 会向 led.bin 添加一个头部，生成新的 load.imx 文件，这个 load.imx 文件就是最终烧写到 SD 卡里面去的。

裸机 C 语言 LED

汇编启动 C 语言要做的事

一、设置处理器模式

设置 6ULL 处于 SVC 模式 下。设置 CPSR 寄存器的 bit4-0, 也就是 M[4:0]为 10011=0X13。读写状态寄存器需要用到 MRS 和 MSR 指令。MRS 将 CPSR 寄存器数据读出到通用寄存器里面，MSR 指令将通用寄存器的值写入到 CPSR 寄存器里面去。

模式	描述
User(USR)	用户模式，非特权模式，大部分程序运行的时候就处于此模式。
FIQ	快速中断模式，进入 FIQ 中断异常
IRQ	一般中断模式。
Supervisor(SVC)	超级管理员模式，特权模式，供操作系统使用。
Monitor(MON)	监视模式？这个模式用于安全扩展模式，只用户安全。
Abort(ABT)	数据访问终止模式，用于虚拟存储以及存储保护。
Hyp(HYP)	超级监视模式？用于虚拟化扩展。
Undef(UND)	未定义指令终止模式。
System(SYS)	系统模式，用于运行特权级的操作系统任务

表 6.2.1 九种运行模式

二、设置 `sp` 指针

`Sp` 可以指向内部 RAM, 也可以指向 DDR, 我们将其指向 DDR。`Sp` 设置到哪里? 512MB 的范围 `0x80000000~0x9FFFFFFF`。栈大小, `0x200000=2MB`。处理器栈增长方式, 对于 A7 而言是向下增长的。设置 `sp` 指向 `0x80200000`。

三、跳转到 C 语言

使用 `b` 指令, 跳转到 C 语言函数, 比如 `main` 函数。

代码段与链接脚本

段就是程序的一部分, 我们把整个程序的所有东西分成了一个一个的段, 给每个段起个名字, 然后在链接时就可以用这个名字来指示这些段。也就是说给段命名就是为了在链接脚本中用段名来让段站在合适的位置。

段名分为2种: 一种是 **编译器** 链接器内部定好的, 先天性的名字; 一种 程序员自己指定的, 自定义的段名。

先天性段名:

代码段: (`.text`), 又叫文本段, 代码段其实就是 **函数** 编译后生成的东西。

数据段: (`.data`), 数据段就是C语言中有显式初始化为非0的全局变量

bss段: (`.bss`), 又叫ZI (zero initial) 段, 就是零初始化段, 对应C语言中初始化为0的全局变量。

后天性段名:

段名由程序员自己定义, 段的属性和特征也由程序员自己定义。

4. 对比 `.bss` 和 `COMMON`

特性	<code>.bss</code> 段	<code>COMMON</code> 段
存储内容	显式初始化为0的全局变量	未初始化的全局变量 (弱符号)
链接行为	直接分配空间	链接时合并重复定义
现代替代	<code>-fno-common</code> 编译选项下直接存到 <code>.bss</code>	旧编译器默认行为

示例代码 10.4.2.1 链接脚本演示代码

```
1 SECTIONS{
2   . = 0x10000000;
3   .text : {*(.text)}
4   . = 0x30000000;
5   .data ALIGN(4) : { *(.data) }
6   .bss ALIGN(4) : { *(.bss) }
7 }
```

第 1 行我们先写了一个关键字“`SECTIONS`”, 后面跟了一个大括号, 这个大括号和第 7 行的大括号是一对, 这是必须的。看起来就跟 C 语言里面的函数一样。

第 2 行对一个特殊符号“`.`”进行赋值, “`.`”在链接脚本里面叫做定位计数器, 默认的定位计数器为 0。我们要求代码链接到以 `0x10000000` 为起始地址的地方, 因此这一行给“`.`”赋值

0X10000000, 表示以 0X10000000 开始, 后面的文件或者段都会以 0X10000000 为起始地址开始链接。

第 3 行的“.text”是段名, 后面的冒号是语法要求, 冒号后面的大括号里面可以填上要链接到“.text”这个段里面的所以文件, “*(.text)”中的“*”是通配符, 表示所有输入文件的.text 段都放到“.text”中。

第 4 行, 我们的要求是数据放到 0X30000000 开始的地方, 所以我们需要重新设置定位计数器“.”, 将其改为 0X30000000。如果不重新设置的话会怎么样? 假设“.text”段大小为 0X10000, 那么接下来的.data 段开始地址就是 0X10000000+0X10000=0X10010000, 这明显不符合我们的要求。所以我们必须调整定位计数器为 0X30000000。

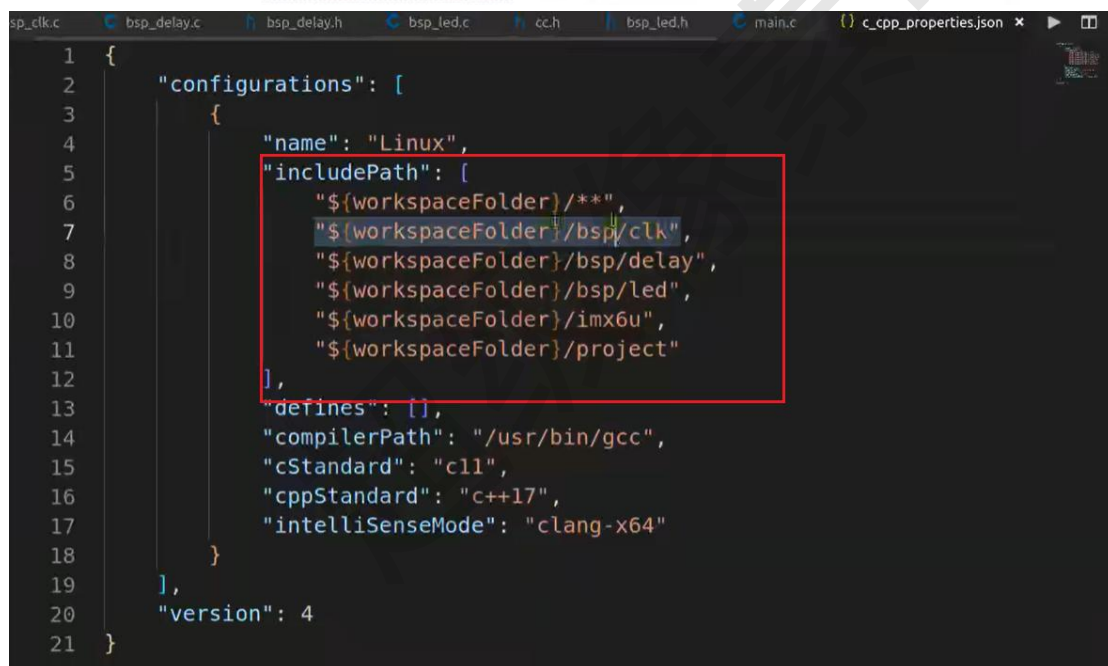
第 5 行跟第 3 行一样, 定义了一个名为“.data”的段, 然后所有文件的“.data”段都放到这里面。但是这一行多了一个“ALIGN(4)”, 这是什么意思呢? 这是用来对“.data”这个段的起始地址做字节对齐的, ALIGN(4)表示 4 字节对齐。也就是说段“.data”的起始地址要能被 4 整除, 一般常见的都是 ALIGN(4)或者 ALIGN(8), 也就是 4 字节或者 8 字节对齐。

第 6 行定义了一个“.bss”段, 所有文件中的“.bss”数据都会被放到这个里面, “.bss”数据就是那些定义了但是没有被初始化的变量。

裸机 BSP 工程模板

设置 vscode 头文件路径

设置 VSCODE 头文件路径。先创建.vscode 目录, 然后打开 c/c++ 配置器, 会在.vscode 目录下生成一个叫做 c_cpp_properties.json 的文件。



```
1 {
2   "configurations": [
3     {
4       "name": "Linux",
5       "includePath": [
6         "${workspaceFolder}/**",
7         "${workspaceFolder}/bsp/clk",
8         "${workspaceFolder}/bsp/delay",
9         "${workspaceFolder}/bsp/led",
10        "${workspaceFolder}/imx6u",
11        "${workspaceFolder}/project"
12      ],
13      "defines": [],
14      "compilerPath": "/usr/bin/gcc",
15      "cStandard": "c11",
16      "cppStandard": "c++17",
17      "intelliSenseMode": "clang-x64"
18    }
19  ],
20  "version": 4
21 }
```

Makefile 改进

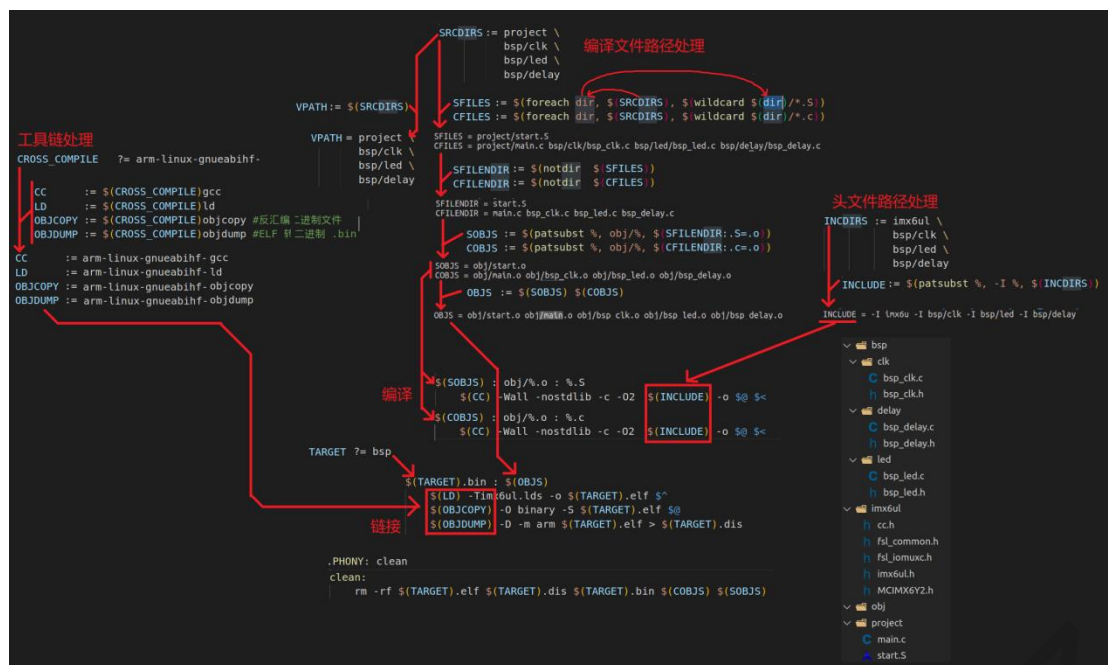
Makefile 指定头文件路径, 需要-I。我们编译源码的时候需要指定头文件路径。比如 bsp/clk/bsp_clk.h 变为-I bsp/clk/bsp_clk.h

通过一堆的变量, 将要编译的原材料准备好了。

Makefile 静态模式

<targets ...>: <target-pattern>: <prereq-patterns ...>

\$(OBJS): obj/%.o: %.S 表示将所有的.S 文件编译为.o 并且存放到 obj 目录下去。



一些函数的实际应用，具体知识点看前面

```
20 INCLUDE := $(patsubst %, -I %, $(INC_DIRS))

20
21 SFILES := $(foreach dir, $(SRC_DIRS), $(wildcard $(dir)/*.S))
22 CFILES := $(foreach dir, $(SRC_DIRS), $(wildcard $(dir)/*.c))
23
```

VPATH 是 Makefile 中的 **虚拟路径 (Virtual Path)** 变量，用于告诉 **make** 在哪些目录中查找源文件 (如 **.c** 和 **.S** 文件)。

当 Makefile 中规则的目标或依赖文件不在当前目录时，**make** 会根据 **VPATH** 中定义的路径去搜索这些文件。

清除 BSS 段的 4 字节对齐问题

·三、加上清除 BSS 段，代码不运行。

__bss_start = 0X87800289。对于 32 位的 SOC 来说，一般访问是 4 字节访问的。0X0,0X4,0X8,0XC。芯片处理的时候以 4 字节访问，因此会从 0X878000288 开始清除 BSS 段。然而 0X878000288 不属于 BSS 段。所以我们需要对 __bss_start 进行四字节对其。按照四字节对其的原理，__bss_start=0X8780028C。所以需要设置 __bss_start 为四字节对其。

裸机时钟树

1、7 路 PLL

为了方便生成时钟,6 从 24MHz 晶振生出来 7 路 PLL。这 7 路 PLL 中有的又生出来 PFD。

PLL1: ARM PLL 供给 ARM 内核。

PLL2: sysstem PLL, 528MHz, 528_PLL, 此路 PLL 分出了 4 路 PFD, 分别为 PLL2_PFD0~PFD3

PLL3: USB1 PLL, 480MHz 480_PLL, 此路 PLL 分出了 4 路 PFD, 分别为 PLL3_PFD0~PFD3。

PLL4: Audio PLL, 主供音频使用。

PLL5: Video PLL, 主供视频外设, 比如 RGB LCD 接口, 和图像处理有关的外设。

PLL6: ENET PLL, 主供网络外设。

PLL7: USB2_PLL ,480MHz, 无 PFD。

5、要初始化的 PLL 和 PFD

PLL1,

PLL2, 以及 PLL2_PFD0~PFD3,

PLL3 以及 PLL3_PFD0~PFD3.

1、系统主频的配置

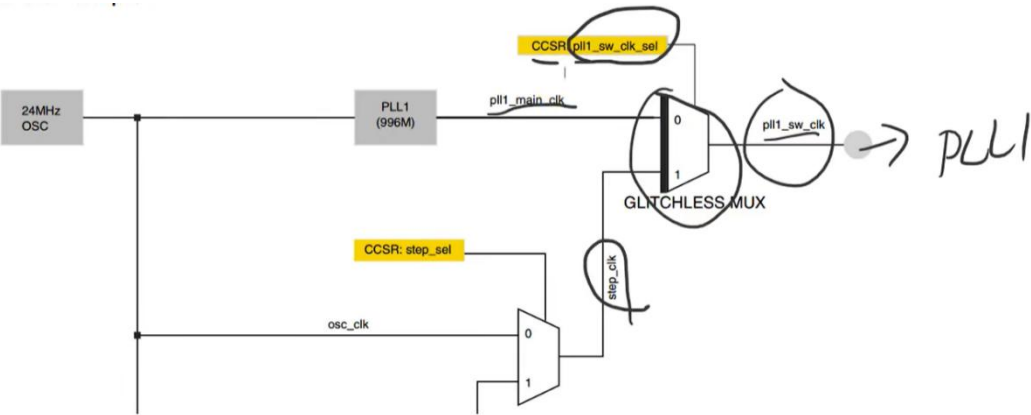
①、要设置 ARM 内核主频为 528MHz, 设置 CACRR 寄存器的 ARM_PODF 位为 2 分频, 然后设置 PLL1=1056MHz 即可。CACRR 的 bit3~0 为 ARM_PODF 位, 可设置 0~7, 分别对应 1~8 分频。应该设置 CACRR 寄存器的 ARM_PODF=1。

Table 18-4. System Clock Frequency Values

Clock Root	Default Frequency (MHz)	Maximum Frequency (MHz)
ARM_CLK_ROOT	12	528



②、设置 PLL1=1056MHz。PLL1=pll1_sw_clk。pll1_sw_clk 有两路可以选择, 分别为 pll1_main_clk, 和 step_clk, 通过 CCSR 寄存器的 pll1_sw_clk_sel 位(bit2)来选择。为 0 的时候选择 pll1_main_clk, 为 1 的时候选额 step_clk。



③、在修改 PLL1 的时候，也就是设置系统时钟的时候需要给 PLL1 一个临时的时钟，也就是 step_clk。在修改 PLL1 的时候需要将 pll1_sw_clk 切换到 step_clk 上。

③、设置 step_clk。Step_clk 也有两路来源，由 CCSR 的 step_sel 位(bit8)来设置，为 0 的时候设置 step_clk 为 osc=24MHz。为 1 的时候不重要，不用。

④、时钟切换成功以后就可以修改 PLL1 的值。

⑤、通过 CCM_ANALOG_PLL_ARM 寄存器的 DIV_SELECT 位(bit6~0)来设置 PLL1 的频率，公式位：

$$\text{Output} = \text{fref} * \text{DIV_SEL} / 2 \quad 1056 = 24 * \text{DIV_SEL} / 2 \Rightarrow \text{DIV_SEL} = 88。$$

设置 CCM_ANALOG_PLL_ARM 寄存器的 DIV_SELECT 位=88 即可。PLL1=1056MHz
还要设置 CCM_ANALOG_PLL_ARM 寄存器的 ENABLE 位(bit13)为 1，也就是使能输出。

⑥、在切换回 PLL1 之前，设置 CACRR 寄存器的 ARM_PODF=1!! 切记。

裸机中断

一、回顾 STM32 中断系统

1、STM32 中断向量表

ARM 芯片从 0X00000000 开始运行，执行指令。在程序开始的地方存放着中断向量表。

中断向量表主要功能是描述中断对应的中断服务函数。

对于 STM32 来说代码最开始的地址存放堆栈栈顶指针。

2、中断向量偏移

一般 ARM 从 0X00000000 地址开始运行，对于 STM32 我们设置连接首地址为 0X80000000。

如果代码一定要从 0X80000000 开始运行，那么需要告诉一下 soc 内核。也就是设置中断向量偏移。设置 SCB 的 VTOR 寄存器为新的中断向量表起始地址即可。

3、NVIC 中断控制器。

NVIC 就是中断管理机构。使能和关闭指定的中断、设置中断优先级。

4、中断服务函数的编写

中断服务函数就是中断要做的事情。

二、Cortex-A7 中断系统

1、Cortex-A 中断向量表

Cortex-A 中断向量表有 8 个中断，其中重点关注 IRQ。Cortex-A 的中断向量表需要用户自己去定义。

2、中断向量偏移

我们的裸机历程都是从 0X87800000 开始的，因此要设置中断向量偏移。

3、GIC 中断控制器。

同 NVIC 一样，GIC 用于管理 Cortex-A 的中断。GIC 提供了开关中断，设置中断优先级。

4、IMX6U 中断号

为了区分不同的中断，引入了中断号。ID0~ID15 是给 SGI, ID16~ID31 是给 PPI。剩下的 ID32~1019 给 SPI，也就是按键中断、串口中断。。。。
6ULL 支持 128 个中断。

4、中断服务函数的编写

一个是 IRQ 中断服务函数的编写，另一个就是在 IRQ 中断服务函数里面去查找并运行的具体的外设中断服务函数，|

图 17.1.3.1 中左侧部分就是中断源，中间部分就是 GIC 控制器，最右侧就是中断控制器向处理器内核发送中断信息。我们重点要看的肯定是中间的 GIC 部分，GIC 将众多的中断源分为三类：

①、SPI(Shared Peripheral Interrupt), 共享中断，顾名思义，所有 Core 共享的中断，这个是最常见的，那些外部中断都属于 SPI 中断(注意！不是 SPI 总线那个中断)。比如按键中断、串口中断等等，这些中断所有的 Core 都可以处理，不限定特定 Core。

②、PPI(Private Peripheral Interrupt)，私有中断，我们说了 GIC 是支持多核的，每个核肯定有自己独有的中断。这些独有的中断肯定是要指定的核心处理，因此这些中断就叫做私有中断。

③、SGI(Software-generated Interrupt)，软件中断，由软件触发引起的中断，通过向寄存器 GICD_SGIR 写入数据来触发，系统会使用 SGI 中断来完成多核之间的通信。

四、中断实验编写

1、编写按键中断例程。

KEY0 使用 UART1_CTS 这个 IO。编写 UART1_CTS 的中断代码。

2、修改 start.S

添加中断向量表，编写复位中断服务函数和 IRQ 中断服务函数。

编写复位中断服务函数，内容如下：

①、关闭 I/D Cache 和 MMU。

②、设置处理器 9 中工作模式下对应的 SP 指针。要使用中断那么必须设置 IRQ 模式下的 SP 指针。索性直接设置所有模式下的 SP 指针。

③、清除 bss 段。

④、跳到 C 函数，也就是 main 函数

Start.s 开头根据机器发生中断跳转到的固定地址信息，在对应地址的位置放置跳转函数，完成中断向量表的构建


```

22 start:
23     ldr pc, =Reset_Handler      /* 复位中断 */
24     ldr pc, =Undefined_Handler /* 未定义中断 */
25     ldr pc, =SVC_Handler        /* SVC(Supervisor)中断 */
26     ldr pc, =PrefAbort_Handler  /* 预取终止中断 */
27     ldr pc, =DataAbort_Handler  /* 数据终止中断 */
28     ldr pc, =NotUsed_Handler    /* 未使用中断 */
29     ldr pc, =IRQ_Handler        /* IRQ中断 */
30     ldr pc, =FIQ_Handler        /* FIQ(快速中断)未定义中断 */

```

3、CP15 协处理器

MRC: 将 CP15 协处理器中的寄存器数据读到 ARM 寄存器中。

MRC 就是读 CP15 寄存器，MCR 就是写 CP15 寄存器，MCR 指令格式如下：

MRC{cond} p15, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>

MRC p15, 0, r0, c0, c0, 0

现在要关闭 I/D cache 和 MMU，打开 Cortex-A7 参考手册到 105 页，找到 SCTLR 寄存器。也就是系统控制寄存器，此寄存器 bit0 用于打开和关闭 MMU，bit1 对其控制位，bit2 控制 D Cache 的打开和关闭。Bit11 用于控制分支预测。Bit12 用于控制 I Cache。

中断向量偏移设置

将新的中断向量表首地址写入到 CP15 协处理器的 VBAR 寄存器。

MRC{cond} p15, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>

MRC p15, 0, r0, c12, c0, 0 //

MCR p15, 0, r0, c12, c0, 0

IRQ 中断服务函数

mrc p15, 4, r1, c15, c0, 0 读取 CP15 的 CBAR 寄存器。CBAR 寄存器保存了 GIC 控制器的寄存器组首地址。GIC 寄存器组偏移 0x1000~0x1fff 为 GIC 的分发器。0x2000~0x3fff 为 CPU 接口端。意味我们可以访问 GIC 控制器了！

代码中，R1 寄存器吧保存着 GIC 控制器的 CPU 接口端基地址。读取 CPU 接口段的 GICC_IAR 寄存器的值保存到 R0 寄存器里面。可以从 GICC_IAR 的 bit9~0 读取中断 ID，我们读取中断 ID 的目的就是为了得到对应的中断处理函数。

system_irqhandler 就是具体的中断处理函数，此函数有一个参数，为 GICC_IAR 寄存器的值。

system_irqhandler 处理完具体的中断以后，需要将对应的中断 ID 值写入到 GICC_EOIR 寄存器里面。

GPIO 中断设置

①、我们首先要设置 GPIO 的中断触发方式，也就是 GPIO_ICR1 或 ICR2 寄存器。触发方式有低电平、高电平、上升沿和下降沿。对于本例程来说，我们设置为 KEY0，也就是 UART1_CTS 这个 IO 为下降沿触发。

②、使能 GPIO 对应的中断，设置 GPIO_IMR 寄存器。

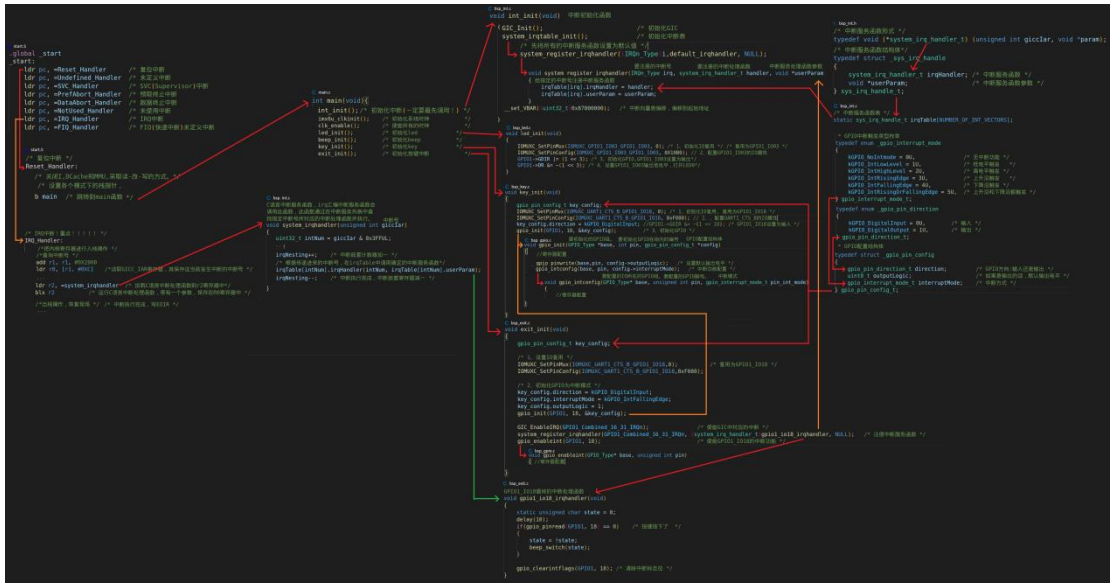
③、处理完中断以后，需要清除中断标志位，也就是清除 GPIO_ISR 寄存器相应的位。GPIO_ISR 寄存器是写 1 清零。

GIC 配置

①、使能相应的中断 ID。GPIO1_IO18 对应的中断 ID 位 67+32=99。

②、设置中断优先级

③、注册 GPIO1_IO18 的中断处理函数。



裸机定时器

一、EPIT 简介

- 1、EPIT 是 32 位的一个向下计数器。
- 2、EPIT 的时钟源可以选择，我们选择 `ipg_clk=66MHz`。
- 3、可以对时钟源进行分频，12 位的分频器，0~4095 分别代表 1~4096 分频。
- 4、开启定时器以后，计数寄存器会每个时钟减 1，如果和比较寄存器里面的值相等的话就会触发中断。
- EPIT 有两种工作模式：
Set-add-forget，一个是 `free-running`
- 5、6ULL 有两个 EPIT 定时器。
- EPIT_CR 寄存器用于配置 EPIT。

二、实验原理简介

EPIT_CR bit0 为 1，设置 EPIT 使能，bit1 为 1，设置计数器的初始值为记载寄存器的值。Bit2 为 1 使能比较中断，bit3 为 1 设置定时器工作在 set-and-forget 模式下。Bit15~bit4 设置分频值。Bit25:24 设置时钟源的选择，我们设置为 1，那么 EPIT 的时钟源就为 `ipg_clock=66MHz`

EPIT_SR 寄存器，只有 bit0 有效，表示中断状态，写 1 清零。当 OCIF 位为 1 的时候表示中断发生，为 0 的时候表示中断未发生。我们处理完定时器中断以后一定要清除中断标志位。

EPIT_LR 寄存器设置计数器的加载值。计数器每次计时到 0 以后就会读取 LR 寄存器的值重新开始计时。

CMPR 比较计数器，当计数器的值和 CMPR 相等以后就会产生比较中断。

使用 EPIT 实现 500ms 周期的定时器。我们在 EPIT 中断服务函数里面让 LED 灯亮灭。

Uboot 与移植

简介

一、何为 uboot?

1、uboot 是一个裸机程序，比较复杂。

2、uboot 就是一个 bootloader，作用就是用于启动 Linux 或其他系统。Uboot 最主要的工作就是初始化 DDR。因为 Linux 是运行在 DDR 里面的。一般 Linux 镜像 zImage(ulmage)+设备树(dtb)存放在 SD、EMMC、NAND、SPI FLASH 等等外置存储区域。

这里就牵扯到一个问题，需要将 Linux 镜像从外置 flash 拷贝到 DDR 中，再去启动。

Uboot 的主要目的就是为系统的启动做准备。

Uboot 不仅仅能启动 Linux，也可以启动其他系统，比如 vxworks。

Linux 不仅仅能通过 uboot 启动。

Uboot 是个通用的 bootloader，他支持多种架构。

Uboot 获取

1、首先就是 uboot 官网。缺点就是支持少，比如某一款具体芯片驱动等不完善。

2、SOC 厂商会从 uboot 官网下载某一个版本的 uboot，然后在这个版本的 uboot 上加入相应的 SOC 以及驱动。这就是 SOC 厂商定制版的 uboot。NXP 官方的 i.MX6ULL EVK 板子，

3、做开发板的厂商，开发板会参考 SOC 厂商的板子。开发板必然会和官方的板子不一样。因此开发板厂商又会去修改 SOC 厂商做好的 uboot，以适应自己的板子。

编译方法

二、正点原子官方 uboot 编译

1、编译 UBOOT 的时候需要先配置

1、编译完成以后就会生成一个 u-boot.bin。必须向 u-boot.bin 添加头部信息。Uboot 编译最后会通过/tools/mkimage 软件添加头部信息，生成 u-boot.imx。

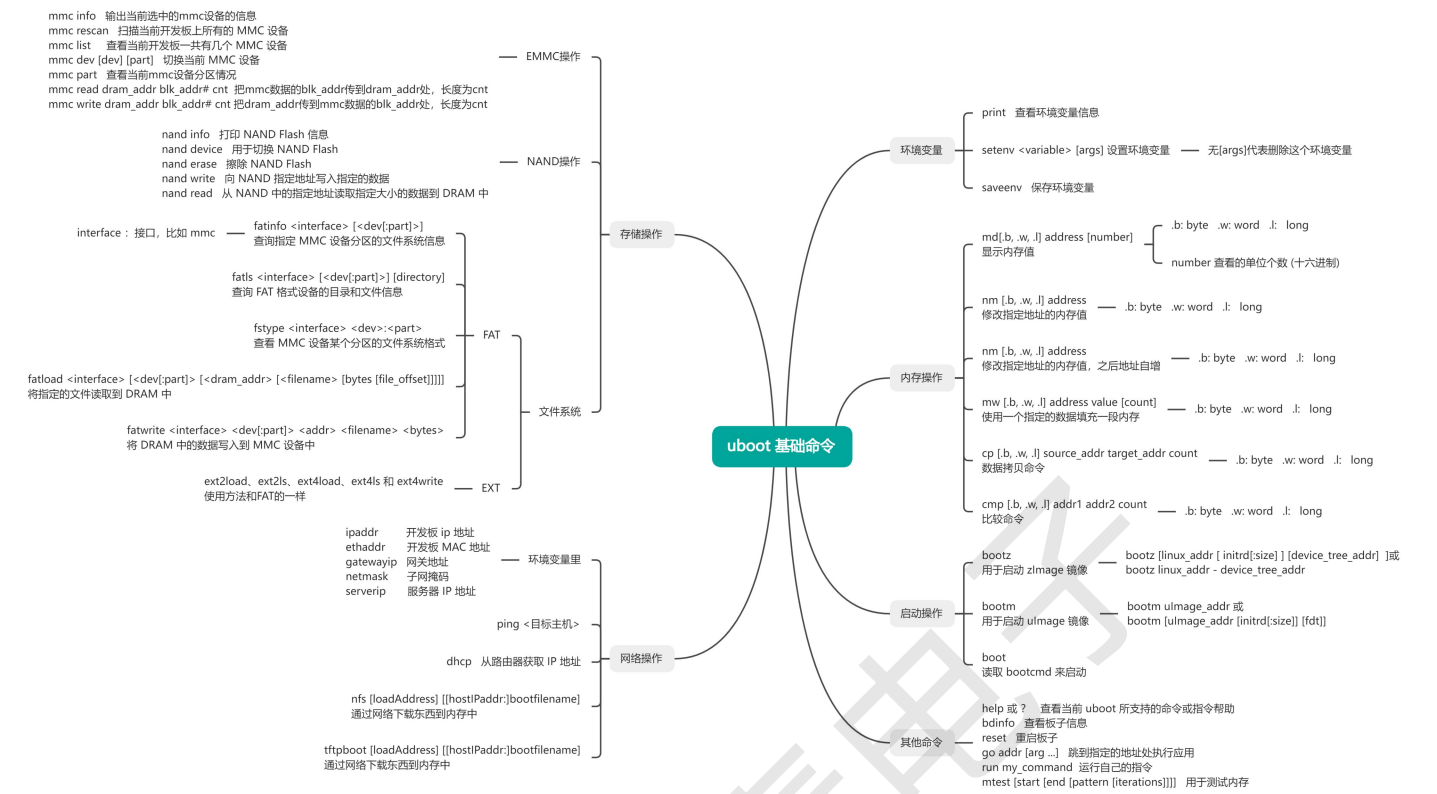
3、如果配置过 uboot，那么一定要注意 shell 脚本会清除整个工程，那么配置的文件也会被删除，配置项也会被删除掉。

```
U-Boot 2016.03 (Apr 01 2025 - 12:50:27 +0800)
CPU: Freescale i.MX6ULL rev1.1 792 MHz (running at 396 MHz)
CPU: Industrial temperature grade (-40C to 105C) at 44C
Reset cause: POR
Board: i.MX6U ALPHA/MINI
I2C: ready
DRAM: 512 MiB
MMC: FSL_SDHC: 0, FSL_SDHC: 1
*** Warning - bad CRC, using default environment

In: serial
Out: serial 输入输出和错误流位于串口
Err: serial
Switch to partitions #0, OK
mmc0 is current device
Net: FEC1
Error: FEC1 address not set.

Normal Boot
Hit any key to stop autoboot: 0
=>
```


Uboot 命令



bdinfo 使用举例

```
=> bdinfo
arch_number = 0x00000000
boot_params = 0x80000100 启动参数存放地址
DRAM bank = 0x00000000
-> start = 0x80000000
-> size = 0x20000000
eth0name = FEC1
ethaddr = (not set)
current eth = FEC1
ip_addr = <NULL>
baudrate = 115200 bps
TLB addr = 0x9FFF0000
relocaddr = 0x9FF55000
reloc off = 0x18755000
irq_sp = 0x9EF52E40
sp_start = 0x9EF52E90
```

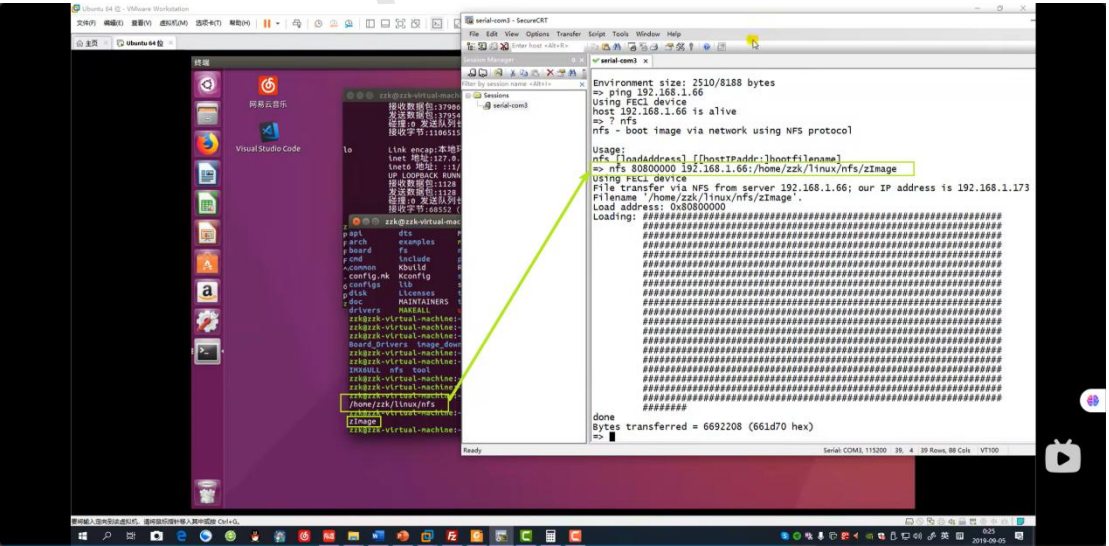
setenv 使用举例

```
=> setenv name 'pixel'
=> setenv bootdelay 9
```

saveenv 使用举例

```
=> saveenv
Saving Environment to MMC...
Writing to MMC(0)... done
```

nfs 命令应用举例



mmc 命令举例

命令	描述
mmc info	输出 MMC 设备信息
mmc read	读取 MMC 中的数据。
mmc write	向 MMC 设备写入数据。
mmc rescan	扫描 MMC 设备。
mmc part	列出 MMC 设备的分区。
mmc dev	切换 MMC 设备。
mmc list	列出当前有效的所有 MMC 设备。
mmc hwpartition	设置 MMC 设备的分区。
mmc bootbus.....	设置指定 MMC 设备的 BOOT_BUS_WIDTH 域的值。
mmc bootpart.....	设置指定 MMC 设备的 boot 和 RPMB 分区的大小。
mmc partconf.....	设置指定 MMC 设备的 PARTITION_CONFIG 域的值。
mmc rst	复位 MMC 设备
mmc setdsr	设置 DSR 寄存器的值。

表 30.4.5.1 mmc 命令

Uboot 源码分析

Uboot 目录结构

一、Uboot 源码目录分析

- 1、因为 uboot 会使用到一些经过编译才会生成的文件, 因此我们在分析 uboot 的时候, 需要先编译一下 uboot。
- 2、arch\arm\cpu\u-boot.lds 就是整个 UBOOT 的连接脚本
- 3、board\freescale\mx6ullevk. 重点。
- 4、configs 目录是 uboot 的默认配置文件目录。此目录下都是以_defconfig 结尾的。这些配置文件对应不同的板子。

mx6ul_14x14_dds_armv7_tsc_defconfig	2019-09-03 10:33	文件	1 KB
mx6ull_14x14_ddr256_emmc_defconfig	2019-09-03 10:33	文件	1 KB
mx6ull_14x14_ddr256_nand_defconfig	2019-09-03 10:33	文件	1 KB
mx6ull_14x14_ddr256_nand_sd_defconfig	2019-09-03 10:33	文件	1 KB
mx6ull_14x14_ddr512_emmc_defconfig	2019-09-03 10:33	文件	1 KB
mx6ull_14x14_ddr512_nand_defconfig	2019-09-03 10:33	文件	1 KB
mx6ull_14x14_ddr512_nand_sd_defconfig	2019-09-03 10:33	文件	1 KB

mx6ull_alientek_alpha_ddr256_emmc_defconfig

- 5、我们移植 uboot 的时候重点要关注。

board\freescale

\configs, 主要是_defconfig

- 6、当我们执行 make xxx_defconfig 以后就会生成.config 文件, 此文件保存了详细的 uboot 配置信息。

- 7、顶层 README, 非常重要, 建议大家阅读, 介绍 uboot。

- 8、u-boot. 这个就是编译出来带 ELF 信息的 uboot 可执行文件。

类型	名字	描述	备注
文件夹	api	与硬件无关的 API 函数。	uboot 自带
	arch	与架构体系有关的代码。	
	board	不同板子(开发板)的定制代码。	
	cmd	命令相关代码。	
	common	通用代码。	
	configs	配置文件。	
	disk	磁盘分区相关代码	
	doc	文档。	
	drivers	驱动代码。	
	dts	设备树。	
	examples	示例代码。	
	fs	文件系统。	
	include	头文件。	
	lib	库文件。	
	Licenses	许可证相关文件。	
	net	网络相关代码。	

	post	上电自检程序。	
	scripts	脚本文件。	
	test	测试代码。	
	tools	工具文件夹。	
文件	.config	配置文件，重要的文件。	编译生成的文件
	.gitignore	git 工具相关文件。	uboot 自带
	.mailmap	邮件列表。	
	.u-boot.xxx.cmd (一系列)	这是一系列的文件，用于保存着一些命令。	编译生成的文件
	config.mk	某个 Makefile 会调用此文件。	uboot 自带
	imxdownload	正点原子编写的 SD 卡烧写软件。	正点原子提供
	Kbuild	用于生成一些和汇编有关的文件。	uboot 自带
	Kconfig	图形配置界面描述文件。	
	MAINTAINERS	维护者联系方式文件。	
	MAKEALL	一个 shell 脚本文件，帮助编译 uboot 的。	
	Makefile	主 Makefile，重要文件！	
	mx6ull_alientek_emmc.sh	上一章编写的编译脚本文件	上一章编写的。
	mx6ull_alientek_nand.sh	上一章编写的编译脚本文件	
	README	相当于帮助文档。	uboot 自带
	snapshot.commint	？ ？ ？	
	System.map	系统映射文件	编译出来的文件
	u-boot	编译出来的 u-boot 文件。	
	u-boot.xxx (一系列)	生成的一些 u-boot 相关文件，包括 u-boot.bin、u-boot.imx 等	

表 31.1.1 uboot 目录列表

Uboot makefile 分析

主 makefile 调用 subdir 下的 makefile

```
$(MAKE) -C subdir
```

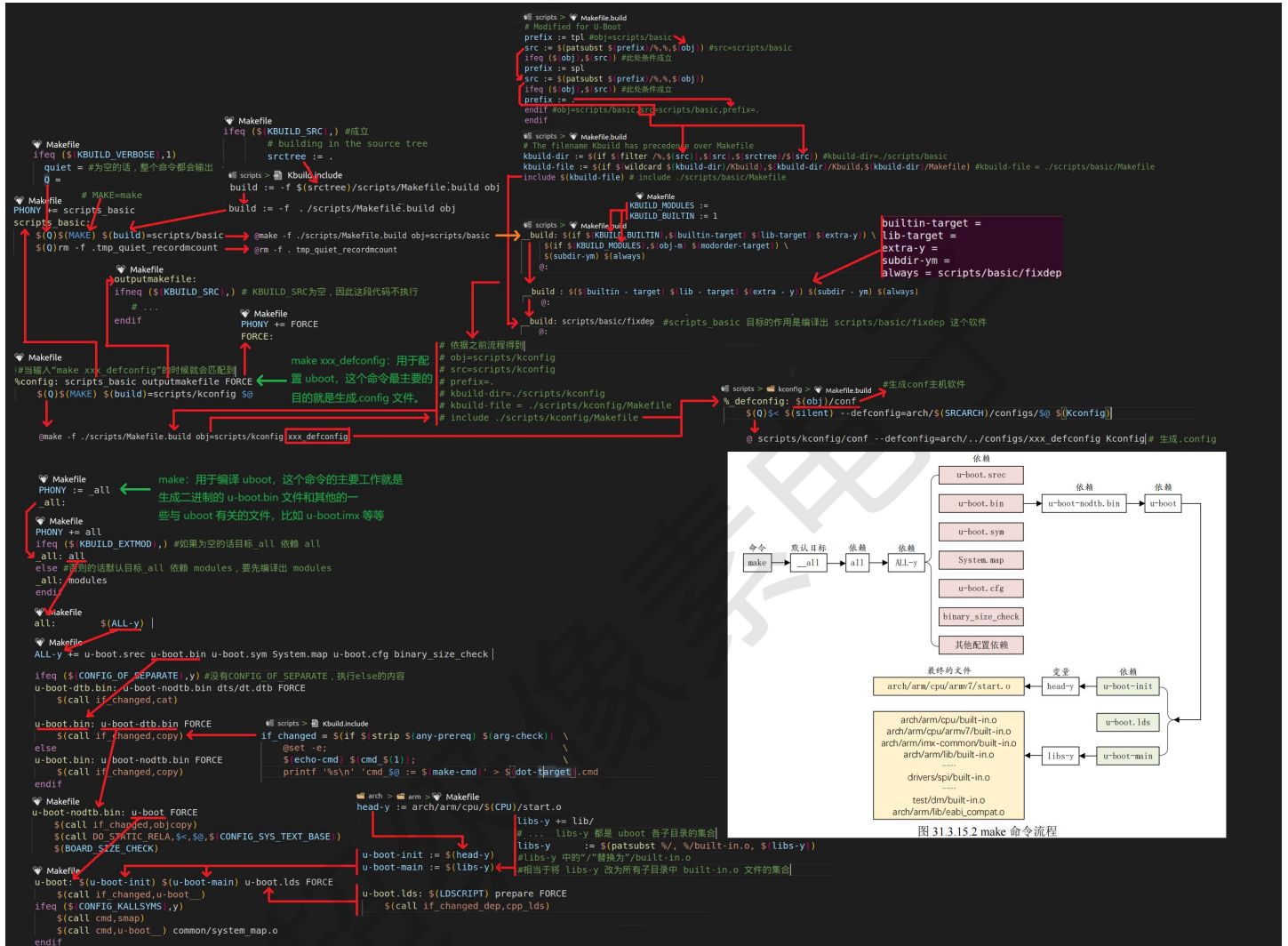
传递变量到其他的 makefile

```
export VARIABLE ..... //导出变量给子 make 。
unexport VARIABLE..... //不导出变量给子 make。
```

有两个特殊的变量：“SHELL”和“MAKEFLAGS”，这两个变量除非使用“unexport”声明，否则的话在整个make的执行过程中，它们的值始终自动的传递给子make。

make xxx_defconfig: 用于配置 uboot，这个命令最主要的目的就是生成.config 文件。

make: 用于编译 uboot，这个命令的主要工作就是生成二进制的 u-boot.bin 文件和其他的一些与 uboot 有关的文件，比如 u-boot.imx 等等。



Uboot 启动源码分析

特性	cpu_init_cp15:	ENTRY(cpu_init_cp15)
语法	直接汇编标签	宏展开后的标签（可能包含额外信息）
作用域	默认局部（取决于编译器）	通常声明为全局符号（如 <code>.global</code> ）
调试信息	无	可能包含类型和大小信息
常见使用场景	普通汇编代码	内核或复杂项目（如 Linux 内核）

位置无关码

```

826 static int rel_a = 0;
827
828 void rel_test(void)
829 {
830     rel_a = 100;
831     printf("rel_test\r\n");
832 }
833
834 int board_init(void)
835 {
836     rel_test();

```

示例代码 32.2.6.3 汇编文件代码段

```

1  87804184 <rel_test>:
2  87804184: e59f300c ldr r3, [pc, #12] ; 87804198 <rel_test+0x14>
3  87804188: e3a02064 mov r2, #100 ; 0x64
4  8780418c: e59f0008 ldr r0, [pc, #8] ; 8780419c <rel_test+0x18>
5  87804190: e5832000 str r2, [r3]
6  87804194: ea00d668 b 87839b3c <printf>
7  87804198: 8785da50 ; <UNDEFINED> instruction: 0x8785da50
8  8780419c: 878426a2 strhi r2, [r4, r2, lsr #13]
9
10 878041a0 <board_init>:
11 878041a0: e92d4010 push {r4, lr}
12 878041a4: ebf0ffff bl 87804184 <rel_test>
13
14 .....
15
16 8785da50 <rel_a>:
17 8785da50: 00000000 andeq r0, r0, r0

```

总结一下 `rel_a=100` 的汇编执行过程:

- ①、在函数 `rel_test` 末尾处有一个地址为 `0X87804198` 的内存空间(示例代码 32.2.6.3 第 7 行), 此内存空间保存着变量 `rel_a` 的地址。
- ②、函数 `rel_test` 要想访问变量 `rel_a`, 首先访问末尾的 `0X87804198` 来获取变量 `rel_a` 的地址, 而访问 `0X87804198` 是通过偏移来访问的, 很明显是个位置无关的操作。
- ③、通过 `0X87804198` 获取到变量 `rel_a` 的地址, 对变量 `rel_a` 进行操作。
- ④、可以看出, 函数 `rel_test` 对变量 `rel_a` 的访问没有直接进行, 而是使用了一个第三方偏移地址 `0X87804198`, 专业术语叫做 `Label`。这个第三方偏移地址就是实现重定位后运行不会出错的重要原因!

重定位以后

```

9ffa4cf8 <rel_a>:
9ffa4cf8: 00000000 andeq r0, r0, r0

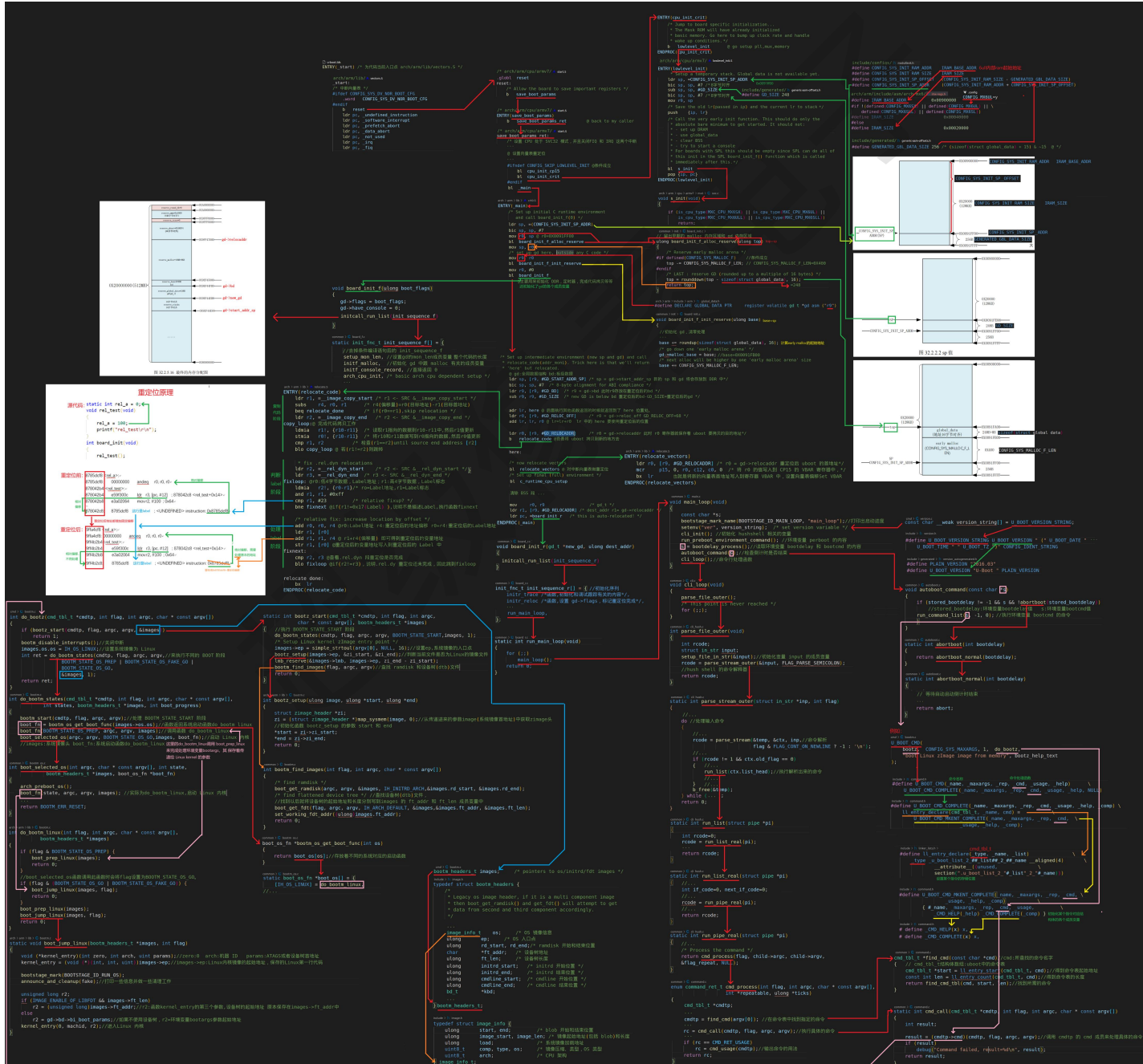
9ff4b2b4 <rel_test>:
9ff4b2b4: e59f300c ldr r3, [pc, #12] ; 878042c8 <rel_test+0x14>
9ff4b2b8: e3a02064 mov r2, #100 ; 0x64
9ff4b2bc: e59f0008 ldr r0, [pc, #8] ; 878042cc <rel_test+0x18>
9ff4b2c0: e5832000 str r2, [r3]
9ff4b2c4: ea00d64c b 87839bfc <printf>

```


t

2

根据前面的分析，只要将地址 0X87804198+offset 处的值改为重定位后的变量 rel_a 地址即可



ARGS和command的默认配置

ARGS和command的默认配置

uchar default_environment[]

commands的配置过程

最后实际执行的代码

```
mmc dev 1 //切换到EMMC
fatload mmc 1:1 0x80800000 zimage //读取zimage到
fatload mmc 1:1 0x83000000 imxfull-14x14-evk.dtb //读取设备树到
bootz (0x80800000 - 0x83000000) //启动Linux
```

args的配置过程

Uboot 的板级头文件

```

1  /*
2  * Copyright (C) 2016 Freescale Semiconductor, Inc.
3  *
4  * Configuration settings for the Freescale i.MX6UL 14x14 EVK board.
5  *
6  * SPDX-License-Identifier: GPL-2.0+
7  */
8  #ifndef _MX6ULL_ALBITENK_EMMC_CONFIG_H
9  #define _MX6ULL_ALBITENK_EMMC_CONFIG_H
10
11
12 #include <asm/arch/imx-regs.h>
13 #include <linux/sizes.h>
14 #include "mx6_common.h"
15 #include <asm/imx-common/gpio.h>
16
17 .....
18
19 #define is_mx6ull_9x9_evk() CONFIG_IS_ENABLED(TARGET_MX6ULL_9X9_EVK)
20
21 #ifdef CONFIG_TARGET_MX6ULL_9X9_EVK
22 #define PHYS_SDRAM_SIZE SZ_256M
23 #define CONFIG_BOOTARGS_CMA_SIZE "cma=96M"
24 #else
25 #define PHYS_SDRAM_SIZE SZ_512M
26 #define CONFIG_BOOTARGS_CMA_SIZE ""
27 /* DCDC used on 14x14 EVK, no PMIC */
28 #undef CONFIG_LDO_BYPASS_CHECK
29 #endif
30
31 /* SPL options */
32 /* We default not support SPL
33 * #define CONFIG_SPL_LIBCOMMON_SUPPORT
34 * #define CONFIG_SPL_MMC_SUPPORT
35 * #include "mx6_spl.h"
36 */
37
38 #define CONFIG_ENV_VARS_UBOOT_RUNTIME_CONFIG
39
40 #define CONFIG_DISPLAY_CPUINFO 定义宏 CONFIG_DISPLAY_CPUINFO, uboot 启动的时候可以输出 CPU 信息
41 #define CONFIG_DISPLAY_BOARDINFO 定义宏 CONFIG_DISPLAY_BOARDINFO, uboot 启动的时候可以输出板子信息
42
43 /* Size of malloc() pool */
44 #define CONFIG_SYS_MALLOC_LEN ((16 * SZ_1M) CONFIG_SYS_MALLOC_LEN for malloc 内存池大小, 这里设置为 16MB
45
46 #define CONFIG_BOARD_EARLY_INIT_F 定义宏 CONFIG_BOARD_EARLY_INIT_F, 这样 board_init_f 函数就会调用 board_early_init_f 函数
47 #define CONFIG_BOARD_LATE_INIT 定义宏 CONFIG_BOARD_LATE_INIT, 这样 board_init_r 函数就会调用 board_late_init 函数
48
49 #define CONFIG_MXC_UART
50 #define CONFIG_MXC_UART_BASE UART1_BASE 使能 IMX6ULL 的串口功能
51
52 /* MMC Configs */
53
54 #ifdef CONFIG_FSL_USDHC
55 #define CONFIG_SYS_FSL_ESDHC_ADDR USDHC2_BASE_ADDR 宏 CONFIG_SYS_FSL_ESDHC_ADDR for EMMC 所使用接口的寄存器地址, 也就是 USDHC2 的基地址
56 #define CONFIG_SYS_FSL_USDHC_NUM 1 跟 NAND 相关的宏
57 #else
58 #define CONFIG_SYS_FSL_USDHC_NUM 2
59 #endif
60 #endif
61
62 /* I2C configs */
63 #define CONFIG_CMD_I2C
64 #ifdef CONFIG_CMD_I2C
65 #define CONFIG_SYS_I2C
66 #define CONFIG_SYS_I2C_MXC
67 #define CONFIG_SYS_I2C_MXC_I2C1 /* enable I2C bus 1 */
68 #define CONFIG_SYS_I2C_MXC_I2C2 /* enable I2C bus 2 */
69 #define CONFIG_SYS_I2C_SPEED 100000
70
71 .....
72
73 #define CONFIG_SYS_MMC_IMG_LOAD_PART 1
74
75 #ifdef CONFIG_SYS_BOOT_NAND
76 #define CONFIG_MFG_NAND_PARTITION "mtdparts=gmml-nand:64m(boot),16m(kernel),16m(dtb),1m(misc),-(rootfs)"
77 #else
78 #define CONFIG_MFG_NAND_PARTITION ""
79 #endif
80
81 #define CONFIG_MFG_ENV_SETTINGS \
82 "mfgtool_args=setenv bootargs console=${console},${baudrate}" \
83 .....
84 "bootcmd_mfg=run mfgtool_args;bootz ${loadaddr} ${initrd_addr} ${fdt_addr};\0" \
85
86 #if defined(CONFIG_SYS_BOOT_NAND)
87 #define CONFIG_EXTRA_ENV_SETTINGS \
88 CONFIG_MFG_ENV_SETTINGS \
89 "panel=TF743AB\0" \
90 .....
91 "bootz ${loadaddr} - ${fdt_addr}\0"
92
93 #else
94 #define CONFIG_EXTRA_ENV_SETTINGS \
95 CONFIG_MFG_ENV_SETTINGS \
96 "script=boot.scr\0" \
97 .....
98 "fi;\0" \
99
100 #define CONFIG_BOOTCOMMAND \
101 "run findfdt;" \
102 .....
103 "else run netboot; fi"
104 #endif

```

第 29-39 行, 设置 DRAM 的大小, 宏 PHYS_SDRAM_SIZE 就是板子上 DRAM 的大小, 如果用的是 NXP 官方的 9X9 EVK 开发板的话 DRAM 大小就为 256MB, 否则的话默认为 512MB, 正点原子的 IMX6U-ALPHA 开发板用的是 512MB DDR3。

宏 CONFIG_SYS_FSL_ESDHC_ADDR for EMMC 所使用接口的寄存器地址, 也就是 USDHC2 的基地址

跟 NAND 相关的宏

和 I2C 有关的宏定义

NAND 的分区设置

CONFIG_MFG_ENV_SETTINGS 定义了一些环境变量, 使用 MfgTool 烧写系统时候会用到这里面的环境变量

通过条件编译来设置宏 CONFIG_EXTRA_ENV_SETTINGS, 宏 CONFIG_MFG_ENV_SETTINGS 也是设置一些环境变量, 此宏会设置 bootargs 这个环境变量

设置宏 CONFIG_BOOTCOMMAND, 此宏就是设置环境变量 bootcmd 的值

```

218
219 /* Miscellaneous configurable options */
220 #define CONFIG_CMD_MEMTEST
221 #define CONFIG_SYS_MEMTEST_START 0x40000000 设置命令 memtest 相关宏定义
222 #define CONFIG_SYS_MEMTEST_END (CONFIG_SYS_MEMTEST_START + 0x4000000)
223
224 #define CONFIG_SYS_LOAD_ADDR CONFIG_LOADADDR 宏 CONFIG_SYS_LOAD_ADDR 表示 linux kernel 在 DRAM 中的加载地址, 也就是 linux kernel 在 DRAM 中的存储基地址
225 #define CONFIG_SYS_HZ 1000 宏 CONFIG_SYS_HZ 为系统时钟频率, 这里为 1000Hz
226
227 #define CONFIG_STACKSIZE SZ_128K 宏 CONFIG_STACKSIZE 为栈大小, 这里为 128KB
228
229 /* Physical Memory Map */
230 #define CONFIG_NR_DRAM_BANKS 1 宏 CONFIG_NR_DRAM_BANKS 为 DRAM BANK 的数量
231 #define PHYS_SDRAM MMIO_ASB_BASE_ADDR 宏 PHYS_SDRAM 为 IMX6ULL 的 DRAM 控制器 MMIO 所管理的 DRAM 范围起始地址, 也就是 0x80000000
232 #define CONFIG_SYS_SDRAM_BASE PHYS_SDRAM 宏 CONFIG_SYS_SDRAM_BASE 为 DRAM 起始地址
233 #define CONFIG_SYS_INIT_RAM_ADDR IRAM_BASE_ADDR 宏 CONFIG_SYS_INIT_RAM_ADDR 为 IMX6ULL 内部 IRAM (OCRAM) 的起始地址, 为 0x00900000
234 #define CONFIG_SYS_INIT_RAM_SIZE IRAM_SIZE 宏 CONFIG_SYS_INIT_RAM_SIZE 为 IMX6ULL 内部 IRAM 的大小 (OCRAM 的大小), 为 0x00040000=128KB
235
236 #define CONFIG_SYS_INIT_SP_OFFSET \
237 (CONFIG_SYS_INIT_RAM_SIZE - GENERATED_GBL_DATA_SIZE) 宏 CONFIG_SYS_INIT_SP_OFFSET 和 CONFIG_SYS_INIT_SP_ADDR 与初始 SP 有关, 第一个为初始 SP 偏移, 第二个为初始 SP 地址
238 #define CONFIG_SYS_INIT_SP_ADDR \
239 (CONFIG_SYS_INIT_RAM_ADDR + CONFIG_SYS_INIT_SP_OFFSET)
240
241
242 /* FLASH and environment organization */
243 #define CONFIG_SYS_NO_FLASH
244
245 .....
246
247 #define CONFIG_SYS_MMC_ENV_DEV 1 /* USDHC2 */ 宏 CONFIG_SYS_MMC_ENV_DEV 为默认以 MMC 设备, 这里默认为 USDHC2, 也就是 EMMC
248 #define CONFIG_SYS_MMC_ENV_PART 0 /* user area */ 宏 CONFIG_SYS_MMC_ENV_PART 为默认分区, 默认为第 0 个分区
249 #define CONFIG_MMCROOT "/dev/mmcblkp1" /* USDHC2 */ 宏 CONFIG_MMCROOT 设置进入 linux 系统的根文件系统所在的分区
250
251 #define CONFIG_CMD_BMODE
252
253 .....
254
255 /* NAND stuff */
256 #ifdef CONFIG_SYS_USE_NAND
257 #define CONFIG_CMD_NAND
258 #define CONFIG_CMD_NAND_TRIMFFS
259
260 #define CONFIG_NAND_MXS
261 #define CONFIG_SYS_NAND_DEVICE 1
262 #define CONFIG_SYS_NAND_BASE 0x40000000
263 #define CONFIG_SYS_NAND_1_ADDR_CYCLE
264 #define CONFIG_SYS_NAND_ONFI_DETECTION
265
266 /* DMA stuff, needed for GPMI/MXS NAND support */
267 #define CONFIG_APBH_DMA
268 #define CONFIG_APBH_DMA_BURST
269 #define CONFIG_APBH_DMA_BURST9
270 #endif
271
272 #define CONFIG_ENV_SIZE SZ_8K 宏 CONFIG_ENV_SIZE 为环境变量大小, 默认为 8KB
273 #if defined(CONFIG_ENV_IS_IN_MMC)
274 #define CONFIG_ENV_OFFSET ((1 * SZ_64K)
275 #elif defined(CONFIG_ENV_IS_IN_SPI_FLASH)
276 #define CONFIG_ENV_OFFSET (0x4 * 1024)
277 #define CONFIG_ENV_SECT_SIZE (64 * 1024) 宏 CONFIG_ENV_OFFSET 为环境变量偏移地址
278 #define CONFIG_ENV_SPT_SIZE CONFIG_SF_DEFAULT_SIZE
279 #define CONFIG_ENV_SPT_CS CONFIG_SF_DEFAULT_CS
280 #define CONFIG_ENV_SPT_MODE CONFIG_SF_DEFAULT_MODE
281 #define CONFIG_ENV_SPT_MAX_HZ CONFIG_SF_DEFAULT_SPEED
282 #elif defined(CONFIG_ENV_IS_IN_NAND)
283 #define CONFIG_ENV_SIZE 304
284 #define CONFIG_ENV_OFFSET ((0 << 20)
285 #define CONFIG_ENV_SECT_SIZE (128 << 10)
286 #define CONFIG_ENV_SIZE CONFIG_ENV_SECT_SIZE
287 #endif
288
289
290
291 /* USB Configs */
292 #define CONFIG_CMD_USB
293 #ifdef CONFIG_CMD_USB
294 #define CONFIG_USB_EHCI
295 #define CONFIG_USB_EHCI_MX6
296 #define CONFIG_USB_STORAGE
297 #define CONFIG_USB_EHCI_HCD_INIT_AFTER_RESET
298 #define CONFIG_USB_HOST_ETHER
299 #define CONFIG_USB_ETHER_ASIX
300 #define CONFIG_MXC_USB_PORTC (PORT_PTS_UTMI | PORT_PTS_PTW)
301 #define CONFIG_MXC_USB_FLAGS 0
302 #define CONFIG_USB_MAX_CONTROLLER_COUNT 2
303 #endif
304
305 #ifdef CONFIG_CMD_NET
306 #define CONFIG_CMD_PING
307 #define CONFIG_CMD_DHCP
308 #define CONFIG_CMD_MII
309 #define CONFIG_FEC_MXC
310 #define CONFIG_MII
311 #define CONFIG_FEC_ENET_DEV 1
312
313 #if (CONFIG_FEC_ENET_DEV == 0)
314 #define IMX_FEC_BASE ENET_BASE_ADDR
315 #define CONFIG_FEC_MXC_PHYADDR 0x0
316 #define CONFIG_FEC_XCV_TYPE RMII
317 #elif (CONFIG_FEC_ENET_DEV == 1)
318 #define IMX_FEC_BASE ENET1_BASE_ADDR
319 #define CONFIG_FEC_MXC_PHYADDR 0x1
320 #define CONFIG_FEC_XCV_TYPE RMII
321 #endif
322 #define CONFIG_ETHERNAME "FEC"
323
324 #define CONFIG_PHYLIB
325 #define CONFIG_PHY_MICREL
326 #endif
327
328 #define CONFIG_IMX_THERMAL
329
330 #ifndef CONFIG_SPL_BUILD
331 #define CONFIG_VIDEO
332 #ifdef CONFIG_VIDEO
333 #define CONFIG_CFB_CONSOLE
334 #define CONFIG_VIDEO_MXS
335 #define CONFIG_VIDEO_LOGO
336 #define CONFIG_VIDEO_RM_CURSOR
337 #define CONFIG_VGA_AS_SINGLE_DEVICE
338 #define CONFIG_SYS_CONSOLE_IS_IN_ENV
339 #define CONFIG_SPLASH_SCREEN
340 #define CONFIG_SPLASH_SCREEN_ALIGN
341 #define CONFIG_CMD_BMP
342 #define CONFIG_BMP_16PP
343 #define CONFIG_VIDEO_BMP_LBS
344 #define CONFIG_VIDEO_BMP_LOGO
345 #define CONFIG_IMX_VIDEO_SKIP
346 #endif
347 #endif
348
349 #define CONFIG_IOMUX_LPSR
350
351 .....
352 #endif

```

设置命令 memtest 相关宏定义

宏 CONFIG_SYS_LOAD_ADDR 表示 linux kernel 在 DRAM 中的加载地址, 也就是 linux kernel 在 DRAM 中的存储基地址

宏 CONFIG_NR_DRAM_BANKS 为 DRAM BANK 的数量

宏 PHYS_SDRAM 为 IMX6ULL 的 DRAM 控制器 MMIO 所管理的 DRAM 范围起始地址, 也就是 0x80000000

宏 CONFIG_SYS_SDRAM_BASE 为 DRAM 起始地址

宏 CONFIG_SYS_INIT_RAM_ADDR 为 IMX6ULL 内部 IRAM (OCRAM) 的起始地址, 为 0x00900000

宏 CONFIG_SYS_INIT_RAM_SIZE 为 IMX6ULL 内部 IRAM 的大小 (OCRAM 的大小), 为 0x00040000=128KB

宏 CONFIG_SYS_INIT_SP_OFFSET 和 CONFIG_SYS_INIT_SP_ADDR 与初始 SP 有关, 第一个为初始 SP 偏移, 第二个为初始 SP 地址

与 NAND 有关的宏定义

宏 CONFIG_ENV_OFFSET 为环境变量偏移地址

与 USB 相关的宏定义

与网络相关的宏定义

Uboot 移植

加入自己的板级配置

外层配置

```
1 #!/bin/bash 编译脚本文件
2 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- distclean
3 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- mx6ull_14x14_evk emmc_defconfig
4 make V=1 ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- -j16
5

configs > mx6ull_14x14_evk_emmc_defconfig 具体某个板子的配置文件
1 CONFIG_SYS_EXTRA_OPTIONS="IMX_CONFIG=board/freescale/mx6ullevk/imximage.cfg,MX6ULL_EVK_EMMC_REWORK"
2 CONFIG_ARM=y
3 CONFIG_ARCH_MX6=y
4 CONFIG_TARGET_MX6ULL_14X14_EVK=y
5 CONFIG_CMD_GPIO=y
```

板级配置

```
board > freescale > mx6ullevk > imximage.cfg 自动映像配置文件 包含DDR初始化和校准参数, 使用
1 /*
2  * Copyright (C) 2016 Freescale Semiconductor, Inc.
3  *
4  * SPDX-License-Identifier: GPL-2.0+
5  *
6  * Refer docs/README.imximage for more details about how-to configure
7  * and create imximage boot image
8  *
9  * The syntax is taken as close as possible with the kwbimage
10 */
11
12 #define __ASSEMBLY__
13 #include <config.h>
14
15 /* Image version */
16
17 IMAGE_VERSION 2
18
19 /*
20  * Boot Device : one of
21  * spi, sd (the board has no nand neither onenand)
22 */
23
24 BOOT_FROM sd
25
26 #ifdef CONFIG_USE_PLUGIN
27 /*PLUGIN plugin-binary-file IRAM FREE START ADDR*/
28 PLUGIN board/freescale/mx6ullevk/plugin.bin 0x0907000
29 #else
30
31 #ifdef CONFIG_SECURE_BOOT
32 CSF_CONFIG_CSF_SIZE
33 #endif
34 #endif
```

芯片级配置

```
arch > arm > cpu > armv7 > mx6 > {} Kconfig 芯片级配置
193 select DM_THERMAL
194
195 config TARGET_MX6ULL_14X14_EVK
196 bool "Support mx6ull_14x14_evk"
197 select MX6ULL
198 select DM
199 select DM_THERMAL
200
201 config TARGET_MX6ULL_9X9_EVK
202 bool "Support mx6ull_9x9_evk"
203 select MX6ULL
204 select DM
205 select DM_THERMAL
206
207 config TARGET_SECOMX6
208 bool "secomx6 boards"
209
210 config TARGET_TBS2910
211 bool "TBS2910 Matrix ARM mini PC"
212
213 source "board/freescale/mx6ul_14x14_evk/Kconfig"
```

开发板头文件

```
include <configs> | mx6ullevk.h > ... 开发板对应的头文件
1 /*
2  * Copyright (C) 2016 Freescale Semiconductor, Inc.
3  *
4  * Configuration settings for the Freescale i.MX6UL 14x14 EVK board.
5  *
6  * SPDX-License-Identifier: GPL-2.0+
7  */
8
9 #ifndef __MX6ULLEVK_CONFIG_H
10 #define __MX6ULLEVK_CONFIG_H
11
12 #include <asm/arch/imx-regs.h>
13 #include <linux/sizes.h>
14 #include "mx6_common.h"
15 #include <asm/imx-common/gpio.h>
16 #endif
```

1. 弄混 sdb1 和 sdb，导致程序烧不进去

1. /dev/sdb：物理磁盘设备

- 含义：表示系统中的 第二块物理磁盘（a 是第一块，b 是第二块，依此类推）。
- 作用：
 - 对应整块物理磁盘（如 U 盘、SSD、HDD）。
 - 不包含分区信息，直接操作会覆盖整个磁盘（危险！）。

2. /dev/sdb1：磁盘的第一个分区

- 含义：表示 /dev/sdb 这块磁盘上的 第一个分区（数字 1 表示分区编号）。
- 作用：
 - 对应磁盘上划分的逻辑存储区域。
 - 需要挂载到目录才能访问文件（如 /mnt/usb）。

```
alientek@zdyz-imx6ull: ~/linux/IMX6ULL/my_uboot2$ ./imxdownload u-boot.bin /dev/sdb
I.MX6ULL bin download software
Edit by: zuozhongkai
Date: 2019/6/10
Version: V1.1
log: V1.0 initial version, just support 512MB DDR3
      V1.1 and support 256MB DDR3
file u-boot.bin size = 419540Bytes
Board DDR SIZE: 512MB
Delete Old load.imx
Create New load.imx
Download load.imx to /dev/sdb .....
825+1 records in
825+1 records out
422612 bytes (423 kB, 413 KiB) copied, 4.26277 s, 99.1 kB/s
```

网络移植

基本内容

6ULL 网络方案采用内部 MAC+外部 PHY，6ULL 官方开发板使用的 PHY 芯片就是 KSZ8081。正点原子的 ALPHA 开发板没有使用 KSZ8081，我们使用的 LAN8720A。因此要修改驱动。

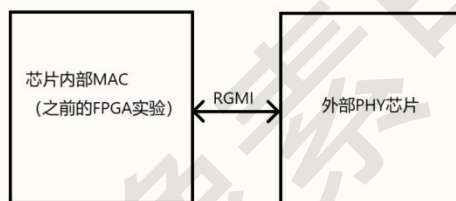
LAN8720 有一个管理接口，叫做 MDIO，两根线，MDIO 和 MDC，一个 MDIO 接口可以管理 32 个 PHY 芯片。MIDO 通过 PHY ADDR 来确定访问那个 PHY 芯片。ALPHA 开发板 ENET1 的 PHY ADDR 是 0x0，ENET2 的 PHY ADDR 是 0x1。

每个 LAN8720 都有一个复位引脚，ENET1 是 SNVS_TAMPER7，ENET2 是 SNVS_TAMPER8。

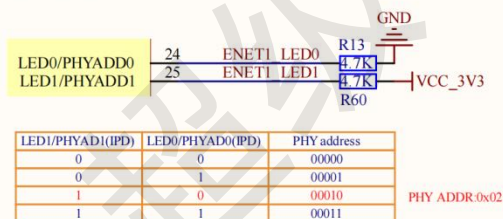
LAN8720 驱动，因为所有的 PHY，其前 32 个寄存器一模一样，因此 uboot 里面已经写好了通用 PHY 驱动，所以理论上不需要修改。

驱动修改

- 1、修改 PHY ADDR
- 2、删除原有的 74LV595 相关代码。
- 3、添加 ALPHA 开发板的网络复位 IO

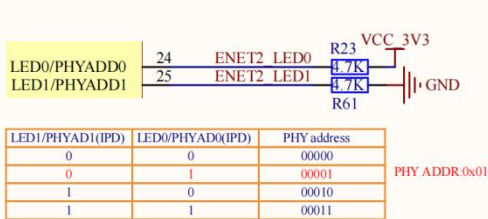


ENET1



```
h my_board.h
#define CONFIG_FEC_ENET_DEV 0
#if (CONFIG_FEC_ENET_DEV == 0)
#define IMX_FEC_BASE ENET_BASE_ADDR
#define CONFIG_FEC_MXC_PHYADDR 0x2
#define CONFIG_FEC_XCV_TYPE RMII
```

ENET2



```
h my_board.h
#define CONFIG_FEC_ENET_DEV 1
#define CONFIG_FEC_ENET_DEV == 1)
#define IMX_FEC_BASE ENET2_BASE_ADDR
#define CONFIG_FEC_MXC_PHYADDR 0x1
#define CONFIG_FEC_XCV_TYPE RMII
```

网络配置内容:

```
setenv ipaddr 192.168.5.9
setenv ethaddr b8:ae:1d:01:00:00
setenv gatewayip 192.168.5.1
setenv netmask 255.255.255.0
setenv serverip 192.168.5.11
saveenv
```


网络配置相关问题

1.桥接网卡一定要 VMnet0



 在 VMware 中，必须使用 **VMnet0** 来配置桥接模式，原因如下：

1. VMnet0 是 VMware 的默认桥接虚拟网络

- 设计逻辑：
VMware 虚拟网络有预定义的用途，**VMnet0** 是专门为 桥接模式 (Bridged) 保留的默认虚拟网络。其他 **VMnet**（如 **VMnet1**、**VMnet8**）分别对应 仅主机 (Host-Only) 和 NAT 模式。
- 系统约定：
VMware 底层网络驱动和虚拟机监控程序会默认将桥接模式的流量关联到 **VMnet0**，直接修改其配置最可靠。

2. 各虚拟网络的作用

(1) vmnet0 (桥接模式 - Bridged)

- 功能：虚拟机直接连接到物理网络，就像一台独立的物理设备。
- IP 分配：由物理网络的 DHCP 服务分配（如家庭路由器）。
- 通信范围：
 - 虚拟机可以访问 局域网 (LAN) 和 互联网。
 - 局域网内的其他设备可以直接访问虚拟机。
- 主机上的表现：
 - vmnet0 在主机上不可见（桥接直接使用物理网卡）。

(2) vmnet1 (仅主机模式 - Host-Only)

- 功能：虚拟机仅与 主机 (Host) 通信，无法连接外部网络。
- IP 分配：由 VMware 的 DHCP 服务分配（默认网段 192.168.56.0/24）。
- 通信范围：
 - 虚拟机 ↔ 主机（双向互通）。
 - 虚拟机 ↔ 其他同模式虚拟机（需在同一 vmnet1 网络）。
 - 无法访问互联网或局域网其他设备。
- 主机上的表现：
 - 主机创建一个虚拟网卡 VMware Network Adapter VMnet1（IP 如 192.168.56.1）。

(3) vmnet8 (NAT 模式 - Network Address Translation)

- 功能：虚拟机通过主机的 NAT 功能共享主机的 IP 访问外部网络。
- IP 分配：由 VMware 的 DHCP 服务分配（默认网段 192.168.xxx.0/24，如 192.168.152.0）。
- 通信范围：
 - 虚拟机 → 互联网（通过主机 NAT 转换）。
 - 主机 → 虚拟机（可直接访问）。
 - 局域网其他设备默认无法访问虚拟机（除非配置端口转发）。
- 主机上的表现：
 - 主机创建一个虚拟网卡 VMware Network Adapter VMnet8（IP 如 192.168.152.1）。

3. 其他 vmnet 接口

- vmnet2~vmnet7、vmnet9~：用户自定义网络（可通过 虚拟网络编辑器 创建）。
- vmnet0 的特殊性：
 - 仅代表桥接的逻辑接口，实际桥接的是主机的物理网卡（如 Wi-Fi 或 以太网），因此不会在主机网络适配器中显示为独立网卡。

4. 查看和配置

- Windows:

2.测试 uboot 网络不能 ping 自己，要 ping 其他设备



DDR

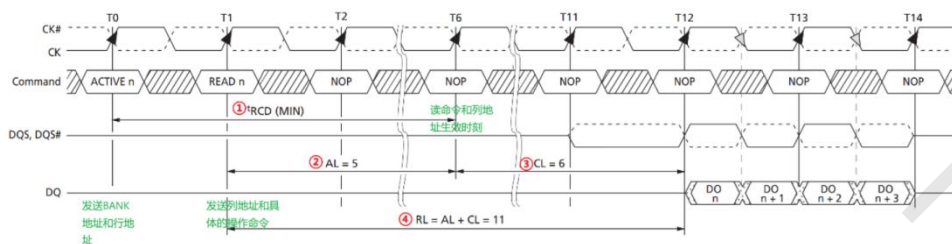
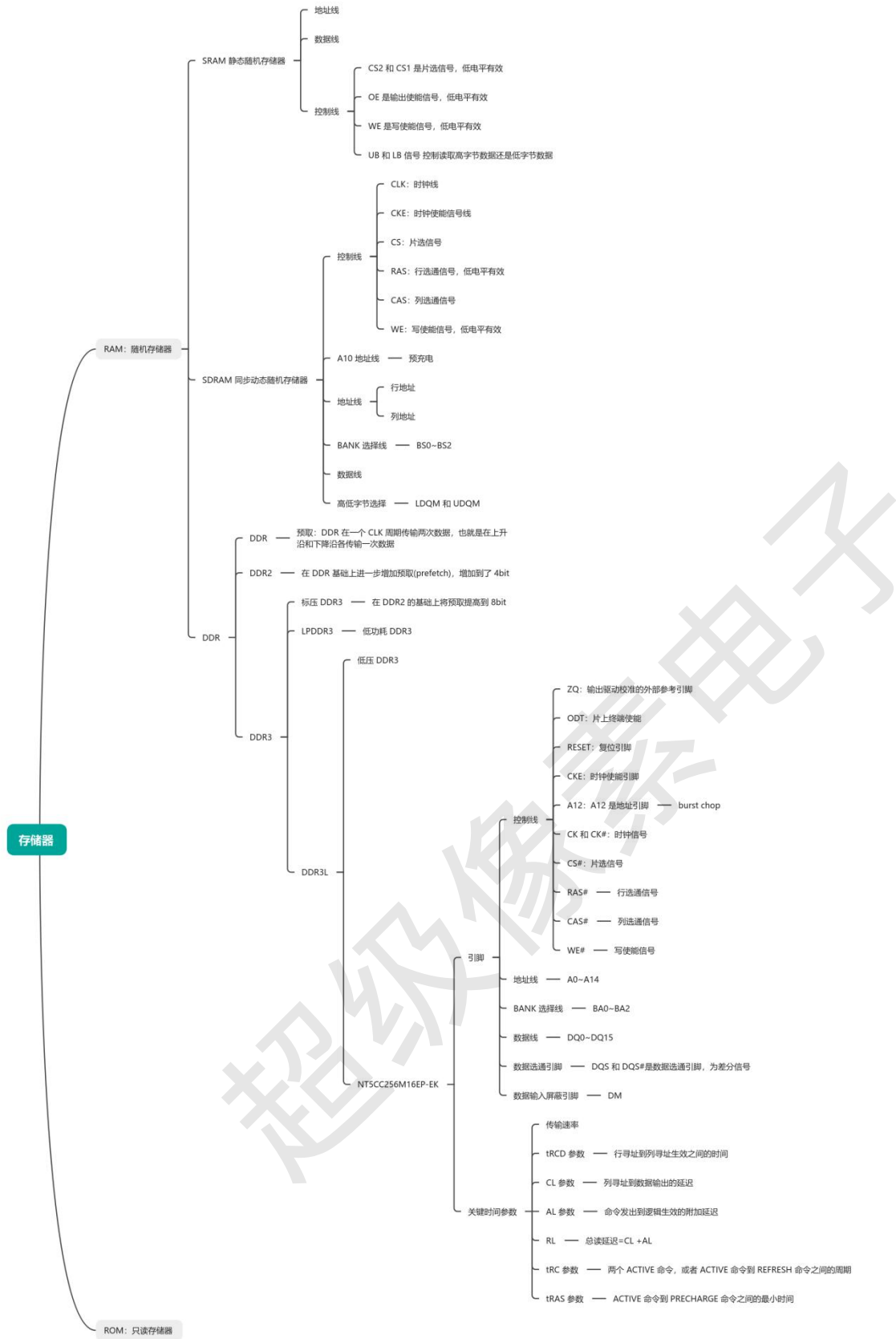


图 23.2.5 加入 AL 后的读时序图

图 32.2.5 就是镁光 DDR3L 的读时序图，我们依次来看一下图中这四部分都是什么内容：

- ①、 t_{RCD} ，前面已经说过了。行寻址到列寻址生效之间的时间
- ②、AL。命令发出到逻辑生效的附加延迟
- ③、CL。列寻址生效到数据输出的延迟
- ④、RL 为读潜伏期， $RL = AL + CL$ 。



· 五、uboot DDR 初始化

· 1、裸机

imxdownload 软件下载, 会在 bin 文件头部添加 IVT DCD 数据,

▪ 2、u-boot

u-boot 编译生成 u-boot.imx。u-boot.imx 已经包含了 IVT DCD 数据。

u-boot.imx 的头部信息是怎么添加的？

u-boot.imx 的 DCD 中的 DDR 初始化代码该怎么修改。

u-boot 编译会输出

```
./tools/mkimage -n board/freescale/mx6ull_alientek_emmc/imximage.cfg.cfgtmp -T  
imximage -e 0x87800000 -d u-boot.bin u-boot.imx
```

可以看出 u-boot 使用 /tools/mkimage 工具，向 u-boot.bin 添加 board/freescale/mx6ull_alientek_emmc/imximage.cfg.cfgtmp 文件信息，从而得到 u-boot.imx。

默认只有 imximage.cfg 文件，imximage.cfg 里面保存的就是 DCD 数据。DDR 初始化也在此文件里面。

我们要修改 DDR 初始化代码，就需要修改 imximage.cfg 文件。此文件默认拷贝的 NXP 给 IMX6ULL EVK 开发板写的，默认是给 512MB DDR3L 写的。

图形化配置的使用

一、Uboot 图形化配置方法

1、通过终端配置。

2、进入到 u-boot 的源码根目录下。

3、首先默认配置

```
make mx6ull_alientek_emmc_defconfig //默认配置
```

4、输入 make menuconfig。打开图形化配置界面。

5、注意，新电脑需要安装 ncurses 库。

图形化配置界面对于一个功能的编译，或者叫做选择有 3 中模式：

Y：对应的功能编译 u-boot 里面。

N：对应的功能不编译进 u-boot 里面

M：将对应的功能编译位模块，.ko，Linux 内核里面常用。

当我们配置好以后，需要保存一下字节的配置文件。

Shift+?查看当前帮助

按下/搜索

图形化配置完成后：

千万不能使用如下命令：

```
./mx6ull_alientek_emmc.sh
```

因为 mx6ull_alientek_emmc.sh 在编译之前会清理工程，会删除掉.config 文件！通过图形化界面配置所有配置项都会被删除，结果就是竹篮打水一场空。

Menuconfig 修改的是 .config 文件，如果 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- distclean make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- my_board_defconfig 会清除 menuconfig 配置

图形化配置的原理

Kconfig 文件的最终目的就是在.config 文件中生成以“CONFIG_”开头的变量

```
alientek@zdyz-imx6ull: ~/linux/IMX6ULL/my_uboot2$ make menuconfig
```

```
489 %config: scripts_basic outputmakefile FORCE
490 $(Q)$(MAKE) $(build)=scripts/kconfig $@
```

顶层makefile 确保基本脚本工具已构建 生成输出 Makefile 强制规则总是执行 (伪目标)

控制输出 调用 make build=-f ./scripts/Makefile.build obj

```
@ make -f ./scripts/Makefile.build obj=scripts/kconfig menuconfig
```

```
36 menuconfig: $(obj)/mconf
37 $< $(silent) $(Kconfig)
```

示例代码 34.2.1.2 scripts/kconfig/Makefile 代码段

obj=scripts/kconfig

第一个目标 Kconfig=Kconfig

```
menuconfig: scripts/kconfig/mconf
scripts/kconfig/mconf Kconfig
```

scripts/kconfig/mconf.c 这个文件会被编译, 生成 mconf 这个可执行文件 具体的编译在scripts/kconfig子文件夹里

mconf 可执行文件会调用 uboot 根目录下的 Kconfig 文件开始构建图形配置界面

1.Mainmenu:主菜单

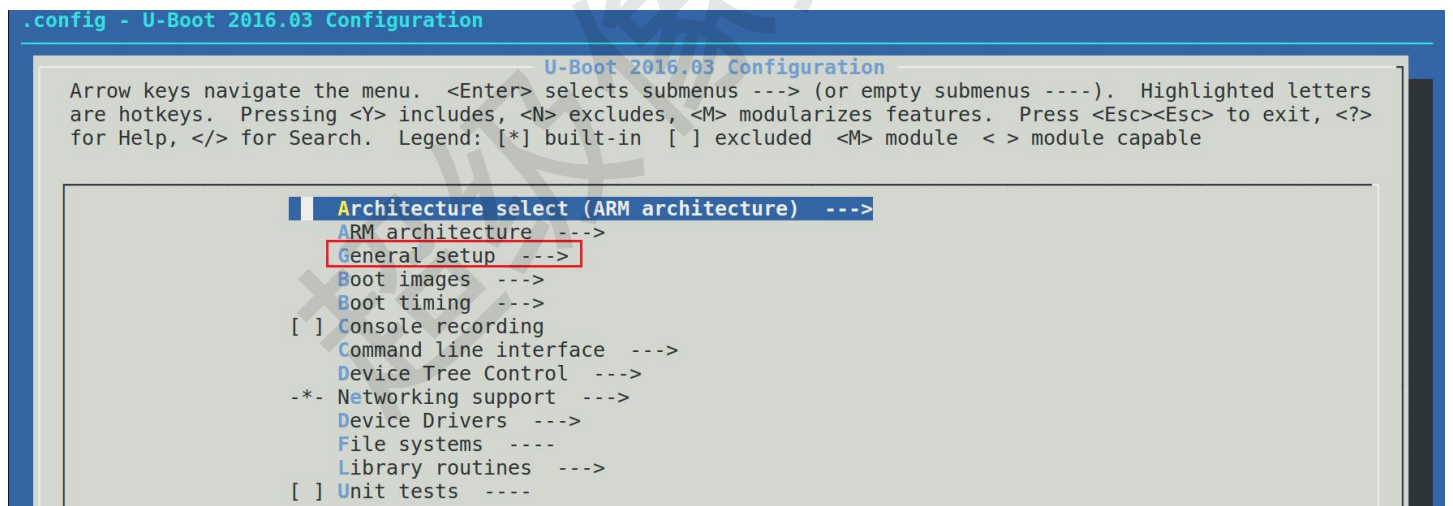
```
mainmenu "U-Boot $UBOOTVERSION Configuration" #定义主菜单
```

2.Source:调用其他目录下的 Kconfig 文件

```
source "arch/Kconfig" #调用其他子目录中的 Kconfig 文件
```

3.menu 用于生成菜单, endmenu 就是菜单结束标志

```
14 menu "General setup"
100 endmenu # General setup
```




```

menu "my config!!!" #最外面的入口标签

config MY_CINFIG_1 #选项1
    bool "my config display 1" #显示文本
    default y #默认数值
    help      #帮助文本
    | this is my config 1!

config MY_CINFIG_2 #选项2
    tristate "my config display 2"
    default y
    help
    | this is my config 2!
config MY_CINFIG_3 #选项3
    string "my config display 3"
    default test_string
    help
    | this is my config 3!

endmenu

```

.config - U-Boot 2016.03 Configuration By Super Pixel!!

U-Boot 2016.03 Configuration By Super Pixel!!
 Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module capable

```

Architecture select (ARM architecture) --->
ARM architecture --->
General setup! --->
Boot images --->
Boot timing --->
[ ] Console recording
Command line interface --->
Device Tree Control --->
*- Networking support --->
Device Drivers --->
File systems ----
Library routines --->
[ ] Unit tests ----
my config!!! --->

```

.config - U-Boot 2016.03 Configuration By Super Pixel!!

→ my config!!!

my config!!!
 Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module capable

```

[*] my config display 1
[ ] my config display 2
(test_string2) my config display 3

```

.config 文件:

```

#
# my config!!!
#
CONFIG_MY_CINFIG_1=y
# CONFIG_MY_CINFIG_2 is not set
CONFIG_MY_CINFIG_3="test_string2"

```

4.choice/endchoice 代码段定义了一组可选择项，将多个类似的配置项组合在一起，供用户单选或者多选。

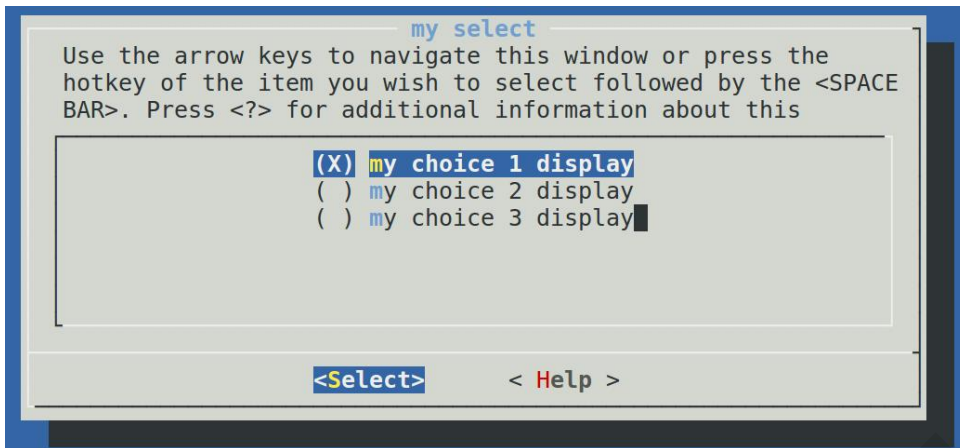

```
choice
|   prompt "my select"

config MY_CHOICE1
|   bool "my choice 1 display"

config MY_CHOICE2
|   bool "my choice 2 display"

config MY_CHOICE3
|   bool "my choice 3 display"

endchoice
```



5.menuconfid

menuconfig 和 menu 很类似，但是 menuconfig 是个带选项的菜单，其一般用法为：

示例代码 34.2.2.8 menuconfig 用法

```
1 menuconfig MODULES
2     bool "菜单"
3 if MODULES
4 ...
5 endif # MODULES
```

只有当 bool 被激活后，if 底下的语句才会被执行

```
menuconfig MY_MODULES
bool "my_modules_display"
if MY_MODULES
config MY_SUB_MODULES
|   bool "my sub_modules display"
endif
```

情况 1:

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module capable

```
[*] my config display 1
[ ] my config display 2
(test_string2) my config display 3
    my select (my choice 1 display) --->
[ ] my modules display ----
```

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module capable

```
--- my_modules_display
```

情况 2:

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module capable

```
[*] my config display 1
[ ] my config display 2
(test_string2) my config display 3
my select (my choice 1 display) --->
[*] my modules display --->
```

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module capable

```
--- my modules display
[ ] my sub_modules display
```

6.comment 标注了一行注释

```
comment "wow!!! so good?!!!"
```

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < > module capable

```
[*] my config display 1
[ ] my config display 2
(test_string2) my config display 3
my select (my choice 1 display) --->
[ ] my modules display (NEW) ----
*** wow!!! so good?!!! ***
```

Uboot 移植总结

uboot 移植到此结束，简单总结一下 uboot 移植的过程：

①、不管是购买的开发板还是自己做的开发板，基本都是参考半导体厂商的 dmeo 板，而半导体厂商会在他们自己的开发板上移植好 uboot、linux kernel 和 rootfs 等，最终制作好 BSP 包提供给用户。我们可以在官方提供的 BSP 包的基础上添加我们的板子，也就是俗称的移植。

②、我们购买的开发板或者自己做的板子一般都不会原封不动的照抄半导体厂商的 demo 板，都会根据实际的情况来做修改，既然有修改就必然涉及到 uboot 下驱动的移植。

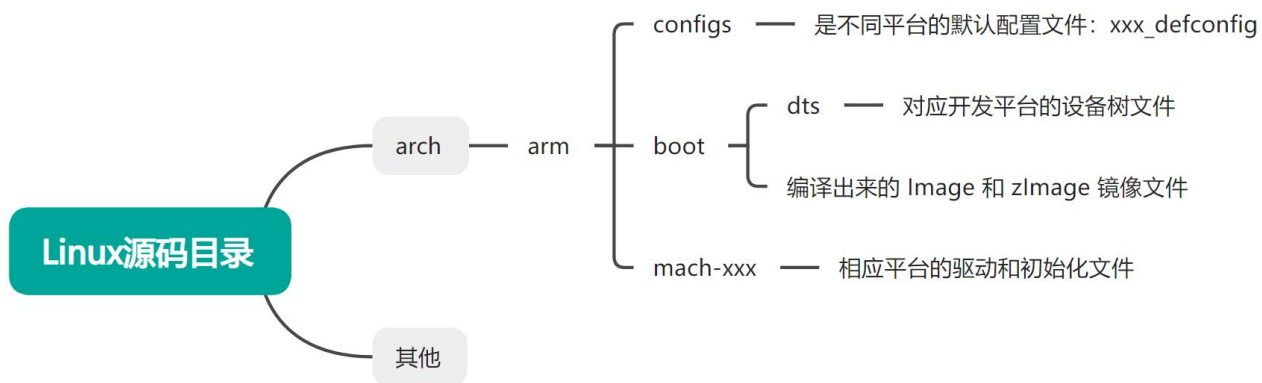
③、一般 uboot 中需要解决串口、NAND、EMMC 或 SD 卡、网络 and LCD 驱动，因为 uboot 的主要目的就是启动 Linux 内核，所以不需要考虑太多的外设驱动。

④、在 uboot 中添加自己的板子信息，根据自己板子的实际情况来修改 uboot 中的驱动。

Linux 内核和移植

源码目录分析

类型	名字	描述	备注
文件夹	arch	架构相关目录。	Linux 自带
	block	块设备相关目录。	
	crypto	加密相关目录。	
	Documentation	文档相关目录。	
	drivers	驱动相关目录。	
	firmeare	固件相关目录。	
	fs	文件系统相关目录。	
	include	头文件相关目录。	
	init	初始化相关目录。	
	ipc	进程间通信相关目录。	
	kernel	内核相关目录。	
	lib	库相关目录。	
	mm	内存管理相关目录。	
	net	网络相关目录。	
	samples	例程相关目录。	
	scripts	脚本相关目录。	
	security	安全相关目录。	
	sound	音频处理相关目录。	
	tools	工具相关目录。	
	usr	与 initramfs 相关的目录，用于生成 initramfs。	
	virt	提供虚拟机技术(KVM)。	
文件	.config	Linux 最终使用的配置文件。	编译生成的文件。
	.gitignore	git 工具相关文件。	Linux 自带
	.mailmap	邮件列表。	
	.missing-syscalls.d	。	编译生成的文件
	.tmp_xx	这是一系列的文件，作用目前笔者还不是很清楚。	编译生成的文件
	.version	好像和版本有关。	
	.vmlinux.cmd	cmd 文件，用于连接生成 vmlinux。	
	COPYING	版权声明。	Linux 自带
	CREDITS	Linux 贡献者。	
	Kbuild	Makefile 会读取此文件。	
	Kconfig	图形化配置界面的配置文件。。	
	MAINTAINERS	维护者名单。	
	Makefile	Linux 顶层 Makefile	
	Module.xx modules.xx	一系列文件，和模块有关。	编译生成的文件
	mx6ull_alientek_emmc.sh mx6ull_alientek_nand.sh	正点原子提供的，Linux 编译脚本。	正点原子提供
	README	Linux 描述文件。	Linux 自带
	REPORTING-BUGS	BUG 上报指南	
	System.map	符号表。	
	vmlinux	编译出来的、未压缩的 ELF 格式 Linux 文件	编译生成的文件
	vmlinux.o	编译出来的 vmlinux.o 文件。	



编译过程分析

vmlinux :编译出来的最原始的内核文件，是未压缩的

Image :Linux 内核镜像文件，但是 Image 仅包含可执行的二进制数据

zImage :经过 gzip 压缩后的 Image

ulImage :老版本 uboot 专用的镜像文件

源码分析

```

Makefile
config-targets := 1
mixed-targets := 0
dot-config := 1

ifeq ($($config-targets),1)
include arch/$(SRCARCH)/Makefile #引用 arch/arm/Makefile 这个文件
export KBUILD_DEFCONFIG KBUILD_KCONFIG

%config: scripts/basic/outputmakefile FORCE
    $(Q)$(MAKE) $(build)=scripts/kconfig $@

@make -f ./scripts/Makefile.build obj=scripts/kconfig xxx_defconfig

scripts % Makefile.build
# The filename build has precedence over Makefile src= scripts/kconfig
kbuild-dir := $(if $(filter %/, $(src)),$(src),$(srctree)/$(src)) kbuild-dir = ./scripts/kconfig
kbuild-file := $(if $(wildcard $(kbuild-dir)/kbuild,$(kbuild-dir)/kbuild,$(kbuild-dir)/Makefile)
include $(kbuild-file) include ./scripts/kconfig/Makefile
kbuild-file = ./scripts/kconfig/Makefile

scripts % kconfig % Makefile
%defconfig: $(obj)/conf
    $(Q)$(cc) $(silent) --defconfig=arch/$(SRCARCH)/configs/$@ $(Kconfig)

%_defconfig: scripts/kconfig/conf
@scripts/kconfig/conf --defconfig=arch/arm/configs/%_defconfig Kconfig

将%_defconfig 中的配置输出到kconfig 文件中，最终生成
Linux kernel 根目录下的config 文件

scripts/kconfig/conf 是 Linux 内核配置系统 (Kconfig) 的实现文件，负责解析用户配置 (如 menuconfig、defconfig) 生成最终的 config 文件

```

```
step1:make_xxxdeconfig
step2:编译vmlinux
```

```

# That's why default target when none is given on the command line
PHONY := _all
_all:

PHONY += all
ifeq ($KBUILD_EXTMOD,)
all: all # 编译内核本身, all 依赖 all
else
all: modules # 编译外部模块, _all 目标依赖 modules
endif

export KBUILD_VMLINUX_INIT := $(head-y) $(init-y)
export KBUILD_VMLINUX_MAIN := $(core-y) $(libs-y) $(drivers-y) $(net-y)
export KBUILD_LDS := arch/$(vmlinux-deps)/kernel/vmlinux.lds

all: vmlinux
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
+$(call if_changed,link-vmlinux)

调用函数 if_changed, link-vmlinux 是函数 if_changed 的参数

scripts % kbuild.include
if_changed = $(if $(strip $(any-prereq) $(arg-check)),
@set -e; rm -rf $(vmlinux)
+$(echo-cmd -s 'cmd $1');
printf '%s\n' 'cmd_$@' := $(make-cmd) ' > $(dot-target).cmd')

Kbuildfile
cmd link-vmlinux = $(CONFIG_SHELL) $< $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux)
quiet_cmd link-vmlinux = LINK $@
cmd link-vmlinux = $(CONFIG_SHELL) $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux)

scripts % link-vmlinux.sh
info LD link-vmlinux.sh
vmlinux link "$(kallsyms0)" vmlinux
vmlinux link()
local lds="$(objtree)/$(KBUILD_LDS)"

if [ "$(SRCARCH)" != "um" ]; then
$(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@
-$(LD) $(KBUILD_VMLINUX_INIT)
-$(LD) $(KBUILD_VMLINUX_MAIN) --start-group $(KBUILD_VMLINUX_MAIN) --end-group $@
else
$(CC) $(CFLAGS_vmlinux) -o $@
-WL,-T,$< $(ld) $(KBUILD_VMLINUX_INIT)
-WL,--start-group
$(KBUILD_VMLINUX_MAIN)
-WL,--end-group
-Lutil $@
fi
rm -f linux

```

in2c 将二进制文件（如固件）转换为 C 数组，方便内核直接引用

fixdep 优化头文件依赖关系，减少不必要的重新编译

scripts/basic 目标的作用就是编译出 scripts/basic/fixdep 和 scripts/basic/bin2c 这两个软件

```

build := -f $(srcdir)/scripts/Makefile.build obj

PHONY += scripts_basic
scripts_basic:
    $(Q)$(MAKE) $(build)=scripts/basic
    $(Q)rm -f .tmp_quiet_recordcount

scripts_basic:
@make -f $(srcdir)/scripts/Makefile.build obj=scripts/basic /也可以没有@, 视配置而定
@rm -f .tmp_quiet_recordcount /也可以没有@

scripts:
scripts: ❖ Makefile.build
# The filename Kbuild has precedence over Makefile
kbuild-dir := $(if $(filter %, $(src)),$(src),$(srcdir)/$(src)) -> kbuild-dir=/scripts/basic
kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild,$(kbuild-dir)/Kbuild,$(kbuild-dir)/Makefile)
include $(kbuild-file) -> include ./scripts/Makefile kbuild-file= ./scripts/basic/Makefile

build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target) $(extra-y)) \ kbuild默认目标
    $(if $(KBUILD_MODULES),obj-m = $(modorder-target)) \
    $(subst -ym,$(always))
    @:
    builtin-target =
    lib-target =
    extra-y =
    subdir-ym =
    _build$(builtin-target) $(lib-target) $(extra-y) $(subdir-ym) $(always)
    @:
    always = scripts/basic/fixdep scripts/basic/bin2c

build: scripts/basic/fixdep scripts/basic/bin2c

```

head-y、init-y、core-y、libs-y、drivers-y 和 net-y: 将相应目录中的源码文件进行编译, 然后在各自目录下生成 built-in.o 文件

Makefile

```

arch > arm
head-y := arch/arm/kernel/heads[MMUEXT].o
# 使用arch/arm/kernel/heads为头文件
# head-y = arch/arm/kernel/head.o

Makefile
init-y := init/
drivers-y := drivers/ sound/ firmware/
net-y := net/

Makefile
libs-y := lib/

arch > arm
Makefile
libs-y := arch/arm/lib/ $(libs-y)
libs-y = arch/arm/lib/ lib/

Makefile
core-y := usr/
core-y += kernel/ mm/ fs/ ipc/
        security/ crypto/ block/

core-y =
usr/built-in.o
arch/arm/vfp/built-in.o
arch/arm/vdso/built-in.o
arch/arm/mmu/built-in.o
arch/arm/common/built-in.o
arch/arm/kernel/built-in.o
arch/arm/firmware/built-in.o
arch/arm/mach-imx/built-in.o
mm/built-in.o
fs/built-in.o
ipc/built-in.o
security/built-in.o
crypto/built-in.o
block/built-in.o

arch > arm
Makefile
core-% $(CONFIG_FPU_MWPE) += arch/arm/mwpe/
core-% $(CONFIG_FASTFP) += $(FASTFPU_OB3)
core-% $(CONFIG_VFP) += arch/arm/vfp/
core-% $(CONFIG_XEN) += arch/arm/xen/
core-% $(CONFIG_KVM_ARM_HOST) += arch/arm/kvm/
core-% $(CONFIG_VDOSO) += arch/arm/vdoso/

arch > arm
Makefile
core-y := arch/arm/kernel/ arch/arm/mm/ arch/arm/common/
core-y := arch/arm/probes/
core-y := arch/arm/net/
core-y := arch/arm/crypto/
core-y := arch/arm/firmware/
core-y := $(machdirs) $(platdirs)

Makefile
core-y := $(patsubst %, %built-in.o, %core-y)

根据不同的配置用core-y来添加不同的
模块，大致的变量名可能会
在config文件里被赋予=y

```

```
$ (sort $(vmlinux-deps) $(vmlinux-dirs) ;  
init arch/arm/vdso arch/arm/kernel arch/arm/common arch/arm/crtyo  
kernel arch/arm/generic arch/arm/firmware fs security  
net block firmware net  
vmlinux-dirs := $(patsubst %, $(filter %, $(init-y) $(init-m) \br/>$(core-y) $(core-m) $(drivers-y) $(drivers-m) \br/>$(net-y) $(net-m) $(libs-y) $(libs-m))
```

```
PHONY += $(vmlinux-dirs)
$(vmlinux-dirs): prepare scripts
$(Q)$(MAKE) $(build)=$@ → make -f ./scripts/Makefile.build obj=
                           @ make -f ./scripts/Makefile.build obj=init
                           @ make -f ./scripts/Makefile.build obj=usr
                           @ make -f ./scripts/Makefile.build obj=arch/arm/vfp
```

scripts > Makefile.build

```

build: $(if $(KBUILD_BUILTIN), $builtin-target $(lib-target) $(extra-y)) \
      $(if $(KBUILD_MODULES), $obj-m) $(modorder-target) \
      $(subst -ym, $(always))
@:

```

↓

scripts > Makefile.build

```

built-in-target: $(lib-target) $(lib-target) $(extra-y) $(subst -ym, $(always))
@:

```

ifneq (\$(strip \$(obj-y) \$(obj-m) \$(obj-) \$(subst -r-m, \$(lib-target),) \
built-in-target := \$(obj)/built-in.o

endit

scripts > Makefile.build

```

$(builtin-target): $(obj-y) FORCE
$(call if_changed,link.o)

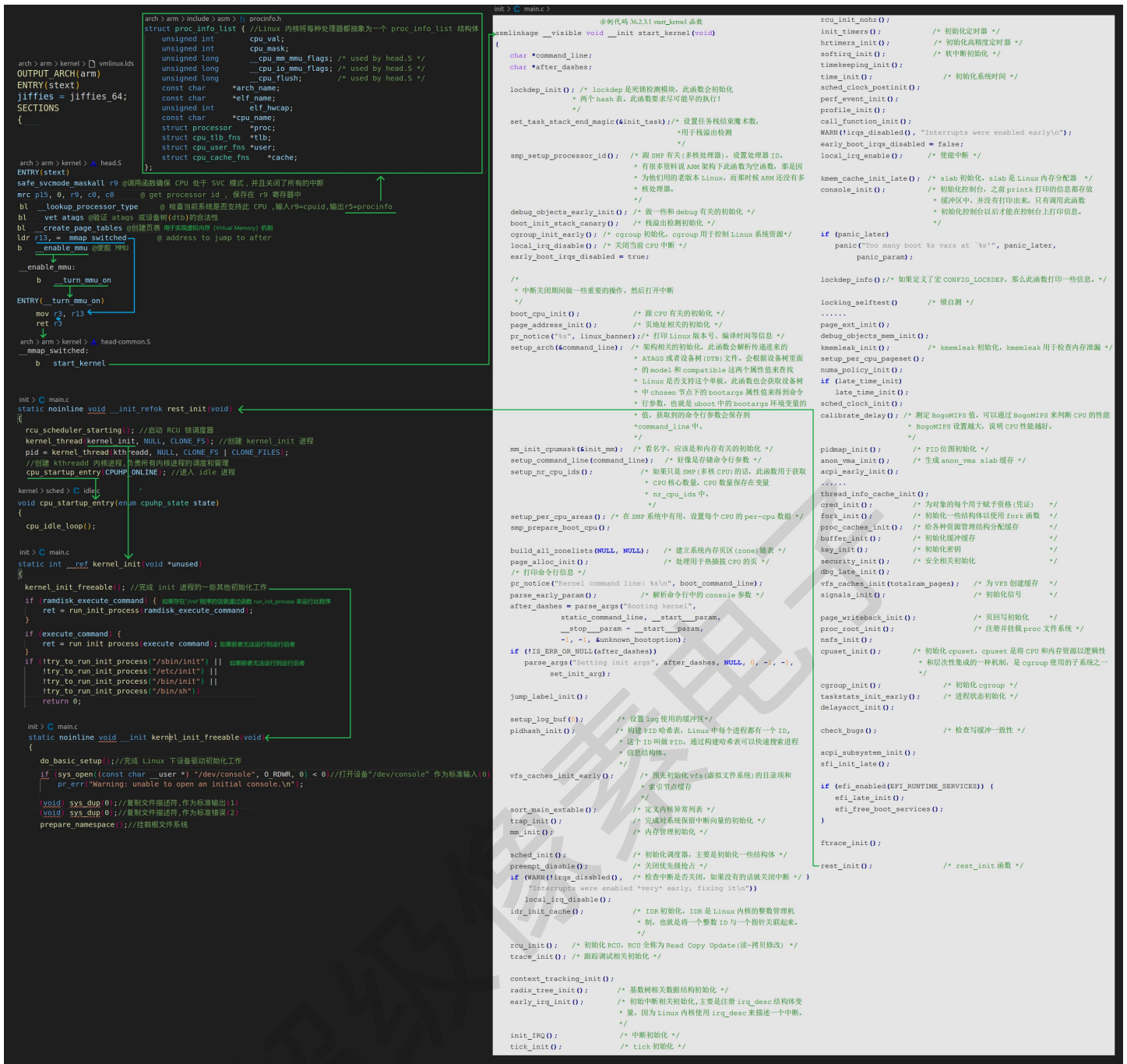
```

调用函数 if_changed, 参数为 link.o_target, 其返回值为具体的命令
if_changed 使用 cmd, \$(1) 所对应的命令(1)就是函数的第 1 个参数, 在这里就是 cmd, link.o_target 所对应的命令

```
cmd_link_o_target = $(if $(strip $(obj-y)),\
$(LD) $(ld_flags) -r -o $@ $(filter $(obj-y), $*) \
$(cmd_secanalysis),\
rm -f $@; $(AR) -r $@ $(KRUIID) $(ARFLAGS) $@)
```

step3:编译zImage

```
BOOT_TARGETS = zImage Image xipImage bootpImage uImage
INSTALL_TARGETS = install uninstall install
PHONY += bzImage $(BOOT_TARGETS) $(INSTALL_TARGETS)
$(BOOT_TARGETS): vmlinux
$(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/%
## make 1 / scripts/Makefile.build obj=arch/arm/boot MACHINE=arch/arm/boot/zImage
```



相关问题总结

在代码上加了注释时不能在注释和代码之间输入空格

```
252 ARCH           ?= arm #my
253 CROSS_COMPILE ?= arm-linux-gnueabihf- #my
```

```
alientek@zdyz-imx6ull: ~/linux/IMX6ULL/my_linux/linux-imx-rel_imx_4.1.15_2.1.0_ga$ make clean
make: *** arch/arm: Is a directory. Stop.
```

```
ARCH           ?= arm#my
CROSS_COMPILE ?= arm-linux-gnueabihf-#my
```

```
alientek@zdyz-imx6ull: ~/linux/IMX6ULL/my_linux/linux-imx-rel_imx_4.1.15_2.1.0_ga$ make clean
alientek@zdyz-imx6ull: ~/linux/IMX6ULL/my_linux/linux-imx-rel_imx_4.1.15_2.1.0_ga$
```

[https://blog.csdn.net/qq_44774965/article/details/128769270:](https://blog.csdn.net/qq_44774965/article/details/128769270)

记录makefile错误

原创

液体小猫

于 2023-01-27 10:11:49 发布

阅读量173

收藏

点赞数

分类专栏：


linux

文章标签：

linux

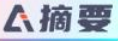
ubuntu

CC 4.0 BY-SA版权

 linux 专栏收录该内容

3 篇文章

订阅专栏

 摘要

博主在学习Linux时添加注释后遇到报错，发现提示处的一个反斜杠颜色与其他不同。经百度得知makefile对空格和Tab敏感，回忆写注释时敲过空格，在颜色不同的反斜杠后按删除键，最终解决问题。

摘要生成于 C知道 ，由 DeepSeek-R1 满血版支持， [前往体验 >](#)

在跟着原子学linux时，遇到了如下错误

```
Makefile:11: *** recipe commences before first target. 停止。
```

奇怪的是之前输出的时候可以输出，在我加了注释后报错了，我就开始找是不是哪里注释搞错了，结果发现提示处的一个跟其他的颜色不一样

```
#所有的头文件目录
INCUDIRS := imx6u \
            bsp/clock \
            bsp/delay \
            bsp/led
```

百度后发现 makefile 对空格和Tab十分敏感，我想到我在写注释的时候好像敲了空格，于是在颜色不一样的后敲了一下删除键

```
#所有的头文件目录
INCUDIRS := imx6u \
            bsp/clock \
            bsp/delay \
            bsp/led
```

问题得以解决

网络适配器和虚拟网络编辑器的区别

特性	网络适配器	虚拟网络编辑器
本质	网络接口（硬件或虚拟）	虚拟网络的管理工具
作用层级	数据链路层/网络层（IP配置）	网络拓扑层（规则配置）
配置内容	IP地址、子网掩码、DNS、网关	NAT规则、DHCP范围、端口转发
影响范围	单台设备（主机或虚拟机）	所有使用该虚拟网络的虚拟机
典型工具	Windows网络设置、ifconfig、ip addr	VMware虚拟网络编辑器、VirtualBox管理器

问题：win 无法 ping 虚拟机

特性	DTS	DTSI	DTB
文件类型	文本文件（可编辑）	文本文件（可编辑）	二进制文件（不可直接编辑）
用途	描述具体板级的硬件配置	定义SoC或通用配置	供内核/U-Boot读取的硬件描述
编译流程	需编译为DTB	被DTS包含后间接编译	由DTS直接编译生成
修改方式	直接编辑文本	直接编辑文本	需反编译为DTS后修改
示例工具	手动编写或工具生成	手动编写	dtc 编译器

特性	DTS	DTSI	DTB
文件类型	文本文件（可编辑）	文本文件（可编辑）	二进制文件（不可直接编辑）
用途	描述具体板级的硬件配置	定义SoC或通用配置	供内核/U-Boot读取的硬件描述
编译流程	需编译为DTB	被DTS包含后间接编译	由DTS直接编译生成
修改方式	直接编辑文本	直接编辑文本	需反编译为DTS后修改
示例工具	手动编写或工具生成	手动编写	dtc 编译器

Linux 内核移植总结

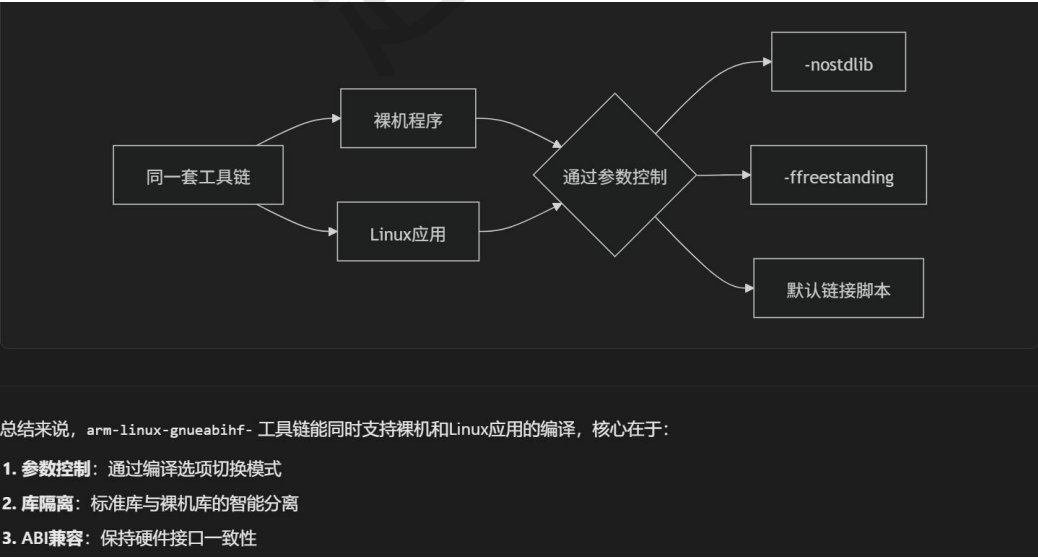
关于 Linux 内核的移植就讲解到这里，简单总结一下移植步骤：

- ①、在 Linux 内核中查找可以参考的板子，一般都是半导体厂商自己做的开发板。
- ②、编译出参考板子对应的 zImage 和.dtb 文件。
- ③、使用参考板子的 zImage 文件和.dtb 文件在我们所使用的板子上启动 Linux 内核，看能否启动。
- ④、如果能启动的话就万事大吉，如果不能启动那就悲剧了，需要调试 Linux 内核。不过一般都会参考半导体官方的开发板设计自己的硬件，所以大部分情况下都会启动起来。启动 Linux 内核用到的外设不多，一般就 DRAM(Uboot 都初始化好的)和串口。作为终端使用的串口一般都会参考半导体厂商的 Demo 板。
- ⑤、修改相应的驱动，像 NAND Flash、EMMC、SD 卡等驱动官方的 Linux 内核都是已经提供好了，基本不会出问题。重点是网络驱动，因为 Linux 驱动开发一般都要通过网络调试代码，所以一定要确保网络驱动工作正常。如果是处理器内部 MAC+外部 PHY 这种网络方案的话，一般网络驱动都很好处理，因为在 Linux 内核中是有外部 PHY 通用驱动的。只要设置好复位引脚、PHY 地址信息基本上都可以驱动起来。
- ⑥、Linux 内核启动以后需要根文件系统，如果没有根文件系统的话肯定会崩溃，所以确定 Linux 内核移植成功以后就要开始根文件系统的构建。

根文件系统与移植

相关问题

我在编译Linux内核的时候和编译Linux上面的应用程序时都是用arm-linux-gnueabi- 为什么它可以同时编译裸机和操作系统上的程序



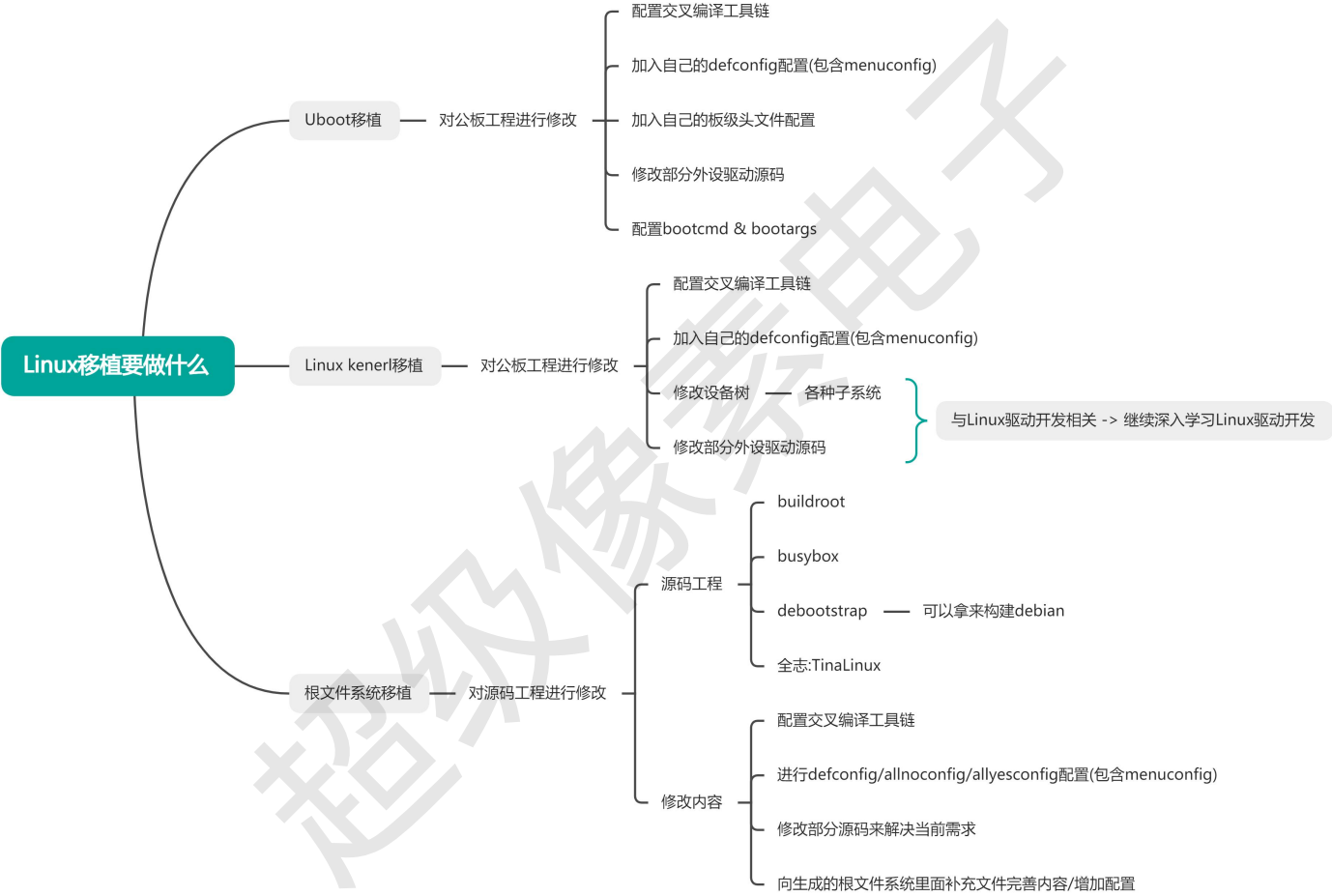
从网络启动 Linux 的配置

```
setenv bootargs 'console=ttyMX0,115200 root=/dev/nfs nfsroot=192.168.5.11:/home/alientek/linux/nfs/my_roofts,proto=tcp
rw ip=192.168.5.9:192.168.5.11:192.168.5.1:255.255.255.0::eth0:off'
```

```
setenv bootcmd 'tftp 80800000 zImage; tftp 83000000 my_device.dtb; bootz 80800000 - 83000000'
```

```
saveenv
```

系统移植总结



关于学习路线的思考

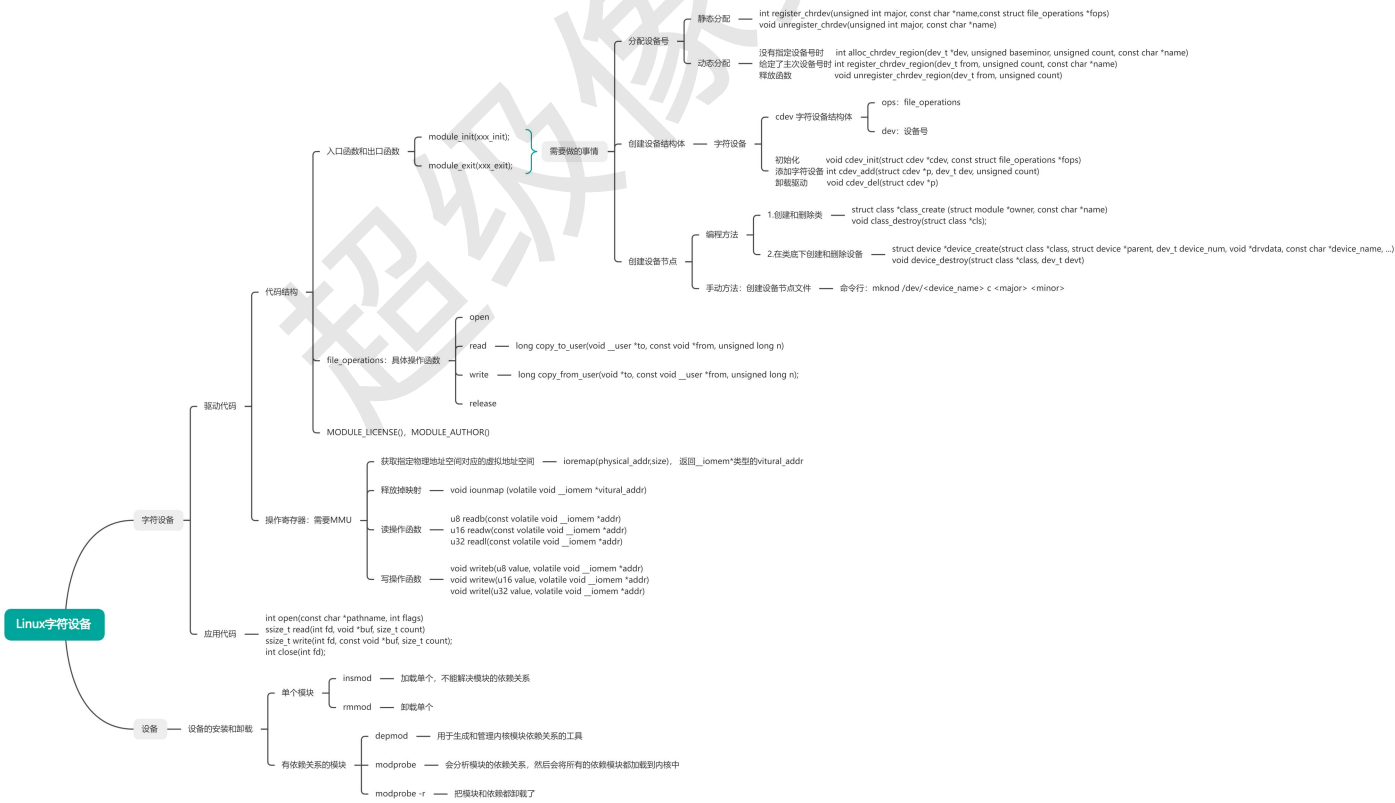
如何选择复刻的核心板

	正点原子 -rk3588	野火鲁班猫 5-rk3588	香 橙 派 5plus-rk3588	嘉立创泰山 派-rk3566	树莓派 4B	迅为 rk3588	Firefly rk3588
文档	驱 动 应 用 qt andriod ai...	驱 动 应 用 qt android ai 内核 ..	基础使用手 册	硬件设计	基础使用手 册	驱 动 应 用 android ai	未知
系统	Buildroot	鲁班猫系统	香橙派系统				
硬件资料	底板原理图 PDF	底板原理图 源文件 底板 PCB 源 文件 核心板原理 图 PDF	原理图 PDF	原理图源文 件 Pcb 源文件	无	底板原理图 源文件 底板 PCB 源 文件	底板相关文 件
内 核 源 码 /SDK	有	有	有	有	不明	有	未知
性能	A76*4+A55*4	A76*4+A55*4	A76*4+A55*4	A55*4	不明	A76*4+A55*4	A76*4+A55*4
咸鱼最低价 格	900	800	800	200	不明	1000	1000

荔枝派 香蕉派：非 rk/arm

Linux 字符设备

基本编程思路



设备的安装和卸载：
insmod 命令不能解决模块的依赖关系
modprobe 会分析模块的依赖关系，然后会将所有的依赖模块都加载到内核中
depmod 是 Linux 系统中用于生成和管理内核模块依赖关系的工具

lsmod 命令可查看当前系统中存在的模块

设备号的分配:

静态分配设备号: register_chrdev(200, "chrtest", &test_fops);

unregister_chrdev(200, "chrtest");

动态分配设备号: int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name) 没有指定设备号时

int register_chrdev_region(dev_t from, unsigned count, const char *name) 给定了设备的主设备号和次设备号时

void unregister_chrdev_region(dev_t from, unsigned count) 释放函数

内存的映射:

ioremap 函数: 用于获取指定物理地址空间对应的虚拟地址空间

iounmap 函数: 释放掉 ioremap 函数所做的映射

映射之后:

写操作函数: writeb、writew 和 writel 这三个函数分别对应 8bit、16bit 和 32bit 写操作

读操作函数: readb、readw 和 readl 这三个函数分别对应 8bit、16bit 和 32bit 读操作

寄存器操作方法: 先 ioremap 映射, 然后和 writel 和 readl 读写, 不用时 iounmap

(下面是韦东山)

2.3 请猜猜: 怎么编写驱动程序?

① 确定主设备号

② 定义自己的 file_operations 结构体

③ 实现对应的 open/read/write 等函数, 填入 file_operations 结构体

④ 把 file_operations 结构体告诉内核: 注册驱动程序

register_chrdev (major, minor)

⑤ 谁来注册驱动程序啊? 得有一个入口函数: 安装驱动程序时, 就会去调用这个入口函数

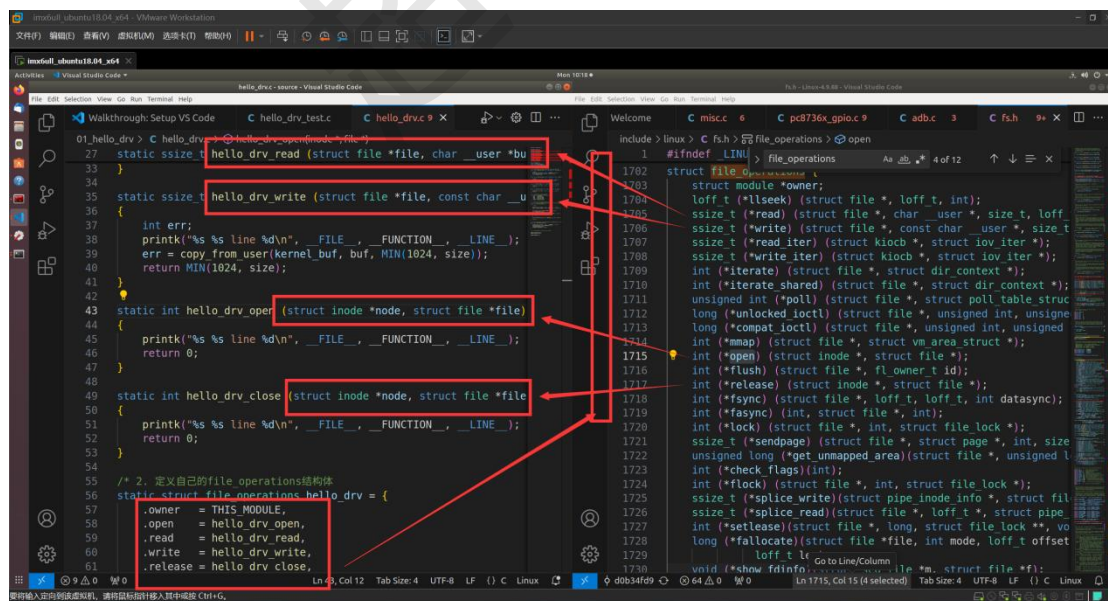
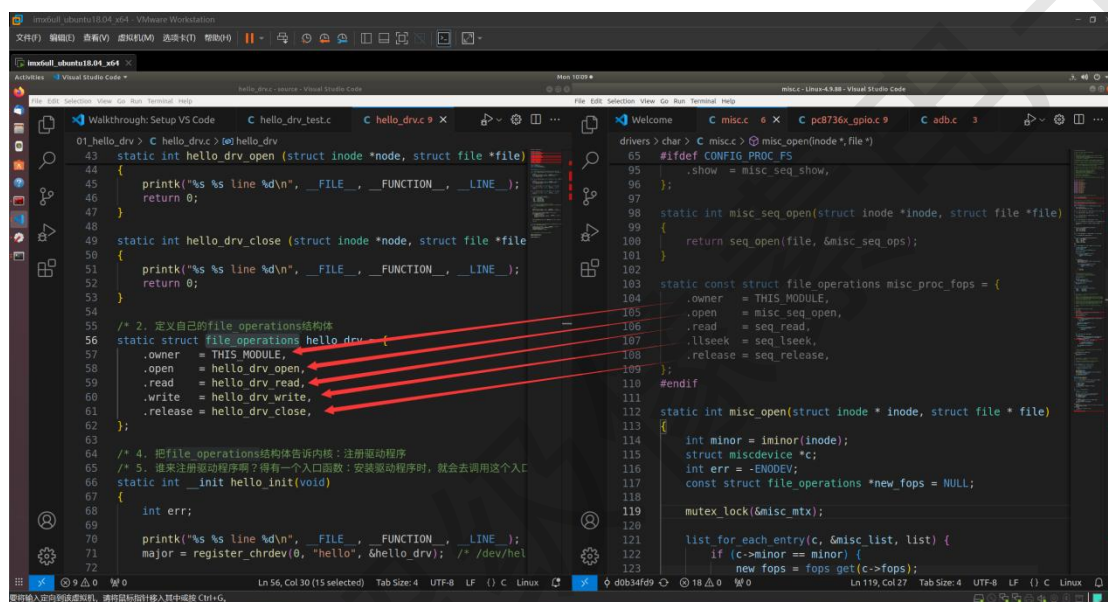
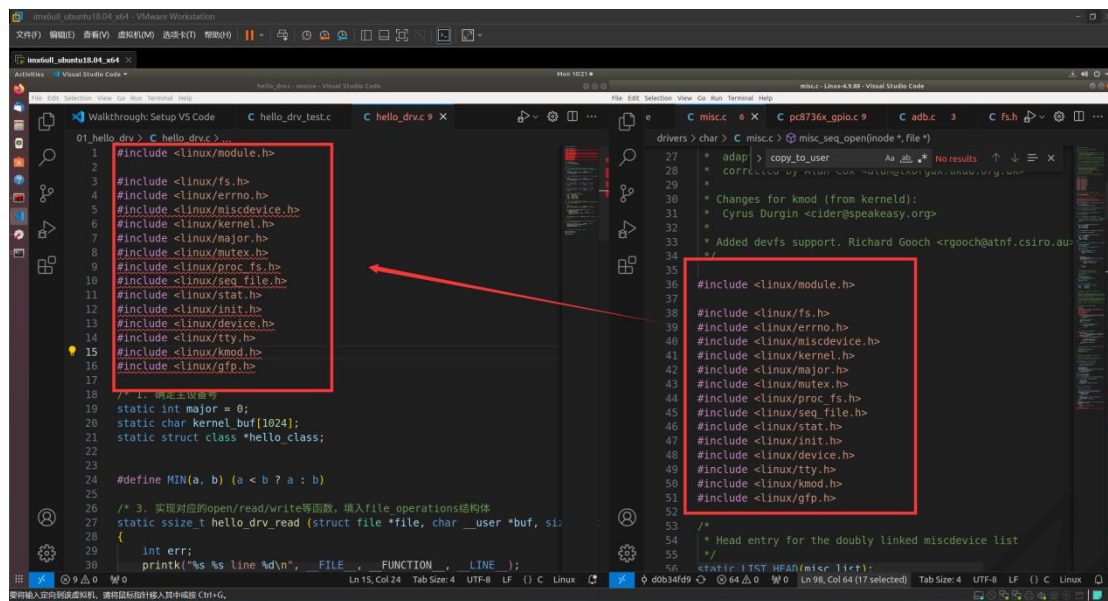
⑥ 有入口函数就应该有出口函数: 卸载驱动程序时, 就会去调用这个出口函数

= unregister_chrdev (major, minor)

⑦ 其他完善: 提供设备信息, 自动创建设备节点



怎么根据已有代码写出自己的驱动代码 (韦东山)



```
01_hello_drv.c: C hello_drv.c > C hello_drv_test.c > C hello_drv.c > X
18 /* 1. 确定主设备号 */
19 static int major = 0;
20 static char kernel_buf[1024];
21 static struct class *hello_class;
22
23
24 #define MIN(a, b) (a < b ? a : b)
25
26 /* 3. 实现对应的open/read/write等函数, 填入file_operations结构体 */
27 static ssize_t hello_drv_read(struct file *file, char __user *buf, size_t size, loff_t *offset)
28 {
29     int err;
30     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
31     err = copy_to_user(buf, kernel_buf, MIN(1024, size));
32     return MIN(1024, size);
33 }
34
35 static ssize_t hello_drv_write(struct file *file, const char __user *buf, size_t size, loff_t *offset)
36 {
37     int err;
38     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
39     err = copy_from_user(kernel_buf, buf, MIN(1024, size));
40     return MIN(1024, size);
41 }
42
43 static int hello_drv_open(struct inode *node, struct file *file)
44 {
45     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
46     return 0;
47 }
```

```
01_hello_drv.c: C hello_drv.c > C hello_drv_test.c > C hello_drv.c > X
56 static struct file_operations hello_drv = {
60     .write = hello_drv_write,
61     .release = hello_drv_close,
62 };
63
64 /* 4. 把file_operations结构体告诉内核: 注册驱动程序 */
65 /* 5. 谁来注册驱动程序呢? 得有一个入口函数: 安装驱动程序时, 就会去调用这个入口函数 */
66 static int __init hello_init(void)
67 {
68     int err;
69     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
70     major = register_chrdev(0, "hello", &hello_drv); /* /dev/hello */
71
72     hello_class = class_create(THIS_MODULE, "hello_class");
73     err = PTR_ERR(hello_class);
74     if (IS_ERR(hello_class)) {
75         printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
76         unregister_chrdev(major, "hello");
77         return -1;
78     }
79     device_create(hello_class, NULL, MKDEV(major, 0), NULL, "hello");
80
81     return 0;
82 }
83
84 /* 6. 有入口函数就应该有出口函数: 卸载驱动程序时, 就会去调用这个出口函数 */
85 static void __exit hello_exit(void)
86 {
87     unregister_chrdev(major, "hello");
88     class_destroy(hello_class);
89     return 0;
90 }
```

```
01_hello_drv.c: C hello_drv.c > C hello_drv_test.c > C hello_drv.c > X
56 static struct file_operations hello_drv = {
60     .write = hello_drv_write,
61     .release = hello_drv_close,
62 };
63
64 /* 4. 把file_operations结构体告诉内核: 注册驱动程序 */
65 /* 5. 谁来注册驱动程序呢? 得有一个入口函数: 安装驱动程序时, 就会去调用这个入口函数 */
66 static int __init hello_init(void)
67 {
68     int err;
69     printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
70     major = register_chrdev(0, "hello", &hello_drv); /* /dev/hello */
71
72     hello_class = class_create(THIS_MODULE, "hello_class");
73     err = PTR_ERR(hello_class);
74     if (IS_ERR(hello_class)) {
75         printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
76         unregister_chrdev(major, "hello");
77         return -1;
78     }
79     device_create(hello_class, NULL, MKDEV(major, 0), NULL, "hello");
80
81     return 0;
82 }
83
84 /* 6. 有入口函数就应该有出口函数: 卸载驱动程序时, 就会去调用这个出口函数 */
85 static void __exit hello_exit(void)
86 {
87     unregister_chrdev(major, "hello");
88     class_destroy(hello_class);
89     return 0;
90 }
```



```
01 hello_drv.c > C hello_drv.c > hello_exit(void)
65 /* 5. 谁来注册驱动程序啊？得有一个入口函数；安装驱动程序时，就会去调用这个入口函数 */
66 static int __init hello_init(void)
67 {
68     int err;
69
70     printk("%%s %%s line %%d\n", __FILE__, __FUNCTION__, __LINE__);
71     major = register_chrdev(0, "hello", &hello_drv); /* /dev/hello */
72
73     hello_class = class_create THIS_MODULE, "hello_class");
74     err = PTR_ERR(hello_class);
75     if (IS_ERR(hello_class)) {
76         printk("%%s %%s line %%d\n", __FILE__, __FUNCTION__, __LINE__);
77         unregister_chrdev(major, "hello");
78         return -1;
79     }
80
81     device_create hello_class, NULL, MKDEV(major, 0), NULL, "hello"); /* /dev/hello */
82
83     return 0;
84
85 }
86
87 /* 6. 有入口函数就应该有出口函数：卸载驱动程序时，就会去调用这个出口函数 */
88 static void __exit hello_exit(void)
89 {
90     printk("%%s %%s line %%d\n", __FILE__, __FUNCTION__, __LINE__);
91     device_destroy hello_class, MKDEV(major, 0);
92     class_destroy hello_class);
93     unregister_chrdev(major, "hello");
94 }
95
```

```
01 hello_drv.c > C hello_drv.c > module_init(hello_init)
66 static int __init hello_init(void)
67 {
68     if (IS_ERR(hello_class)) {
69         return 0;
70     }
71     device_create(hello_class, NULL, MKDEV(major, 0), NULL, "hell
72 }
73
74 /* 6. 有入口函数就应该有出口函数：卸载驱动程序时，就会去调用这个出口函数 */
75 static void __exit hello_exit(void)
76 {
77     printk("%%s %%s line %%d\n", __FILE__, __FUNCTION__, __LINE__);
78     device_destroy(hello_class, MKDEV(major, 0));
79     class_destroy(hello_class);
80     unregister_chrdev(major, "hello");
81 }
82
83 /* 7. 其他完善：提供设备信息，自动创建设备节点 */
84
85 module_init(hello_init);
86 module_exit(hello_exit);
87
88 MODULE_LICENSE("GPL");
89
90
91
92
93
94
95
```

GPIO 的操作方法 (带 MMU) (韦东山)

翻译一下：

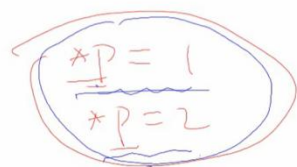
- ① 设置 CCM_CCGRx 寄存器中某位使能对应的 GPIO 模块 // 默认是使能的，上图省略了
- ② 设置 IOMUX 来选择引脚用于 GPIO
- ③ 设置 GPIOx_GDIR 中某位为 0，把该引脚设置为输入功能
- ④ 读 GPIOx_DR 或 GPIOx_PSR 得到某位的值（读 GPIOx_DR 返回的是 GPIOx_PSR 的值）

怎么用指针操作寄存器

volatile



int (*p) = >addr (Reg)

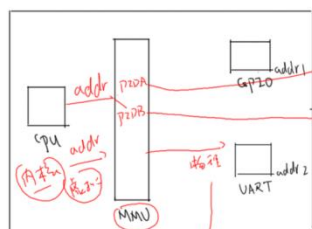


val = *p ;

不能乱

他意思是告诉编译器不要随便来优化我

进行地址映射，通过映射后的地址来操作原有地址来对物理寄存器进行操作



v = ioremap()

```
int g_a = 123;
int main(int argc, char **argv)
{
    printf("g_a's addr = 0x%x\n", &g_a);
    g_a = 456;
    while (1);
    return 0;
}
```

同一个程序，每次运行的结果应该一样。
这个程序，同时运行2次，每次的结果也应该一样。
同时运行2次，在内存中有两份代码，它们的地址不同，
为何打印出的地址还是一样的？

这是谁在起作用！ MMU: Memory Manager Unit

写APP的人水平有高低，
总不能让完全没有硬件知识的人去直接访问硬件吧？
怎么禁止他们访问硬件？

要用MMU！

MMU有两大作用：

1. 地址映射：CPU发出同样的地址（虚拟地址），执行不同的APP时，访问的是不同的物理地址
由MMU执行这个转换

2. 权限保护：CPU发出的地址，要经过MMU审核之后才可以访问具体硬件

```
/* registers */
// IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 地址: 0x02290000 + 0x14
static volatile unsigned int *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3;
```

```
/* ioremap */
// IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 地址: 0x02290000 + 0x14
IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 = ioremap(0x02290000 + 0x14, 4);
```

```
static int led_open(struct inode *inode, struct file *filp)
```

```
{
    /* enable gpio5
    * configure gpio5_io3 as gpio
    * configure gpio5_io3 as output
    */
    *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 &= ~0xf;
    *IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 |= 0x5;
```

```
// 卸载驱动的时候取消内存映射
iounmap(IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3);
```

有关韦东山按键驱动+MMU 的


```

struct iomux {
    volatile unsigned int unnames[23];
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I000; /* offset 0x5c */
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I001;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I002;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I003;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I004;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I005;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I006;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I007;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I008;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_I009;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_UART1_CTS_B;
};

```

```

iomux = ioremap(0x20e0000, sizeof(struct iomux));

```

```

struct imx6ull_gpio {
    volatile unsigned int dr;
    volatile unsigned int gdir;
    volatile unsigned int pscr;
    volatile unsigned int icr1;
    volatile unsigned int icr2;
    volatile unsigned int imr;
    volatile unsigned int isr;
    volatile unsigned int edge_sel;
};

```

```

gpio4 = ioremap(0x020A0000, sizeof(struct imx6ull_gpio));

```

32.6.7 SW_MUX_CTL_PAD_GPIO1_I000 SW MUX Control Register (IOMUXC_SW_MUX_CTL_PAD_GPIO1_I000)

SW_MUX_CTL Register

Address: 20E_0000h base: 20E_005Ch

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Field	Description
31-5	This field is reserved. Reserved.
4	Software Input On Field.
SION	Force the selected mux mode Input path no matter of MUX_MODE functionality.

LMX 6ULL Applications Processor Reference Manual, Rev. 1, 11/2017

GPIO memory map

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/page
209_C000	GPIO data register (GPIO1_DR)	32	R/W	0x00_0000h	28.5.1/1358
209_C004	GPIO direction register (GPIO1_GDIR)	32	R/W	0x00_0000h	28.5.2/1359
209_C008	GPIO pad status register (GPIO1_PSR)	32	R	0x00_0000h	28.5.3/1359
209_C00C	GPIO interrupt configuration register1 (GPIO1_ICR1)	32	R/W	0x00_0000h	28.5.4/1360
209_C010	GPIO interrupt configuration register2 (GPIO1_ICR2)	32	R/W	0x00_0000h	28.5.5/1364
209_C014	GPIO interrupt mask register (GPIO1_IMR)	32	R/W	0x00_0000h	28.5.6/1367
209_C018	GPIO interrupt status register (GPIO1_ISR)	32	W/C	0x00_0000h	28.5.7/1368
209_C01C	GPIO edge select register (GPIO1_EDGE_SEL)	32	R/W	0x00_0000h	28.5.8/1369
20A_0000	GPIO data register (GPIO2_DR)	32	R/W	0x00_0000h	28.5.1/1358
20A_0004	GPIO direction register (GPIO2_GDIR)	32	R/W	0x00_0000h	28.5.2/1359
20A_0008	GPIO pad status register (GPIO2_PSR)	32	R	0x00_0000h	28.5.3/1359
20A_000C	GPIO interrupt configuration register1 (GPIO2_ICR1)	32	R/W	0x00_0000h	28.5.4/1360
20A_0010	GPIO interrupt configuration register2 (GPIO2_ICR2)	32	R/W	0x00_0000h	28.5.5/1364
20A_0014	GPIO interrupt mask register (GPIO2_IMR)	32	R/W	0x00_0000h	28.5.6/1367

韦东山的按钮驱动代码

```

C button_drv.c: X
int button_init(void)
{
    major = register_chrdev(0, "100ask_button", &button_fops);
    button_class = class_create(THIS_MODULE, "100ask_button");
}

C button_drv.c: X
module_init(button_init);
module_exit(button_exit);

void button_exit(void)
{
    class_destroy(button_class);
    unregister_chrdev(major, "100ask_button");
}

C button_drv.c: X
static struct file_operations button_fops = {
    .open = button_open,
    .read = button_read,
};

C button_drv.c: X
static int button_open(struct inode *inode, struct file *file)
{
    int minor = iminor(inode);
    p_button_opr->init(minor);
    return 0;
}

C button_drv.c: X
static ssize_t button_read(struct file *file, char __user *buf, size_t size, loff_t *off)
{
    unsigned int minor = iminor(file->f_inode);
    char level;
    int err;

    level = p_button_opr->read(minor);
    err = copy_to_user(buf, &level, 1);
    return 1;
}

C button_drv.c: X
static struct button_operations *p_button_opr; // 全局变量

C board_xxx.c: X
static void board_xxx_button_init_gpio(int which)
{
    printk("%s %s %d, init gpio for button %d\n", __FILE__, __FUNCTION__, __LINE__, which);
}

static int board_xxx_button_read_gpio(int which)
{
    printk("%s %s %d, read gpio for button %d\n", __FILE__, __FUNCTION__, __LINE__, which);
    return 1;
}

C board_xxx.c: X
static struct button_operations my_buttons_ops = {
    .init = board_xxx_button_init_gpio,
    .read = board_xxx_button_read_gpio,
};

C board_xxx.c: X
int board_xxx_button_init(void)
{
    register_button_operations(&my_buttons_ops);
    return 0;
}

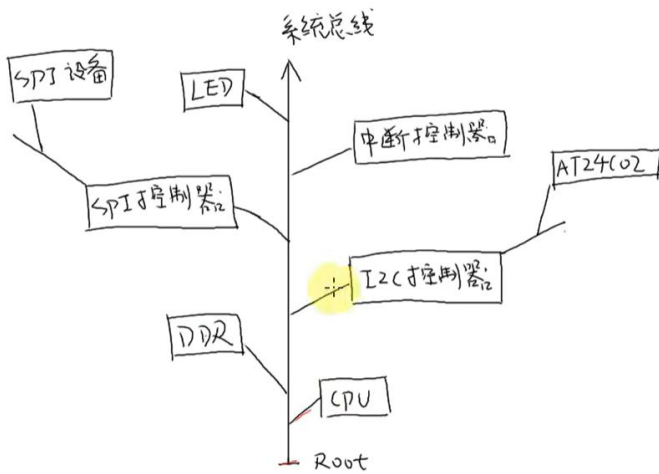
void board_xxx_button_exit(void)
{
    unregister_button_operations();
}

C button_drv.c: X
void register_button_operations(struct button_operations *opr)
{
    p_button_opr = opr;
    device_create(button_class, NULL, MKDEV(major, 1), NULL, "100ask_button%d", 1);
}

C button_drv.c: X
void unregister_button_operations(void)
{
    device_destroy(button_class, MKDEV(major, 1));
}

```

Linux 设备树



怎么写设备树

Linux设备树

文件分类

- .dtsi 描述 SOC 级信息
- .dts 描述板级信息
- .dtsb 编译后的文件

组成

- #include: 引用.h, .dtsi 和 dts 文件
- /: 根节点
- aliases 节点: 定义别名
- chosen 节点: 含有uboot代码加入的 bootargs 参数

节点

- 名字: label: node-name@unit-address 通过&label 来访问这个节点, 可以在不改源码文件的情况下向内部添加内容
- 属性类型
 - 字符串 eg: compatible = "arm,cortex-a7"
 - 32 位无符号整数 eg: reg = <0>;
 - 字符串列表 eg: compatible = "fsl,imx6ull-gpmi-nand", "fsl,imx6ull-gpmi-nand";
- 兼容性属性: compatible="manufacturer,model" 兼容性属性 如果在设备树中有哪个节点的 compatible 属性值与驱动代码里面描述的相等, 那么这个节点就会使用此驱动文件
- model = "<model_name>"; 描述设备模块信息
- status 属性
 - okay 设备可操作
 - disabled 设备当前不可操作, 但是在未来可以变为可操作的
 - fail 设备不可操作, 检测到错误
 - fail-sss 含义和 "fail" 相同, sss 部分是错误内容
- ranges 属性: 父子地址之间的物理转换
 - child-bus-address: 子总线地址空间的物理地址
 - parent-bus-address: 父总线地址空间的物理地址
 - length: 子地址空间的长度
- reg 属性: 设备地址空间资源信息 eg: reg = <address1 length1 address2 length2 address3 length3.....>
- name 属性: 记录节点名字
- device_type: 描述设备的 FCode

运行

- 内核在 /proc/device-tree 目录下根据节点名字创建不同文件夹

写法

- 看Linux内核源码的 /Documentation/device-tree/bindings

设备树读取: OF 函数

step1 查找节点

- 直接查找
 - struct device_node *of_find_node_by_name(struct device_node *start_search_node, const char *search_name): 通过节点名字查找指定的节点
 - struct device_node *of_find_node_by_type(struct device_node *start_search_node, const char *device_type): 通过 device_type 属性查找指定的节点
 - struct device_node *of_find_compatible_node(struct device_node *start_search_node, const char *device_type, const char *compatible): 根据 device_type 和 compatible 查找节点
 - struct device_node *of_find_matching_node_and_match(struct device_node *start_search_node, const struct of_device_id *matches, const struct of_device_id **return_matched_things): 通过 of_device_id 匹配表来查找指定的节点
 - inline struct device_node *of_find_node_by_path(const char *path): 通过路径来查找指定的节点
- 查找父/子节点
 - struct device_node *of_get_parent(const struct device_node *node): 获取指定节点的父节点
 - struct device_node *of_get_next_child(const struct device_node *parent_node, struct device_node *child_node_prev): 用迭代的方式查找子节点

step2 提取属性

- property *of_find_property(const struct device_node *device_node, const char *property_name, int *property_length): 查找指定的属性
- int of_property_count_elems_of_size(const struct device_node *device_node, const char *property_name, int element_size): 获取属性中元素的数量
- int of_property_read_u32_index(const struct device_node *device_node, const char *property_name, u32 read_index, u32 *return_values): 从属性中获取指定标号的 u32 类型数据值
- int of_property_read_string(struct device_node *device_node, const char *property_name, const char **return_string): 读取属性中字符串值
- int of_n_addr_cells(struct device_node *device_node): 用于获取 #address-cells 属性值
- int of_n_size_cells(struct device_node *device_node): 获取 #size-cells 属性值
- int of_device_is_compatible(const struct device_node *device_node, const char *compat): 查看节点的 compatible 属性是否有包含 compat 指定的字符串
- const _be32 *of_get_address(struct device_node *device_node, int read_index, u64 *addr_size, unsigned int *arguments): 获取地址相关属性
- u64 of_translate_address(struct device_node *device_node, const _be32 *input_addr): 将从设备树读取到的地址转换为物理地址
- int of_address_to_resource(struct device_node *device_node, int resource_index, struct resource *return_resource): 从设备树里面提取 reg 属性值, 转换为 resource 结构体类型
- void __iomem *of_iomap(struct device_node *device_node, int reg_index): 获取内存地址所对应的虚拟地址
- int of_property_read_u8_array(const struct device_node *device_node, const char *property_name, u8 *return_values, size_t read_element_num): 一次读取属性中 u8 类型的所有数组数据
- int of_property_read_u16_array(const struct device_node *device_node, const char *property_name, u16 *return_values, size_t read_element_num): 一次读取属性中 u16 类型的所有数组数据
- int of_property_read_u32_array(const struct device_node *device_node, const char *property_name, u32 *return_values, size_t read_element_num): 一次读取属性中 u32 类型的所有数组数据
- int of_property_read_u64_array(const struct device_node *device_node, const char *property_name, u64 *return_values, size_t read_element_num): 一次读取属性中 u64 类型的所有数组数据
- int of_property_read_u8(const struct device_node *device_node, const char *property_name, u8 *return_values): 读取只有一个 u8 类型值的属性
- int of_property_read_u16(const struct device_node *device_node, const char *property_name, u16 *return_values): 读取只有一个 u16 类型值的属性
- int of_property_read_u32(const struct device_node *device_node, const char *property_name, u32 *return_values): 读取只有一个 u32 类型值的属性
- int of_property_read_u64(const struct device_node *device_node, const char *property_name, u64 *return_values): 读取只有一个 u64 类型值的属性

根据原理图确定"驱动程序无法确定的硬件资源", 再在设备树文件中填写对应内容。
那么, 所填写内容的格式是什么?

① 看绑定文档

内核文档 Documentation/devicetree/bindings/
做得好的厂家也会提供设备树的说明文档

② 参考同类型单板的设备树文件

③ 网上搜索

④ 实在没办法时, 只能去研究驱动源码

设备匹配代码分析(使用设备树前后)

1、使用设备树之前设备匹配方法

arch/arm/mach-imx/mach-mx35_3ds.c

```
613 MACHINE_START(MX35_3DS, "Freescale MX35PDK")
614 /* Maintainer: Freescale Semiconductor, Inc */
615 .atag_offset = 0x100,
616 .map_io = mx35_map_io,
617 .init_early = imx35_init_early,
618 .init_irq = mx35_init_irq,
619 .init_time = mx35pdk_timer_init,
620 .init_machine = mx35_3ds_init,
621 .reserve = mx35_3ds_reserve,
622 .restart = mx35_restart,
623 MACHINE_END
```

arch/arm/include/asm/mach/arch.h

```
示例代码 43.3.4.3 MACHINE_START 和 MACHINE_END 宏定义
#define MACHINE_START(_type, _name) \
static const struct machine_desc __mach_desc_##_type \
__used \
__attribute__((section("__arch.info.init"))) = { \
    .nr = MACH_TYPE_##_type, \
    .name = _name, \
};
#define MACHINE_END
```



示例代码 43.3.4.3 展开以后

```
1 static const struct machine_desc __mach_desc_MX35_3DS
2 __used
3 __attribute__((section("__arch.info.init"))) = {
4     .nr = MACH_TYPE_MX35_3DS,
5     .name = "Freescale MX35PDK",
6     /* Maintainer: Freescale Semiconductor, Inc */
7     .atag_offset = 0x100,
8     .map_io = mx35_map_io,
9     .init_early = imx35_init_early,
10    .init_irq = mx35_init_irq,
11    .init_time = mx35pdk_timer_init,
12    .init_machine = mx35_3ds_init,
13    .reserve = mx35_3ds_reserve,
14    .restart = mx35_restart,
15 };
```

示例代码 43.3.4.3 mach-types.h 文件中的 machine id

```
15 #define MACH_TYPE_EBSA110 0
16 #define MACH_TYPE_RISPCFC 1
17 #define MACH_TYPE_EBSA285 4
18 #define MACH_TYPE_NETWINDER 5
19 #define MACH_TYPE_CATS 6
20 #define MACH_TYPE_SHARK 15
21 #define MACH_TYPE_BRUTUS 16
22 #define MACH_TYPE_PERSONAL_SERVER 17
.....
287 #define MACH_TYPE_MX35_3DS 1645
.....
1000 #define MACH_TYPE_PFLA03 4575
```

前面说了, uboot 会给 Linux 内核传递 machine id 这个参数, Linux 内核会检查这个 machine id, 其实就是将 machine id 与示例代码 43.3.4.3 中的这些 MACH_TYPE_XXX 宏进行对比, 看看有没有相等的, 如果相等的话就表示 Linux 内核支持这个设备, 如果不支持的话那么这个设备就没法启动 Linux 内核。

示例代码 43.3.4.7 setup_machine_fdt 函数内容

```
204 const struct machine_desc * __init setup_machine_fdt(unsigned int dt_phys)
205 {
206     const struct machine_desc *mdesc, *mdesc_best = NULL;
207     .....
214     if (!dt_phys || !early_init_dt_verify(phys_to_virt(dt_phys)))
215         return NULL;
216     mdesc = of_flat_dt_match_machine(mdesc_best,
217                                     默认 machine_desc
218                                     arch_get_next_mach);
219     .....
247     __machine_arch_type = mdesc->nr;
248     .....
249     return mdesc;
250 }
```

示例代码 43.3.4.8 of_flat_dt_match_machine 函数内容

```
705 const void * __init of_flat_dt_match_machine(const void
706 *default_match,
707 const void * (*get_next_compat)(const char * const**))
708 {
709     const void *data = NULL;
710     const void *best_data = default_match;
711     const char *compat;
712     unsigned long dt_root;
713     unsigned int best_score = ~1, score = 0;
714     dt_root = of_get_flat_dt_root(); 获取设备树根节点
715     while ((data = get_next_compat(&compat))) {
716         score = of_flat_dt_match(dt_root, compat);
717         if (score > 0 && score < best_score) {
718             best_data = data;
719             best_score = score;
720         }
721     }
722     .....
739     pr_info("Machine model: %s\n", of_flat_dt_get_machine_name());
740     return best_data;
741     .....
742     .....
743 }
```

将根节点 compatible 属性的值和每个 machine_desc 结构中 dt_compat 的值进行比较，直至找到匹配的那个 machine_desc

查找匹配的 machine_desc

arch/arm/kernel/setup.c 示例代码 43.3.4.6 setup_arch 函数内容

```
913 void __init setup_arch(char **cmdline_p)
914 {
915     const struct machine_desc *mdesc;
916     .....
917     setup_processor();
918     mdesc = setup_machine_fdt(__atags_pointer);
919     if (!mdesc)
920         mdesc = setup_machine_tags(__atags_pointer,
921                                   __machine_arch_type);
922     machine_desc = mdesc;
923     machine_name = mdesc->name;
924     .....
986 }
```

输入: uboot传递给Linux内核的dtb文件首地址
返回: 最匹配的 machine_desc

arch/arm/mach-imx/mach-imx6ul.c 示例代码 43.3.4.5 imx6ul 设备

```
208 static const char *imx6ul_dt_compat[] __initconst = {
209     "fsl,imx6ul",
210     "fsl,imx6ull",
211     NULL,
212 };
213
214 DT_MACHINE_START(IMX6UL, "Freescale i.MX6 Ultralite (Device Tree)")
215     .map_io = imx6ul_map_io,
216     .init_irq = imx6ul_init_irq,
217     .init_machine = imx6ul_init_machine,
218     .init_late = imx6ul_init_late,
219     .dt_compat = imx6ul_dt_compat,  "fsl,imx6ul" "fsl,imx6ull"
220 MACHINE_END
221
222 只要某个设备(板子)根节点"/"的 compatible 属性值与 imx6ul_dt_compat 表中的任何一个值相等，那么就表示 Linux 内核支持此设备。
```

示例代码 43.3.4.4 DT_MACHINE_START 宏

```
#define DT_MACHINE_START(name, namestr) \
static const struct machine_desc __mach_desc_##name \
__used \
__attribute__((__section__(".arch.info.init"))) = { \
    .nr = ~0, \
    .name = namestr, \
}
```

节点解析代码分析

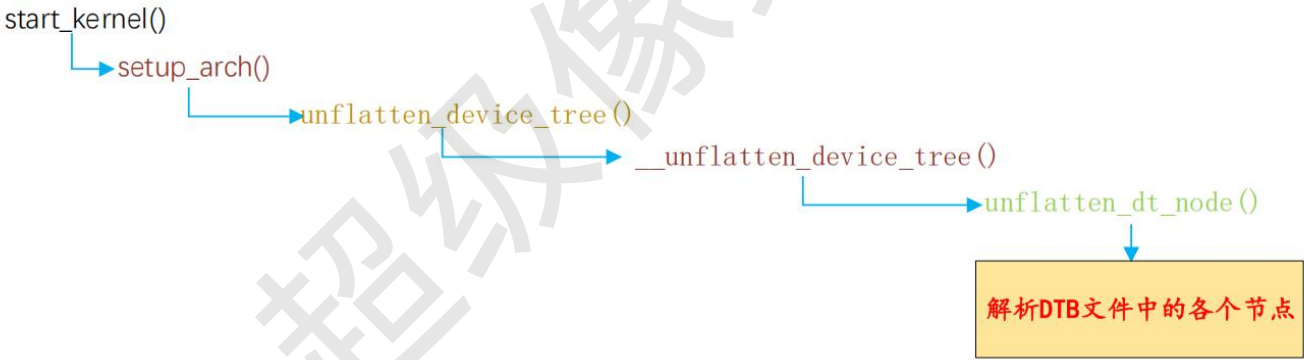


图 43.7.1 设备树节点解析流程。

从图 43.7.1 中可以看出，在 start_kernel 函数中完成了设备树节点解析的工作，最终实际工作的函数为 unflatten_dt_node。

字长 (Word Length) 是计算机体系结构中的核心概念，指CPU一次能处理的二进制数据的位数，直接影响计算性能、内存寻址能力和硬件设计。以下是全方位解析：

1. 核心定义

- 字 (Word) : CPU在单次操作中处理的基本数据单元。
- 字长: 该数据单元的二进制位数，例如:
 - 32位系统: 字长为32位 (4字节)
 - 64位系统: 字长为64位 (8字节)

设备树是怎么发挥作用的(韦东山)

dtb 文件经过 `gcc -E` 预编译后，然后用设备树的编译器编译生产 dtb 文件

dtb 文件在板子的 `/boot` 文件夹下，重启后 uboot 会自动调用，启动板子后按下空格进入 uboot，然后输入 `print` 找到

```
fdt_file=100ask_imx6ull-14x14.dtb
```

即为当前使用设备树

启动系统后，查看当前设备树文件目录：

```
[root@roc-rk3399-pc:~]# cd /sys/firmware/devicetree/
[root@roc-rk3399-pc:/sys/firmware/devicetree]# ls
base
[root@roc-rk3399-pc:/sys/firmware/devicetree]# cd base/
[root@roc-rk3399-pc:/sys/firmware/devicetree/base]# ls
#address-cells      iommu@ff903f00      qos@ffac0080
#size-cells         iommu@ff901400      qos@ffac8000
```

11.4 内核对设备树的处理

从源代码文件 dtb 文件开始，设备树的处理过程为：

DTS $\xrightarrow{\text{PC}}$ DTB $\xrightarrow{\text{内核}}$ device_node $\xrightarrow{\text{内核}}$ platform_device

- ① dtb 在 PC 机上被编译为 dtb 文件；
- ② u-boot 把 dtb 文件传给内核；
- ③ 内核解析 dtb 文件，把每一个节点都转换为 device_node 结构体；
- ④ 对于某些 device_node 结构体，会被转换为 platform_device 结构体。

设备树文件目录反编译(韦东山)

11.3.4 板子启动后查看设备树

板子启动后执行下面的命令：

```
# ls /sys/firmware/devicetree
devicetree fdt
```

`/sys/firmware/devicetree` 目录下是以目录结构呈现的 dtb 文件，根节点对应 `base` 目录，每一个节点对应一个目录，每一个属性对应一个文件。

这些属性的值如果是字符串，可以使用 `cat` 命令把它打印出来；对于数值，可以用 `hexdump` 把它打印出来。

还可以看到 `/sys/firmware/fdt` 文件，它就是 dtb 格式的设备树文件，可以把它复制出来放到 ubuntu 上，执行下面的命令反编译出来（-I dtb：输入格式是 dtb，-O dts：输出格式是 dts）：

```
cd 板子所用的内核源码目录
./scripts/dtc/dtc -I dtb -O dts /从板子上/复制出来的/fdt -o tmp.dts
```

pinctrl 和 gpio 子系统



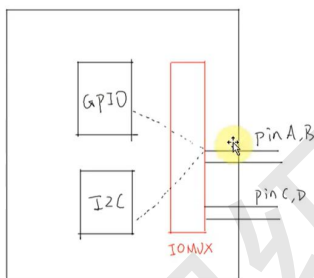
pinctl 子系统

为什么要引入这个子系统(韦东山)

16.1 Pinctrl 子系统重要概念

16.1.1 引入

无论是哪种芯片，都有类似下图的结构：

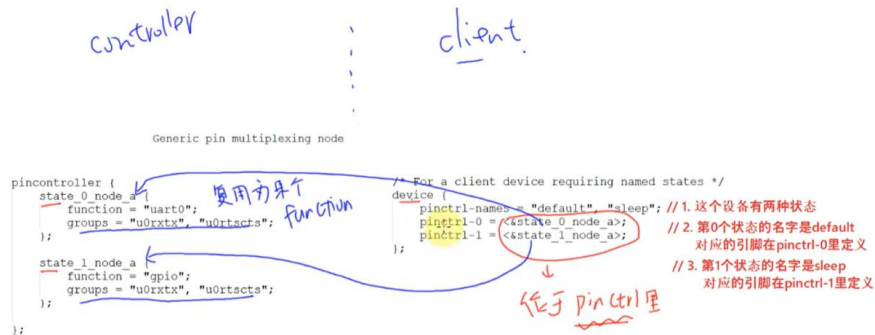


要想让 pinA、B 用于 GPIO，需要设置 IOMUX 让它们连接到 GPIO 模块：

要想让 pinA、B 用于 I2C，需要设置 IOMUX 让它们连接到 I2C 模块。

所以 GPIO、I2C 应该是并列的关系，它们能够使用之前，需要设置 IOMUX。有时候并不仅仅是设置 IOMUX，还要配置引脚，比如上拉、下拉、开漏等等。

设备树写法示例(需要参考厂商文档)(韦东山)



16.1.4 代码中怎么引用 pinctrl

这是透明的，我们的驱动基本不用管。当设备切换状态时，对应的 pinctrl 就会被调用。
比如在 platform_device 和 platform_driver 的枚举过程中，流程如下：

```
really probe {
    // 1. 引脚被设置为某个状态：根本不用我们自己去调用代码
    ret = pinctrl_bind_pins(dev);
    dev->pins->default_state = pinctrl_lookup_state(dev->pins->p,
        PINCTRL_STATE_DEFAULT); /* 获得"default"状态的pinctrl */
    dev->pins->init_state = pinctrl_lookup_state(dev->pins->p,
        PINCTRL_STATE_INIT); /* 获得"init"状态的pinctrl */

    ret = pinctrl_select_state(dev->pins->p, dev->pins->init_state); /* 优先设置"init"状态的引脚 */
    ret = pinctrl_select_state(dev->pins->p, dev->pins->default_state); /* 如果没有init状态，则设置"default"状态的引脚 */

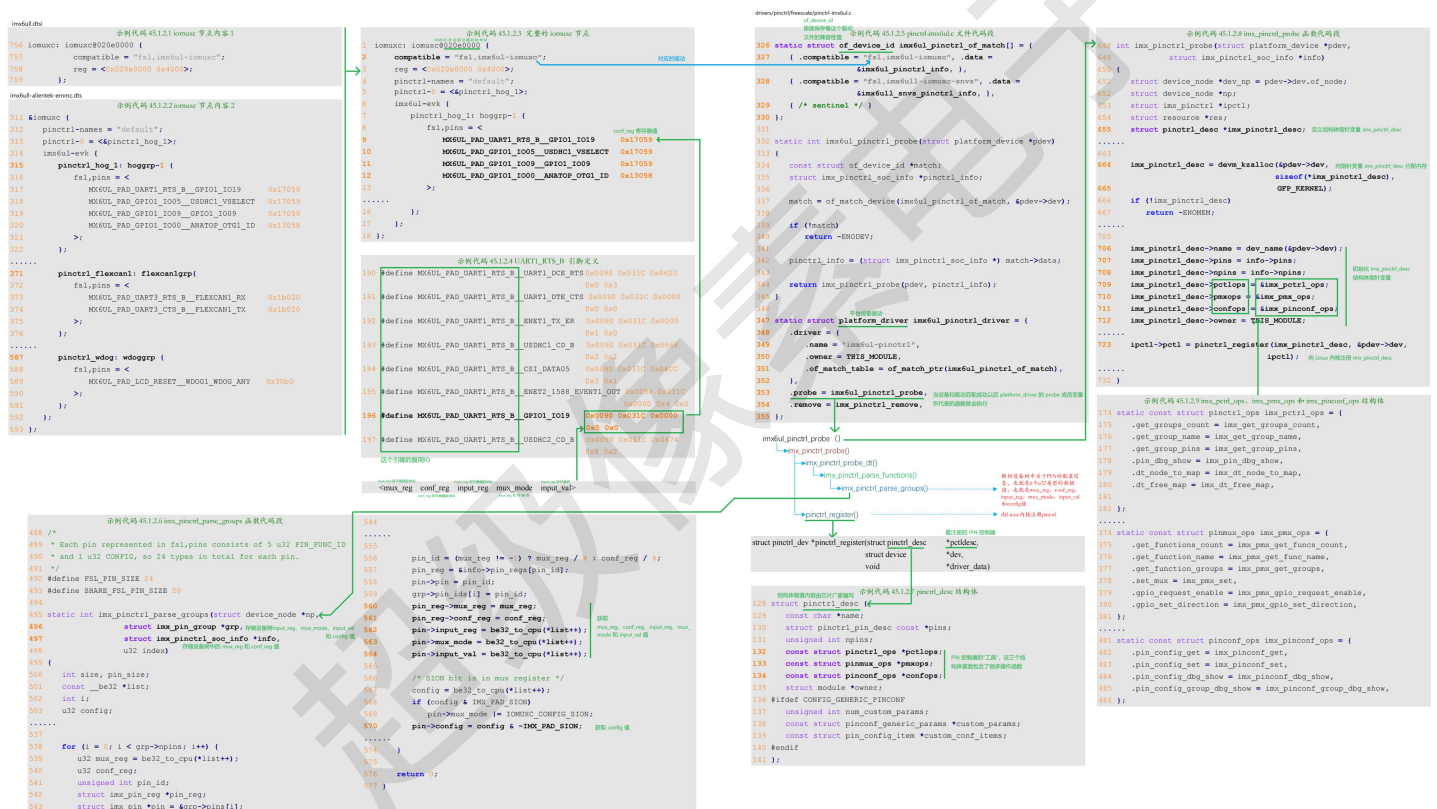
    *****
    ret = drv->probe(dev); // 2. 调用到我们的代码
}
```

非要自己调用，也有函数：

```
devm_pinctrl_get_select_default(struct device *dev); // 使用"default"状态的引脚
pinctrl_get_select(struct device *dev, const char *name); // 根据 name 选择某种状态的引脚
pinctrl_put(struct pinctrl *p); // 不再使用，退出时调用
```

I

设备树里面的配置是如何发挥作用的



gpio 子系统

设备树里面的配置是如何发挥作用的

```
imx6ul.dts
304 gpio1: gpio080209c000 {
305     compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
306     reg = <0x0209c000 0x4000>;
307     interrupts = <GIC_SPI 66 IRQ_TYPE_LEVEL_HIGH>;
308     <GIC_SPI 67 IRQ_TYPE_LEVEL_HIGH>;
309     gpio-controller;
310     #gpio-cells = <2>;
311     interrupt-controller;
312     #interrupt-cells = <2>;
313 };

316 pinctrl_hog_1: hoggrp-1 {
317     fsl,pins = <
318         MX6UL_PAD_UART1_RTS_B_GPIO1_IO19 0x17059 /* SD1 CD */
319     >;
320 };

322 >;
323 };

760 busdch1 {
761     pinctrl-names = "default", "state_100mhz", "state_200mhz";
762     pinctrl-0 = <&pinctrl_usdhc1>;
763     pinctrl-1 = <&pinctrl_usdhc1_100mhz>;
764     pinctrl-2 = <&pinctrl_usdhc1_200mhz>;
765     /* pinctrl-3 = <&pinctrl_hog_1>; */
766     cd-gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
767     keep-power-in-suspend;
768     enable-sdio-wakeup;
769     vmmc-supply = <&reg_sdi_vmmc>;
770     status = "okay";
771 };

152 static const struct of_device_id mx6_gpio_dt_ids[] = {
153     { .compatible = "fsl,imx31-gpio", .data = },
154     { .compatible = "fsl,imx21-gpio", .data = },
155     { .compatible = "fsl,imx31-gpio", .data = },
156     { .compatible = "fsl,imx35-gpio", .data = },
157     { .compatible = "fsl,imx35-gpio", .data = },
158 };

496 static struct platform_driver mx6_gpio_driver = {
497     .driver = {
498         .name = "gpio-mxc",
499         .of_match_table = mx6_gpio_dt_ids,
500     },
501     .probe = mx6_gpio_probe,
502     .id_table = mx6_gpio_devtype,
503 };
```

Documentation/devicetree/bindings/gpio/fsl-imx-gpio.txt
* Freescale i.MX/MXC GPIO controller

Required properties:

- compatible : Should be "fsl,imx35-gpio"
- reg : Address and length of the register set for the device
- interrupts : Should be the port interrupt shared by all 32 pins, if one number. If two numbers, the first one is the interrupt shared by low 16 pins and the second one is for high 16 pins.
- gpio-controller : Marks the device node as a gpio controller.
- #gpio-cells : Should be two. The first cell is the pin number and the second cell is used to specify the gpio polarity:
 - 0 = active high
 - 1 = active low
- interrupt-controller : Marks the device node as an interrupt controller.
- #interrupt-cells : Should be 2. The first cell is the GPIO number. The second cell bits[3:0] is used to specify trigger type and level flags:
 - 1 = low-to-high edge triggered.
 - 2 = high-to-low edge triggered.
 - 4 = active high level-sensitive.
 - 8 = active low level-sensitive.

```
static int mx6_gpio_probe(struct platform_device *pdev)
404 {
405     struct device_node *np = pdev->dev.of_node;
406     struct mx6_gpio_port *port;
407     struct resource *iores;
408     int irq_base;
409     int err;
410
411     mx6_gpio_get_hw(pdev);
412
413     port = devm_kzalloc(pdev->dev, sizeof(*port), GFP_KERNEL);
414     if (!port)
415         return -ENOMEM;
416
417     iores = platform_get_resource(pdev, IORESOURCE_MEM, 0);
418     port->base = devm_ioremap_resource(pdev->dev, iores);
419     if (IS_ERR(port->base))
420         return PTR_ERR(port->base);
421
422     port->irq_high = platform_get_irq(pdev, 1);
423     port->irq = platform_get_irq(pdev, 0);
424     if (port->irq < 0)
425         return port->irq;
426
427     /* disable the interrupt and clear the status */
428     writel(0, port->base + GPIO_IMR);
429     writel(0, port->base + GPIO_ISR);
430
431     if (mx6_gpio_hwttype == IMX21_GPIO) {
432         /*
433          * Setup one handler for all GPIO interrupts. Actually
434          * setting the handler is needed only once, but doing it for
435          * every port is more robust and easier.
436          */
437         irq_set_chained_handler(port->irq, mx21_irq_handler);
438     } else {
439         /* setup one handler for each entry */
440         irq_set_chained_handler(port->irq, mx3_gpio_irq_handler);
441         irq_set_handler_data(port->irq, port);
442         if (port->irq_high > 0) {
443             /* setup handler for GPIO 16 to 31 */
444             irq_set_chained_handler(port->irq_high,
445                                     mx3_gpio_irq_handler);
446             irq_set_handler_data(port->irq_high, port);
447         }
448     }
449     bkgc = gpiochip_add(&port->bkgc, pdev->dev, 0,
450                        port->base + GPIO_FSR,
451                        port->base + GPIO_DR, NULL,
452                        port->base + GPIO_GDIR, NULL, 0);
453     if (err)
454         goto out_bkgc;
455     bkgc->gc->to_irq = mx6_gpio_to_irq;
456     port->bkgc.gc.base = (pdev->id < 0) ? of_alias_get_id(np, "gpio")
457                                : 32 + pdev->id * 32;
458     err = gpiochip_add(&port->bkgc.gc);
459     if (err)
460         goto out_bkgc_remove;
461     irq_base = irq_alloc_descs(-1, 0, 32, numa_node_id());
462     if (irq_base < 0) {
463         err = irq_base;
464         goto out_irqdesc_free;
465     }
466     port->domain = irq_domain_add_legacy(np, 32, irq_base, 0,
467                                         &irq_domain_simple_ops, NULL);
468     if (!port->domain) {
469         err = -ENODEV;
470         goto out_irqdesc_free;
471     }
472     /* gpio-mxc can be a generic irq chip */
473     mx6_gpio_init_gc(port, irq_base);
474     list_add_tail(&port->node, &mx6_gpio_ports);
475     return 0;
476 }
477
478
479
480
481
482
483
484 }
```

```
mx6_gpio_port: 对 IMX35 GPIO 的封装
61 struct mx6_gpio_port {
62     struct list_head node;
63     void __iomem *base;
64     int irq;
65     int irq_high;
66     struct irq_domain *domain;
67     struct bgpio_chip bkgc;
68     u32 both_edges;
69 };

364 static void mx6_gpio_get_hw(struct platform_device *pdev)
365 {
366     const struct of_device_id *of_id =
367         of_match_device(mx6_gpio_dt_ids, pdev->dev);
368     enum mx6_gpio_hwttype hwttype;
369
370     if (hwttype == IMX35_GPIO)
371         mx6_gpio_hwdatas = &mx35_gpio_hwdatas;
372     else if (hwttype == IMX31_GPIO)
373         mx6_gpio_hwdatas = &mx31_gpio_hwdatas;
374     else
375         mx6_gpio_hwdatas = &imx1_imx21_gpio_hwdatas;
376     mx6_gpio_hwttype = hwttype;
377 }

101 static struct mx6_gpio_hwdatas imx35_gpio_hwdatas = {
102     .dr_reg = 0x000,
103     .gdir_reg = 0x004,
104     .par_reg = 0x008,
105     .icr1_reg = 0x00c,
106     .icr2_reg = 0x010,
107     .imr_reg = 0x014,
108     .isr_reg = 0x018,
109     .edge_sel_reg = 0x01c,
110     .low_level = 0x000,
111     .high_level = 0x001,
112     .rise_edge = 0x002,
113     .fall_edge = 0x003,
114 };

74 struct gpio_chip {
75     const char *label;
76     struct device *dev;
77     struct module *owner;
78     struct list_head list;
79
80     int (*request)(struct gpio_chip *chip,
81                  unsigned offset);
82     void (*free)(struct gpio_chip *chip,
83                unsigned offset);
84     int (*get_direction)(struct gpio_chip *chip,
85                        unsigned offset);
86     int (*direction_input)(struct gpio_chip *chip,
87                          unsigned offset);
88     int (*direction_output)(struct gpio_chip *chip,
89                           unsigned offset, int value);
90     int (*get)(struct gpio_chip *chip,
91              unsigned offset);
92     void (*set)(struct gpio_chip *chip,
93               unsigned offset, int value);
94     .....
145 };
```


用户写的代码是如何运行的(韦东山)

```
/* 2. 在入口函数注册platform driver */
static int __init led_init(void)
{
    err = platform_driver_register(&chip_deno_gpio_driver);
    if (err)
        return err;

    static struct platform_driver chip_deno_gpio_driver = {
        .probe = chip_deno_gpio_probe,
        .remove = chip_deno_gpio_remove,
        .driver = {
            .name = "100ask_led",
            .of_match_table = ask100_leds,
        },
    };

    static const struct of_device_id ask100_leds[] = {
        { .compatible = "100ask_led", },
        { },
    };

    /* 3. 在入口函数就注册出口函数：即驱动运行时，就会去调用这个出口函数？ */
    platform_driver_unregister(&chip_deno_gpio_driver);
    static void __exit led_exit(void)
    {
        printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
        platform_driver_unregister(&chip_deno_gpio_driver);
    }
}
```

/* 4.1 设备树中定义？ led-gpio->dev, "led", 0; */
led_gpio = gpio_get(&pdev->dev, "led", 0);
/* 4.2 设备树中定义？ */
major = register_chrdev(0, "100ask_led", &led_drv);
led_class = class_create(THIS_MODULE, "100ask_led");
device_create(led_class, NULL, MDEV(major, 0), NULL, "100ask_led", 0);
/* /dev/100ask_led0 */
static int __chip_deno_gpio_remove(struct platform_device *pdev)
{
 device_destroy(led_class, MDEV(major, 0));
 class_destroy(led_class);
 unregister_chrdev(major, "100ask_led");
 gpio_put(led_gpio, 0);
 return 0;
}

/* 定义自己的file operations结构？ */
static struct file_operations led_drv = {
 .owner = THIS_MODULE,
 .open = led_drv_open,
 .read = led_drv_read,
 .write = led_drv_write,
 .release = led_drv_close,
};

static int led_drv_open(struct inode *node, struct file *file)
{
 /* 根据设备号初始化LED */
 gpio_direction_output(led_gpio, 0);

 /* 3. 实现对应的open/read/write等函数。填入file operations结构？ */
 static size_t led_drv_read(struct file *file, char __user *buf, size_t size, loff_t *offset)
 {
 /* 根据设备号从GPIO读取LED */
 static size_t led_drv_write(struct file *file, const char __user *buf, size_t size, loff_t *offset)
 {
 err = copy_from_user(status, buf, 1);
 /* 根据设备号从GPIO写入LED */
 gpio_set_value(led_gpio, status);
 }

 static int led_drv_close(struct inode *node, struct file *file)
 {
 /* 根据设备号从GPIO关闭LED */
 }
 }
}

/* 设备树中定义？ */
/* 设备树中定义？ */
/* 设备树中定义？ */

设备树中定义？

设备树中定义？

设备树中定义？

设备树中定义？

设备树中定义？

设备树中定义？

并发与竞争-解决数据安全问题

原子变量操作 API 函数

函数	描述
atomic_t; //定义 a	
atomic_init(int i)	定义原子变量的时候对其初始化。
int atomic_read(atomic_t *v)	读取 v 的值，并返回。
void atomic_set(atomic_t *v, int i)	向 v 写入 i 值。
void atomic_add(atomic_t *v, int i)	给 v 加上 i 值。
void atomic_sub(atomic_t *v, int i)	从 v 减去 i 值。
void atomic_inc(atomic_t *v)	给 v 加 1，也就是自增。
void atomic_dec(atomic_t *v)	从 v 减 1，也就是自减。
int atomic_dec_return(atomic_t *v)	从 v 减 1，并返回 v 的值。
int atomic_inc_return(atomic_t *v)	给 v 加 1，并返回 v 的值。
int atomic_sub_and_test(int i, atomic_t *v)	从 v 减 i，如果结果为 0 就返回真，否则返回假。
int atomic_dec_and_test(atomic_t *v)	从 v 减 1，如果结果为 0 就返回真，否则返回假。
int atomic_inc_and_test(atomic_t *v)	给 v 加 1，如果结果为 0 就返回真，否则返回假。
int atomic_sub_and_negative(int i, atomic_t *v)	给 v 减 i，如果结果为负就返回真，否则返回假。

顺序锁

函数	描述
void set_bit(nr, void *p)	将 p 地址的第 nr 位置 1。
void clear_bit(nr, void *p)	将 p 地址的第 nr 位置 0。
void change_bit(nr, void *p)	将 p 地址的第 nr 位置进行翻转。
int test_and_set(nr, void *p)	获取 p 地址的第 nr 位置的值。
int test_and_set_bit(nr, void *p)	将 p 地址的第 nr 位置 1，并返回 nr 位置原来的值。
int test_and_clear_bit(nr, void *p)	将 p 地址的第 nr 位置清零，并返回 nr 位置原来的值。
int test_and_change_bit(nr, void *p)	将 p 地址的第 nr 位置翻转，并返回 nr 位置原来的值。

自旋锁 spinlock_t lock; //定义自旋锁

函数	描述
DEFINE_SPINLOCK(spinlock_t lock)	定义并初始化一个自旋锁。
int spin_lock_init(spinlock_t *lock)	初始化自旋锁。
void spin_lock(spinlock_t *lock)	获取指定的自旋锁，也叫加锁。
void spin_unlock(spinlock_t *lock)	释放指定的自旋锁。
int spin_trylock(spinlock_t *lock)	尝试获取指定的自旋锁，如果没有获取到就返回 0。
int spin_is_locked(spinlock_t *lock)	检查指定的自旋锁是否被获取，如果没有被获取就返回 0，否则返回 1。

获取锁之前关本地中断

函数	描述
void spin_lock_irq(spinlock_t *lock)	禁止本地中断，并获取自旋锁。
void spin_unlock_irq(spinlock_t *lock)	激活本地中断，并释放自旋锁。
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)	保存中断状态，禁止本地中断，并获取自旋锁。
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)	将中断状态恢复到以前的状态，并激活本地中断，并释放自旋锁。
void spin_lock_bh(spinlock_t *lock)	关闭下半部，并获取自旋锁。
void spin_unlock_bh(spinlock_t *lock)	打开下半部，并释放自旋锁。

自旋锁 API 函数

自旋锁：线程访问共享资源时，只能被一个线程持有，否则原地打转。

原子操作：不能再进一步分割的操作。

读写自锁：读和写不能同时进行，但是可以多人并发的读取。

顺序锁：可同时进行读，但是不允许同时进行并发的写操作。

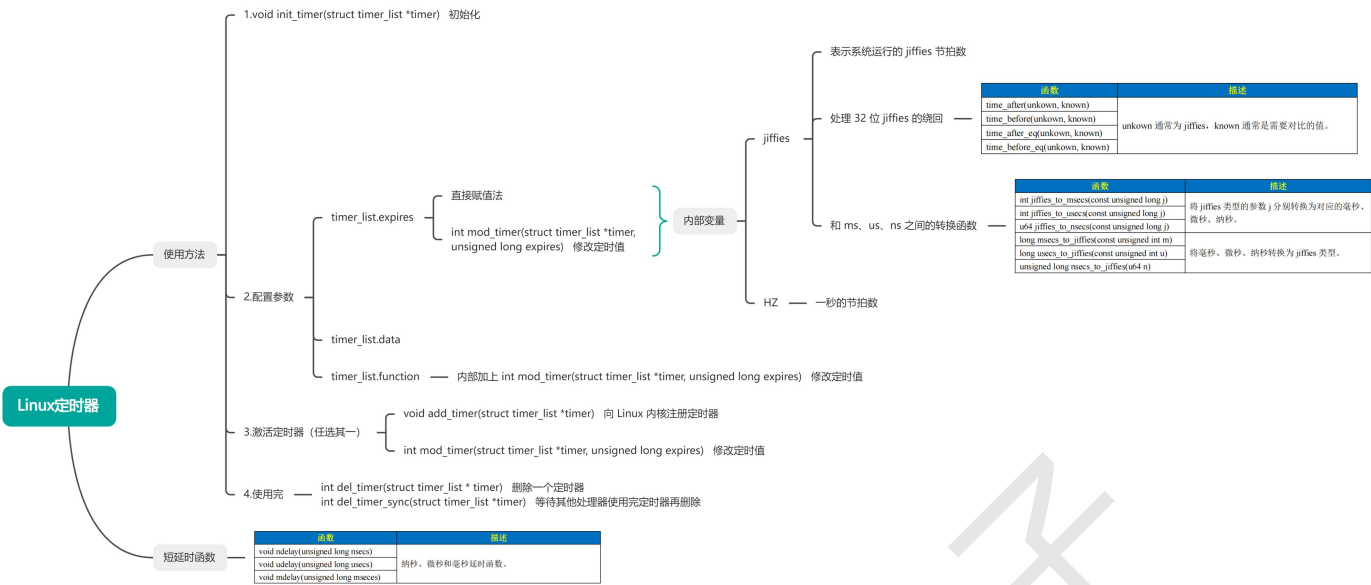
信号量：二值信号量：资源数 = 1；计数信号量：资源数 > 1。

互斥体：相当于二值信号量。

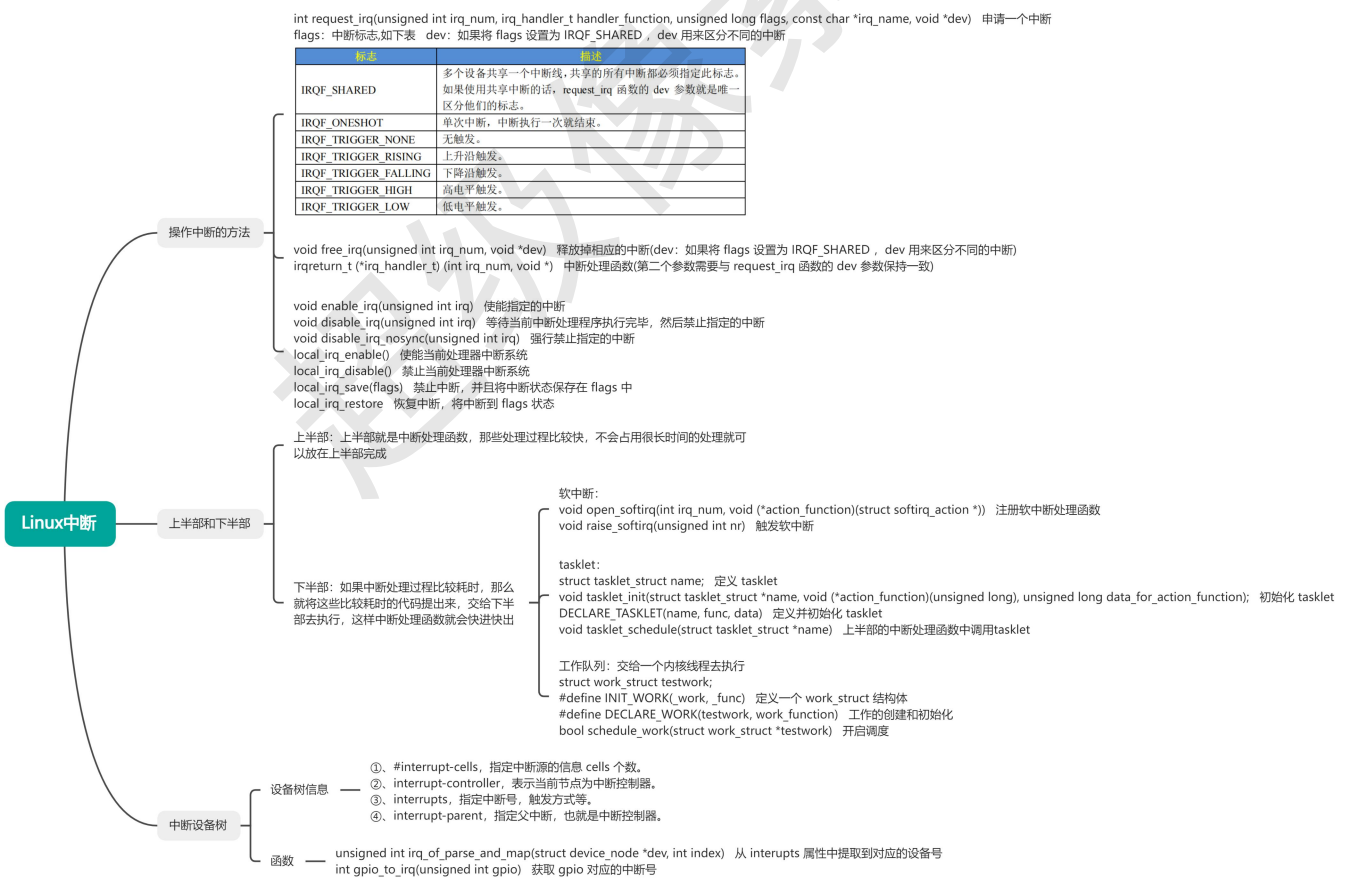
函数	描述
DEFINE_MUTEX(mutex_t name)	定义并初始化一个 mutex 变量。
void mutex_init(mutex_t *mutex, int val)	初始化 mutex，设置信号量为 val。
void mutex_lock(mutex_t *mutex)	获取 mutex，也就是给 mutex 上锁。如果获取不到就阻塞。
void mutex_unlock(mutex_t *mutex)	释放 mutex，也就是给 mutex 解锁。
int mutex_trylock(mutex_t *mutex)	尝试获取 mutex，如果成功就返回 1，如果失败就返回 0。
int mutex_is_locked(mutex_t *mutex)	判断 mutex 是否被获取，如果是的话就返回 1，否则返回 0。
int mutex_lock_interruptible(mutex_t *mutex)	使用此函数获取信号量失败时会被信号打断。

函数	描述
DEFINE_SEMAPHORE(sem_t name)	定义一个信号量，并且设置信号量的值为 1。
void sema_init(struct semaphore *sem, int val)	初始化信号量 sem，设置信号量为 val。
void down(struct semaphore *sem)	获取信号量，因为会导致休眠，因此不能在中断中使用。
int down_trylock(struct semaphore *sem)	尝试获取信号量，如果能获取到信号量就获取，并且返回 0，如果不能就返回非 0，并且不会进入休眠。
int down_interruptible(struct semaphore *sem)	获取信号量，和 down 类似，只是使用 down 进入休眠状态的线程不能信号打断，而使用此函数进入休眠以后是可以被信号打断的。
void up(struct semaphore *sem)	释放信号量。

Linux 内核定时器



Linux 中断



裸机的中断流程 (韦东山)

ARM 对异常(中断)的使用过程:

- a. 设置中断源 (屏蔽, 优先级)
- b. 设置中断控制器 (屏蔽, 优先级)
- c. 设置 CPU 寄存器 (使能中断)

① 初始化: 设置中断源, 使 CPU 产生中断

② 执行程序:

③ 产生中断: 按下按钮 → 中断控制器 → CPU

硬件

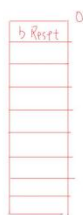
④ CPU 由执行完指令, 检查有无中断, 异常

⑤ 发现有异常/中断产生, 开始处理

对于不同的异常, 跳至不同的地址执行程序

异常向量

这些地址上, 只是一条跳转指令, 跳去执行某个函数。



这些地址上, 只是一条跳转指令, 跳去执行某个函数。

我们通常说的 PC 寄存器

对于 CPU 跳去执行一个函数

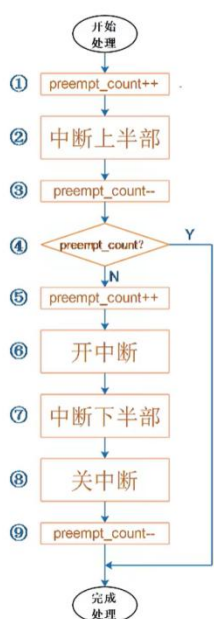
保持现场
调用处理函数
恢复现场

⑥ 这些函数:

软件

处理函数 (各种寄存器) 分配到中断源
处理异常 (中断) 再调用不同的函数
恢复现场

Linux 中断流程 (韦东山)



1. A 中断

2. A 中断再次发生

2

执行上半部
结果

2 次

执行上半部

中断下半部

恢复下半部

1 次

上: 下
1: 1

Linux 中断源码解析

```
include/linux/interrupt.h

示例代码 51.1.2.1 softirq_action 结构体
433 struct softirq_action
434 {
435     void (*action)(struct softirq_action *);
436 };

kernel/softirq.c

示例代码 51.1.2.2 softirq_vec 数组
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

示例代码 51.1.2.3 softirq_vec 数组

```
enum
{
    HI_SOFTIRQ=0,           /* 高优先级软中断 */
    TIMER_SOFTIRQ,         /* 定时器软中断 */
    NET_TX_SOFTIRQ,         /* 网络数据发送软中断 */
    NET_RX_SOFTIRQ,         /* 网络数据接收软中断 */
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,        /* tasklet 软中断 */
    SCHED_SOFTIRQ,          /* 调度软中断 */
    HRTIMER_SOFTIRQ,        /* 高精度定时器软中断 */
    RCU_SOFTIRQ,            /* RCU 软中断 */
    NR_SOFTIRQS = 10
};
```

中断设备树的写法

imx6ull.dtsi

示例代码 51.1.3.1 中断控制器 intc 节点

```
1 intc: interrupt-controller@00a01000 {
2     compatible = "arm,cortex-a7-gic";
3     #interrupt-cells = <3>;
4     interrupt-controller;
5     reg = <0x00a01000 0x1000>,
6         <0x00a02000 0x100>;
7 };
```

第一个 cells: 中断类型, 0 表示 SPI 中断, 1 表示 PPI 中断。
第二个 cells: 中断号, 对于 SPI 中断来说中断号的范围为 0~987, 对于 PPI 中断来说中断号的范围为 0~15。
第三个 cells: 标志, bit[3:0]表示中断触发类型, 为 1 的时候表示上升沿触发, 为 2 的时候表示下降沿触发, 为 4 的时候表示高电平触发, 为 8 的时候表示低电平触发。bit[15:8]为 PPI 中断的 CPU 掩码。

gpio5: gpio@020ac000 {

示例代码 51.1.3.2 gpio5 设备节点

```
2 compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
3 reg = <0x020ac000 0x4000>;
4 interrupts = <GIC_SPI 74 IRQ_TYPE_LEVEL_HIGH>,
5             <GIC_SPI 75 IRQ_TYPE_LEVEL_HIGH>;
6 gpio-controller;
7 #gpio-cells = <2>;
8 interrupt-controller;
9 #interrupt-cells = <2>;
10 };
```

IRQ	Interrupt Source	LOGIC	Interrupt Description
74	gpio5	-	Combined interrupt indication for GPIO5 signal 0 throughout 15
75	gpio5	-	Combined interrupt indication for GPIO5 signal 16 throughout 31

gpio5 节点也是个中断控制器, 用于控制 gpio5 所有 IO 的中断

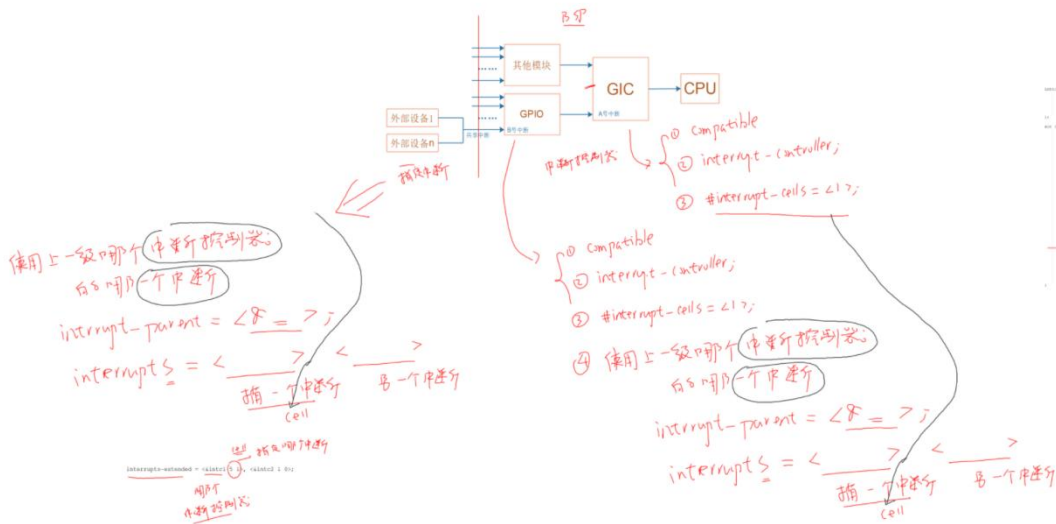
fxls8471@1e {

示例代码 51.1.3.3 fxls8471 设备节点

```
2 compatible = "fsl,fxls8471";
3 reg = <0x1e>;
4 position = <0>;
5 interrupt-parent = <&gpio5>; 使用 gpio5 作为中断控制器
6 interrupts = <0 8>;
7 };
```

GPIO5_IO00 低电平触发

①、#interrupt-cells, 指定中断源的信息 cells 个数。
②、interrupt-controller, 表示当前节点为中断控制器。
③、interrupts, 指定中断号, 触发方式等。
④、interrupt-parent, 指定父中断, 也就是中断控制器。



用户的驱动程序里中断代码的写法(韦东山)

中断上半部可以运行多次，下半部只能运行 1 次

```

/* 2. 在入口函数注册platform driver */
static int __init gpio_key_init(void)
{
    err = platform_driver_register(&gpio_keys_driver);
}

/* 1. 定义platform driver */
static struct platform_driver gpio_keys_driver = {
    .probe = gpio_key_probe,
    .remove = gpio_key_remove,
    .name = "gpio-keys",
    .of_match_table = of_match_table,
};

static const struct of_device_id gpio_keys_of_match_table[] = {
    { .compatible = "gpio-keys", },
    { }
};

/* 3. 在入口函数注册出口函数：即驱动启动时，就会去调用这个出口函数 */
static void __exit gpio_key_exit(void)
{
    platform_driver_unregister(&gpio_keys_driver);
}

static struct gpio_key {
    int gpio;
    struct gpio_desc *gpio_desc;
    int flags;
    int irq;
};

static int gpio_key_remove(struct platform_device *pdev)
{
    //int err;
    struct device_node *node = pdev->dev.of_node;
    int count;
    int i;

    count = of_gpio_count(node);
    for (i = 0; i < count; i++)
    {
        free_irq(gpio_keys_100ask[i].irq, &gpio_keys_100ask[i]);
        kfree(gpio_keys_100ask[i]);
    }
    return 0;
}

static struct gpio_key *gpio_keys_100ask;

/* 1. 从platform device获取gpio */
/* 2. gpio-irq */
/* 3. request_irq */

static int gpio_key_probe(struct platform_device *pdev)
{
    struct device_node *node = pdev->dev.of_node;
    count = of_gpio_count(node); // 计算与设备节点关联的GPIO数量
    for (i = 0; i < count; i++)
    {
        gpio_keys_100ask[i].gpio = of_get_gpio_flags(node, i, &flags); // 从设备树中获取第i个GPIO编号及其标志
        gpio_keys_100ask[i].gpio_desc = gpio_desc_to_gpio_desc(gpio_keys_100ask[i].gpio); // 将GPIO编号转换为gpio_desc指针
        gpio_keys_100ask[i].flags = flags & OF_GPIO_ACTIVE_LOW;
        if (flags & OF_GPIO_ACTIVE_LOW)
            flags |= GPIO_ACTIVE_LOW;
        err = devm_gpio_request_one(&pdev->dev, gpio_keys_100ask[i].gpio, flags, NULL); // 请求并配置GPIO，设置为输入模式，必要时设置为低电平有效
        gpio_keys_100ask[i].irq = gpio_to_irq(gpio_keys_100ask[i].gpio); // 将GPIO编号转换为IRQ编号
        for (i = 0; i < count; i++)
        {
            err = request_irq(gpio_keys_100ask[i].irq, gpio_key_isr, IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING, "gpio-keys", &gpio_keys_100ask[i]);
        }
    }

    static irqreturn_t gpio_key_isr(int irq, void *dev_id)
    {
        struct gpio_key *gpio_key = dev_id;
        int val;
        val = gpio_get_value(gpio_key->gpio_desc);
        printk("key %d %d\n", gpio_key->gpio, val);
        return IRQ_HANDLED;
    }
}

```

中断下半部 tasklet 的写法(韦东山)

19.6.2 tasklet 使用方法

先定义 tasklet，需要使用时调用 tasklet_schedule，驱动卸载前调用 tasklet_kill。

tasklet_schedule 只是把 tasklet 放入内核队列，它的 func 函数会在软件中断的执行过程中被调用。

```

struct tasklet_struct tasklet;

```

```
tasklet_init(&gpio_keys_100ask[i].tasklet, key_tasklet_func, &gpio_keys_100ask[i]);
tasklet_schedule(&gpio_key->tasklet);

static void key_tasklet_func(unsigned long data)
{
    tasklet_kill(&gpio_keys_100ask[i].tasklet);
}
```

工作队列的写法(韦东山)

第1步 构造一个 work_struct 结构体，里面有函数：

```
struct work_struct work;
INIT_WORK(&gpio_keys_100ask[i].work, key_work_func);

static void key_work_func(struct work_struct *work)
{
}
```

第2步 把这个 work_struct 结构体放入工作队列，内核线程就会运行 work 中的函数。

2 使用 work: schedule_work

调用 schedule_work 时，就会把 work_struct 结构体放入队列中，并唤醒对应的内核线程。内核线程就会从队列里把 work_struct 结构体取出来，执行里面的函数。

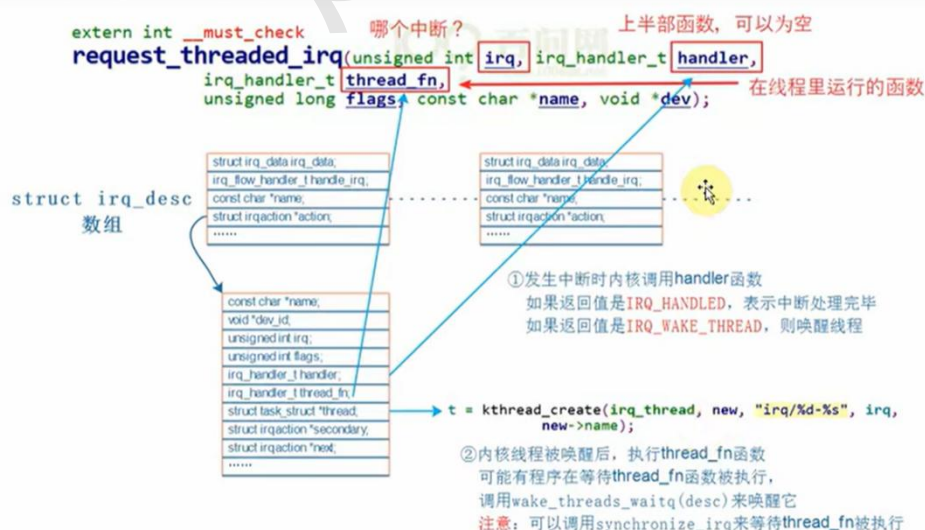
```
schedule_work(&gpio_key->work);
```

中断的线程化处理 (韦东山)

复杂、耗时的事情，尽量使用内核线程来处理。上节视频介绍的工作队列用起来挺简单，但是它有一个缺点：工作队列中有多个 work，前一个 work 没处理完会影响后面的 work。解决方法有很多种，比如干脆自己创建一个内核线程，不跟别的 work 凑在一块了。在 Linux 系统中，对于存储设备比如 SD/TF 卡，它的驱动程序就是这样做的，它有自己的内核线程。

对于中断处理，还有另一种方法：threaded irq，线程化的中断处理。中断的处理仍然可以认为分为上半部、下半部。上半部用来处理紧急的事情，下半部用一个内核线程来处理，这个内核线程专用于这个中断。

1. 调用 request_threaded_irq 后内核的数据结构



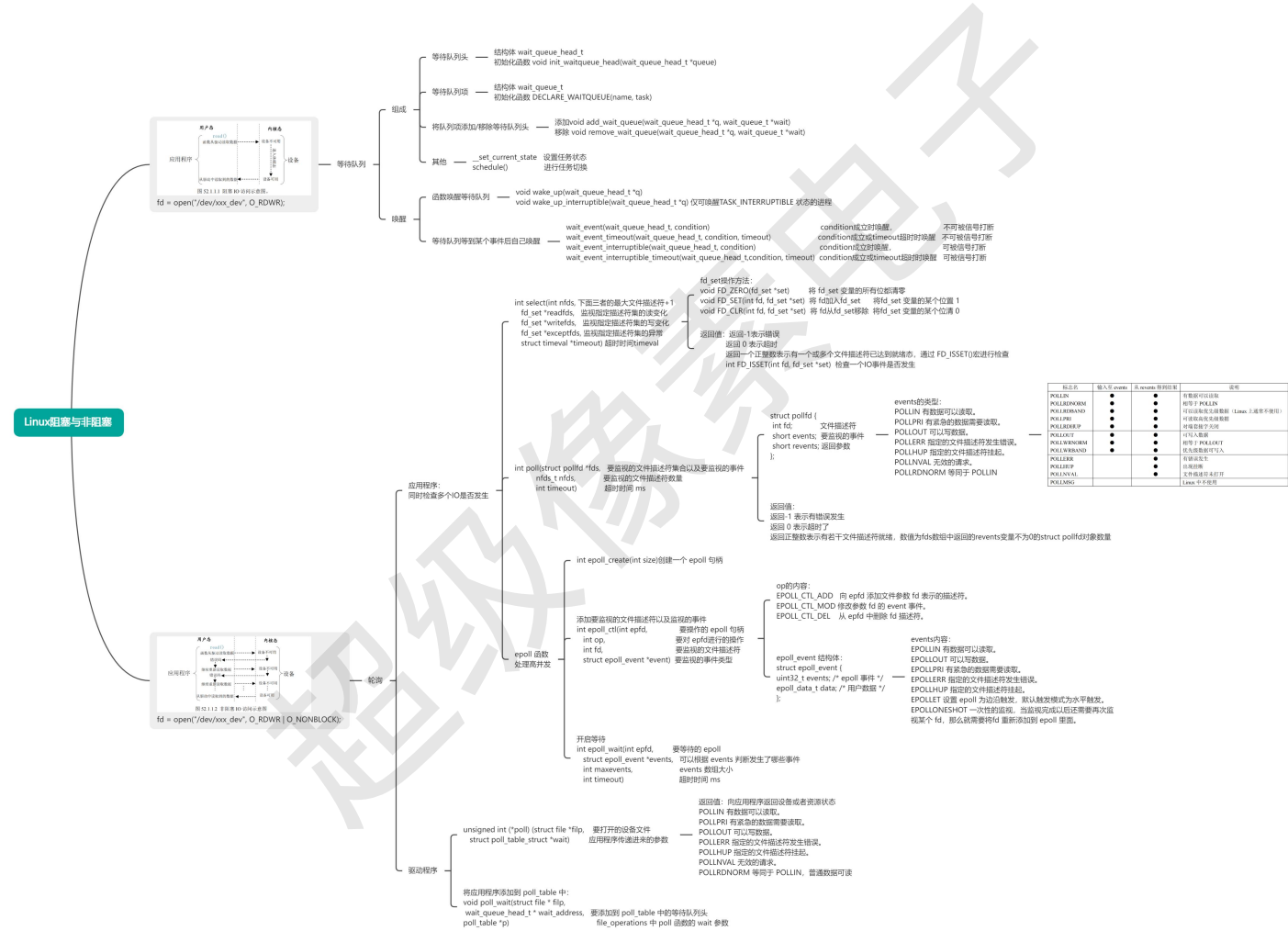
```
err = request_threaded_irq(gpio_keys_100ask[i].irq, gpio_key_isr, gpio_key_thread_func, IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING, "100ask gpio key", &gpio_keys_100ask[i]);
```

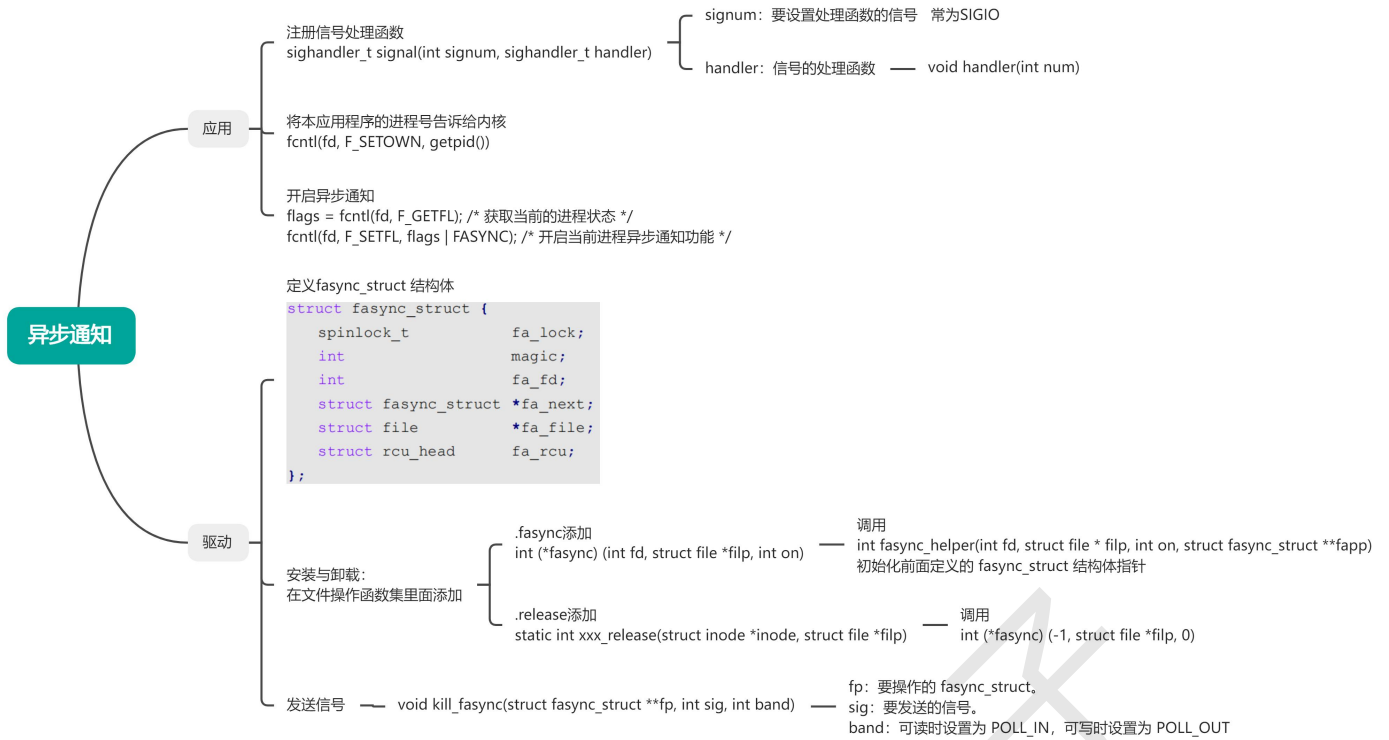
```
static irqreturn_t gpio_key_isr(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

static irqreturn_t gpio_key_thread_func(int irq, void *data)
{
    return IRQ_HANDLED;
}
```

提高 IO 效率的管理方法

程序编写方法





驱动和应用的交互方式 (韦东山)

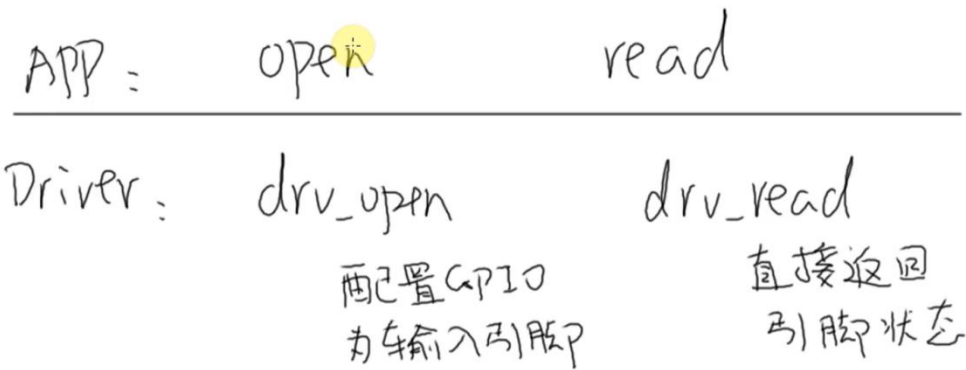
7.3.2 APP 访问硬件的 4 种方式：妈妈怎么知道孩子醒了



妈妈怎么知道卧室里小孩醒了？

- ① 时不时进房间看一下：**查询方式**
简单，但是累
- ② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：**休眠-唤醒**
不累，但是妈妈干不了活了
- ③ 妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟：**poll 方式**
要浪费点时间，但是可以继续干活。
妈妈要么是被小孩吵醒，要么是被闹钟吵醒。
- ④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：**异步通知**
妈妈、小孩互不耽误。

查询方式

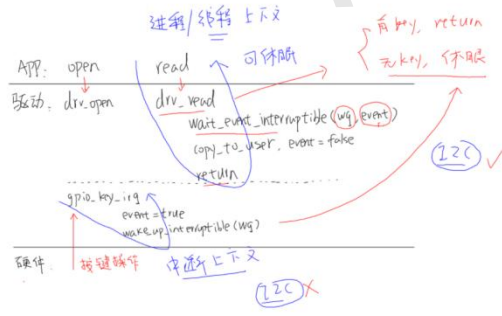
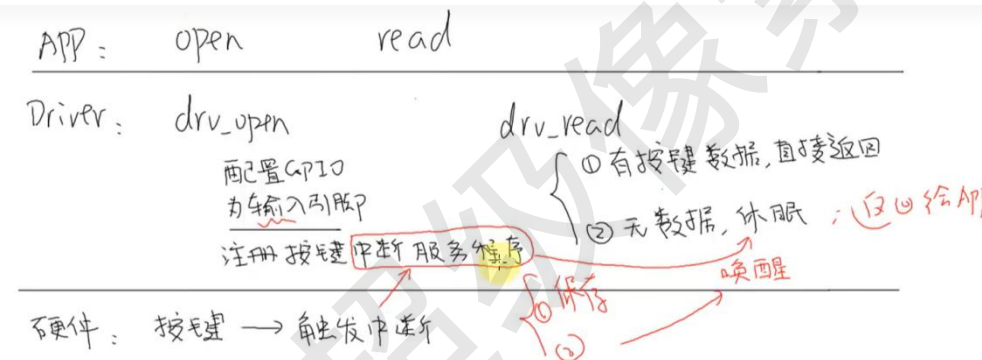


7.3.4 查询方式

APP 调用 open 函数时，传入“O_NONBLOCK”表示“非阻塞”。
APP 调用 read 函数读取数据时，如果驱动程序中有数据，那么 APP 的 read 函数会返回数据，否则也会立刻返回错误。

```
fd = open(argv[1], O_RDWR | O_NONBLOCK);  
len = read(fd, &event, sizeof(event));
```

阻塞中的休眠唤醒机制



- 13:
- ① 初始化 wg
 - ② wait_event (wg, event);
 - ③ wake_up (wg)
- wait queue

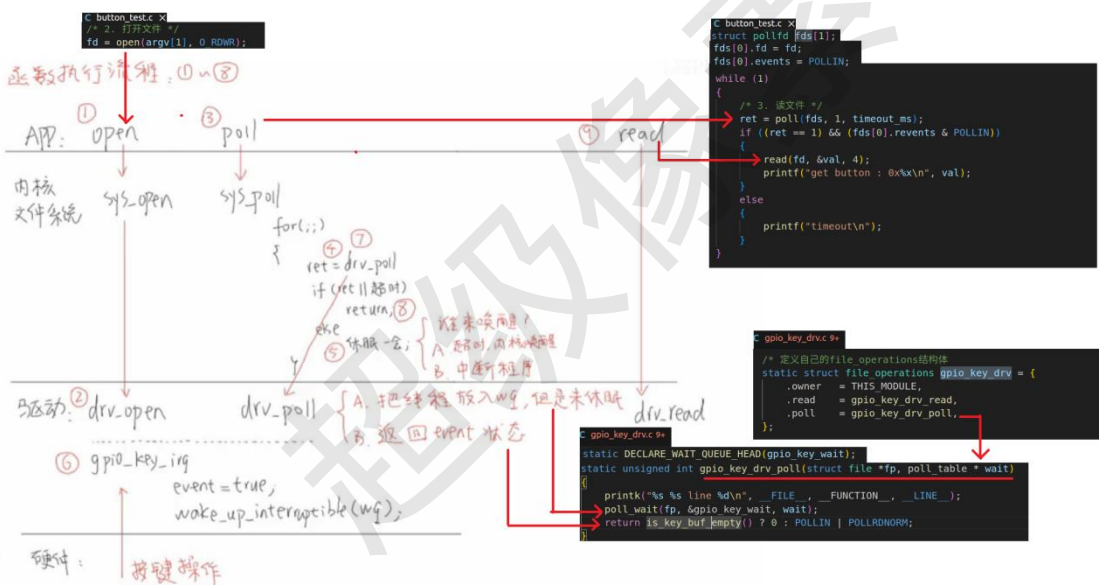
7.3.5 休眠-唤醒方式

APP 调用 open 函数时，不要传入 “O_NONBLOCK”。

APP 调用 read 函数读取数据时，如果驱动程序中有数据，那么 APP 的 read 函数会返回数据；否则 APP 就会在内核态休眠，当有数据时驱动程序会把 APP 唤醒，read 函数恢复执行并返回数据给 APP。

```
fd = open(argv[1], O_RDWR);  
len = read(fd, &event, sizeof(event));
```

非阻塞的 poll 机制



POLL 机制、SELECT 机制是完全一样的，只是 APP 接口函数不一样。

简单地说，它们就是“定个闹钟”：在调用 poll、select 函数时可以传入“超时时间”。在这段时间内，条件合适时(比如有数据可读、有空间可写)就会立刻返回，否则等到“超时时间”结束时返回错误。

```
fd = open(argv[1], O_RDWR | O_NONBLOCK); // 打开设备文件。  
  
// 设置 pollfd 结构体。  
fds[0].fd = fd; // 想查询哪个文件(fd) ?  
fds[0].events = POLLIN; // 想查询什么事件(POLLIN) ?  
fds[0].revents = 0; // 先清除“返回的事件”(revents)。  
  
ret = poll(fds, nfd, 5000); // 使用 poll 函数查询事件, 指定超时时间为 5000(ms)。  
// 在 poll(fds, nfd, 5000);  
// 这个调用中, nfd 参数表示你想要监视的文件描述符的数量,  
// 即 pollfd 结构体数组 fds 中元素的数量。这个参数告诉  
// poll() 函数需要检查多少个文件描述符的状态。
```

```

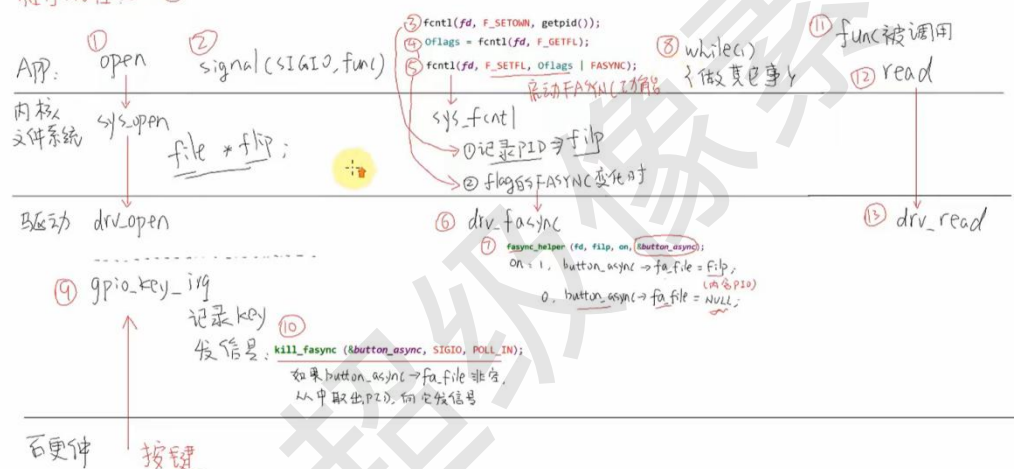
if (ret > 0)
{
    if (fds[0].revents == POLLIN)
    {
        while (read(fd, &event, sizeof(event)) == sizeof(event))
        {
            printf("get event: type = 0x%x, code = 0x%x, value = 0x%x\n", event.type, event.code, event.value);
        }
    }
}

```

异步通知



程序流程: ①~⑬



1. 编写信号处理函数:

```

16 void my_sig_handler(int sig)
17 {
18     struct input_event event;
19     while (read(fd, &event, sizeof(event)) == sizeof(event))
20     {
21         printf("get event: type = 0x%x, code = 0x%x, value = 0x%x\n", event.type, event.code, event.value);
22     }
23 }

```

2. 注册信号处理函数

```

71 /* 注册信号处理函数 */
72 signal(SIGIO, my_sig_handler);

```

3. 打开驱动:

```

74 /* 打开驱动程序 */
75 fd = open(argv[1], O_RDWR | O_NONBLOCK);

```

4. 把进程 ID 告诉驱动:

```

108 /* 把APP的进程号告诉驱动程序 */
109 fcntl(fd, F_SETOWN, getpid());

```

5.使能驱动的 FASYNC 功能:

```
111 /* 使能"异步通知" */
112 flags = fcntl(fd, F_GETFL);
113 fcntl(fd, F_SETFL, flags | FASYNC);
```

platform 平台总线

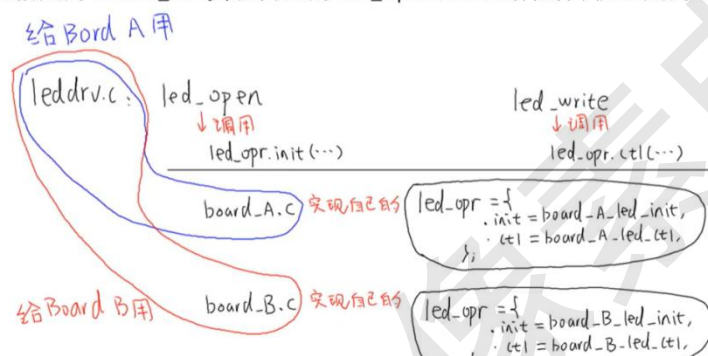
引入：自己做一个升级驱动模型 (韦东山)

2. 以面向对象的思想, 改进代码:

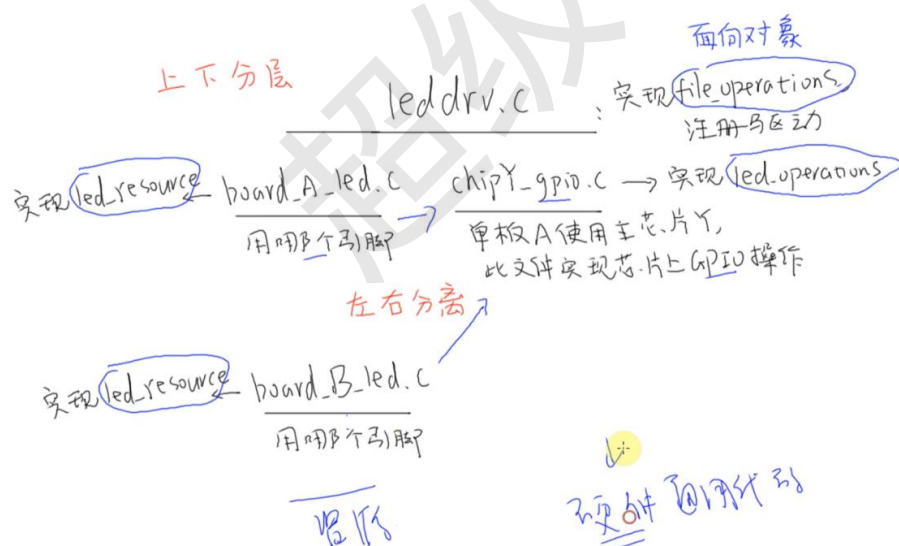
抽象出一个结构体:

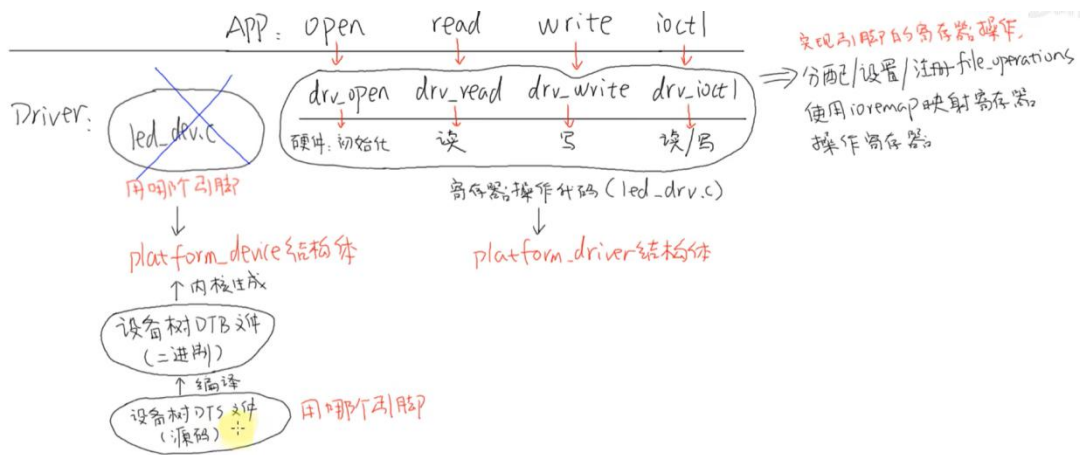
```
struct led_operations {
    int (*init) (int which); /* 初始化LED, which-哪个LED */
    int (*ctl) (int which, int status); /* 控制LED, which-哪个LED, status:1-亮,0-灭 */
};
```

每个单板相关的 board_X.c 实现自己的 led_operations 结构体, 供上层的 leddrv.c 调用:



把涉及寄存器（芯片）部分的代码和点灯动作的代码进行分离





超级像素电子

Device 的 driver 的匹配规则

源码分析

platform总线

```
include/linux/device.h
示例代码 54.2.1.1 bus_type 结构体代码段
1 struct bus_type {
2     const char *name; /* 总线名字 */
3     const char *dev_name;
4     struct device *dev_root;
5     struct device_attribute *dev_attrs;
6     const struct attribute_group **bus_groups; /* 总线属性 */
7     const struct attribute_group **dev_groups; /* 设备属性 */
8     const struct attribute_group **drv_groups; /* 驱动属性 */
9
10    int (*match)(struct device *dev, struct device_driver *drv); /* 完成设备与驱动之间匹配 */
11    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
12    int (*probe)(struct device *dev);
13    int (*remove)(struct device *dev);
14    void (*shutdown)(struct device *dev);
15
16    int (*online)(struct device *dev);
17    int (*offline)(struct device *dev);
18    int (*suspend)(struct device *dev, pm_message_t state);
19    int (*resume)(struct device *dev);
20    const struct dev_pm_ops *pm;
21    const struct iommu_ops *iommu_ops;
22    struct subsys_private *p;
23    struct lock_class_key lock_key;
24 }
```

driver/base/platform.c

```
示例代码 54.2.1.2 platform 总线实例
1 struct bus_type platform_bus_type = {
2     .name = "platform",
3     .dev_groups = platform_dev_groups,
4     .match = platform_match, /* 匹配函数 */
5     .uevent = platform_uevent,
6     .pm = &platform_dev_pm_ops,
7 };
```

示例代码 54.2.1.3 platform 总线实例

```
static int platform_match(struct device *dev,
                          struct device_driver *drv)
2 {
3     struct platform_device *pdev = to_platform_device(dev);
4     struct platform_driver *pdrv = to_platform_driver(drv);
5
6     /* When driver_override is set, only bind to the matching driver */
7     if (pdev->driver_override)
8         return !strcmp(pdev->driver_override, drv->name);
9
10    /* Attempt an OF style match first */
11    if (of_driver_match_device(dev, drv))
12        return 1;
13
14    /* Then try ACPI style match */
15    if (acpi_driver_match_device(dev, drv))
16        return 1;
17
18    /* Then try to match against the id table */
19    if (pdrv->id_table)
20        return platform_match_id(pdrv->id_table, pdev) != NULL;
21
22    /* fall-back to driver name match */
23    return (strcmp(pdev->name, drv->name) == 0);
24 }
```

platform驱动

```
示例代码 54.2.2.1 platform_driver 结构体
1 struct platform_driver {
2     int (*probe)(struct platform_device *); /* 当驱动与设备匹配成功以后 probe 函数就会执行 */
3     int (*remove)(struct platform_device *);
4     void (*shutdown)(struct platform_device *);
5     int (*suspend)(struct platform_device *, pm_message_t state);
6     int (*resume)(struct platform_device *);
7     struct device_driver driver; /* 相当于基类, 提供了最基本的驱动框架 */
8     const struct platform_device_id *id_table; /* id table 表 */
9     bool prevent_deferred_probe;
10 };
1 struct platform_device_id {
2     char name[PLATFORM_NAME_SIZE];
3     kernel_ulong_t driver_data;
4 };
```

```
include/linux/device.h
1 struct device_driver {
2     const char *name;
3     struct bus_type *bus;
4
5     struct module *owner;
6     const char *mod_name; /* used for built-in modules */
7
8     bool suppress_bind_attrs; /* disables bind/unbind via sysfs */
9
10    const struct of_device_id *of_match_table; /* 采用设备树的时候驱动使用的匹配表 */
11    const struct acpi_device_id *acpi_match_table;
12
13    int (*probe)(struct device *dev);
14    int (*remove)(struct device *dev);
15    void (*shutdown)(struct device *dev);
16    int (*suspend)(struct device *dev, pm_message_t state);
17    int (*resume)(struct device *dev);
18    const struct attribute_group **groups;
19
20    const struct dev_pm_ops *pm;
21
22    struct driver_private *p;
23 };
```

include/linux/mod_devicetable.h 示例代码 54.2.2.4 of_device_id 结构体

```
1 struct of_device_id {
2     char name[32];
3     char type[32];
4     char compatible[128];
5     const void *data;
6 };
```

platform设备

```
include/linux/platform_device.h
示例代码 54.2.3.1 platform_device 结构体代码段
22 struct platform_device {
23     const char *name; /* 设备名字 */
24     int id;
25     bool id_auto;
26     struct device dev;
27     u32 num_resources; /* 资源数量 */
28     struct resource *resource; /* 设备信息 */
29
30     const struct platform_device_id *id_entry;
31     char *driver_override; /* Driver name to force a match */
32
33     /* MFD cell pointer */
34     struct mfd_cell *mfd_cell;
35
36     /* arch specific additions */
37     struct pdev_archdata archdata;
38 };
```

示例代码 54.2.3.2 resource 结构体代码段

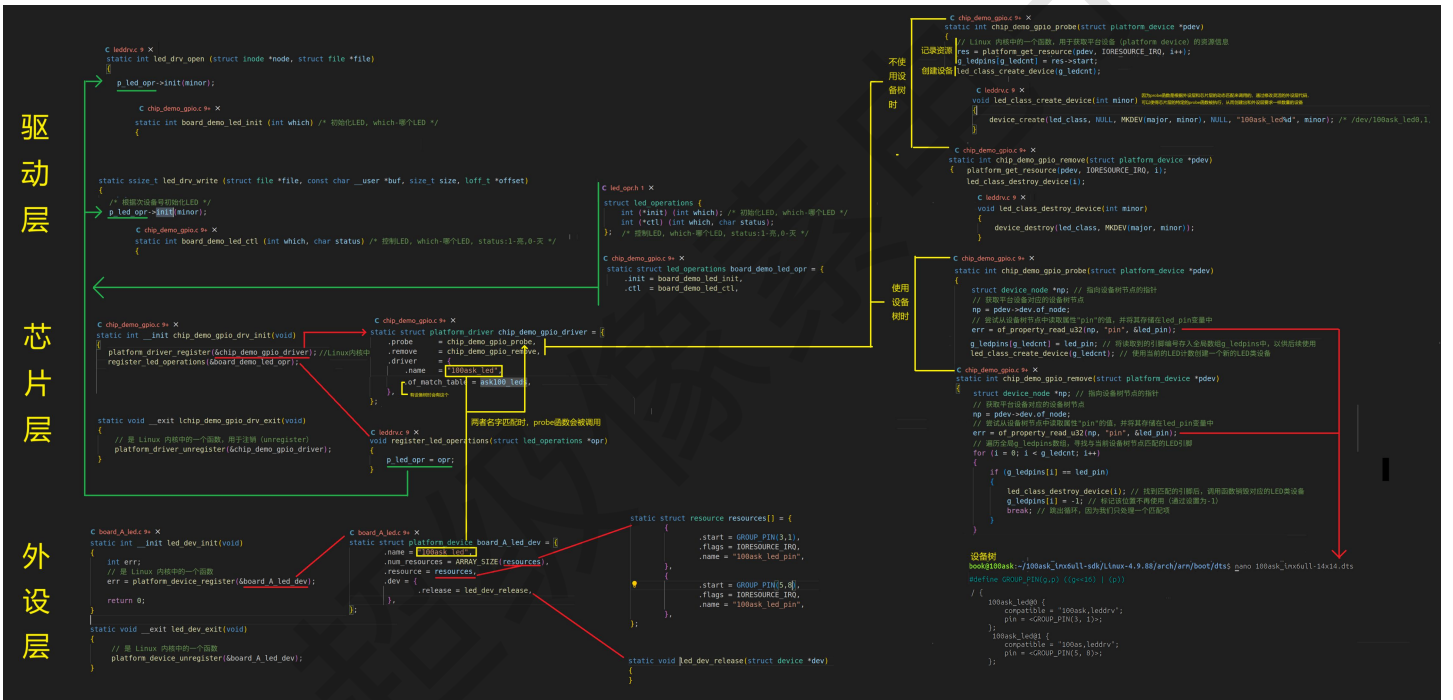
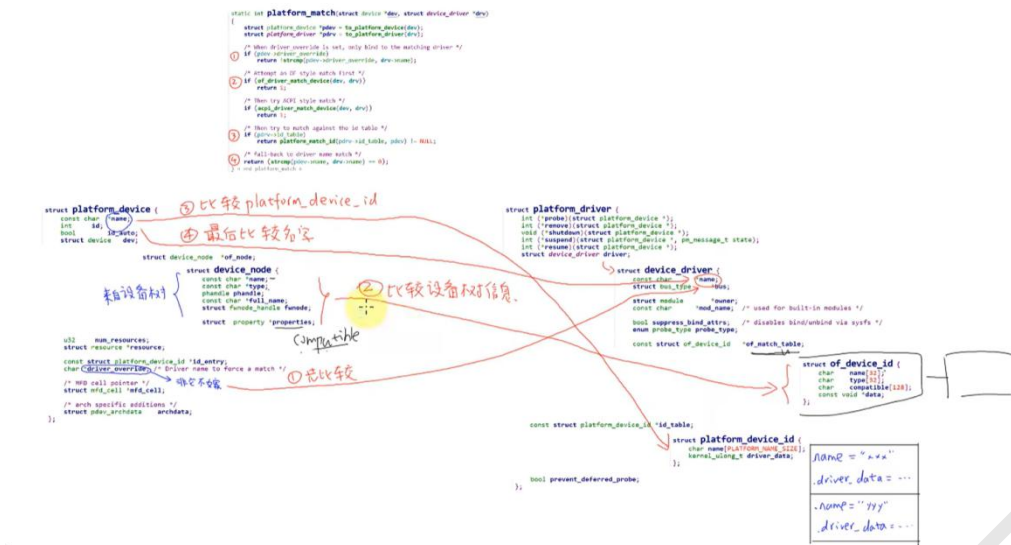
```
1 struct resource {
2     resource_size_t start; /* 起始信息或起始地址 */
3     resource_size_t end; /* 终止信息或终止地址 */
4     const char *name; /* 资源名字 */
5     unsigned long flags; /* 资源类型 */
6     struct resource *parent, *sibling, *child;
7 };
```

include/linux/ioport.h

示例代码 54.2.3.3 资源类型

```
29 #define IORESOURCE_BITS 0x000000ff /* Bus-specific bits */
30
31 #define IORESOURCE_TYPE_BITS 0x00001f00 /* Resource type */
32 #define IORESOURCE_IO 0x00000100 /* PCI/ISA I/O ports */
33 #define IORESOURCE_MEM 0x00000200
34 #define IORESOURCE_REG 0x00000300 /* Register offsets */
35 #define IORESOURCE_IRQ 0x00000400
36 #define IORESOURCE_DMA 0x00000800
37 #define IORESOURCE_BUS 0x00001000
38
39 .....
104 /* PCI control bits. Shares IORESOURCE_BITS with above PCI ROM. */
105 #define IORESOURCE_PCI_FIXED (1<<4) /* Do not move resource */
```


当有设备树时，匹配代码的样子 (韦东山)



哪些设备树节点会被转换为 platform_device

- A. 根节点下含有 compatible 属性的子节点
- I
- B. 含有特定 compatible 属性的节点的子节点
- 如果一个节点的 compatible 属性，它的值是这 4 者之一："simple-bus","simple-mfd","isa","arm,amba-bus",
- 那么它的子节点(需含 compatible 属性)也可以转换为 platform_device。
- C. 总线 I2C、SPI 节点下的子节点：不转换为 platform_device
- 某个总线下到子节点，应该交给对应的总线驱动程序来处理，它们不应该被转换为 platform_device。

使用设备树信息来判断 dev 和 drv 是否配对时，

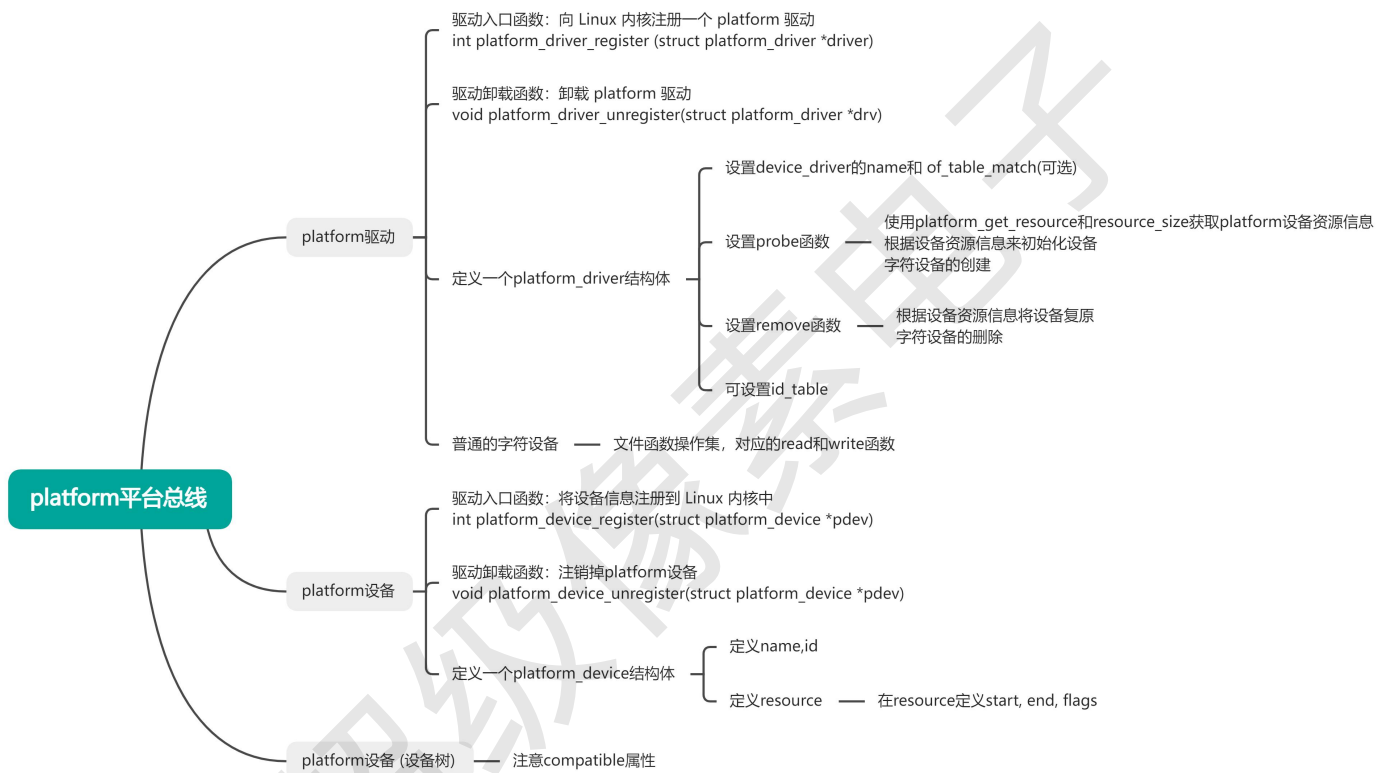
首先，如果 of_match_table 中含有 compatible 值，就跟 dev 的 compatible 属性比较，若一致则成功，否则返回失败；

其次，如果 of_match_table 中含有 type 值，就跟 dev 的 device_type 属性比较，若一致则成功，否则返回失败；

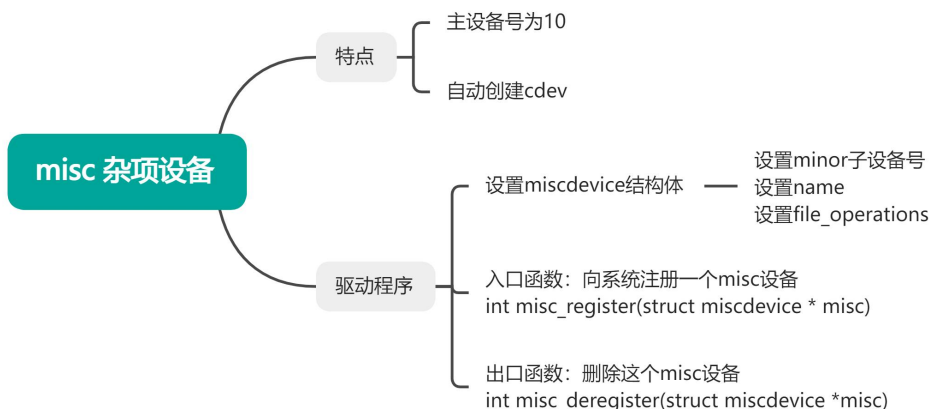
最后，如果 of_match_table 中含有 name 值，就跟 dev 的 name 属性比较，若一致则成功，否则返回失败。

而设备树中建议不再使用 device_type 和 name 属性，所以基本上只使用设备节点的 compatible 属性来寻找匹配的 platform_driver。

代码的写法

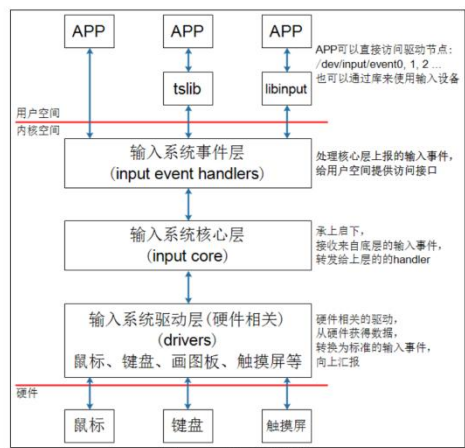


Misc 杂项设备



Input 子系统

基本介绍(韦东山)



驱动程序上报的数据含义三项重要内容:

type: 哪类? 比如EV_KEY, 按键类

code: 哪个? 比如KEY_A

value: 值, 比如0-按下, 1-松开

可以得到一系列的输入事件, 就是一个一个“struct input_event”, 它定义如下:

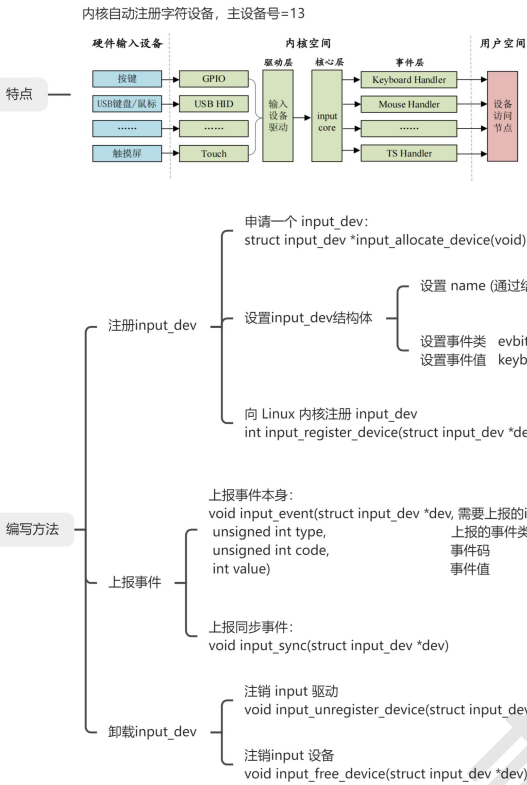
```
struct input_event { // include/uapi/linux/input.h
    struct timeval time;
    __u16 type;
    __u16 code;
    __s32 value;
};

struct timeval { // include/uapi/linux/time.h
    __kernel_time_t tv_sec; /* seconds */
    __kernel_suseconds_t tv_usec; /* microseconds */
};
```

每个输入事件 input_event 中都含有发生时间: timeval 表示的是“自系统启动以来过了多少时间”, 它是一个结构体, 含有“tv_sec、tv_usec”两项(即秒、微秒)。

代码编写方法

input子系统



input_dev内容

```
121 struct input_dev {
122     const char *name;
123     const char *phys;
124     const char *uniq;
125     struct input_id id;
126
127     unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];
128
129     unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; /* 事件类型的位图 */
130     unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; /* 按键值的位图 */
131     unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; /* 相对坐标的位图 */
132     unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; /* 绝对坐标的位图 */
133     unsigned long msckbit[BITS_TO_LONGS(MSC_CNT)]; /* 杂项事件的位图 */
134     unsigned long ledbit[BITS_TO_LONGS(LED_CNT)]; /* LED 相关的位图 */
135     unsigned long sndbit[BITS_TO_LONGS(SND_CNT)]; /* sound 有关的位图 */
136     unsigned long ffb[BITS_TO_LONGS(FF_CNT)]; /* 压力反馈的位图 */
137     unsigned long swbit[BITS_TO_LONGS(SW_CNT)]; /* 开关状态的位图 */
138
139     bool devres_managed;
140 };
```

input_event 结构体

```
24 struct input_event {
25     struct timeval time;
26     __u16 type;
27     __u16 code;
28     __s32 value;
29 };
```

驱动源码分析

drivers/input/input.c

示例代码 58.1.2.1 input 核心层创建字符设备过程

```
1767 struct class input_class = {
1768     .name      = "input",
1769     .devnode   = input_devnode,
1770 };
.....
2414 static int __init input_init(void)
2415 {
2416     int err;
2417
2418     err = class_register(&input_class);注册一个 input 类
2419     if (err) {
2420         pr_err("unable to register input_dev class\n");
2421         return err;
2422     }
2423
2424     err = input_proc_init();
2425     if (err)
2426         goto fail1;
2427
2428     err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),
2429                                INPUT_MAX_CHAR_DEVICES, "input");
2430     if (err) {
2431         pr_err("unable to register char major %d", INPUT_MAJOR);
2432         goto fail2;
2433     }
2434
2435     return 0;
2436
2437 fail2:    input_proc_exit();
2438 fail1:    class_unregister(&input_class);
2439     return err;
2440 }
```

include/uapi/linux/input.h

示例代码 58.1.2.4 按键值

```
215 #define KEY_RESERVED      0
216 #define KEY_ESC            1
217 #define KEY_1              2
218 #define KEY_2              3
219 #define KEY_3              4
220 #define KEY_4              5
221 #define KEY_5              6
222 #define KEY_6              7
223 #define KEY_7              8
224 #define KEY_8              9
225 #define KEY_9             10
226 #define KEY_0             11
.....
794 #define BTN_TRIGGER_HAPPY39 0x2e6
795 #define BTN_TRIGGER_HAPPY40 0x2e7
```

ls sys/class/

ata_device
ata_link
ata_port
backlight
bdi
block
dma
drm
firmware
/ # █

gpio
graphics
i2c-dev
ieee80211
input
lcd
leds
mdio_bus
mem

misc
mmc_host
mtd
net
power_supply
pps
ptp
pwm
rc

regulator
rfkill
rtc
scsi_device
scsi_disk
scsi_host
sound
spi_master
thermal

tty
ubi
udc
vc
video4linux
vtconsole
watchdog

include/linux/input.h

示例代码 58.1.2.2 input_dev 结构体

```
121 struct input_dev {
122     const char *name;
123     const char *phys;
124     const char *uniq;
125     struct input_id id;
126
127     unsigned long proppbit[BITS_TO_LONGS(INPUT_PROP_CNT)];
128
129     unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; /* 事件类型的位图 */
130     unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; /* 按键值的位图 */
131     unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; /* 相对坐标的位图 */
132     unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; /* 绝对坐标的位图 */
133     unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)]; /* 杂项事件的位图 */
134     unsigned long ledbit[BITS_TO_LONGS(LED_CNT)]; /* LED 相关的位图 */
135     unsigned long sndbit[BITS_TO_LONGS(SND_CNT)]; /* sound 有关的位图 */
136     unsigned long ffbbit[BITS_TO_LONGS(FF_CNT)]; /* 压力反馈的位图 */
137     unsigned long swbit[BITS_TO_LONGS(SW_CNT)]; /* 开关状态的位图 */
.....
189     bool devres_managed;
190 }
```

示例代码 58.1.2.3 事件类型

```
#define EV_SYN      0x00 /* 同步事件 */
#define EV_KEY      0x01 /* 按键事件 */
#define EV_REL      0x02 /* 相对坐标事件 */
#define EV_ABS      0x03 /* 绝对坐标事件 */
#define EV_MSC      0x04 /* 杂项 (其他) 事件 */
#define EV_SW       0x05 /* 开关事件 */
#define EV_LED      0x11 /* LED */
#define EV_SND      0x12 /* sound (声音) */
#define EV_REP      0x14 /* 重复事件 */
#define EV_FF       0x15 /* 压力事件 */
#define EV_PWR      0x16 /* 电源事件 */
#define EV_FF_STATUS 0x17 /* 压力状态事件 */
```

如何查询 input 设备节点(韦东山)

查找有什么设备节点

```
[root@100ask:~]# ls /dev/input/* -l
crw-rw---- 1 root input 13, 64 Feb  7 15:51 /dev/input/event0
crw-rw---- 1 root input 13, 65 Feb  7 15:51 /dev/input/event1

/dev/input/by-path:
total 0
lrwxrwxrwx 1 root root 9 Feb  7 15:51 platform-5c002000.i2c-event -> ../event0
lrwxrwxrwx 1 root root 9 Feb  7 15:51 platform-joystick-event -> ../event1
[root@100ask:~]# █
```

查看这个节点到底是什么设备


```
[root@100ask:~]# cat /proc/bus/input/devices
I: Bus=0018 Vendor=0416 Product=038f Version=1060
N: Name="Goodix Capacitive TouchScreen"
P: Phys=input/ts
S: Sysfs=/devices/platform/soc/5c002000.i2c/i2c-2/2-005d/input/input0
U: Uniq=
H: Handlers=kbd event0
B: PROP=2
B: EV=b 1
B: KEY=400 0 0 0 0 0 0 20000000 0 0 0
B: ABS=2658000 3
```

-
- ② N:name of the device
设备名称
 - ③ P:physical path to the device in the system hierarchy
系统层次结构中设备的物理路径。
 - ④ S:sysfs path
位于 sys 文件系统的路径
 - ⑤ U:unique identification code for the device(if device has it)
设备的唯一标识码
 - ⑥ H:list of input handles associated with the device.
与设备关联的输入句柄列表。
 - ⑦ B:bitmaps(位图)

EV=b: 上面的这个 b 转化为二进制后，如果最低位为 1，表示这个设备的 EV（输入事件）支持 0 号事件，以此类推。

ABS=2658000 3: 上面的这个 2658000 3 转化为二进制后，如果最低位为 1，表示这个设备的 ABS（绝对位移事件）支持 0 号事件，以此类推。

其他

如何查看事件原始值

```
hexdump /dev/input/event1
```

示例代码 58.4.2.1 input_event 类型的原始事件值

```
/* ****input_event 类型**** */
/* 编号 */ /* tv_sec */ /* tv_usec */ /* type */ /* code */ /* value */
00000000 0c41 0000 d7cd 000c 0001 000b 0001 0000
0000010 0c41 0000 d7cd 000c 0000 0000 0000 0000
0000020 0c42 0000 54bb 0000 0001 000b 0000 0000
0000030 0c42 0000 54bb 0000 0000 0000 0000 0000
```

Lcd 驱动

代码编写方法

- 1.修改设备树里面的 lcd 驱动 GPIO 配置信息
- 2.修改设备树里 lcdif 里面的 lcd 配置信息
- 3.修改设备树里面的 lcd 背光信息

驱动源码分析：

示例代码 59.12.1 imx6ull.dtsi 文件中 lcdif 节点内容

```
1 lcdif: lcdif@021c8000 {
2     compatible = "fsl,imx6ul-lcdif", "fsl,imx28-lcdif";
3     reg = <0x021c8000 0x4000>;
4     interrupts = <GIC_SPI 5 IRQ_TYPE_LEVEL_HIGH>;
5     clocks = <&clks IMX6UL_CLK_LCDIF_APB>;
6     <&clks IMX6UL_CLK_LCDIF_APB>;
7     <&clks IMX6UL_CLK_DUMMY>;
8     clock-names = "pix", "axi", "disp_axi";
9     status = "disabled";
10 };
```

drivers/video/fbdev/mxsfb.c

示例代码 59.12.2 platform 下的 LCD 驱动

```
1362 static const struct of_device_id mxsfb_dt_ids[] = {
1363     { .compatible = "fsl,imx6ul-lcdif", .data = &mxsfb_devtype[0], },
1364     { .compatible = "fsl,imx28-lcdif", .data = &mxsfb_devtype[1], },
1365     /* sentinel */
1366 };
1367
1368 .....
1369 static struct platform_driver mxsfb_driver = {
1370     .probe = mxsfb_probe,
1371     .remove = mxsfb_remove,
1372     .shutdown = mxsfb_shutdown,
1373     .id_table = mxsfb_devtype,
1374     .driver = {
1375         .name = DRIVER_NAME,
1376         .of_match_table = mxsfb_dt_ids,
1377         .pm = &mxsfb_pm_ops,
1378     },
1379 };
1380
1381 module_platform_driver(mxsfb_driver);
```

示例代码 59.12.3 fb_info 结构体

```
448 struct fb_info {
449     atomic_t count;
450     int node;
451     int flags;
452     struct mutex lock; /* 互斥锁 */
453     struct mutex mm_lock; /* 互斥锁，用于 fb_mmap 和 lmem_* 域 */
454     struct fb_var_screeninfo var; /* 当前可变参数 */
455     struct fb_fix_screeninfo fix; /* 当前固定参数 */
456     struct fb_monspecs monspecs; /* 当前显示器特性 */
457     struct work_struct queue; /* 帧缓冲事件队列 */
458     struct fb_pixmap pixmap; /* 图像硬件映射 */
459     struct fb_pixmap sprite; /* 光栅硬件映射 */
460     struct fb_cmap cmap; /* 当前调色板 */
461     struct list_head modelist; /* 当前模式列表 */
462     struct fb_videomode *mode; /* 当前视频模式 */
463
464 #ifdef CONFIG_FB_BACKLIGHT /* 如果 LCD 支持背光的话 */
465     /* assigned backlight device */
466     /* set before framebuffer registration,
467      * remove after unregister */
468     struct backlight_device *bl_dev; /* 背光设备 */
469
470     /* Backlight level curve */
471     struct mutex bl_curve_mutex;
472     u8 bl_curve[FB_BACKLIGHT_LEVELS];
473 #endif
474
475     .....
476     struct fb_ops *fbops; /* 帧缓冲操作函数集 */
477     struct device *device; /* 父设备 */
478     struct device *dev; /* 当前 fb 设备 */
479     int class_flag; /* 私有 sysfs 标志 */
480
481     .....
482     char __iomem *screen_base; /* 虚拟内存基地址 (屏幕显存) */
483     unsigned long screen_size; /* 虚拟内存大小 (屏幕显存大小) */
484     void *pseudo_palette; /* 伪 16 位调色板 */
485
486     .....
487 };
488
```

示例代码 59.12.5 mxsfb_ops 操作集合

```
987 static struct fb_ops mxsfb_ops = {
988     .owner = THIS_MODULE,
989     .fb_check_var = mxsfb_check_var,
990     .fb_set_par = mxsfb_set_par,
991     .fb_setcolreg = mxsfb_setcolreg,
992     .fb_ioctl = mxsfb_ioctl,
993     .fb_blank = mxsfb_blank,
994     .fb_pan_display = mxsfb_pan_display,
995     .fb_mmap = mxsfb_mmap,
996     .fb_fillrect = cfb_fillrect,
997     .fb_copyarea = cfb_copyarea,
998     .fb_imageblit = cfb_imageblit,
999 };

```

示例代码 59.12.4 mxsfb_probe 函数

```
1370 static int mxsfb_probe(struct platform_device *pdev)
1371 {
1372     const struct of_device_id *of_id =
1373         of_match_device(mxsfb_dt_ids, &pdev->dev);
1374     struct resource *res;
1375     struct mxsfb_info *host;
1376     struct fb_info *fb_info;
1377     struct pinctrl *pinctrl;
1378     int irq = platform_get_irq(pdev, 0);
1379     int gpio, ret;
1380
1381     .....
1382     /* 从设备树中获取 eLCDIF 接口控制器的寄存器首地址 */
1383     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
1384     if (!res) {
1385         dev_err(&pdev->dev, "Cannot get memory IO resource\n");
1386         return -ENODEV;
1387     }
1388
1389     /* 给 host 申请内存，host 为 mxsfb_info 类型结构体指针 */
1390     host = devm_kzalloc(&pdev->dev, sizeof(struct mxsfb_info),
1391         GFP_KERNEL);
1392
1393     if (!host) {
1394         dev_err(&pdev->dev, "Failed to allocate IO resource\n");
1395         return -ENOMEM;
1396     }
1397
1398     /* 给 fb_info 申请内存 */
1399     fb_info = framebuffer_alloc(sizeof(struct fb_info), &pdev->dev);
1400     if (!fb_info) {
1401         dev_err(&pdev->dev, "Failed to allocate fbdev\n");
1402         devm_kfree(&pdev->dev, host);
1403         return -ENOMEM;
1404     }
1405
1406     /* 将前面申请的 host 和 fb_info 联系在一起 */
1407     host->fb_info = fb_info;
1408     fb_info->par = host;
1409
1410     /* 申请中断，中断服务函数为 mxsfb_irq_handler */
1411     ret = devm_request_irq(&pdev->dev, irq, mxsfb_irq_handler, 0,
1412         dev_name(&pdev->dev), host);
1413     if (ret) {
1414         dev_err(&pdev->dev, "request_irq (%d) failed with
1415             error %d\n", irq, ret);
1416         ret = -ENODEV;
1417         goto fb_release;
1418     }
1419
1420     /* 对从设备树中获取到的寄存器首地址(res)进行内存映射，得到虚拟地址 */
1421     host->base = devm_ioremap_resource(&pdev->dev, res);
1422     if (IS_ERR(host->base)) {
1423         dev_err(&pdev->dev, "ioremap failed\n");
1424         ret = PTR_ERR(host->base);
1425         goto fb_release;
1426     }
1427
1428     /* 给 fb_info 中的 pseudo_palette 申请内存 */
1429     fb_info->pseudo_palette = devm_kzalloc(&pdev->dev, sizeof(u32) *
1430         16, GFP_KERNEL);
1431     if (!fb_info->pseudo_palette) {
1432         ret = -ENOMEM;
1433         goto fb_release;
1434     }
1435
1436     INIT_LIST_HEAD(&fb_info->modelist);
1437     pm_runtime_enable(&host->pdev->dev);
1438
1439     ret = mxsfb_init_fbinfo(host); /* 初始化 fb_info */
1440     if (ret != 0)
1441         goto fb_pm_runtime_disable;
1442     mxsfb_dispdrv_init(pdev, fb_info);
1443
1444     if (!host->dispdrv) {
1445         pinctrl = devm_pinctrl_get_select_default(&pdev->dev);
1446         if (IS_ERR(pinctrl)) {
1447             ret = PTR_ERR(pinctrl);
1448             goto fb_pm_runtime_disable;
1449         }
1450     }
1451     if (!host->enabled) {
1452         writel(0, host->base + LCDC_CTRL);
1453         mxsfb_set_par(fb_info); /* 设置 eLCDIF 控制器的相应寄存器 */
1454         mxsfb_enable_controller(fb_info);
1455         pm_runtime_get_sync(&host->pdev->dev);
1456     }
1457
1458     ret = register_framebuffer(fb_info);
1459     if (ret != 0) {
1460         dev_err(&pdev->dev, "Failed to register framebuffer\n");
1461         goto fb_destroy;
1462     }
1463
1464     .....
1465     return ret;
1466 }

```

RTC 驱动

驱动源码分析

示例代码 60.2.1 imx6ull.dtsi 文件 rtc 设备节点

```
1 snvs_rtc: snvs-rtc-lp {
2     compatible = "fsl,sec-v4.0-mon-rtc-lp";
3     regmap = <&snvs>;
4     offset = <0x34>;
5     interrupts = <GIC_SPI 19 IRQ_TYPE_LEVEL_HIGH>, <GIC_SPI 20
        IRQ_TYPE_LEVEL_HIGH>;
6 };
```

示例代码 60.2.2 rtc 设备 platform 驱动框架

```
380 static const struct of_device_id snvs_dt_ids[] = {
381     { .compatible = "fsl,sec-v4.0-mon-rtc-lp", },
382     { /* sentinel */ }
383 };
384 MODULE_DEVICE_TABLE(of, snvs_dt_ids);
385
386 static struct platform_driver snvs_rtc_driver = {
387     .driver = {
388         .name = "snvs_rtc",
389         .pm = SNVS_RTC_PM_OPS,
390         .of_match_table = snvs_dt_ids,
391     },
392     .probe = snvs_rtc_probe,
393 };
394 module_platform_driver(snvs_rtc_driver);
```

示例代码 60.2.4 snvs_rtc_ops 操作集

```
200 static const struct rtc_class_ops snvs_rtc_ops = {
201     .read_time = snvs_rtc_read_time,
202     .set_time = snvs_rtc_set_time,
203     .read_alarm = snvs_rtc_read_alarm,
204     .set_alarm = snvs_rtc_set_alarm,
205     .alarm_irq_enable = snvs_rtc_alarm_irq_enable,
206 };
```

示例代码 60.2.5 snvs_rtc_read_time 函数代码段

```
126 static int snvs_rtc_read_time(struct device *dev,
        struct rtc_time *tm)
127 {
128     struct snvs_rtc_data *data = dev_get_drvdata(dev);
129     unsigned long time = rtc_read_lp_counter(data); // 获取 RTC 计数值
130
131     rtc_time_to_tm(time, tm); // 将获取到的秒数转换为时间值
132
133     return 0;
134 }
```

示例代码 60.2.6 rtc_time 结构体类型

```
20 struct rtc_time {
21     int tm_sec;
22     int tm_min;
23     int tm_hour;
24     int tm_mday;
25     int tm_mon;
26     int tm_year;
27     int tm_wday;
28     int tm_yday;
29     int tm_isdst;
30 };
```

示例代码 60.2.7 rtc_read_lp_counter 函数代码段

```
50 static u32 rtc_read_lp_counter(struct snvs_rtc_data *data)
51 {
52     u64 read1, read2;
53     u32 val;
54
55     do {
56         regmap_read(data->regmap, data->offset + SNVS_LPSRTCMR,
            &val);
57         read1 = val;
58         read1 <= 32;
59         regmap_read(data->regmap, data->offset + SNVS_LPSRTCLR,
            &val);
60         read1 |= val;
61
62         regmap_read(data->regmap, data->offset + SNVS_LPSRTCMR,
            &val);
63         read2 = val;
64         read2 <= 32;
65         regmap_read(data->regmap, data->offset + SNVS_LPSRTCLR,
            &val);
66         read2 |= val;
67
68         /*
69          * when CPU/BUS are running at low speed, there is chance that
70          * we never get same value during two consecutive read, so here
71          * we only compare the second value.
72          */
73     } while ((read1 >> CNTR_TO_SECS_SH) != (read2 >>
        CNTR_TO_SECS_SH));
74
75     /* Convert 47-bit counter to 32-bit raw second count */
76     return (u32) (read1 >> CNTR_TO_SECS_SH); // 返回时间值
77 }
```

示例代码 60.2.3 snvs_rtc_probe 函数代码段

```
238 static int snvs_rtc_probe(struct platform_device *pdev)
239 {
240     struct snvs_rtc_data *data;
241     struct resource *res;
242     int ret;
243     void __iomem *mmio;
244
245     data = devm_kzalloc(&pdev->dev, sizeof(*data), GFP_KERNEL);
246     if (!data)
247         return -ENOMEM;
248
249     data->regmap =
250         syscon_regmap_lookup_by_phandle(pdev->dev.of_node, "regmap");
251
252     if (IS_ERR(data->regmap)) {
253         dev_warn(&pdev->dev, "snvs rtc: you use old dts file,
            please update it\n");
254         res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
255         // 从设备树中获取到 RTC 外设寄存器基地址
256         mmio = devm_ioremap_resource(&pdev->dev, res);
257         if (IS_ERR(mmio))
258             return PTR_ERR(mmio); // 完成内存映射, 得到 RTC 外设寄存器物理基地址
259         // 对应的虚拟地址
260         data->regmap = devm_regmap_init_mmio(&pdev->dev, mmio,
            &snvs_rtc_config); // 读写 RTC 底层硬件寄存器
261     } else {
262         data->offset = SNVS_LPREGISTER_OFFSET;
263         of_property_read_u32(pdev->dev.of_node, "offset",
            &data->offset);
264     }
265
266     if (!data->regmap) {
267         dev_err(&pdev->dev, "Can't find snvs syscon\n");
268         return -ENODEV;
269     }
270
271     data->irq = platform_get_irq(pdev, 0); // 从设备树中获取 RTC 的中断号
272     if (data->irq < 0)
273         return data->irq;
274
275     platform_set_drvdata(pdev, data);
276
277     /* Initialize glitch detect */
278     regmap_write(data->regmap, data->offset + SNVS_LPPGDR,
        SNVS_LPPGDR_INIT); // 对寄存器进行写操作
279
280     /* Clear interrupt status */
281     regmap_write(data->regmap, data->offset + SNVS_LPSR,
        0xffffffff);
282
283     /* Enable RTC */
284     snvs_rtc_enable(data, true); // 使能 RTC
285
286     device_init_wakeup(&pdev->dev, true);
287
288     ret = devm_request_irq(&pdev->dev, data->irq, snvs_rtc_irq_handler,
        IRQF_SHARED, "rtc alarm", &pdev->dev); // 请求 RTC 中断
289
290     if (ret) {
291         dev_err(&pdev->dev, "failed to request irq %d: %d\n",
            data->irq, ret);
292         goto error_rtc_device_register;
293     }
294
295     data->rtc = devm_rtc_device_register(&pdev->dev, pdev->name,
        &snvs_rtc_ops, THIS_MODULE); // 向系统注册 rtc_device
296
297     if (IS_ERR(data->rtc)) {
298         ret = PTR_ERR(data->rtc);
299         dev_err(&pdev->dev, "failed to register rtc: %d\n", ret);
300         goto error_rtc_device_register;
301     }
302
303     return 0;
304
305 error_rtc_device_register:
306     if (data->clk)
307         clk_disable_unprepare(data->clk);
308
309     return ret;
310 }
```

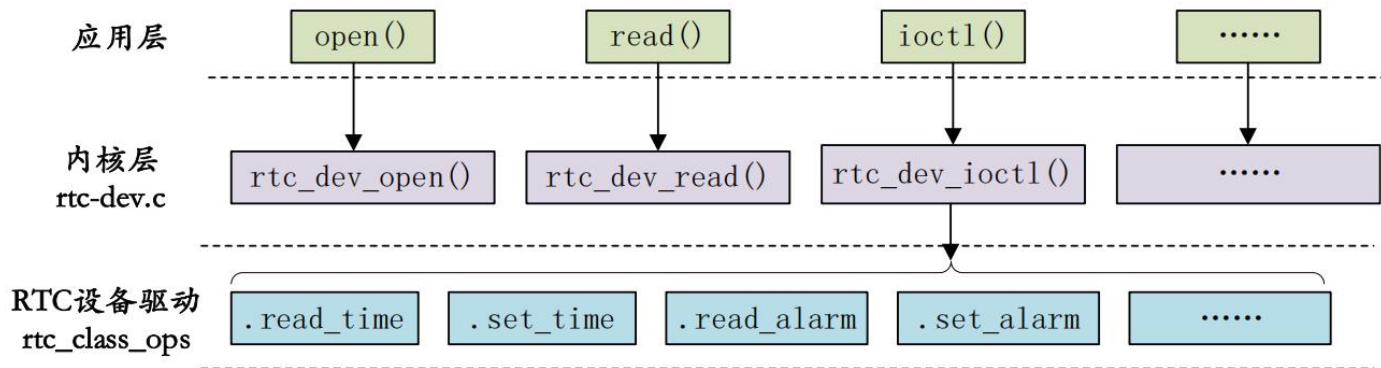


图 60.1.1 Linux RTC 驱动调用流程

操作方法

查看时间:

Date

结果: Thu Jan 1 01:23:05 UTC 1970

设置时间:

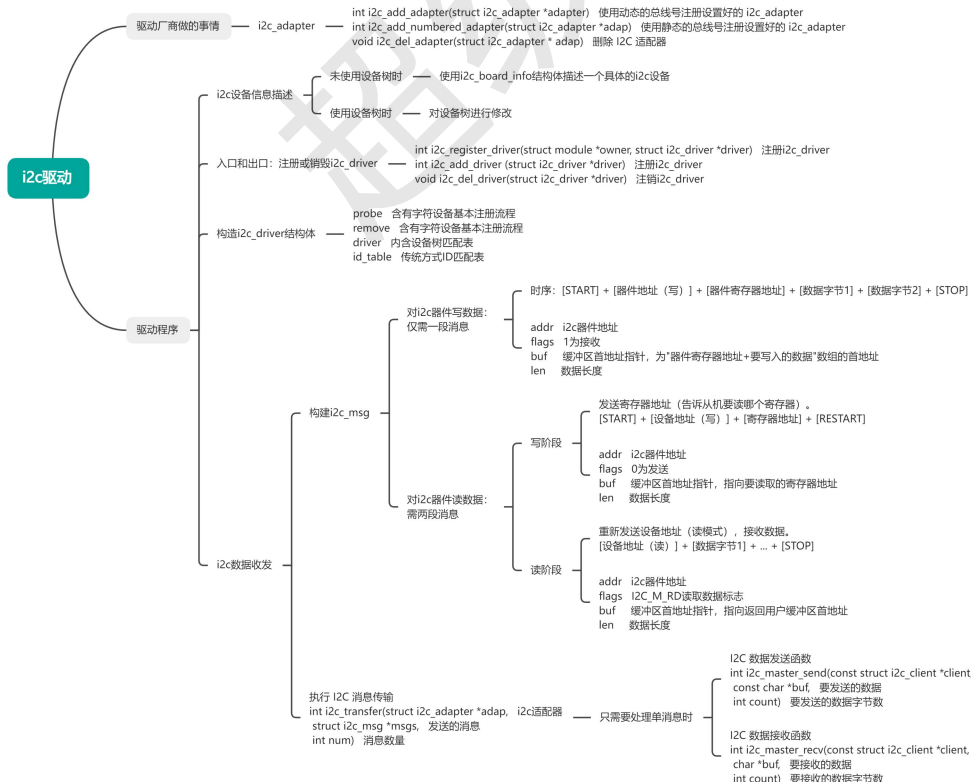
date -s "2025-09-28 18:11:00"

把时间写入 rtc 芯片:

hwclock -w

I2C 驱动

驱动编写方法



驱动源码分析

```

include/linux/i2c.h
// 示例代码为 i2c_12c_adapter 结构体
490 struct i2c_adapter {
491     struct module *owner;
492     unsigned int class; /* classes to allow probing for */
501     const struct i2c_algorithm *algo; /* 总线访问算法 */
502     void *algo_data;
503
504     /* data fields that are valid for all devices */
505     struct rt_mutex bus_lock;
506
507     int timeout; /* in jiffies */
508     int retries;
509     struct device dev; /* the adapter device */
510
511     int nr;
512     char name[40];
513     struct completion dev_released;
514
515     struct mutex userspace_clients_lock;
516     struct list_head userspace_clients;
517
518     struct i2c_bus_recovery_info *bus_recovery_info;
519     const struct i2c_adapter_quirks *quirks;
520 };

include/linux/i2c.h
// 示例代码为 i2c_12c_algorithm 结构体
391 struct i2c_algorithm {
392
393     int (*master_xfer)(struct i2c_adapter *adap, i2c 适配器的传输函数
394
395         int num); // SMBUS 总线的传输函数
396
397     int (*ambus_xfer)(struct i2c_adapter *adap, u16 addr,
398         unsigned short flags, char read_write,
399         u8 command, int size, union i2c_smbus_data *data);
400
401     /* To determine what the adapter supports */
402     u32 (*functionality) (struct i2c_adapter *);
403
404     .....
411 };

include/linux/i2c.h
// 示例代码为 i2c_client 结构体
517 struct i2c_client {
518     unsigned short flags; /* 标志 */
519     unsigned short addr; /* 芯片地址，7位，存放在低7位中 */
520     .....
521     char name[I2C_NAME_SIZE]; /* 名字 */
522     struct i2c_adapter *adapter; /* 对应的I2C适配器 */
523     struct device dev; /* 设备结构体 */
524     int irq; /* 中断 */
525     struct list_head detected;
526     .....
530 };

include/linux/i2c.h
// 示例代码为 i2c_driver 结构体
561 struct i2c_driver {
562     unsigned int class;
563
564     /* Notifies the driver that a new bus has appeared. You should
565      * avoid using this, it will be removed in a near future.
566      */
567     int (*attach_adapter)(struct i2c_adapter *) __deprecated;
568
569     /* Standard driver model interfaces */
570     int (*probe)(struct i2c_client *, const struct i2c_device_id *);
571     int (*remove)(struct i2c_client *);
572
573     /* Driver model interfaces that don't relate to enumeration */
574     void (*shutdown)(struct i2c_client *);
575
576     /* Alert callback, for example for the SMBus alert protocol.
577      * The format and meaning of the data value depends on the
578      * protocol. For the SMBus alert protocol, there is a single bit
579      * of data passed as the alert response's low bit ("event
580       flag").
581      */
582     void (*alert)(struct i2c_client *, unsigned int data);
583
584     /* A ioctl like command that can be used to perform specific
585      * functions with the device.
586      */
587     int (*command)(struct i2c_client *client, unsigned int cmd,
588         void *arg);
589
590     struct device_driver driver; // device_driver 内有of_match_table成员变量
591     const struct i2c_device_id *id_table; // 未使用设备树的设备匹配ID表
592
593     /* Device detection callback for automatic device creation */
594     int (*detect)(struct i2c_client *, struct i2c_board_info *);
595     const unsigned short *address_list;
596     struct list_head clients;
597 };

```

```

736 struct bus_type i2c_bus_type = {
737     .name = "i2c/*",
738     .match = i2c_device_match,
739     .probe = i2c_device_probe,
740     .remove = i2c_device_remove,
741     .shutdown = i2c_device_shutdown,
742 };

```

```

437 static int i2c_device_match(struct device *dev, struct
438     device_driver *drv)
439 {
440     struct i2c_client *client = i2c_verify_client(dev);
441     struct i2c_driver *driver;
442
443     if (!client)
444         return 0;
445
446     /* Attempt an OF style match */
447     if (of_device_match(dev, drv)) 设备树设备和驱动匹配
448         return 1;
449
450     /* Then ACPI style match */
451     if (acpi_device_match(dev, drv)) ACPI 形式的匹配
452         return 1;
453
454     driver = to_i2c_driver(drv);
455     /* match on an id table if there is one */
456     if (driver->id_table) 无设备树的 i2c 设备和驱动匹配过程
457         return i2c_match_id(driver->id_table, client) != NULL;
458
459     return 0;
460 }

```

12.3适配驱动分析

示例代码21.1 DCI 控制寄存器

```
1201: 120201a00000 {
2 #address-cells = <1>;
3 #size-cells = <0>;
4 compatible = "fsl,imx21-dci", "fsl,imx21-l20";
5 reg = <0x02010000 0x4000>;
6 interrupts = <GIC_SPI 36 IRQ_TYPE_LEVEL_HIGH>;
7 clocks = <clks IMX6Q_CLK_I2C1>;
8 status = "disabled";
9 };
```

示例代码21.2 DCI 支持代码段

```
244 static struct platform_device_id imx_i2c_devtype[] = {
245 {
246 .name = "imx1-i2c",
247 .driver_data = (kernel_ulong_t)&imx1_i2c_hdata,
248 }, {
249 .name = "imx2-i2c",
250 .driver_data = (kernel_ulong_t)&imx2l_i2c_hdata,
251 }, {
252 /* sentinel */
253 };
254 };
255 MODULE_DEVICE_TABLE(platform, imx_i2c_devtype);
256
257 static const struct of_device_id i2c_imx_of_id[] = {
258 { .compatible = "fsl,imx21-i2c", .data = &imx1_i2c_hdata, },
259 { .compatible = "fsl,imx21-i2c", .data = &imx2l_i2c_hdata, },
260 { .compatible = "fsl,vf610-i2c", .data = &vf610_i2c_hdata, },
261 { /* sentinel */ }
262 };
263 MODULE_DEVICE_TABLE(of, i2c_imx_of_id);
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

示例代码21.3 DCI 适配代码段

```
static int i2c_probe(struct platform_device *pdev)
{
    const struct of_device_id *of_id =
        match_device(i2c_imx_of_id, &pdev->dev);
    struct i2c_adapter *i2c_adapter;
    struct resource *res;
    struct imx12c_platform_data *pdata =
        dev_get_platdata(&pdev->dev);
    void __iomem *base;
    int irq, ret;
    dma_addr_t phy_addr;
    dev_dbg(&pdev->dev, "<la>la", __func__);
    .....
    irq = platform_get_irq(pdev, 0); 获取中断号
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    base = devm_ioremap_resource(&pdev->dev, res);
    if (IS_ERR(base))
        return PTR_ERR(base);
    phy_addr = (dma_addr_t)res->start;
    i2c_imx = devm_kalloc(&pdev->dev, sizeof(*i2c_imx),
        GFP_KERNEL); 申请内存
    if (!i2c_imx)
        return -ENOMEM;
    if (of_id)
        i2c_imx->hdata = of_id->data;
    else
        i2c_imx->hdata = (struct imx12c_hdata *)
            platform_get_device_id(pdev->driver_data);
    /* Setup i2c_imx driver structure */
    strscpy(i2c_imx->adapter.name, pdev->name,
        sizeof(i2c_imx->adapter.name));
    i2c_imx-&
```

```

// 示例代码 61.24 i2c_imx_algo 结构体
static struct i2c_algorithm i2c_imx_algo = {
    .master_xfer    = i2c_imx_xfer,
    .functionality   = i2c_imx_func,
};

// 示例代码 61.25 i2c_imx_func 函数
static u32 i2c_imx_func(struct i2c_adapter *adapter)
{
    return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL
        | I2C_FUNC_SMBUS_READ_BLOCK_DATA;
}

// 示例代码 61.26 i2c_imx_xfer 函数
static int i2c_imx_xfer(struct i2c_adapter *adapter,
    struct i2c_msg *msgs, int num)
{
    unsigned int i, temp;
    int result;
    bool is_lastmsg = false;
    struct imx_i2c_struct *i2c_imx = i2c_get_adapdata(adapter);

    dev_dbg(&i2c_imx->adapter.dev, "<ts>\n", __func__);

    /* Start I2C transfer */
    result = i2c_imx_start(i2c_imx);  // 开启 I2C 通信
    if (result)
        goto fail0;

    /* read/write data */
    for (i = 0; i < num; i++) {
        if (i == num - 1)
            is_lastmsg = true;

        if (i) {
            dev_dbg(&i2c_imx->adapter.dev,
                "<ts> repeated start\n", __func__);
            temp = imx_i2c_read_reg(i2c_imx, IMX_I2C_I2CR);
            temp |= I2CR_RSTA;
            imx_i2c_write_reg(temp, i2c_imx, IMX_I2C_I2CR);
            result = i2c_imx_bus_busy(i2c_imx, i);
            if (result)
                goto fail0;
        }

        dev_dbg(&i2c_imx->adapter.dev,
            "<ts> transfer message: %d\n", __func__, i);
        /* write/read data */
        if (msgs[i].flags & I2C_M_RD)
            result = i2c_imx_read(i2c_imx, &msgs[i], is_lastmsg);
        else {
            // 从 I2C 设备读取数据
            if (i2c_imx->dma && msgs[i].len >= DMA_THRESHOLD)  // 向 I2C 设备
                                                                    // 写数据
                result = i2c_imx_dma_write(i2c_imx, &msgs[i]);
            else
                result = i2c_imx_write(i2c_imx, &msgs[i]);
        }

        if (result)
            goto fail0;
    }

fail0:
    /* Stop I2C transfer */
    i2c_imx_stop(i2c_imx);  // I2C 通信完成以后调用 i2c_imx_stop 函数停止 I2C 通信

    dev_dbg(&i2c_imx->adapter.dev, "<ts> exit with: %s: %d\n",
        __func__,
        (result < 0) ? "error" : "success" msg,
        (result < 0) ? result : num);
    return (result < 0) ? result : num;
}

```

spi 驱动

驱动源码分析

SPI 主机驱动分析

示例代码 6.2.2.1 imx6ull.dtsi 文件中的 ecspi3 节点内容

```
1 ecspi3: ecspi@02010000 {
2     #address-cells = <1>;
3     #size-cells = <0>;
4     compatible = "fsl,imx6ul-ecspi", "fsl,imx51-ecspi";
5     reg = <0x02010000 0x4000>;
6     interrupts = <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>;
7     clocks = <clks IMX6UL_CLK_ECSP1>;
8     <clks IMX6UL_CLK_ECSP1>;
9     clock-names = "ipg", "per";
10    dmas = <dma 7 7 1>, <dma 8 7 2>;
11    dma-names = "rx", "tx";
12    status = "disabled";
13};
```

1.SPI硬件控制器检测

示例代码 6.2.2 spi_imx_driver 结构体

```
694 static struct platform_device_id spi_imx_devtype[] = {
695     {
696         .name = "imx1-cspi",
697         .driver_data = (kernel_ulong_t) &imx1_cspi_devtype_data,
698     }, {
699         .name = "imx21-cspi",
700         .driver_data = (kernel_ulong_t) &imx21_cspi_devtype_data,
701     }, {
702         .name = "imx6ul-ecspi", SPI 无设备树匹配表
703         .driver_data = (kernel_ulong_t) &imx6ul_ecspi_devtype_data,
704     }, {
705         /* sentinel */
706     },
707 };
708 static const struct of_device_id spi_imx_dt_ids[] = { SPI 设备树匹配表
709     { .compatible = "fsl,imx1-cspi", .data =
710         &imx1_cspi_devtype_data, },
711     { .compatible = "fsl,imx6ul-ecspi", .data =
712         &imx6ul_ecspi_devtype_data, },
713     { /* sentinel */ }
714 };
715 MODULE_DEVICE_TABLE(of, spi_imx_dt_ids);
716
717 static struct platform_driver spi_imx_driver = {
718     .driver = {
719         .name = DRIVER_NAME,
720         .of_match_table = spi_imx_dt_ids,
721         .pm = IMX_SPI_PM,
722     },
723     .id_table = spi_imx_devtype,
724     .probe = spi_imx_probe,
725     .remove = spi_imx_remove,
726 };
727 module_platform_driver(spi_imx_driver);
```

2.用户SPI驱动和SPI外设的匹配

spi_imx_probe 函数

```
// spi_imx_probe 函数
static int spi_imx_probe(struct platform_device *pdev)
{
    // 1. 创建 spi_master
    master = spi_alloc_master(&pdev->dev, sizeof(struct spi_imx_data));
    if (!master)
        return -ENOMEM;

    // 2. 设置 master 的操作函数
    master->bits_per_word_mask = SPI_BPMW_MASK(1, 10);
    master->transfer_one_message = spi_imx_transfer_one_message;
    master->prepare_message = spi_imx_prepare_message;
    master->unprepare_message = spi_imx_unprepare_message;
    master->mode_bits = SPI_CS0 | SPI_CS1 | SPI_CS_HDM;

    // 3. 【关键】关联到 spi_bus_type
    master->dev.bus = &spi_bus_type; // 这里建立关联！

    // 4. 注册到系统
    ret = spi_register_master(master);
    if (ret)
        return ret;

    // spi_imx_probe 函数完成从设备树中匹配到的设备驱动，
    // 返回设备树中的设备名称，
    // 返回设备树中的设备名称，
    // 返回设备树中的设备名称
```

SPI 设备和驱动匹配过程

示例代码 6.2.1.1 spi_master 结构体

```
include/linux/spi/cspi.h
315 struct spi_master { SPI 控制器硬件
316     struct device dev;
317     struct list_head list;
318     struct list_head list;
319     .....
320     s16 bus_num;
321     .....
322     /* chipselects will be integral to many controllers; some others
323     * might use board-specific GPIOs.
324     */
325     u16 num_chipselect;
326     .....
327     /* some SPI controllers pose alignment requirements on DMAable
328     * buffers; let protocol drivers know about these requirements.
329     */
330     u16 dma_alignment;
331     .....
332     /* spi_device.mode flags understood by this controller driver */
333     u16 mode_bits;
334     /* bitmask of supported bits_per_word for transfers */
335     u32 bits_per_word_mask;
336     .....
337     /* limits on transfer speed */
338     u32 min_speed_hz;
339     u32 max_speed_hz;
340     .....
341     /* other constraints relevant to this driver */
342     u16 flags;
343     .....
344     /* lock and mutex for SPI bus locking */
345     spinlock_t bus_lock_spinlock;
346     struct mutex bus_lock_mutex;
347     .....
348     /* flag indicating that the SPI bus is locked for exclusive use
349     */
350     bool bus_lock_flag;
351     .....
352     int (*setup)(struct spi_device *spi);
353     .....
354     int (*transfer)(struct spi_device *spi,
355                     struct spi_message *msg);
356     .....
357     int (*transfer_one_message)(struct spi_device *master,
358                                struct spi_message *msg); // 用于发送一个 spi_message
359     .....
360 };
```

SPI 设备驱动

示例代码 6.2.2.3 spi_imx_buf_tx_u8 函数

```
152 #define MXC_SPI_BUF_TX(type) \
153 static void spi_imx_buf_tx_##type(struct spi_imx_data *spi_imx) \
154 { \
155     type val = 0; \
156     \
157     if (spi_imx->tx_buf) { \
158         val = *(type *)spi_imx->tx_buf; \
159         spi_imx->tx_buf += sizeof(type); \
160     } \
161     \
162     spi_imx->count -= sizeof(type); \
163     \
164     writel(val, spi_imx->base + MXC_CSPTXDATA); \
165 } // 将要发送的数据值写入到 ECSP1 的 TXDATA 寄存器里面去
166 \
167 MXC_SPI_BUF_RX(u8) \
168 MXC_SPI_BUF_TX(u8)
```

SPI 设备驱动

示例代码 6.2.1.2 spi_driver 结构体

```
180 struct spi_driver {
181     const struct spi_device_id *id_table;
182     int (*probe)(struct spi_device *spi);
183     int (*remove)(struct spi_device *spi);
184     void (*shutdown)(struct spi_device *spi);
185     struct device_driver driver;
186 };
```

SPI 设备驱动

示例代码 6.2.1.3 spi_match_device 函数

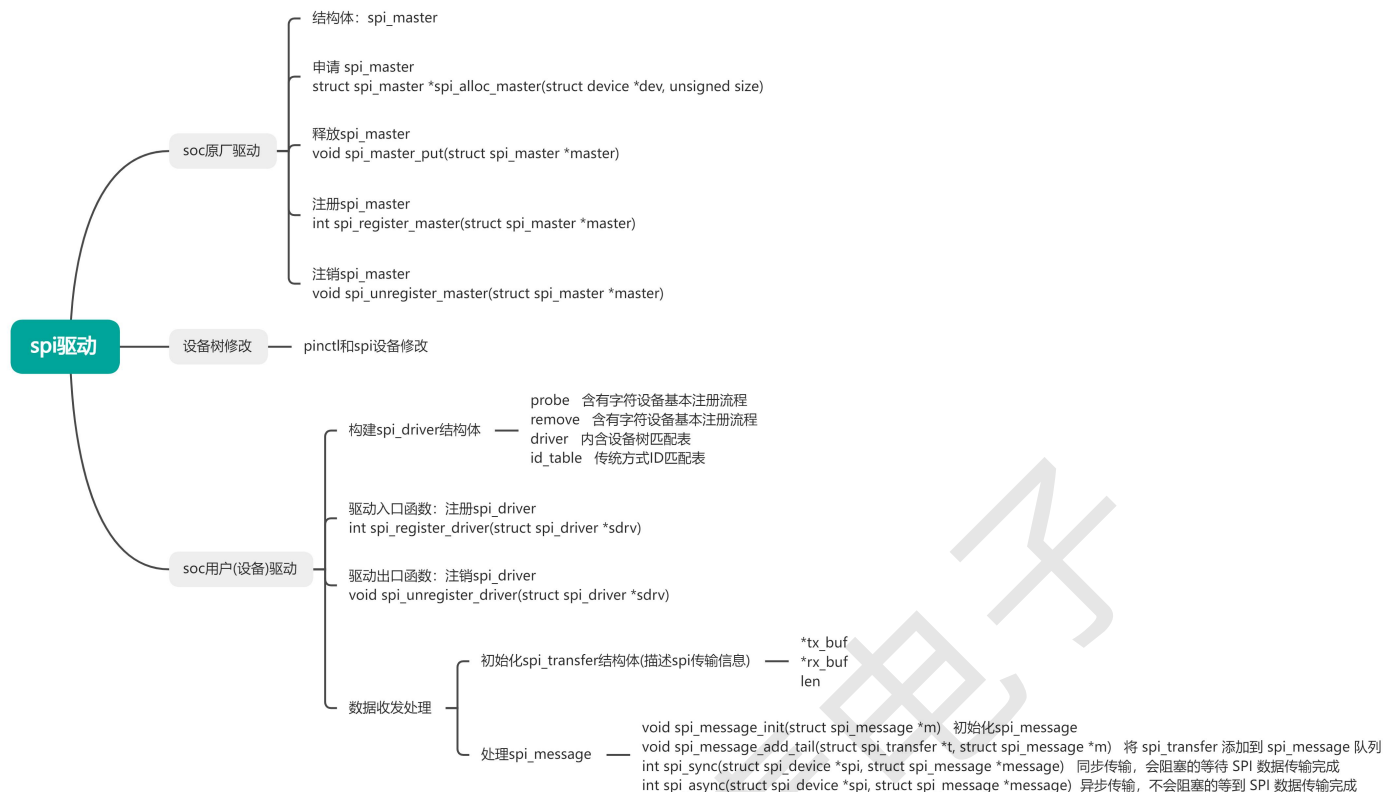
```
99 static int spi_match_device(struct device *dev,
100                             struct device_driver *drv)
101 {
102     const struct spi_device *spi = to_spi_device(dev);
103     const struct spi_driver *sdrv = to_spi_driver(drv);
104     .....
105     /* Attempt an OF style match */
106     if (of_driver_match_device(dev, drv)) // 完成设备树设备和驱动匹配
107         return 1;
108     .....
109     /* Then try ACPI */
110     if (acpi_driver_match_device(dev, drv)) // 用于 ACPI 形式的匹配
111         return 1;
112     if (sdrv->id_table)
113         return !spi_match_id(sdrv->id_table, spi);
114     // 用于传统的、无设备树的 SPI 设备和驱动匹配过程
115     return strcmp(spi->modalias, drv->name) == 0;
116 }
```

SPI 设备驱动

```
graph TD
    A[设备树节点] --> C[平台驱动匹配]
    B[硬件控制器驱动] --> C
    C --> D[注册spi_master]
    D --> E[SPI总线枚举]
    D --> F[设备协议驱动]
    E --> G[SPI设备驱动匹配]
    F --> G
    G --> H[具体设备功能]
```

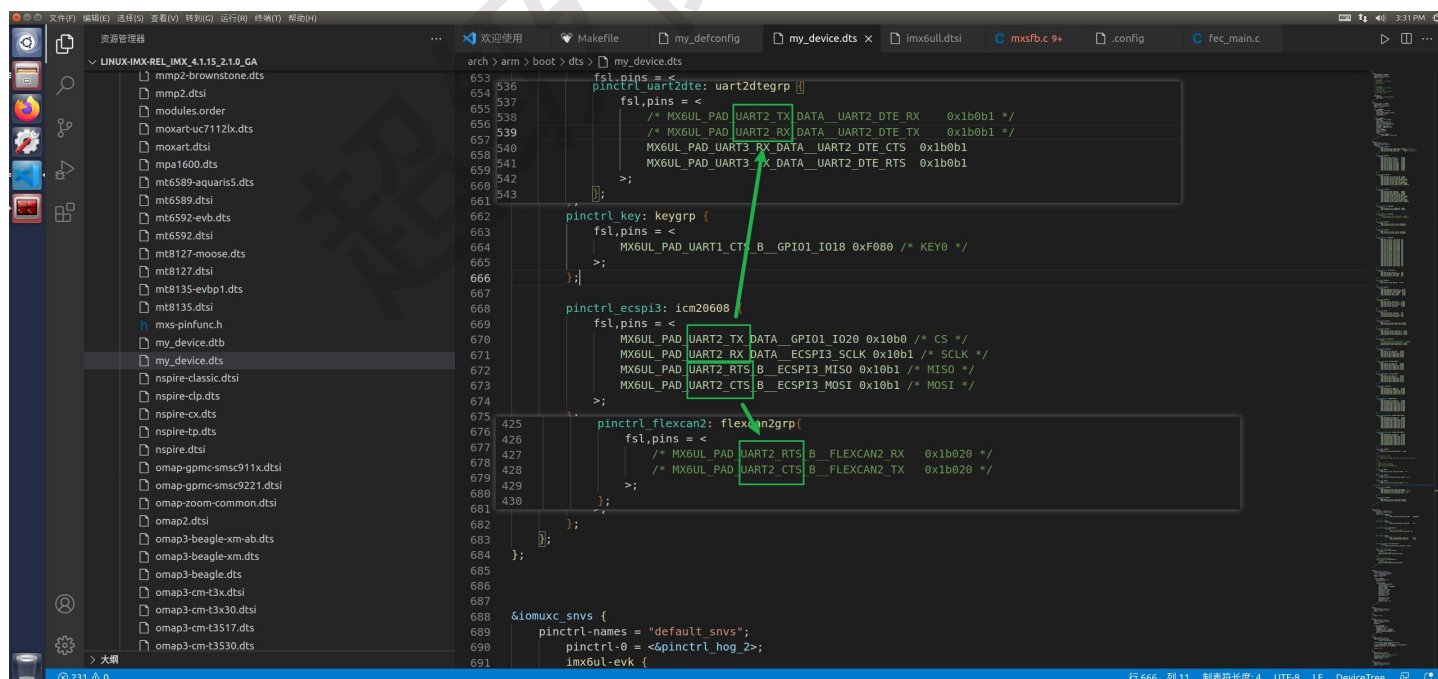
The flowchart illustrates the SPI driver initialization process. It starts with '设备树节点' (Device Tree Node) and '硬件控制器驱动' (Hardware Controller Driver) both pointing to '平台驱动匹配' (Platform Driver Matching). From '平台驱动匹配', the process flows to '注册spi_master' (Register spi_master). This step then branches into 'SPI总线枚举' (SPI Bus Enumeration) and '设备协议驱动' (Device Protocol Driver). Both of these lead to 'SPI设备驱动匹配' (SPI Device Driver Matching), which finally leads to '具体设备功能' (Specific Device Function).

驱动代码编写方法



其他

注意在设备树中的引脚冲突问题，否则新设备无法添加成功



Uart 驱动

驱动源码分析

```
1 uart3: serial@021ec000 {
2     compatible = "fsl,imx6ul-uart",
3       "fsl,imx6q-uart", "fsl,imx21-uart";
4     reg = <0x021ec000 0x4000>;
5     interrupts = <GIC_SPI 28 IRQ_TYPE_LEVEL_HIGH>;
6     clocks = <&clks IMX6UL_CLK_UART3_IPG>;
7     <&clks IMX6UL_CLK_UART3_SERIAL>;
8     clock-names = "ipg", "per";
9     dmas = <&dma 29 4 0>, <&dma 30 4 0>;
10    dma-names = "rx", "tx";
11    status = "disabled";
12 };
```

```
267 static struct platform_device_id imx_uart_devtype[] = {
268     {
269         .name = "imx1-uart",
270         .driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX1_UART],
271     }, {
272         .name = "imx21-uart",
273         .driver_data = (kernel_ulong_t)
274             &imx_uart_devdata[IMX21_UART],
275     }, {
276         .name = "imx6q-uart",
277         .driver_data = (kernel_ulong_t)
278             &imx_uart_devdata[IMX6Q_UART],
279     }, {
280         /* sentinel */
281     }
282 };
283 MODULE_DEVICE_TABLE(platform, imx_uart_devtype);
284
285 static const struct of_device_id imx_uart_dt_ids[] = {
286     { .compatible = "fsl,imx6q-uart", .data =
287         &imx_uart_devdata[IMX6Q_UART], },
288     { .compatible = "fsl,imx21-uart", .data =
289         &imx_uart_devdata[IMX21_UART], },
290     { .compatible = "fsl,imx1-uart", .data =
291         &imx_uart_devdata[IMX1_UART], },
292     { /* sentinel */ }
293 };
294
295 static struct platform_driver serial_imx_driver = {
296     .probe = serial_imx_probe,
297     .remove = serial_imx_remove,
298
299     .suspend = serial_imx_suspend,
300     .resume = serial_imx_resume,
301     .id_table = imx_uart_devtype,
302     .driver = {
303         .name = "imx-uart",
304         .of_match_table = imx_uart_dt_ids,
305     },
306 };
307
308 static int __init imx_serial_init(void)
309 {
310     int ret = uart_register_driver(&serial_imx_driver);
311     if (ret)
312         return ret;
313
314     ret = platform_driver_register(&serial_imx_driver);
315     if (ret != 0)
316         uart_unregister_driver(&serial_imx_driver);
317
318     return ret;
319 }
320
321 static void __exit imx_serial_exit(void)
322 {
323     platform_driver_unregister(&serial_imx_driver);
324     uart_unregister_driver(&serial_imx_driver);
325 }
326
327 module_init(imx_serial_init);
328 module_exit(imx_serial_exit);
```

```
295 struct uart_driver
296 {
297     struct module *owner; /* 模块所有者 */
298     const char *driver_name; /* 驱动名字 */
299     const char *dev_name; /* 设备名字 */
300     int major; /* 主设备号 */
301     int minor; /* 次设备号 */
302     int nr; /* 设备数 */
303     struct console *cons; /* 控制台 */
304
305     /*
306      * these are private; the low level driver should not
307      * touch these; they should be initialised to NULL
308      */
309     struct uart_state *state;
310     struct tty_driver *tty_driver;
311 };
312
```

```
196 static int serial_imx_probe(struct platform_device *pdev)
197 {
198     struct imx_port *sport; 定义一个 imx_port 类型的结构体指针变量 sport
199     void __iomem *base;
200     int ret = 0;
201     struct resource *res;
202     int txirq, rxirq, rtsirq;
203
204     sport = devm_kzalloc(&pdev->dev, sizeof(*sport), GFP_KERNEL);
205     if (!sport)
206         return -ENOMEM;
207
208     ret = serial_imx_probe_dt(sport, pdev);
209     if (ret > 0)
210         serial_imx_probe_pdata(sport, pdev);
211     else if (ret < 0)
212         return ret;
213
214     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
215     base = devm_ioremap_resource(&pdev->dev, res);
216     if (IS_ERR(base))
217         return PTR_ERR(base);
218
219     rxirq = platform_get_irq(pdev, 0);
220     txirq = platform_get_irq(pdev, 1);
221     rtsirq = platform_get_irq(pdev, 2);
222
223     sport->port.dev = &pdev->dev;
224     sport->port.mapbase = res->start;
225     sport->port.membase = base;
226     sport->port.type = PORT_IMX;
227     sport->port.iotype = UPIO_MEM;
228     sport->port.irq = rxirq;
229     sport->port.fifosize = 32;
230     sport->port.ops = &imx_pops;
231     sport->port.rs485_config = imx_rs485_config;
232     sport->port.rs485_flags = 0;
233     SER_RS485_RTS_ON_SEND | SER_RS485_RX_DURING_TX;
234     sport->port.flags = UPP_BOOT_AUTOCONF;
235     init_timer(&sport->timer);
236     sport->timer.function = imx_timeout;
237     sport->timer.data = (unsigned long)sport;
238
239     sport->clk_ipg = devm_clk_get(&pdev->dev, "ipg");
240     if (IS_ERR(sport->clk_ipg)) {
241         ret = PTR_ERR(sport->clk_ipg);
242         dev_err(&pdev->dev, "failed to get ipg clk: %d\n", ret);
243         return ret;
244     }
245
246     sport->clk_per = devm_clk_get(&pdev->dev, "per");
247     if (IS_ERR(sport->clk_per)) {
248         ret = PTR_ERR(sport->clk_per);
249         dev_err(&pdev->dev, "failed to get per clk: %d\n", ret);
250         return ret;
251     }
252
253     sport->uartclk = clk_get_rate(sport->clk_per);
254     if (sport->uartclk > IMX_MODULE_MAX_CLK_RATE) {
255         ret = clk_set_rate(sport->clk_per, IMX_MODULE_MAX_CLK_RATE);
256         if (ret < 0) {
257             dev_err(&pdev->dev, "clk_set_rate() failed\n");
258             return ret;
259         }
260     }
261
262     sport->uartclk = clk_get_rate(sport->clk_per);
263
264     /*
265      * Allocate the IRQ(s) i.MX1 has three interrupts whereas later
266      * chips only have one interrupt.
267      */
268     if (txirq > 0) {
269         ret = devm_request_irq(&pdev->dev, rxirq, imx_rxint, 0,
270             dev_name(&pdev->dev), sport);
271         if (ret)
272             return ret;
273
274         ret = devm_request_irq(&pdev->dev, txirq, imx_txint, 0,
275             dev_name(&pdev->dev), sport);
276         if (ret)
277             return ret;
278     } else {
279         ret = devm_request_irq(&pdev->dev, rxirq, imx_int, 0,
280             dev_name(&pdev->dev), sport);
281         if (ret)
282             return ret;
283     }
284
285     imx_ports[sport->port.line] = sport;
286
287     platform_set_drvdata(pdev, sport);
288
289     return uart_add_one_port(&imx_reg, &sport->port);
290 }
```

```
117 struct uart_port {
118     spinlock_t lock; /* port lock */
119     unsigned long iobase; /* in/out[bwl] */
120     unsigned char __iomem *membase; /* read/write[bwl] */
121     .....
122
123     const struct uart_ops *ops; ops 包含了串口的具体驱动函数,
124     custom_divisor;
125     unsigned int line; /* port index */
126     unsigned int minor;
127     resource_size_t mapbase; /* for ioremap */
128     resource_size_t mapsize;
129     struct device *dev; /* parent device */
130     .....
131 };
132
```

```
49 struct uart_ops {
50     unsigned int (*tx_empty)(struct uart_port *);
51     void (*set_mctrl)(struct uart_port *, unsigned int mctrl);
52     unsigned int (*get_mctrl)(struct uart_port *);
53     void (*stop_tx)(struct uart_port *);
54     void (*start_tx)(struct uart_port *);
55     void (*throttle)(struct uart_port *);
56     void (*unthrottle)(struct uart_port *);
57     void (*send_xchar)(struct uart_port *, char ch);
58     void (*stop_rx)(struct uart_port *);
59     void (*enable_ms)(struct uart_port *);
60     void (*break_ctl)(struct uart_port *, int ctl);
61     int (*startup)(struct uart_port *);
62     void (*shutdown)(struct uart_port *);
63     void (*flush_buffer)(struct uart_port *);
64     void (*set_termios)(struct uart_port *, struct ktermios *new,
65         struct ktermios *old);
66     void (*set_ldisc)(struct uart_port *, struct ktermios *);
67     void (*pm)(struct uart_port *, unsigned int state,
68         unsigned int oldstate);
69
70     /*
71      * Return a string describing the type of the port
72      */
73     const char *(*type)(struct uart_port *);
74
75     /*
76      * Release IO and memory resources used by the port.
77      * This includes iounmap if necessary.
78      */
79     void (*release_port)(struct uart_port *);
80
81     /*
82      * Request IO and memory resources used by the port.
83      * This includes iomapping the port if necessary.
84      */
85     int (*request_port)(struct uart_port *);
86     void (*config_port)(struct uart_port *, int);
87     int (*verify_port)(struct uart_port *, struct serial_struct *);
88     int (*ioctl)(struct uart_port *, unsigned int, unsigned long);
89 #ifdef CONFIG_CONSOLE_POLL
90     int (*poll_init)(struct uart_port *);
91     void (*poll_put_char)(struct uart_port *, unsigned char);
92     int (*poll_get_char)(struct uart_port *);
93 #endif
94 };
95
```

```
1611 static struct uart_ops imx_pops = {
1612     .tx_empty = imx_tx_empty,
1613     .set_mctrl = imx_set_mctrl,
1614     .get_mctrl = imx_get_mctrl,
1615     .stop_tx = imx_stop_tx,
1616     .start_tx = imx_start_tx,
1617     .stop_rx = imx_stop_rx,
1618     .enable_ms = imx_enable_ms,
1619     .break_ctl = imx_break_ctl,
1620     .startup = imx_startup,
1621     .shutdown = imx_shutdown,
1622     .flush_buffer = imx_flush_buffer,
1623     .set_termios = imx_set_termios,
1624     .type = imx_type,
1625     .config_port = imx_config_port,
1626     .verify_port = imx_verify_port,
1627 #if defined(CONFIG_CONSOLE_POLL)
1628     .poll_init = imx_poll_init,
1629     .poll_get_char = imx_poll_get_char,
1630     .poll_put_char = imx_poll_put_char,
1631 #endif
1632 };
1633
```

platform 驱动框架
结构体
serial_imx_driver

为 sport 申请内存

从设备树中获取 IMX 系列 SOC UART 外设寄存器首地址

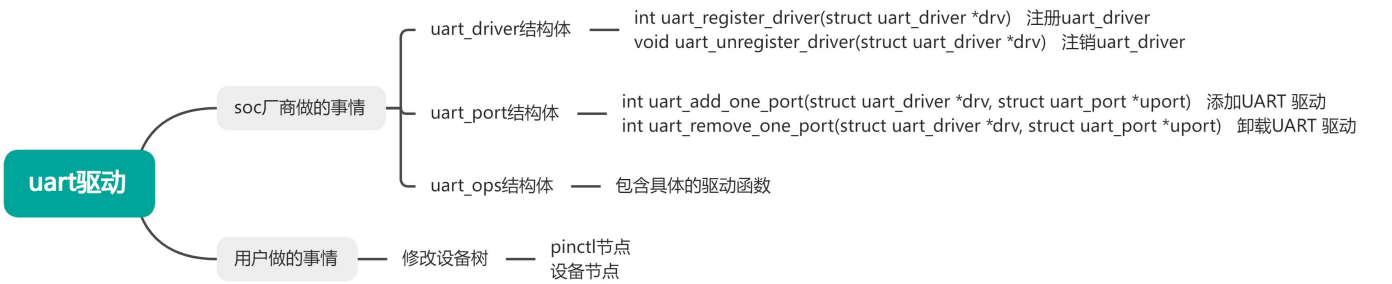
获取中断信息

示例代码 63.1.2 uart_port 结构体

示例代码 63.1.3 uart_ops 结构体

示例代码 63.2.6 imx_pops 结构体
IMX6ULL 串口最底层
的操作函数

驱动编写方法



触摸屏 mt 协议

驱动源码分析

typeA

示例代码 64.1.2.2 stl232_ts_irq_handler 函数代码段

```
100 static irqreturn_t stl232_ts_irq_handler(int irq, void *dev_id)
104 {
111     ret = stl232_ts_read_data(ts); 获取所有触摸点信息
112     if (ret < 0)
113         goto end;
114
115     /* multi touch protocol */
116     for (i = 0; i < MAX_FINGERS; i++) {
117         if (!finger[i].is_valid)
118             continue;
119
120         input_report_abs(input_dev, ABS_MT_TOUCH_MAJOR, finger[i].t);
121         input_report_abs(input_dev, ABS_MT_POSITION_X, finger[i].x);
122         input_report_abs(input_dev, ABS_MT_POSITION_Y, finger[i].y);
123         input_mt_sync(input_dev);
124         count++;
125     }
126
127     /* SYN_REPORT */
128     input_sync(input_dev);
129
130     return IRQ_HANDLED;
131 }
```

1 ABS_MT_POSITION_X x[0] 上报第一个触摸点的 X 坐标数据 通过input_report_abs 函数实现
2 ABS_MT_POSITION_Y y[0] 上报第一个触摸点的 Y 坐标数据
3 SYN_MT_REPORT
4 ABS_MT_POSITION_X x[1] 上报第二个触摸点的 X 坐标数据
5 ABS_MT_POSITION_Y y[1] 上报第二个触摸点的 Y 坐标数据
6 SYN_MT_REPORT
7 SYN_REPORT 通过调用 input_sync 函数来实现

typeB

示例代码 64.1.3.3 ili210x_report_events 函数代码段

```
75 static void ili210x_report_events(struct input_dev *input,
76                                 const struct touchdata *touchdata)
79 {
80     int i;
81     bool touch;
82     unsigned int x, y;
83     const struct finger *finger;
84
85     for (i = 0; i < MAX_TOUCHES; i++) {
86         input_mt_slot(input, i);
87
88         finger = &touchdata->finger[i];
89
90         touch = touchdata->status & (1 << i);
91         input_mt_report_slot_state(input, MT_TOOL_FINGER, touch);
92         if (touch) {
93             x = finger->x_low | (finger->x_high << 5);
94             y = finger->y_low | (finger->y_high << 5);
95
96             input_report_abs(input, ABS_MT_POSITION_X, x);
97             input_report_abs(input, ABS_MT_POSITION_Y, y);
98         }
99     }
100
101     input_mt_report_pointer_emulation(input, false);
102     input_sync(input);
103 }
```

1 ABS_MT_SLOT 0 上报标识槽位
2 ABS_MT_TRACKING_ID 45 上报唯一标识符 使用input_mt_report_slot_state函数
3 ABS_MT_POSITION_X x[0] 上报触摸点 0 的 X 轴坐标 使用函数 input_report_abs
4 ABS_MT_POSITION_Y y[0] 上报触摸点 0 的 Y 轴坐标 使用函数 input_report_abs
5 ABS_MT_SLOT 1
6 ABS_MT_TRACKING_ID 46
7 ABS_MT_POSITION_X x[1]
8 ABS_MT_POSITION_Y y[1]
9 SYN_REPORT 使用 input_sync函数来完成
示例代码 64.1.3.2 Type B 触摸点移除时序
1 ABS_MT_TRACKING_ID -1 触摸点被移除 使用 input_mt_report_slot_state函数完成
2 SYN_REPORT

驱动编写方法



碰到的问题

1.如何解决引脚冲突问题

```
/lib/modules/4.1.15 # modprobe ft5x06.ko
edt_ft5x06 1-0038: Failed to request GPIO 9, error -16
edt_ft5x06: probe of 1-0038 failed with error -16
```

可用下面方法：

```
mount -t debugfs none /sys/kernel/debug/
```

```
/lib/modules/4.1.15 # mount -t debugfs none /sys/kernel/debug/
```

这句话是 Linux 系统中挂载 debugfs 文件系统的命令，让我为你详细解释：

1. 命令分解

bash

mount -t debugfs none /sys/kernel/debug/

部分	含义
mount	Linux 挂载命令
-t debugfs	指定文件系统类型为 debugfs
none	表示没有实际的存储设备（虚拟文件系统）
/sys/kernel/debug/	挂载目标目录

```
/lib/modules/4.1.15 # cat /sys/kernel/debug/gpio | grep -E "gpio-9|GPIO1_9"
gpio-9 (VSD_3V3 ) out lo
```

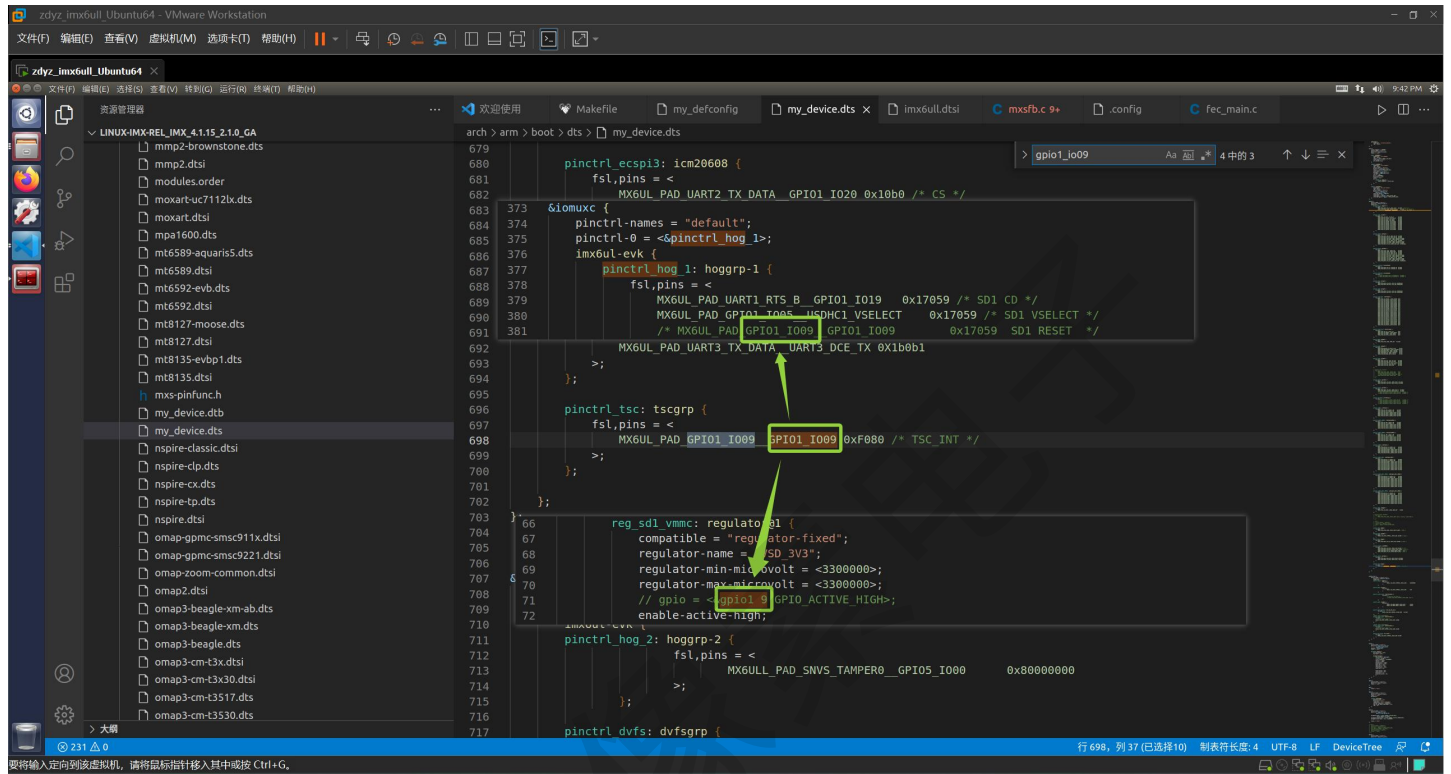
太好了！现在我们已经找到了问题所在。从调试信息可以看到：

复制

```
gpio-9 (VSD_3V3) out Lo
```

GPIO 9 被一个名为 `vsd_3v3` 的稳压器 (regulator) 占用，并且被配置为输出低电平 (out lo)。

在设备树中，同一个引脚可能会以不同形式出现，造成冲突，因此在解决引脚冲突问题时，需要找引脚的几种不同表现形式



2.编译设备树之后运行驱动，发现/dev/input/event2 无信息输出

```
/lib/modules/4.1.15 # cd /dev/input/  
/dev/input # ls  
event0 event1 event2 mice mouse0  
/dev/input # hexdump event2  
^C
```

- 1.开始认为设备树写错，于是照着教程文档再次检查设备树，检查过后再次编译设备树，发现还是不行
- 2.之后观看教程视频，主要删除了一些原本电阻触摸屏冗余的设备节点，更换了同级别部分代码的位置，还是不行
- 3.之后使用~/linux/IMX6ULL/linux/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/arch/arm/boot/dts/imx6ull-alientek-emmc.dts 代替原本的 my_device.dts，我认为这个是正点原子官方的，发现还是不行
- 4.我忽然发现 23_multitouch 文件夹下除了 ft5x06.c，还有 gt9147.c，发现同样的屏幕居然还可能有多多种触摸芯片，于是怀疑可能是触摸屏驱动芯片不兼容，问购买屏幕来源的闲鱼卖家，发现屏幕驱动是 gt911，但是闲鱼卖家说兼容这个驱动程序，可是网上和 gpt 查询，得到的答案是不兼容
- 5.于是我烧录了触摸屏对应的裸机例程 19_touchscreen 进行验证，发现可以正常使用。裸机例程和 Linux 驱动例程内容逻辑是一样的，所以我认为问题应该不在驱动，大概率在设备树那边。可是我已经试过了正点原子官方的设备树，也是不行，那问题出在哪里呢
- 6.继续看视频，视频在触摸中断的 probe 函数加入了 printk，视频里只要触摸屏幕，屏幕向 soc 发出触摸中断，probe 被激活，printk 会工作，可是我的 printk 不工作，于是打算检查是不是中断有问题
- 7.我决定查看 gpio 的电平，问 ai，答案是

```
echo 41 > /sys/class/gpio/export # 请求内核导出GPIO41的控制权  
echo in > /sys/class/gpio/gpio41/direction # 设置为输入模式  
cat /sys/class/gpio/gpio41/value # 读取当前电平值
```

可是我发现得到的电平数值是 0，即使我把屏幕拆下来，还是不行，可是这个中断引脚在开发板上是有配置上拉电阻的，如果拆掉屏幕还是 0，说明肯定代码有问题，于是我质疑 AI，得到了新的答案

```
# 测试GPIO1_I009 (应该是9, 不是41)
echo 9 > /sys/class/gpio/export
echo in > /sys/class/gpio/gpio9/direction
cat /sys/class/gpio/gpio9/value
```

这次我发现不管怎么样，即使按下或没按下触摸屏，结果都是 1，不知道怎么回事

8.第二天早上(2025.9.30)，我发现教程最后面指出了驱动适配的问题，尝试解决，成功解决问题

64.8 7 寸触摸 GT911 驱动

我们新的 7 寸触摸屏换用了 GT911 这颗触摸芯片，因此直接使用本章编写的针对 FT54X6 芯片的触摸驱动肯定会有问题。由于 GT9147、GT911 和 GT1151 这三颗触摸芯片几乎一模一样，所以我们直接编写了一个兼容这三款触摸 IC 的驱动程序，就是 [23_multitouch->gt9147.c](#)，所以使用新版本 7 寸屏的朋友，请使用 gt9147.c 这个触摸驱动。

中断引脚配置发生改变，我猜测可能是因为引脚配置和屏幕硬件芯片差异会导致之前的引脚读取问题

```
pinctrl_tsc: tscgrp {
    fsl,pins = <
        /* MX6UL_PAD_GPIO1_I009_GPIO1_I009 0xF080 */ TSC_INT for FT5426*/
        MX6UL_PAD_GPIO1_I009_GPIO1_I009 0x79 /* TSC_INT for GT911*/
    >;
};
```

核心区别

参数	0x79	0xF080
电气特性	较弱的驱动能力	较强的驱动能力
典型应用	低速信号、普通IO	中断引脚、高速信号
阻抗匹配	较高阻抗	较低阻抗
抗干扰能力	较弱	较强

好的，我来详细对比分析这两个触摸屏驱动程序的区别：

核心区别总结

特性	FT5x06 驱动	GT9147 驱动
芯片厂商	FocalTech	Goodix
通信协议	标准I2C	扩展I2C (16位地址)
中断处理	多点触控完整处理	简化单点处理 (注释说明有多点问题)
寄存器架构	8位地址	16位地址
初始化流程	相对简单	更复杂，需要读取固件信息
复位序列	简单高低电平	复杂序列 (包括软复位)
ID识别	通过设备树兼容性	通过读取芯片ID寄存器动态识别


```
/dev/input # hexdump event2
00000000 04c3 0000 3ae4 000c 0003 0039 0013 0000
00000010 04c3 0000 3ae4 000c 0003 0035 0358 0000
00000020 04c3 0000 3ae4 000c 0003 0036 01a8 0000
00000030 04c3 0000 3ae4 000c 0001 014a 0001 0000
00000040 04c3 0000 3ae4 000c 0003 0000 0358 0000
00000050 04c3 0000 3ae4 000c 0003 0001 01a8 0000
00000060 04c3 0000 3ae4 000c 0000 0000 0000 0000
00000070 04c3 0000 0e43 000d 0003 0039 ffff ffff
00000080 04c3 0000 0e43 000d 0001 014a 0000 0000
00000090 04c3 0000 0e43 000d 0000 0000 0000 0000
```

音频驱动

I2S 协议与硬件电路

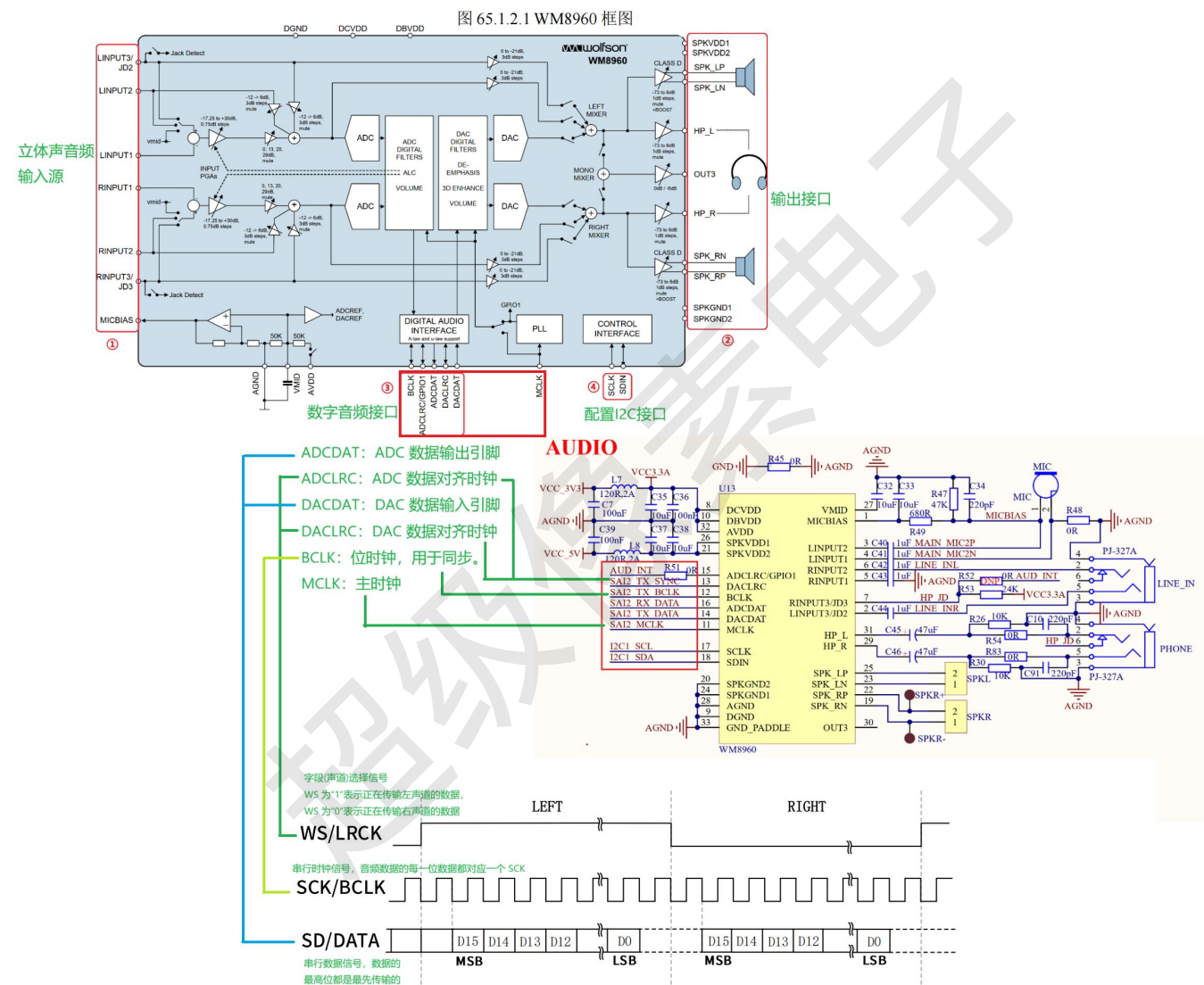
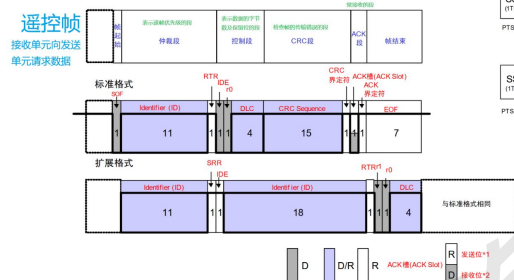
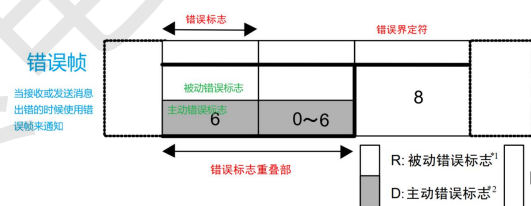
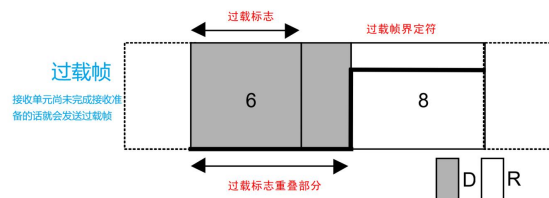
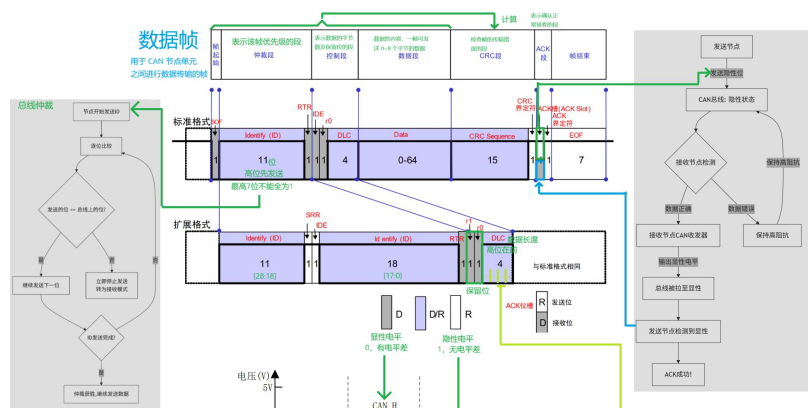


图 65.1.3.1 I2S 时序图

can 协议



USB 驱动

USB 协议

设备描述符用于描述 USB 设备的一般信息

编号	域	大小/位	数据类型	描述
0	M.length	1	数字	此设备描述符长度, 18 个字节
1	bDescriptorType	1	常量	描述符类型, 为 0x01
2	bcdUSB	2	BCD 码	USB 版本号
4	bDeviceClass	1	类	设备类
5	bDeviceSubClass	1	子类	设备子类
6	bDeviceProtocol	1	协议	设备协议
7	bMaxPacketSize0	1	数字	端点 0 的最大包长度
8	idVendor	2	ID	厂商 ID
10	idProduct	2	ID	产品 ID
12	bcdDevice	2	BCD 码	设备版本号
14	iManufacturer	1	字符串	厂商信息字符串描述符索引
15	iProduct	1	字符串	产品信息字符串描述符索引
16	iSerialNumber	1	字符串	产品信息字符串描述符索引
17	bNumConfigurations	1	数字	可能的配置描述符数量

定义了一个 USB 设备的配置描述符数量

编号	域	大小/位	数据类型	描述
0	M.length	1	数字	此配置描述符长度, 9 个字节
1	bDescriptorType	1	常量	配置描述符类型, 为 0x02
2	wTotalLength	2	数字	整个配置信息总长度(包括配置、接口、端点、设备类和“自定义”的描述符)
4	bNumInterfaces	1	数字	此配置所支持的接口数
5	bConfigurationValue	1	数字	此配置的一个设备支持多种配置, 通过配置值来区分不同的配置
6	iConfiguration	1	数字	描述此配置的字串描述符索引
7	bmAttributes	1	数字	该设备的属性: D0: 保留 D6: 自给电源 D7: 远程唤醒 D8-15: 保留
8	bMaxPower	1	数字	此配置下所需的总线电流(单位 2mA)

用于描述一些方便人们阅读的信息

编号	域	大小/位	数据类型	描述
0	M.length	1	N+2	此字符串描述符长度
1	bDescriptorType	1	常量	字符串描述符类型, 为 0x03
2	wLANGID[0]	2	数字	语言标识符 0
N	wLANGID[x]	2	数字	语言标识符 x

指定了该配置下的接口数量

编号	域	大小/位	数据类型	描述
0	M.length	1	数字	此接口描述符长度, 9 个字节
1	bDescriptorType	1	常量	描述符类型, 为 0x04
2	bInterfaceNumber	1	数字	当前接口编号, 从 0 开始
3	bAlternateSetting	1	数字	当前接口备用编号
4	bNumEndpoints	1	数字	当前接口的端点数量
5	bInterfaceClass	1	类	当前接口所属的类
6	bInterfaceSubClass	1	子类	当前接口所属的子类
7	bInterfaceProtocol	1	协议	当前接口所使用的协议
8	iInterface	1	索引	当前接口字符串的索引值

描述了传输类型、方向、数据包大小、端点号等信息

编号	域	大小/位	数据类型	描述
0	M.length	1	数字	此端点描述符长度, 7 个字节
1	bDescriptorType	1	常量	描述符类型, 为 0x05
2	bEndpointAddress	1	数字	端点地址和方向: b7:0: 端点号 b6:4: 保留, 为零。 b7:7: 方向, 0 输出端点(主机到设备), 1 输入端点(设备到主机)
3	bmAttributes	1	数字	端点属性, b7:0 表示传输类型: 00: 控制传输 01: 同步传输 10: 批量传输 11: 中断传输 其他位保留
4	wMaxPacketSize	2	数字	端点能发送或接收的最大数据包长度
6	bInterval	1	子类	端点数据传输中周期时间间隔值。此域对于批量传输和控制传输无效, 同步传输的此域必须为 1ms, 中断传输此域可以设置 1ms~25ms。

Device Descriptor 设备描述符 1

Configuration Descriptor 配置描述符 2

String Descriptor 字符串描述符 3

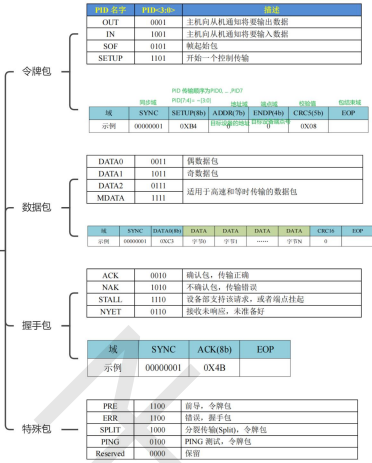
Interface Descriptor 接口字符串 4

Endpoint Descriptor 端点描述符 5

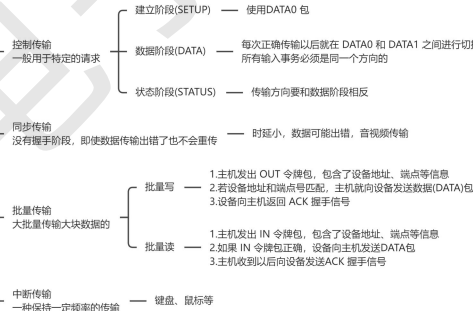
USB 驱动

USB 描述符

USB 数据包类型



USB 传输类型



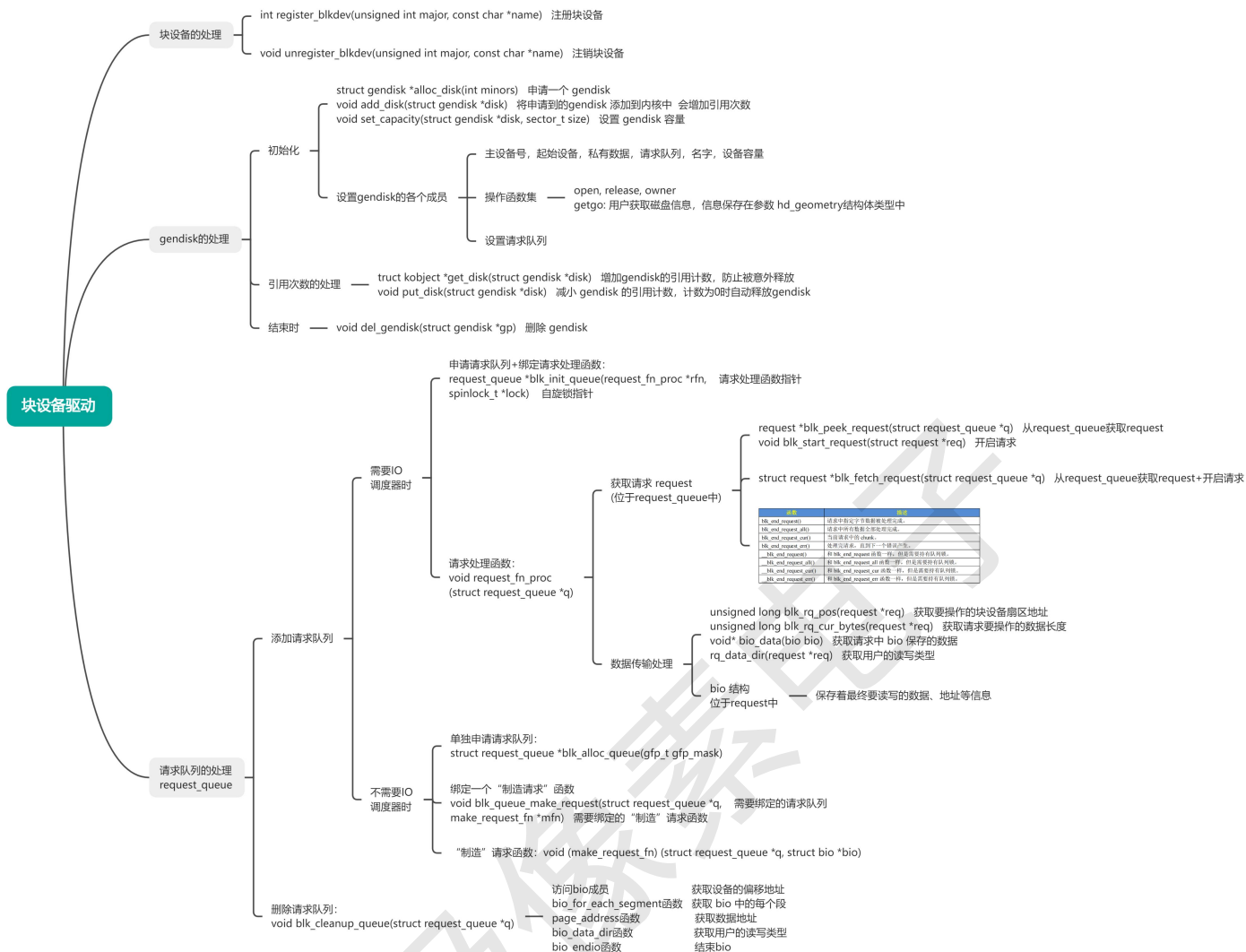
USB 枚举



驱动源码分析

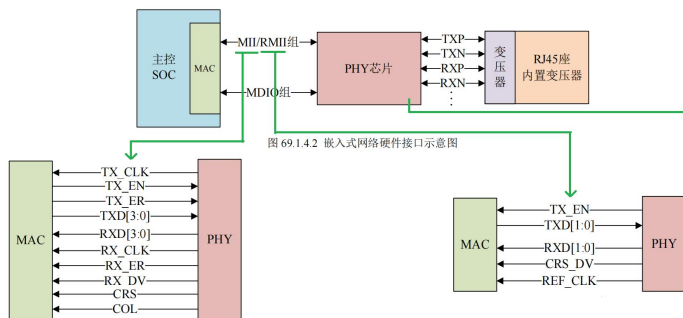


驱动代码编写方法



网络驱动

硬件电路解析



TX_CLK: 发送时钟, 如果网速为 100M 的话时钟频率为 25MHz, 10M 网速的话时钟频率为 2.5MHz, 此时钟由 PHY 产生并发送给 MAC。

TX_EN: 发送使能信号。

TX_ER: 发送错误信号，高电平有效，表示 TX_ER 有效期内传输的数据无效。10Mbps 网
速下 TX_ER 不起作用。

TXD[3:0]: 发送数据信号线, 一共 4 根。

RXD[3:0]: 接收数据信号线, 一共 4 根。

RX_CLK: 接收时钟信号, 如果网速为 100M 的话时钟频率为 25MHz, 10M 网速的话时钟频率为 2.5MHz, RX_CLK 也是由 PHY 产生的。

RX_ER: 接收错误信号，高电平有效，表示 RX_ER 有效期内传输的数据无效。10Mbps 网速下 RX_ER 不起作用。

RX_DV: 接收数据有效, 作用类似 TX_EN。

CRS: 载波侦听信号。

COL: 冲突检测信号。

TX_EN: 发送使能信号。

TXD[1:0]: 发送数据信号线, 一共 2 根。

RXD[1:0]: 接收数据信号线, 一共 2 根。

CRS_DV: 相当于 MII 接口中的 RX_DV 和 CRS 这两个信号的混合。

REF_CLK: 参考时钟, 由外部时钟源提供, 频率为 50MHz。这里与 MII 不同, MII 的接收和发送时钟是独立分开的, 而且都是由 PHY 芯片提供的。

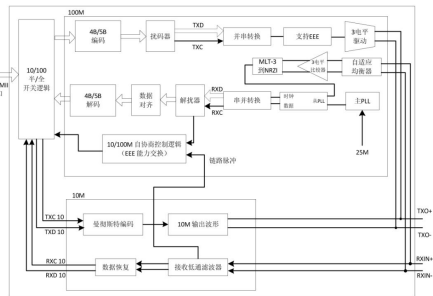
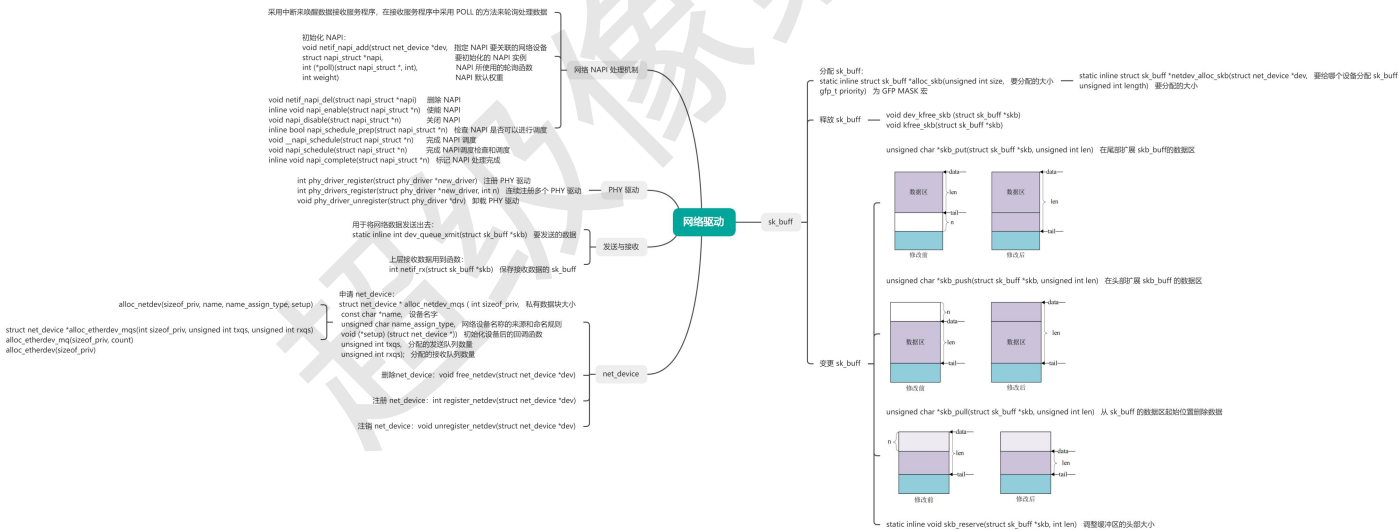


图 69.2.3.1 SR8201F 功能框图

- 1.配置 BCR 寄存器
- 2.从BSR寄存器获取 PHY 芯片的工作状态
- 3.可以从ID寄存器获取其唯一ID值

驱动用到的一些函数



驱动层/实现层代码解析

```
1 struct net_device {
2     char name[IFNAMSIZ]; 网络设备的名字
3     struct hlist_node name_hlist;
4     char *ifalias;
5     /*
6      * I/O specific fields
7      * FIXME: Merge these and struct ifmap into one
8      */
9     unsigned long mem_end; 共享内存结束地址
10    unsigned long mem_start; 共享内存起始地址
11    unsigned long base_addr; 网络设备 I/O 地址
12    int irq; 网络设备的中断号
13
14    atomic_t carrier_changes;
15
16    /*
17     * Some hardware also needs these fields (state, dev_list,
18     * napi_list, unreg_list, close_list) but they are not
19     * part of the usual set specified in Space.c.
20     */
21
22    unsigned long state;
23
24    struct list_head dev_list; 全局网络设备列表
25    struct list_head napi_list; napi 网络设备的列表入口
26    struct list_head unreg_list; 注册(unregister)的网络设备列表入口
27    struct list_head close_list; 关闭的网络设备列表入口
28
29    .....
30    const struct net_device_ops *netdev_ops; 网络设备操作函数集
31    const struct ethtool_ops *ethtool_ops; 网络管理工具相关函数集
32    #ifdef CONFIG_NET_SWITCHDEV
33    const struct swdev_ops *swdev_ops;
34    #endif
35
36    const struct header_ops *header_ops; 头部的相关操作函数集
37
38    unsigned int flags; 网络接口标志
39
40    .....
41    unsigned char if_port; 指定接口的端口类型
42    unsigned char dma; 网络设备所使用的 DMA 通道
43
44    .....
45    unsigned int mtu; 网络最大传输单元
46    unsigned short type; 指定 ARP 模块的类型
47    unsigned short hard_header_len;
48
49    .....
50    unsigned short needed_headroom;
51    unsigned short needed_tailroom;
52
53    /* Interface address info. */
54    unsigned char perm_addr[MAX_ADDR_LEN]; 永久的硬件地址
55    unsigned char addr_assign_type;
56    unsigned char addr_len; 硬件地址长度
57
58    .....
59    /* Cache lines mostly used on receive path (including
60     * eth_type_trans())
61     */
62    .....
63    unsigned long last_rx; 最后接收的数据包时间戳
64
65    /* Interface address info used in eth_type_trans() */
66    unsigned char *dev_addr; 硬件地址, 当前分配的 MAC 地址
67
68    .....
69    #ifdef CONFIG_SYSFS
70    struct netdev_rx_queue *rx; 接收队列
71
72    .....
73    unsigned int num_rx_queues; 接收队列数量
74    unsigned int real_num_rx_queues; 当前活动的队列数量
75
76    #endif
77    .....
78    /* Cache lines mostly used on transmit path
79     */
80    struct netdev_queue *tx; 发送队列
81    .....
82    unsigned int num_tx_queues; 发送队列数量
83    unsigned int real_num_tx_queues; 当前有效的发送队列数量
84    struct Qdisc *qdisc;
85    unsigned long tx_queue_len;
86    spinlock_t tx_global_lock;
87    int watchdog_timeout;
88
89    .....
90    /* These may be needed for future network-power-down code. */
91
92    .....
93    /*
94     * trans_start here is expensive for high speed devices on SMP.
95     * please use netdev_queue->trans_start instead.
96     */
97    .....
98    unsigned long trans_start; 最后的数据包发送的时间戳
99
100    struct phy_device *phydev; 对应的 PHY 设备
101    struct lock_class_key *qdisc_tx_busylock;
102 };
```

```
1 #include <uapi/linux/if.h> 示例代码 69.3.1.2 网络标志类型
2
3 enum net_device_flags {
4     IFF_UP = 1, /* sysfs */
5     IFF_BROADCAST = 2, /* volatile */
6     IFF_DEBUG = 4, /* sysfs */
7     IFF_LOOPBACK = 8, /* volatile */
8     IFF_POINTOPOINT = 16, /* volatile */
9     IFF_NOTRAILERS = 32, /* sysfs */
10    IFF_RUNNING = 64, /* volatile */
11    IFF_NOARP = 128, /* sysfs */
12    IFF_PROMISC = 256, /* sysfs */
13    IFF_ALLMULTI = 512, /* sysfs */
14    IFF_MASTER = 1024, /* volatile */
15    IFF_SLAVE = 2048, /* volatile */
16    IFF_MULTICAST = 4096, /* sysfs */
17    IFF_PORTSEL = 8192, /* sysfs */
18    IFF_AUTOMEDIA = 16384, /* sysfs */
19    IFF_DYNAMIC = 32768, /* sysfs */
20    IFF_LOWER_UP = 65536, /* volatile */
21    IFF_DORMANT = 131072, /* volatile */
22    IFF_ECHO = 262144, /* volatile */
23 };
```

```
1 #include <uapi/linux/netdevice.h> 示例代码 69.3.1.3 端口类型
2
3 enum {
4     IF_PORT_UNKNOWN = 0,
5     IF_PORT_10BASE2,
6     IF_PORT_10BASET,
7     IF_PORT_AUI,
8     IF_PORT_10BASEFX,
9     IF_PORT_10BASETX,
10    IF_PORT_10BASEFX,
11 };
```

```
1 struct net_device_ops {
2     int (*ndo_init)(struct net_device *dev); 第一次注册网络设备的初始化函数
3     void (*ndo_uninit)(struct net_device *dev); 卸载网络设备的函数
4     int (*ndo_open)(struct net_device *dev); 打开网络设备时执行的函数
5     int (*ndo_stop)(struct net_device *dev); 关闭网络设备时执行的函数
6     netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb,
7     struct net_device *dev); 需要发送数据时执行的函数
8     void (*ndo_select_queue)(struct net_device *dev,
9     struct sk_buff *skb,
10    void *accol_priv,
11    select_queue_fallback_t fallback); 支持多传输队列的时候选择使用哪个队列
12    void (*ndo_change_rx_flags)(struct net_device *dev,
13    int flags);
14    void (*ndo_set_rx_mode)(struct net_device *dev); 用于改变地址过滤列表
15    int (*ndo_set_mac_address)(struct net_device *dev,
16    void *addr); 用于修改网卡的 MAC 地址
17    int (*ndo_validate_addr)(struct net_device *dev); 验证 MAC 地址是否合法
18    int (*ndo_do_ioctl)(struct net_device *dev,
19    struct ifreq *ifr, int cmd); 用户程序调用 ioctl 的时候执行此函数
20    int (*ndo_set_config)(struct net_device *dev,
21    struct ifmap *map);
22    int (*ndo_change_mtu)(struct net_device *dev, 更改 MTU 大小
23    int new_mtu);
24    int (*ndo_neigh_setup)(struct net_device *dev,
25    struct neigh_parms *);
26    void (*ndo_tx_timeout)(struct net_device *dev); 当发送超时的时候执行此函数
27
28    .....
29    #ifdef CONFIG_NET_POLL_CONTROLLER
30    void (*ndo_poll_controller)(struct net_device *dev); 使用查询方式来处理网卡数据的收发
31    int (*ndo_netpoll_setup)(struct net_device *dev,
32    struct netpoll_info *info);
33    void (*ndo_netpoll_cleanup)(struct net_device *dev);
34
35    #endif
36    .....
37    int (*ndo_set_features)(struct net_device *dev,
38    netdev_features_t features);
39    .....
40    修改 net_device 的 features 属性, 设置相应的硬件属性
41 };
```

```
1 struct sk_buff {
2     union {
3         struct {
4             /* These two members must be first. */
5             struct sk_buff *next; /* 构成一个双向链表
6             struct sk_buff *prev;
7         };
8         union {
9             ktime_t tstamp; 数据接收时或准备发送时的时间戳
10            struct skb_mstamp skb_mstamp;
11        };
12    };
13    struct rb_node rbnode; /* used in netem & top stack */
14 };
15 struct sock *sk; 当前 sk_buff 所属的 Socket
16 struct net_device *dev; 表示当前 sk_buff 从哪个设备接收到或者发出的
17
18 /*
19  * This is the control buffer. It is free to use for every
20  * layer. Please put your private variables there. If you
21  * want to keep them across layers you have to do a skb_clone()
22  * first. This is owned by whoever has the skb queued ATM.
23  */
24 char cb[4096] __aligned(8); 控制缓冲区
25
26 unsigned long _skb_refdst;
27 void (*destructor)(struct sk_buff *skb);
28
29 .....
30 unsigned int len, data_len; 实际的数据长度
31 unsigned int mac_len, hdr_len; 连接层头部长度
32 .....
33 __be16 protocol; 协议: protocol 协议
34 __u16 transport_header; 传输层头部
35 __u16 network_header; 网络层头部
36 __u16 mac_header; 链路层头部
37
38 /* private: */
39 _u32 headers_end[0];
40 /* public: */
41
42 /* These elements must be at the end, see alloc_skb() for
43 details. */
44 tail; 指向实际数据的尾部
45 sk_buff_data_t end; 指向缓冲区的尾部
46 unsigned char *head, *data; head 指向缓冲区的头部, data 指向实际数据的头部
47 unsigned int truesize;
48 atomic_t users;
```

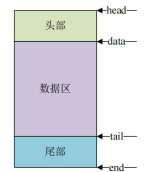


图 69.3.1.2 sk_buff 数据区结构示意图

数据层/接口层代码解析

probe函数解析

[illegible]

remove函数解析

[illegible]

PHY子系统与MIDO总线

[illegible]

网络设备操作集 制代码:03.4.3.16

[illegible]

通用PHY驱动

[illegible]

WIFI 驱动

需要移植的几个库的解析

组件	主要功能	典型工具/库
Wireless Tools	提供基础的 WiFi 设备配置（如扫描网络、设置 ESSID、加密方式等）	iwconfig, iwlist, iwspy
OpenSSL	提供加密算法和 TLS/SSL 协议支持（用于 WPA2/WPA3 的密钥协商和数据加密）	libssl, libcrypto
libnl	提供 Netlink 协议支持（内核与用户态通信，用于现代 WiFi 配置）	libnl-3, libnl-genl, libnl-route
wpa_supplicant	实现 WPA/WPA2/WPA3 认证协议，管理加密连接（依赖 OpenSSL 和 libnl）	wpa_supplicant, wpa_cli

4G 驱动

要做的事情：给内核相关部分添加代码，make menuconfig 配置，移植应用程序

一些讨论

为什么 4G 模块只有少数厂家有在生产，并且网上没有任何的电路原理图/PCB 开源工程(比高性能 Linux 核心板还要少)

核心逻辑链

这条商业逻辑链是这样的：

- 根本属性：**它是一个**无线通信模块**。
- 必然要求：**无线通信会占用**稀缺的公共频谱资源**。
- 管理手段：**为了公平、高效、安全地使用资源，**政府/国际组织强制进行认证**。
- 高额投入：**认证过程**极其昂贵、复杂且耗时**（研发、测试、认证费用）。
- 商业决策：**厂商投入巨资通过认证后，其**硬件设计（尤其是射频部分）就成了核心商业机密和护城河**。
- 最终结果：**为了保护这项投资并维持竞争优势，厂商选择**完全闭源**。

PWM 驱动

```
int pwmchip_add(struct pwm_chip *chip)  向内核注册初始化完成以后的 pwm_chip
int pwmchip_remove(struct pwm_chip *chip)  卸载 PWM 驱动
```

驱动源码分析

示例代码 73.1.1.1 pwm3 节点内容

```
1 pwm3: pwm@02088000 {
2     compatible = "fsl,imx6ul-pwm", "fsl,imx27-pwm";
3     reg = <0x02088000 0x4000>;
4     interrupts = <GIC_SPI 85 IRQ_TYPE_LEVEL_HIGH>;
5     clocks = <&clks IMX6UL_CLK_PWM3>,
6             <&clks IMX6UL_CLK_PWM3>;
7     clock-names = "ipg", "per";
8     #pwm-cells = <2>;
9 };
```

示例代码 73.1.3.1 IMX6ULL PWM 平台驱动

```
1 static const struct of_device_id imx_pwm_dt_ids[] = {
2     { .compatible = "fsl,imx6ul-pwm", .data = &imx_pwm_data_v1, },
3     { .compatible = "fsl,imx27-pwm", .data = &imx_pwm_data_v2, },
4     { /* sentinel */ }
5 };
6
7 .....
8
9 static struct platform_driver imx_pwm_driver = {
10     .driver = {
11         .name = "imx-pwm",
12         .of_match_table = imx_pwm_dt_ids,
13     },
14     .probe = imx_pwm_probe,
15     .remove = imx_pwm_remove,
16 };
17
18 module_platform_driver(imx_pwm_driver);
```

示例代码 73.1.3.3 imx_pwm_probe 函数

```
1 static int imx_pwm_probe(struct platform_device *pdev)
2 {
3     const struct of_device_id *of_id =
4         of_match_device(imx_pwm_dt_ids, &pdev->dev);
5     const struct imx_pwm_data *data;
6     struct imx_chip *imx;
7     struct resource *r;
8     int ret = 0;
9
10    if (!of_id)
11        return -ENODEV;
12
13    imx = devm_kzalloc(&pdev->dev, sizeof(*imx), GFP_KERNEL);
14    if (imx == NULL)
15        return -ENOMEM;
16
17    imx->chip.ops = &imx_pwm_ops; // 初始化 imx 的 chip 成员变量
18    imx->chip.dev = &pdev->dev;
19    imx->chip.base = -1;
20    imx->chip.ngwm = 1;
21    imx->chip.can_sleep = true; // 从设备树中获取 PWM 节点中关于 PWM 控制器的地址信息
22
23    r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
24    imx->mmio_base = devm_ioremap_resource(&pdev->dev, r); // 进行内存映射
25    if (IS_ERR(imx->mmio_base))
26        return PTR_ERR(imx->mmio_base);
27
28    data = of_id->data;
29    imx->config = data->config;
30    imx->set_enable = data->set_enable;
31
32    ret = pwmchip_add(&imx->chip);
33    if (ret < 0)
34        return ret;
35
36    platform_set_drvdata(pdev, imx);
37    return 0;
38 }
```

示例代码 73.1.3.4 imx_pwm_ops 操作集合

```
1 static struct pwm_ops imx_pwm_ops = {
2     .enable = imx_pwm_enable,
3     .disable = imx_pwm_disable,
4     .config = imx_pwm_config,
5     .owner = THIS_MODULE,
6 };
```

示例代码 73.1.3.2 imx_pwm_data_v2 结构体变量

```
1 static struct imx_pwm_data imx_pwm_data_v2 = {
2     .config = imx_pwm_config_v2, // 最终操作 IMX6ULL 的 PWM 外设寄存器, 进行实际配置的函数
3     .set_enable = imx_pwm_set_enable_v2,
4 };
```

用于打开或关闭对应的 PWM

```
1 static void imx_pwm_set_enable_v2(struct pwm_chip *chip, bool enable)
2 {
3     struct imx_chip *imx = to_imx_chip(chip);
4     u32 val;
5
6     val = readl(imx->mmio_base + MX3_PWMCR); // 读取 PWMCR 寄存器的值
7
8     if (enable)
9         val |= MX3_PWMCR_EN; // 使能 PWM
10    else
11        val &= ~MX3_PWMCR_EN; // 关闭 PWM
12
13    writel(val, imx->mmio_base + MX3_PWMCR);
14 }
```

示例代码 73.1.3.6 imx_pwm_config_v2 函数

```
1 static int imx_pwm_config_v2(struct pwm_chip *chip,
2     struct pwm_device *pwm, int duty_ns, int period_ns)
3 {
4     struct imx_chip *imx = to_imx_chip(chip);
5     struct device *dev = chip->dev;
6     unsigned long long c;
7     unsigned long period_cycles, duty_cycles, prescale;
8     unsigned int period_ms;
9     bool enable = test_bit(PWMF_ENABLED, &pwm->flags);
10    int wait_count = 0, fifoav;
11    u32 cr, sr;
12
13    .....
14
15    c = clk_get_rate(imx->clk_per);
16    c = c * period_ns;
17    do_div(c, 1000000000);
18    period_cycles = c;
19
20    prescale = period_cycles / 0x10000 + 1;
21
22    period_cycles /= prescale;
23    c = (unsigned long long)period_cycles * duty_ns;
24    do_div(c, period_ns);
25    duty_cycles = c;
26
27    /*
28     * according to imx pwm FM, the real period value should be
29     * PERIOD value in PWMCR plus 2.
30     */
31    if (period_cycles > 2)
32        period_cycles -= 2;
33    else
34        period_cycles = 0;
35
36    writel(duty_cycles, imx->mmio_base + MX3_PWMCR); // 写寄存器, 设置占空比
37    writel(period_cycles, imx->mmio_base + MX3_PWMCR); // 写寄存器, 设置频率
38
39    cr = MX3_PWMCR_PRESCALER(prescale) |
40        MX3_PWMCR_DOZEEN | MX3_PWMCR_WAITEN |
41        MX3_PWMCR_DBGGEN | MX3_PWMCR_CLKSRC_IPG_HIGH;
42
43    if (enable)
44        cr |= MX3_PWMCR_EN;
45
46    writel(cr, imx->mmio_base + MX3_PWMCR);
47
48    return 0;
49 }
```

include/linux/pwm.h

示例代码 73.1.2.1 pwm_chip 结构体

```
1 struct pwm_chip {
2     struct device *dev;
3     struct list_head list;
4     const struct pwm_ops *ops; // PWM 外设的各种操作函数集合
5     int base;
6     ngwm;
7     struct pwm_device *pwm;
8     struct pwm_device * (*of_xlate)(struct pwm_chip *pc,
9         const struct of_phandle_args *args);
10    unsigned int of_pwm_n_cells;
11    bool can_sleep;
12 };
```

示例代码 73.1.2.2 pwm_ops 结构体

```
1 struct pwm_ops {
2     int (*request)(struct pwm_chip *chip, // 请求 PWM
3         struct pwm_device *pwm);
4     void (*free)(struct pwm_chip *chip, // 释放 PWM
5         struct pwm_device *pwm);
6     int (*config)(struct pwm_chip *chip, // 配置 PWM 周期和占空比
7         struct pwm_device *pwm,
8         int duty_ns, int period_ns);
9     int (*set_polarity)(struct pwm_chip *chip, // 设置 PWM 极性
10         struct pwm_device *pwm,
11         enum pwm_polarity polarity);
12     int (*enable)(struct pwm_chip *chip, // 使能 PWM
13         struct pwm_device *pwm);
14     void (*disable)(struct pwm_chip *chip, // 关闭 PWM
15         struct pwm_device *pwm);
16     struct module *owner;
17 };
```

Regmap API

源码解析

drivers/base/regmap/internal.h 示例代码 74.12.1 regmap 结构体

```
51 struct regmap {
52     union {
53         struct mutex mutex;
54         struct {
55             spinlock_t spinlock;
56             unsigned long spinlock_flags;
57         };
58     };
59     regmap_lock lock;
60     regmap_unlock unlock;
61     void *lock_arg; /* This is passed to lock/unlock functions */
62
63     struct device *dev; /* Device we do I/O on */
64     void *work_buf; /* Scratch buffer used to format I/O */
65     struct regmap_format format; /* Buffer format */
66     const struct regmap_bus *bus;
67     void *bus_context;
68     const char *name;
69
70     bool async;
71     spinlock_t async_lock;
72     wait_queue_head_t async_waitq;
73     struct list_head async_list;
74     struct list_head async_free;
75     int async_ret;
76
77     .....
78     unsigned int max_register;
79     bool (*writeable_reg)(struct device *dev, unsigned int reg);
80     bool (*readable_reg)(struct device *dev, unsigned int reg);
81     bool (*volatile_reg)(struct device *dev, unsigned int reg);
82     bool (*precious_reg)(struct device *dev, unsigned int reg);
83     const struct regmap_access_table *wr_table;
84     const struct regmap_access_table *rd_table;
85     const struct regmap_access_table *volatile_table;
86     const struct regmap_access_table *precious_table;
87     const struct regmap_access_table *precious_table;
88
89     int (*reg_read)(void *context, unsigned int reg, unsigned int *val);
90     int (*reg_write)(void *context, unsigned int reg, unsigned int val);
91
92     .....
93     struct rb_root range_tree;
94     void *selector_work_buf; /* Scratch buffer used for selector */
95 };
```

示例代码 74.12.2 regmap_config 结构体

```
186 struct regmap_config {
187     const char *name; 名字
188
189     int reg_bits; 寄存器地址位数
190     int reg_stride; 寄存器地址步长
191     int pad_bits; 寄存器和值之间的填充位数
192     int val_bits; 寄存器值位数
193
194     bool (*writeable_reg)(struct device *dev, unsigned int reg); 寄存器可写的此回调函数就会被调用
195     bool (*readable_reg)(struct device *dev, unsigned int reg); 寄存器可读的此回调函数就会被调用
196     bool (*volatile_reg)(struct device *dev, unsigned int reg); 当寄存器值不能缓存的时候此回调函数就会被调用
197     bool (*precious_reg)(struct device *dev, unsigned int reg); 当寄存器值不能被读出来的时候此回调函数就会被调用
198     regmap_lock lock;
199     regmap_unlock unlock;
200     void *lock_arg;
201
202     int (*reg_read)(void *context, unsigned int reg, unsigned int *val); 所有读寄存器的操作此回调函数就会执行
203     int (*reg_write)(void *context, unsigned int reg, unsigned int val); 所有写寄存器的操作此回调函数就会执行
204
205     bool fast_io; 快速 I/O, 使用 spinlock 替代 mutex 来提升性能
206
207     unsigned int max_register; 有效的最大寄存器地址
208     const struct regmap_access_table *wr_table; 可写的地址范围
209     const struct regmap_access_table *rd_table;
210     const struct regmap_access_table *volatile_table;
211     const struct regmap_access_table *precious_table;
212     const struct regmap_default *reg_defaults; 寄存器模式值
213     unsigned int num_reg_defaults;
214     unsigned int num_reg_defaults;
215     enum regcache_type cache_type;
216     const void *reg_defaults_raw;
217     unsigned int num_reg_defaults_raw; 默认寄存器表中的元素个数
218
219     u8 read_flag_mask; 读标志掩码
220     u8 write_flag_mask; 写标志掩码
221
222     bool use_single_rw;
223     bool can_multi_write;
224
225     enum regmap_endian reg_format_endian;
226     enum regmap_endian val_format_endian;
227
228     const struct regmap_range_cfg *ranges;
229     unsigned int num_ranges;
230 };
```

regmap_config 掩码设置 示例代码 74.14.2 regmap-spi 代码段

```
105 static struct regmap_bus regmap_spi = { 初始化了一个 regmap_bus 实例: regmap_spi
106     .write = regmap_spi_write,
107     .gather_write = regmap_spi_gather_write,
108     .async_write = regmap_spi_async_write,
109     .async_alloc = regmap_spi_async_alloc,
110     .read = regmap_spi_read,
111     .read_flag_mask = 0x00, 通过掩码设置 spi 数据开头比特来控制读写
112     .reg_format_endian_default = REGMAP_ENDIAN_BIG,
113     .val_format_endian_default = REGMAP_ENDIAN_BIG,
114 };
115
116 .....
117 struct regmap *regmap_init_spi(struct spi_device *spi,
118     const struct regmap_config *config)
119 {
120     return regmap_init(&spi->dev, &regmap_spi, &spi->dev, config);
121 }
122
123
124 示例代码 74.14.3 regmap_init 函数代码段
125 if (config->read_flag_mask || config->write_flag_mask) {
126     map->read_flag_mask = config->read_flag_mask;
127     map->write_flag_mask = config->write_flag_mask;
128 } else if (bus) {
129     map->read_flag_mask = bus->read_flag_mask;
130 }
```

regmap API 函数

regmap_write regmap_read
regmap_bulk_write regmap_bulk_read
.....

regmap core

flat lzo rbtree

物理总线

i2c i3c spi mmio sccb
sdw slimbus irq spmi wl

SPI Regmap 申请与初始化: struct regmap * regmap_init_spi(struct spi_device *spi, const struct regmap_config *config)

图 74.12.1 regmap 框架

驱动编写方法



IIO 框架

它的核心思想是暴露传感器的“通道（Channel）”和“属性（Attribute）”，而不是提供一个简单的、无结构的字节流（那是传统字符设备的工作）。

驱动源码分析

```
474 struct iio_dev {
475     int id;
476
477     int modes;
478     int currentmode; // 当前模式
479     struct device dev;
480
481     struct iio_event_interface *event_interface;
482
483     struct iio_buffer *buffer; // 缓冲区
484     struct list_head buffer_list; // 当前匹配的缓冲区列表
485     int scan_bytes; // 捕获到并且提供给缓冲区的字节数
486     struct mutex mlock;
487
488     const unsigned long *available_scan_masks; // 确定使能哪些通道
489     unsigned masklength;
490     const unsigned long *active_scan_mask; // 缓冲区已经开启的通道掩码
491     bool scan_timestamp; // 扫描时间戳
492     unsigned scan_index_timestamp;
493     struct iio_trigger *trig; // IIO 设备当前触发器
494     struct iio_poll_func *pollfunc; // 在接收到的触发器上运行的函数
495
496     struct iio_chan_spec const *channels; // IIO 设备通道
497     int num_channels; // IIO 设备的通道数
498
499     struct list_head channel_attr_list;
500     struct attribute_group chan_attr_group;
501     const char *name; // IIO 设备名字
502     const struct iio_info *info; // 函数集
503     struct mutex info_exist_lock;
504     const struct iio_buffer_setup_ops *setup_ops;
505     struct cdev cdev;
506
507     .....
508 };
```

```
427 struct iio_buffer_setup_ops {
428     int (*preamble)(struct iio_dev *); // 缓冲区使能之前调用
429     int (*postenable)(struct iio_dev *); // 缓冲区使能之后调用
430     int (*predisable)(struct iio_dev *); // 缓冲区禁用之前调用
431     int (*postdisable)(struct iio_dev *); // 缓冲区禁用之后调用
432     bool (*validate_scan_mask)(struct iio_dev *indio_dev,
433         const unsigned long *scan_mask); // 检查扫描掩码是否有效
434 };
```

```
include/linux/iio/iio.h
352 struct iio_info {
353     struct module *driver_module;
354     struct attribute_group *event_attr;
355     const struct attribute_group *attrs; // 通用的设备属性
356
357     int (*read_raw)(struct iio_dev *indio_dev,
358         struct iio_chan_spec const *chan,
359         int *val, // 最终读写设备内部数据的操作函数
360         int *val2, // 次要通道
361         long mask);
362
363     .....
364
365     int (*write_raw)(struct iio_dev *indio_dev,
366         struct iio_chan_spec const *chan,
367         int val, // 最终读写设备内部数据的操作函数
368         int val2, // 次要通道
369         long mask);
370
371     int (*write_raw_get_fmt)(struct iio_dev *indio_dev,
372         struct iio_chan_spec const *chan,
373         long mask);
374
375     .....
376
377     int (*write_raw_get_fmt)(struct iio_dev *indio_dev,
378         struct iio_chan_spec const *chan,
379         long mask);
380
381     .....
382 };
```

模式	说明
INDIO_DIRECT_MODE	提供 sysfs 接口。
INDIO_BUFFER_TRIGGERED	支持硬件缓冲触发。
INDIO_BUFFER_SOFTWARE	支持软件缓冲触发。
INDIO_BUFFER_HARDWARE	支持硬件缓冲触发。

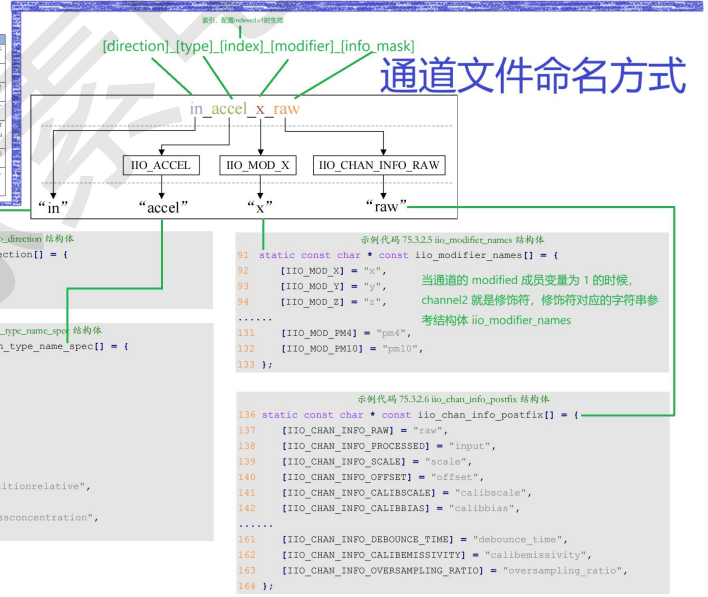
```
224 struct iio_chan_spec {
225     enum iio_chan_type type;
226     int channel; // 当成员变量 indexed == 1, channel 为通道索引
227     int channel2; // 当成员变量 modified 为 1 的时候, channel2 为通道修饰符
228     unsigned long address; // 常设置为通道对应的芯片数据寄存器地址
229     int scan_index; // 扫描索引(有缓冲区时)
230     struct {
231         char sign; // 'u' -> 无符号类型, 's' -> 有符号类型
232         u8 realbits; // 数据真实的有效位数
233         u8 storagebits; // 存储位数
234         u8 shift; // 右移位数
235         u8 repeat; // 实际或存储位的重复数量
236     } enum iio_endian; // 数据的大小端模式
237     scan_type;
238     long info_mask_separate; // 标记某些属性是各通道独立的
239     long info_mask_shared_by_type; // 标记某些属性是各通道相同的
240     long info_mask_shared_by_dir; // 标记某些属性是各通道相同的
241     long info_mask_shared_by_all; // 标记某些属性是所有特征各通道相同的
242     const struct iio_event_spec *event_spec;
243     unsigned int num_event_specs;
244     const struct iio_chan_spec_ext_info *ext_info;
245     const char *extend_name;
246     const char *datasheet_name;
247     unsigned modified; // 当成员变量 indexed == 1 的时候, channel2 为通道修饰符
248     unsigned indexed; // 当成员变量 indexed == 1 的时候, channel2 为通道修饰符
249     unsigned output; // 表示为输出通道
250     unsigned differential; // 表示为差分通道
251 };
```

```
enum iio_chan_type {
14     IIO_VOLTAGE, // 电压类型
15     IIO_CURRENT, // 电流类型
16     IIO_POWER, // 功率类型
17     IIO_ACCEL, // 加速度类型
18     IIO_ANG_VEL, // 角速度类型 (陀螺仪)
19     IIO_MAGN, // 电磁类型 (磁力计)
20     IIO_LIGHT, // 灯光类型
21     IIO_INTENSITY, // 强度类型 (光强传感器)
22     IIO_PROXIMITY, // 接近类型 (接近传感器)
23     IIO_TEMP, // 温度类型
24     IIO_INCL, // 倾角类型 (倾角测量传感器)
25     IIO_ROT, // 旋转角度类型
26     IIO_ANG, // 倾角类型 (电机旋转角度测量传感器)
27     IIO_TIMESTAMP, // 时间类型
28     IIO_CAPACITANCE, // 电容类型
29     IIO_ALTITUDE, // 高度类型
30     IIO_CCT, // 笔管暂时未知的类型
31     IIO_PRESSURE, // 压力类型
32     IIO_HUMIDITY_RELATIVE, // 湿度类型
33     IIO_ACTIVITY, // 活动类型 (计步传感器)
34     IIO_STEPS, // 步数类型
35     IIO_ENERGY, // 能量类型 (卡路里)
36     IIO_DISTANCE, // 距离类型
37     IIO_VELOCITY, // 速度类型
38 };
```

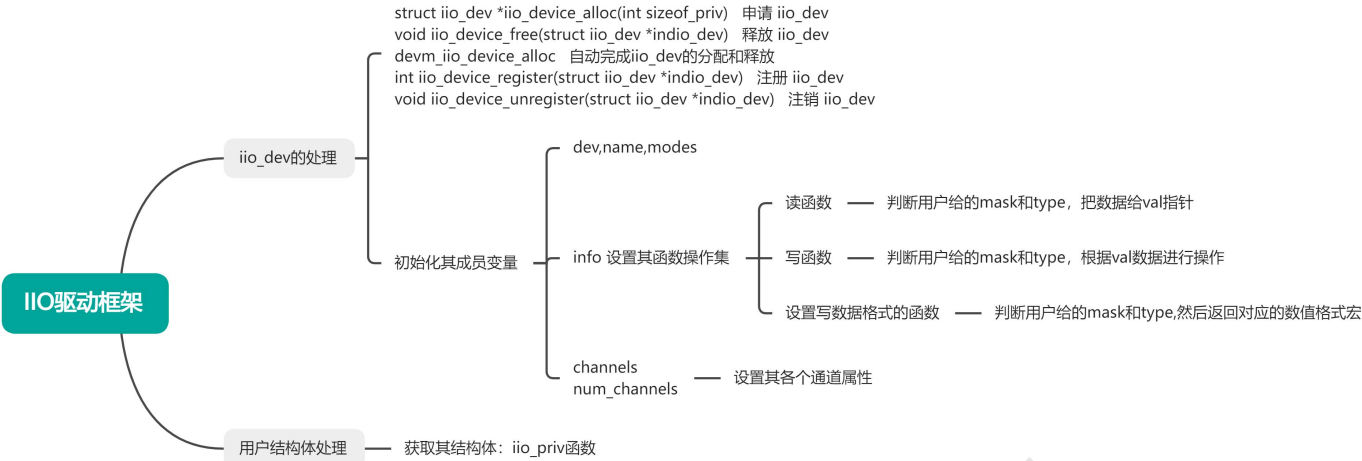
```
include/uapi/linux/iio/types.h
enum iio_modifier {
41     IIO_NO_MOD, // 无修饰
42     IIO_MOD_X, // X 轴
43     IIO_MOD_Y, // Y 轴
44     IIO_MOD_Z, // Z 轴
45
46     .....
47 };
```

```
include/linux/iio/types.h
enum iio_chan_info_enum {
24     IIO_CHAN_INFO_RAW = 0,
25     IIO_CHAN_INFO_PROCESSED,
26     IIO_CHAN_INFO_SCALE,
27     IIO_CHAN_INFO_OFFSET,
28
29     .....
30     IIO_CHAN_INFO_DEBOUNCE_TIME,
31
32     .....
33 };
```

IIO驱动框架



驱动编写方法



其他

- 1.FILE *fopen(const char *pathname, const char *mode) 打开文件流
- 2.int fclose(FILE *stream) 关闭文件流
- 3.size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) 读取文件流
- 4.size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream); 写文件流
- 5.int fscanf(FILE *stream, const char *format, [,argument...]) 格式化输入文件流
- 6.fseek 函数

这行代码 `fseek(data_stream, 0, SEEK_SET);`的意思是: 将文件流 `data_stream`的读写位置重置到文件的开头。

这是一个非常常用且重要的操作, 通常被称为“回绕 (rewind)”文件指针。

详细分解:

1. fseek函数

- 功能: 用于移动文件流的位置指针 (也称为文件位置指示器)。
- 所属头文件: `<stdio.h>`
- 原型: `int fseek(FILE *stream, long offset, int whence);`

2. 三个参数的含义

- `data_stream`: 这是一个 `FILE*`类型的指针, 指向你之前通过 `fopen`打开的文件流。它代表了你所正在操作的那个文件。
- `0`: 这是偏移量 (offset) 。它指定了从基准位置 (由第三个参数决定) 开始移动的字节数。
- `SEEK_SET`: 这是一个常量, 定义了偏移的基准位置 (whence) 。`SEEK_SET`表示基准位置是文件的开头。

ADC 驱动

ADC 驱动源码分析

1 adcl: adc802190000 {
2 compatible = "fsl,imx6ul-adc", "fsl,vf610-adc"; 兼容性属性
3 reg = <0x02190000 0x4000>; ADC 控制器寄存器信息
4 interrupts = <GIC_SPI 100 IRQ_TYPE_LEVEL_HIGH>; 中断属性
5 clocks = <clks IMX6UL_CLK_ADCL>; 时钟属性
6 num-channels = <1>;
7 clock-names = "adc"; 时钟名字
8 status = "disabled"; vref 参考电压引脚
9 };

ADC 外设结构体
1 struct vf610_adc {
2 struct device *dev;
3 void __iomem *regs;
4 struct clk *clk;
5
6 u32 vref_uv;
7 u32 value;
8 struct regulator *vref;
9 struct vf610_adc_feature adc_feature;
10
11 u32 sample_freq_avail[5];
12
13 struct completion completion;
14 };

1 static int vf610_adc_probe(struct platform_device *pdev)
2 {
3 struct vf610_adc *info;
4 struct iio_dev *indio_dev;
5 struct resource *mem;
6 int irq;
7 int ret;
8 u32 channels;
9
10 indio_dev = devm_iio_device_alloc(&pdev->dev, sizeof(struct vf610_adc));
11 if (!indio_dev) {
12 dev_err(&pdev->dev, "Failed allocating iio device\n");
13 return -ENOMEM;
14 }
15
16 info = iio_priv(indio_dev); 从 iio_dev 里面得到 vf610_adc 首地址
17 info->dev = &pdev->dev;
18
19 mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
20 info->regs = devm_ioremap_resource(&pdev->dev, mem);
21 if (IS_ERR(info->regs))
22 return PTR_ERR(info->regs);
23
24 irq = platform_get_irq(pdev, 0); 获取中断号
25 if (irq < 0) {
26 dev_err(&pdev->dev, "no irq resource?\n");
27 return irq;
28 }
29
30 ret = devm_request_irq(info->dev, irq, 申请中断
31 vf610_adc_isr, 0,
32 dev_name(&pdev->dev), info);
33 if (ret < 0) {
34 dev_err(&pdev->dev, "failed requesting irq, irq = %d\n", irq);
35 return ret;
36 }
37
38 info->clk = devm_clk_get(&pdev->dev, "adc");
39 if (IS_ERR(info->clk)) {
40 dev_err(&pdev->dev, "failed getting clock, err = %d\n",
41 PTR_ERR(info->clk));
42 return PTR_ERR(info->clk);
43 }
44
45 info->vref = devm_regulator_get(&pdev->dev, "vref");
46 if (IS_ERR(info->vref))
47 return PTR_ERR(info->vref);
48
49 ret = regulator_enable(info->vref);
50 if (ret)
51 return ret;
52
53 info->vref_uv = regulator_get_voltage(info->vref);
54
55 platform_set_drvdata(pdev, indio_dev);
56
57 init_completion(&info->completion);
58
59 ret = of_property_read_u32(pdev->dev.of_node,
60 "num-channels", &channels);
61 if (ret)
62 channels = ARRAY_SIZE(vf610_adc_iio_channels);
63
64 indio_dev->name = dev_name(&pdev->dev);
65 indio_dev->dev.parent = &pdev->dev;
66 indio_dev->dev.of_node = pdev->dev.of_node;
67 indio_dev->info = &vf610_adc_iio_info;
68 indio_dev->modes = IIO_DIRECT_MODE;
69 indio_dev->channels = vf610_adc_iio_channels;
70 indio_dev->num_channels = (int)channels;
71
72 ret = clk_prepare_enable(info->clk);
73 if (ret) {
74 dev_err(&pdev->dev,
75 "Could not prepare or enable the clock.\n");
76 goto error_adc_clk_enable;
77 }
78
79 vf610_adc_cfg_init(info); ADC 的配置初始化
80 vf610_adc_hw_init(info); 初始化 ADC 硬件
81
82 ret = iio_device_register(indio_dev); 向内核注册 iio_dev
83 if (ret) {
84 dev_err(&pdev->dev, "Could't register the device.\n");
85 goto error_iio_device_register;
86 }
87
88 return 0;
89
90 return ret;
91 };

1 static const struct iio_info vf610_adc_iio_info = {
2 .driver_module = THIS_MODULE,
3 .read_raw = &vf610_read_raw,
4 .write_raw = &vf610_write_raw,
5 .debugfs_reg_access = &vf610_adc_reg_access,
6 .attrs = &vf610_attribute_group,
7 };

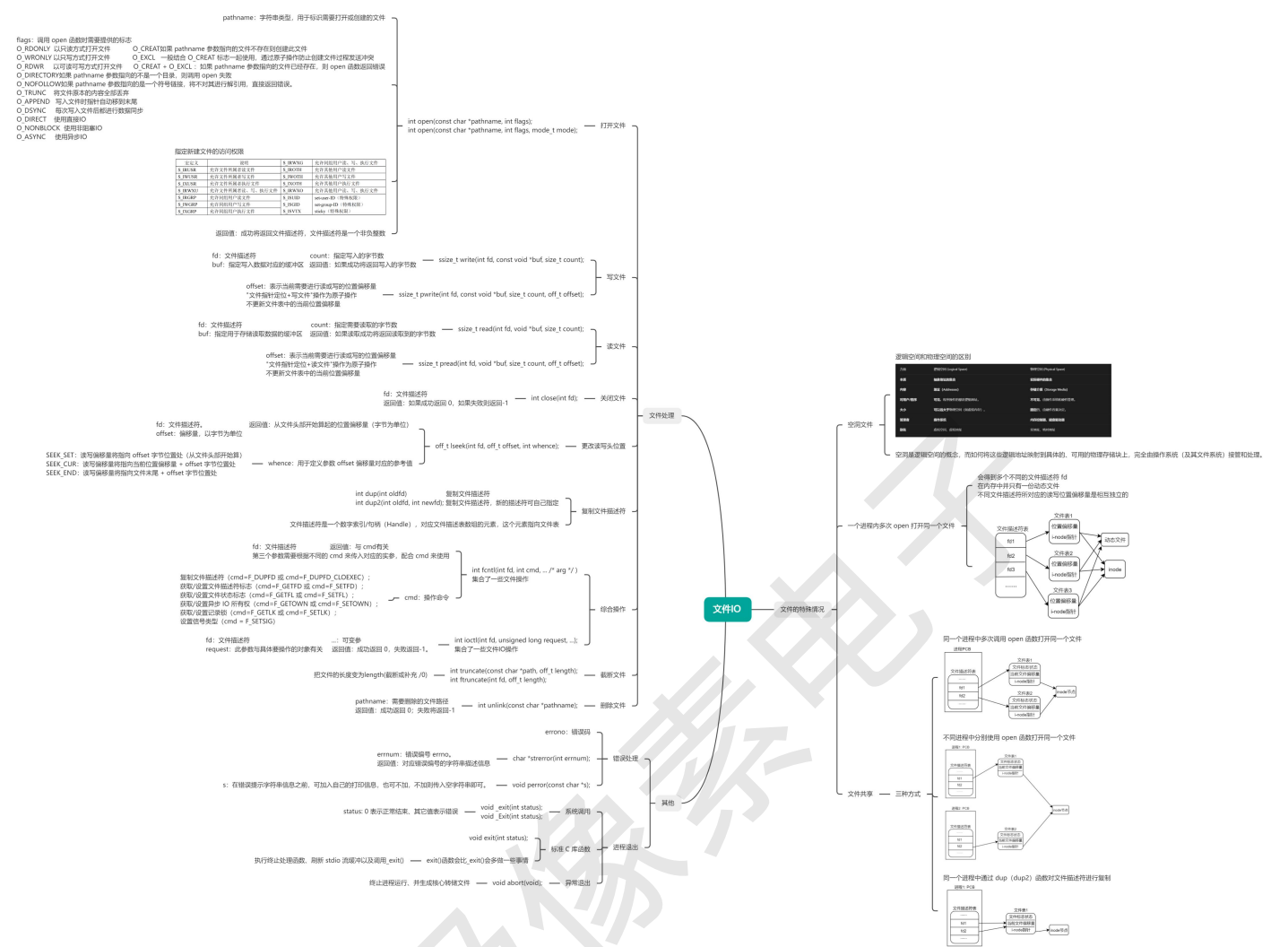
1 static int vf610_read_raw(struct iio_dev *indio_dev,
2 struct iio_chan_spec const *chan,
3 int *val,
4 int *val2,
5 long mask)
6 {
7 struct vf610_adc *info = iio_priv(indio_dev);
8 unsigned int hc_cfg;
9 long ret;
10
11 switch (mask) {
12 case IIO_CHAN_INFO_RAW:
13 case IIO_CHAN_INFO_PROCESSED:
14 mutex_lock(&indio_dev->mlock);
15 reinit_completion(&info->completion);
16
17 hc_cfg = VF610_ADC_ADCH(chan->channel);
18 hc_cfg |= VF610_ADC_AIEN;
19 writel(hc_cfg, info->regs + VF610_REG_ADC_HC0);
20 ret = wait_for_completion_interruptible_timeout
21 (&info->completion, VF610_ADC_TIMEOUT);
22 if (ret == 0) {
23 mutex_unlock(&indio_dev->mlock);
24 return -ETIMEDOUT;
25 }
26 if (ret < 0) {
27 mutex_unlock(&indio_dev->mlock);
28 return ret;
29 }
30
31 switch (chan->type) {
32 case IIO_VOLTAGE: 读取电压值
33 *val = info->value;
34 break;
35 case IIO_TEMP:
36 /* Calculate in degree Celsius times 1000
37 * Using sensor slope of 1.84 mV/°C and
38 * V at 25°C of 696 mV
39 */
40 *val = 25000 - ((int)info->value - 864) * 1000000 /
41 1840;
42 break;
43 default:
44 mutex_unlock(&indio_dev->mlock);
45 return -EINVAL;
46 }
47
48 mutex_unlock(&indio_dev->mlock);
49 return IIO_VAL_INT;
50
51 case IIO_CHAN_INFO_SCALE:
52 *val = info->vref_uv / 1000;
53 *val2 = info->adc_feature.res_mode;
54 return IIO_VAL_FRACTIONAL_LOG2;
55
56 case IIO_CHAN_INFO_SAMP_FREQ:
57 *val = info->sample_freq_avail[info->
58 adc_feature.sample_rate];
59 *val2 = 0;
60 return IIO_VAL_INT;
61
62 default:
63 break;
64 }
65 return -EINVAL;
66 }

1 static irqreturn_t vf610_adc_isr(int irq, void *dev_id)
2 {
3 struct vf610_adc *info = (struct vf610_adc *)dev_id;
4 int coco;
5
6 coco = readl(info->regs + VF610_REG_ADC_HS);
7 if (coco & VF610_ADC_HS_COCON) {
8 info->value = vf610_adc_read_data(info); 读取 ADC 原始值
9 complete(&info->completion);
10 ADC 的原始值保存位置
11
12 return IRQ_HANDLED;
13 }

初始化其成员变量, 初始化ADC

返回 ADC 对应的分辨率

文件 IO



应用和内核的关系

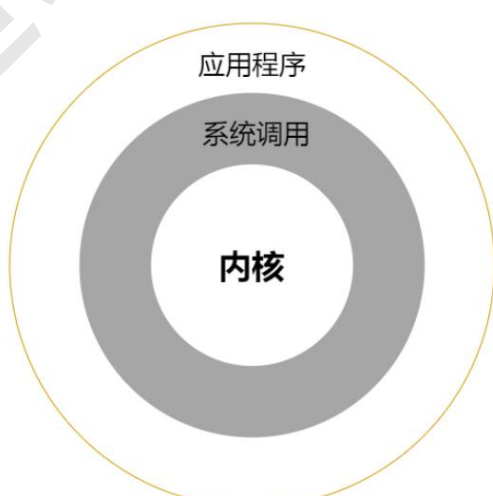
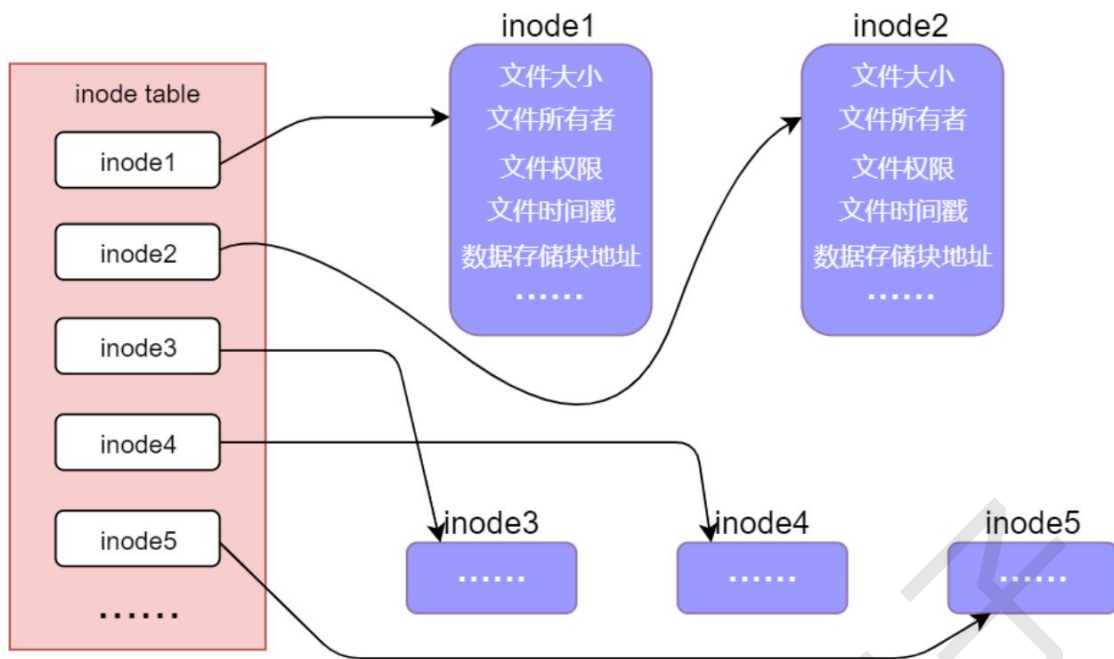
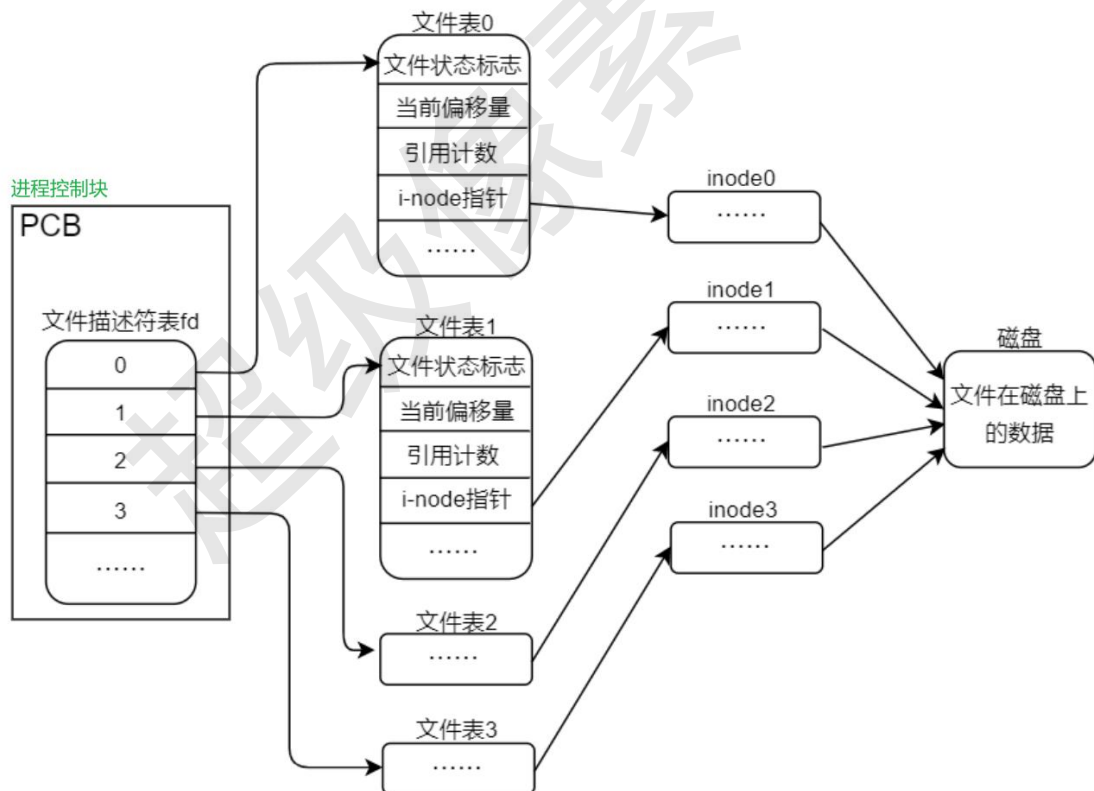


图 1.1.1 内核、系统调用与应用程序

静态文件(文件还没被打开时)



文件打开时的状态



文件操作(韦东山)

打开文件

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
fd = open(argv[1], O_RDWR);
```

函数原型:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

函数说明:

- ① **pathname** 表示打开文件的路径;
- ② **Flags** 表示打开文件的方式, 常用的有以下 6 种,
 - **O_RDWR** 表示可读可写方式打开;
 - **O_RDONLY** 表示只读方式打开;
 - **O_WRONLY** 表示只写方式打开;
 - **O_APPEND** 表示如果这个文件中本来是有内容的, 则新写入的内容会接续到原来内容的后面;
 - **O_TRUNC** 表示如果这个文件中本来是有内容的, 则原来的内容会被丢弃, 截断;
 - **O_CREAT** 表示当前打开文件不存在, 我们创建它并打开它, 通常与 **O_EXCL** 结合使用, 当没有文件时创建文件, 有这个文件时会报错提醒我们;
- ③ **Mode** 表示创建文件的权限, 只有在 **flags** 中使用了 **O_CREAT** 时才有效, 否则忽略。
- ④ 返回值: 打开成功返回文件描述符, 失败将返回-1。

创建文件

```
fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0777);
```

O_RDONLY: 只读模式

O_WRONLY: 只写模式

O_RDWR: 可读可写

O_APPEND: 表示追加, 如果原来文件里面有内容, 则这次写入会写在文件的最末尾。

O_CREAT: 表示如果指定文件不存在, 则创建这个文件

O_EXCL: 表示如果要创建的文件已存在, 则出错, 同时返回-1, 并且修改 **errno** 的值。

O_TRUNC: 表示截断, 如果文件存在, 并且以只写、读写方式打开, 则将其长度截断为 0。

最后创建文件的权限为 777&(~umask 查到的结果)

```
book@100ask:~/fileio/02_create$ umask
0002
```

写文件

写入

write	<pre>#include <unistd.h> ssize_t write(int fd, const void *buf, size_t count);</pre>	<p>作用：将 buf 中的 count 字节数据写入指定文件描述符的文件中。</p> <p>fd：指定要写入的文件描述符</p> <p>buf：缓冲区，一般是一个数组，读取存放于该数组的内容存放于文件中</p> <p>count：要写入的实际字节数</p>
-------	---	---

```
write(fd, "end!!!", 3);
```

修改指针

lseek	<pre>#include <sys/types.h> #include <unistd.h> off_t lseek(int fd, off_t offset, int whence);</pre>	<p>作用：重新定位读/写文件偏移</p> <p>fd：指定要偏移的文件描述符</p> <p>offset：文件偏移量</p> <p>whence：开始添加偏移 offset 的位置</p> <p>SEEK_SET, offset 相对于文件开头进行偏移</p> <p>SEEK_CUR, offset 相对文件当前位置进行偏移</p> <p>SEEK_END, offset 相对于文件末尾进行偏移</p>
-------	---	--

```
lseek(fd, 3, SEEK_SET);
// 文件句柄 开始插入位置 选项
```

读文件

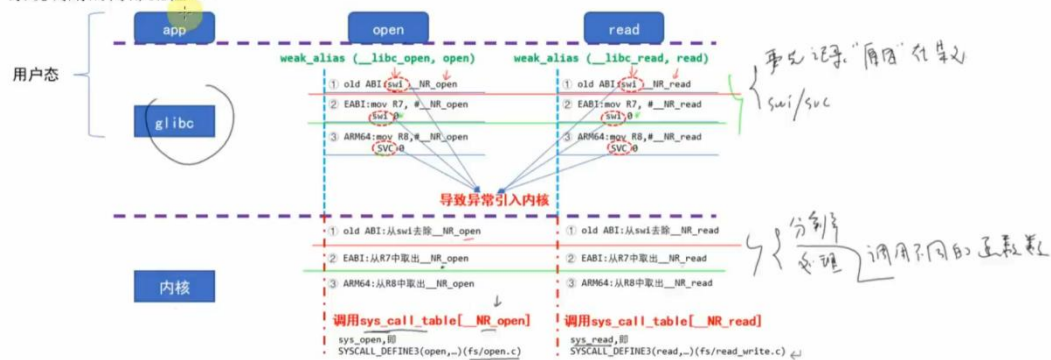
read	<pre>#include <unistd.h> ssize_t read(int fd, void *buf, size_t count);</pre>	<p>作用：从给定的文件描述符指定的文件中，读取 count 个字节的数 据，存放至 buf 中。</p> <p>fd：指定要读写的文件描述符</p> <p>buf：缓冲区，一般是一个数组，用于存放读取的内容</p> <p>count：一次要读取的最大字节数</p>
------	--	---

Eg:

```
//复原文件指针
lseek(fd, 0, SEEK_SET);
// 从fd句柄指向的文件读取 (sizeof(buf) - 1) 长度并放到buf中
len = read(fd, buf, sizeof(buf) - 1);
```

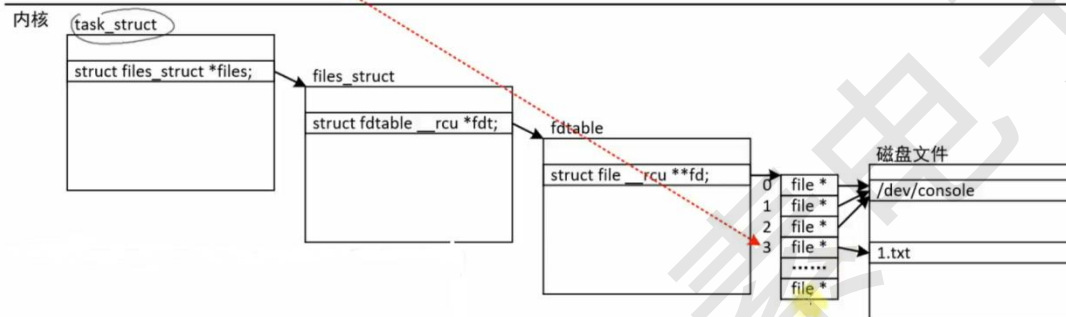
文件与内核操作

系统调用的内部流程:



用户程序把异常信息放入寄存器，然后调用特定汇编指令引发异常让内核去处理文件句柄的结构:

```
APP fd = open("1.txt", O_RDWR); // fd = 3
```



每个 fd 虽然是 int 型的，但是能在内核里映射到一个具体的文件（上图的 **file*** 数组）
dup 函数可以用来复制文件句柄，相当于 C++ 的 & 引用

```
int fd2;  
fd2 = dup(fd);
```

dup2 函数：文件句柄重定向

Dup(a,b): 把 a 文件句柄赋值给 b，这样 b 的值就为 a，对 b 操作，实际就是对 a 进行操作

```
int main(int argc, char **argv)  
{  
    int fd;  
    fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0644);  
    // 0-标准输入  
    // 1-标准输出  
    // 2-标准错误输出  
    dup2(fd, 1);  
    printf("123");  
    return 0;  
}
```

ioctl

头文件:

```
#include <sys/ioctl.h>
```

函数原型:

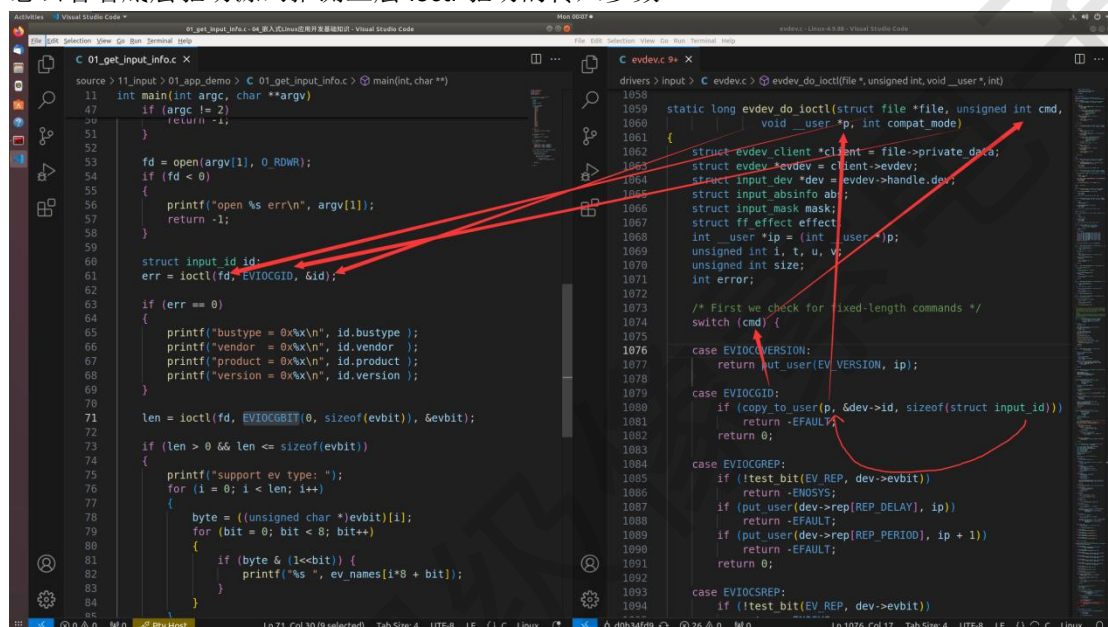
```
int ioctl(int fd, unsigned long request, ...);
```

函数说明:

- ① **fd** 表示文件描述符;
- ② **request** 表示与驱动程序交互的命令, 用不同的命令控制驱动程序输出我们需要的数据;
- ③ ... 表示可变参数 **arg**, 根据 **request** 命令, 设备驱动程序返回输出的数据。
- ④ 返回值: 打开成功返回文件描述符, 失败将返回-1。

ioctl 的作用非常强大、灵活。不同的驱动程序内部会实现不同的 **ioctl**, APP 可以使用各种 **ioctl** 跟驱动程序交互: 可以传数据给驱动程序, 也可以从驱动程序中读出数据。

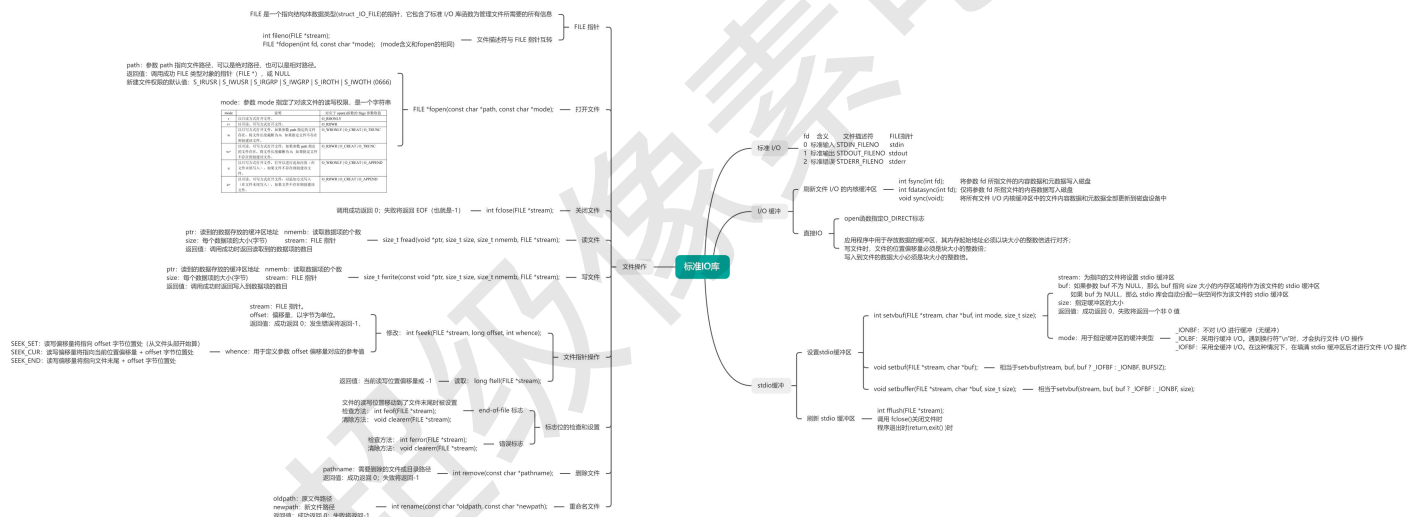
怎么看着底层驱动源码推测上层 **ioctl** 驱动的传入参数



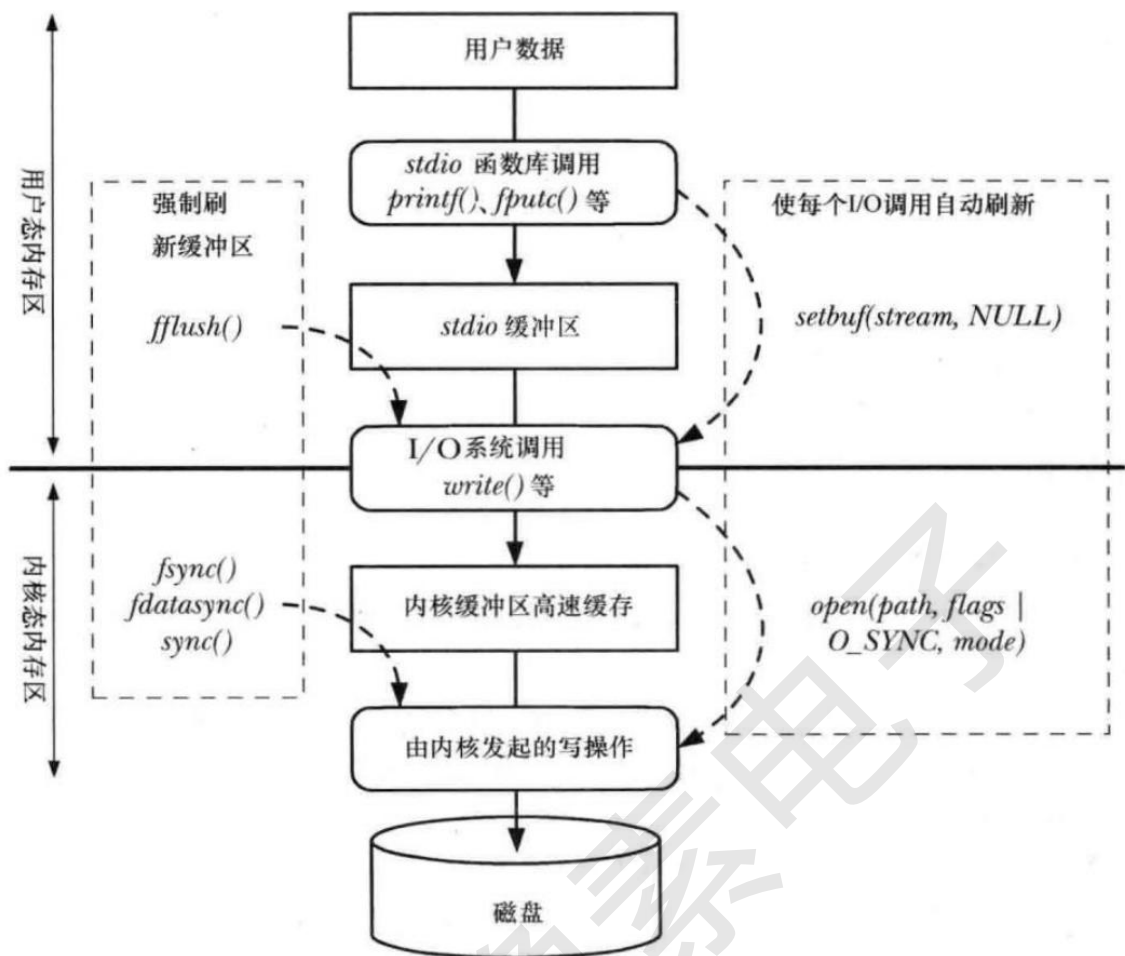
```
len = ioctl(fd, EVIOCGBIT(0, sizeof(evbit)), &evbit);

C input.h
#define EVIOCGBIT(ev,len) _IOC(_IOC_READ, 'E', 0x20 + (ev), len) /* get event bits */

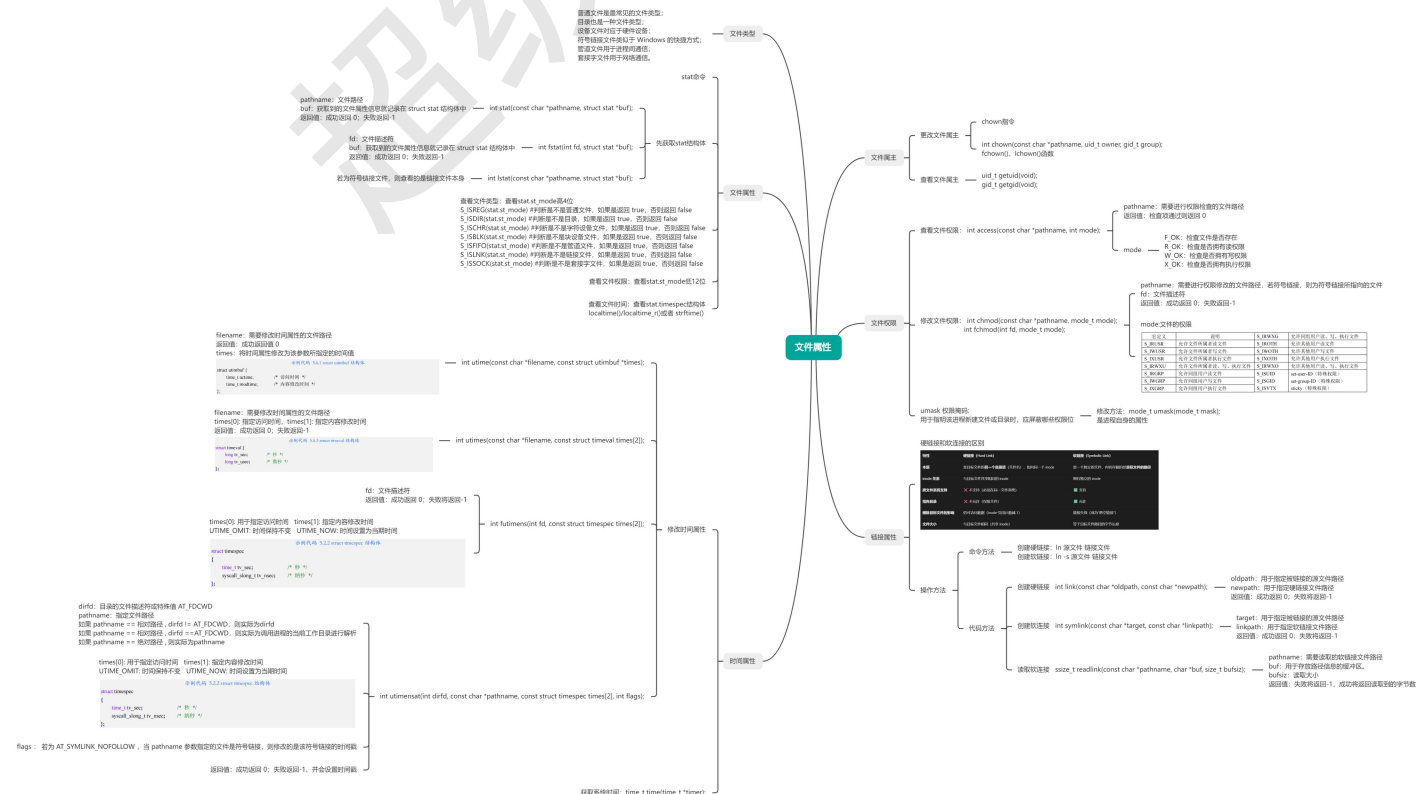
C ioctl.h
69 #define _IOC(dir,type,nr,size) \
70     (((dir) << _IOC_DIRSHIFT) | \
71      ((type) << _IOC_TYPSHIFT) | \
72      ((nr) << _IOC_NRSHIFT) | \
73      ((size) << _IOC_SIZESHIFT))
```

IO 层级



文件属性



Stat 结构体

```
示例代码 5.2.1 struct stat 结构体
struct stat
{
    dev_t st_dev;           /* 文件所在设备的 ID */
    ino_t st_ino;           /* 文件对应 inode 节点编号 */
    mode_t st_mode;         /* 文件对应的模式 */
    nlink_t st_nlink;       /* 文件的链接数 */
    uid_t st_uid;           /* 文件所有者的用户 ID */
    gid_t st_gid;           /* 文件所有者的组 ID */
    dev_t st_rdev;          /* 设备号 (针对设备文件) */
    off_t st_size;          /* 文件大小 (以字节为单位) */
    blksize_t st_blksize;   /* 文件内容存储的块大小 */
    blkcnt_t st_blocks;     /* 文件内容所占块数 */
    struct timespec st_atim; /* 文件最后被访问的时间 */
    struct timespec st_mtim; /* 文件内容最后被修改的时间 */
    struct timespec st_ctim; /* 文件状态最后被改变的时间 */
};
```

表 5.4.1 与进程相关联的用户 ID 和组 ID

ID 类型	作用
实际用户 ID	我们实际上是谁
实际组 ID	
有效用户 ID	用于文件访问权限检查
有效组 ID	
附属组 ID	

```
示例代码 5.2.2 struct timespec 结构体
struct timespec
{
    time_t tv_sec;          /* 秒 */
    syscall_slong_t tv_nsec; /* 纳秒 */
};
```

0000 000 000 000 000
文件类型 S U G O

S_IRWXU	00700	owner has read, write, and execute permission
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	group has read, write, and execute permission
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	others (not in group) have read, write, and execute permission
S_IROTH	00004	others have read permission
S_IWOTH	00002	others have write permission
S_IXOTH	00001	others have execute permission

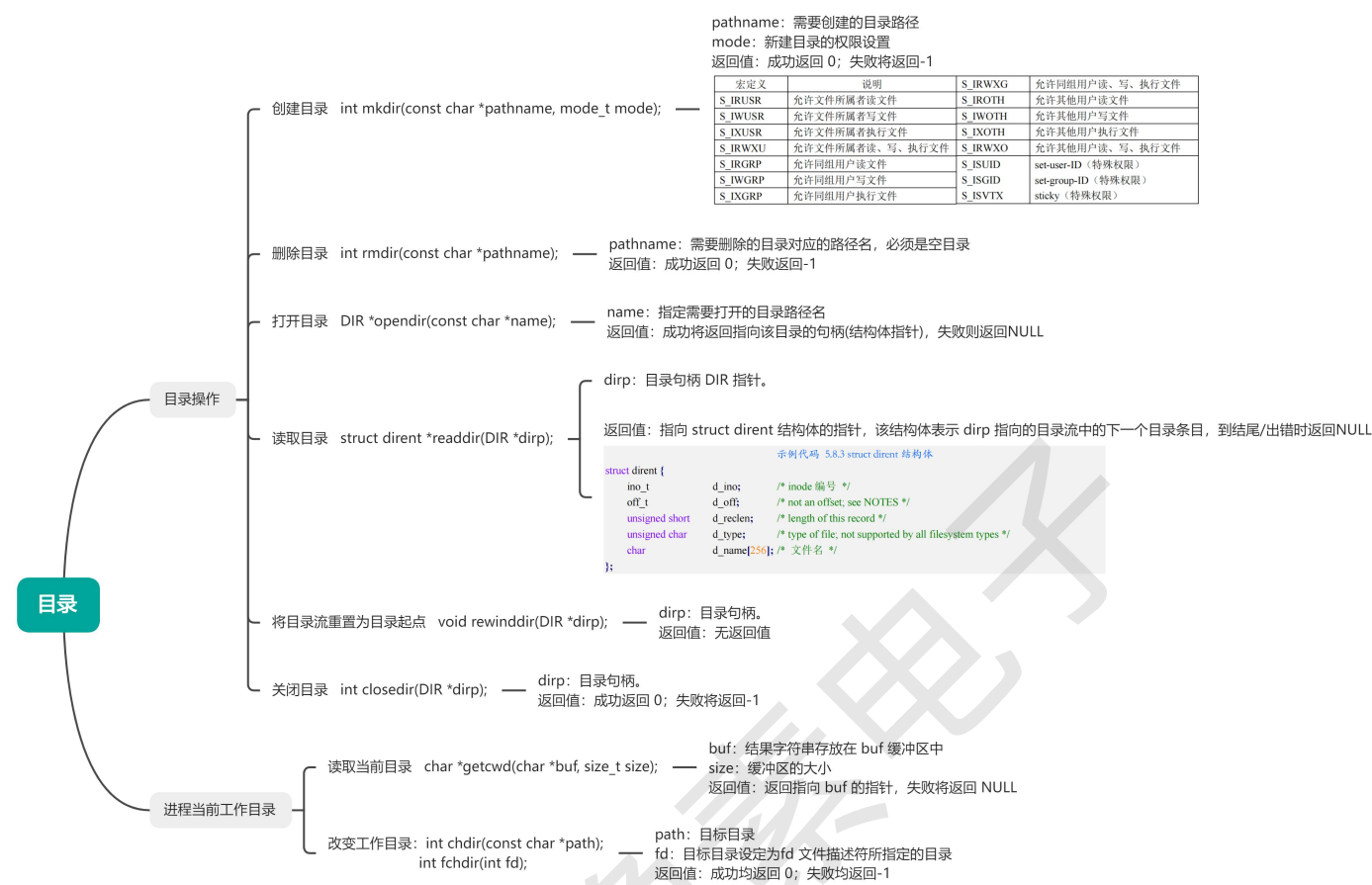
权限 (八进制方式表示)

S_IFSOCK	0140000	socket (套接字文件)
S_IFLNK	0120000	symbolic link (链接文件)
S_IFREG	0100000	regular file (普通文件)
S_IFBLK	0060000	block device (块设备文件)
S_IFDIR	0040000	directory (目录)
S_IFCHR	0020000	character device (字符设备文件)
S_IFIFO	0010000	FIFO (管道文件)

文件类型

S_ISUID	04000	set-user-ID bit	内核会将进程的有效 ID 设置为该文件的用户 ID	特殊权限
S_ISGID	02000	set-group-ID bit (see below)	内核会将进程的有效用户组 ID 设置为该文件的用户组 ID	
S_ISVTX	01000	sticky bit (see below)		

目录



目录在文件系统中的存储形式

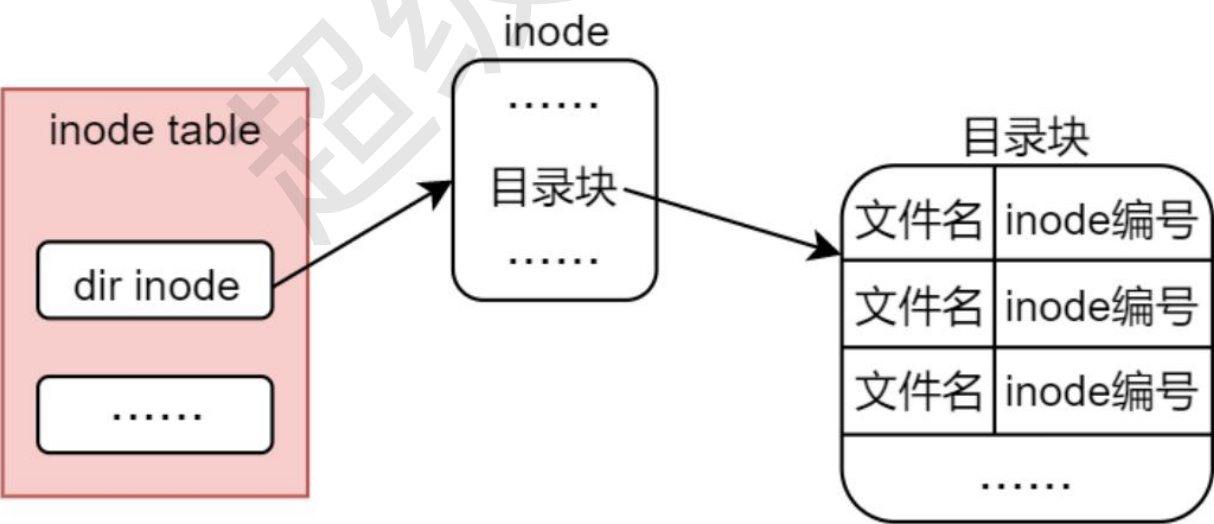
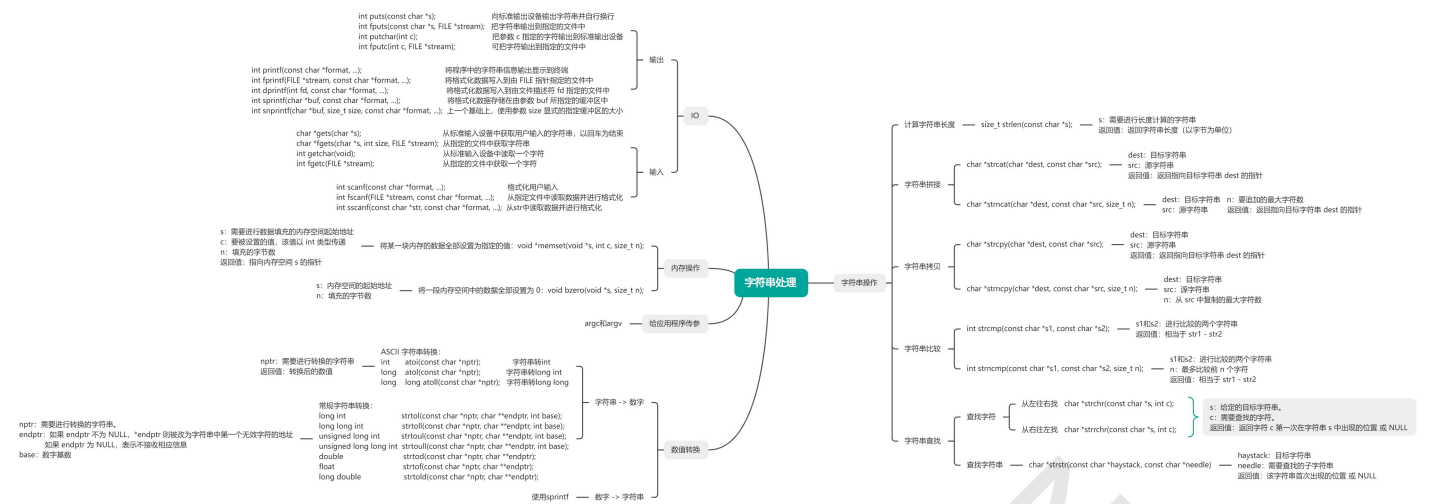
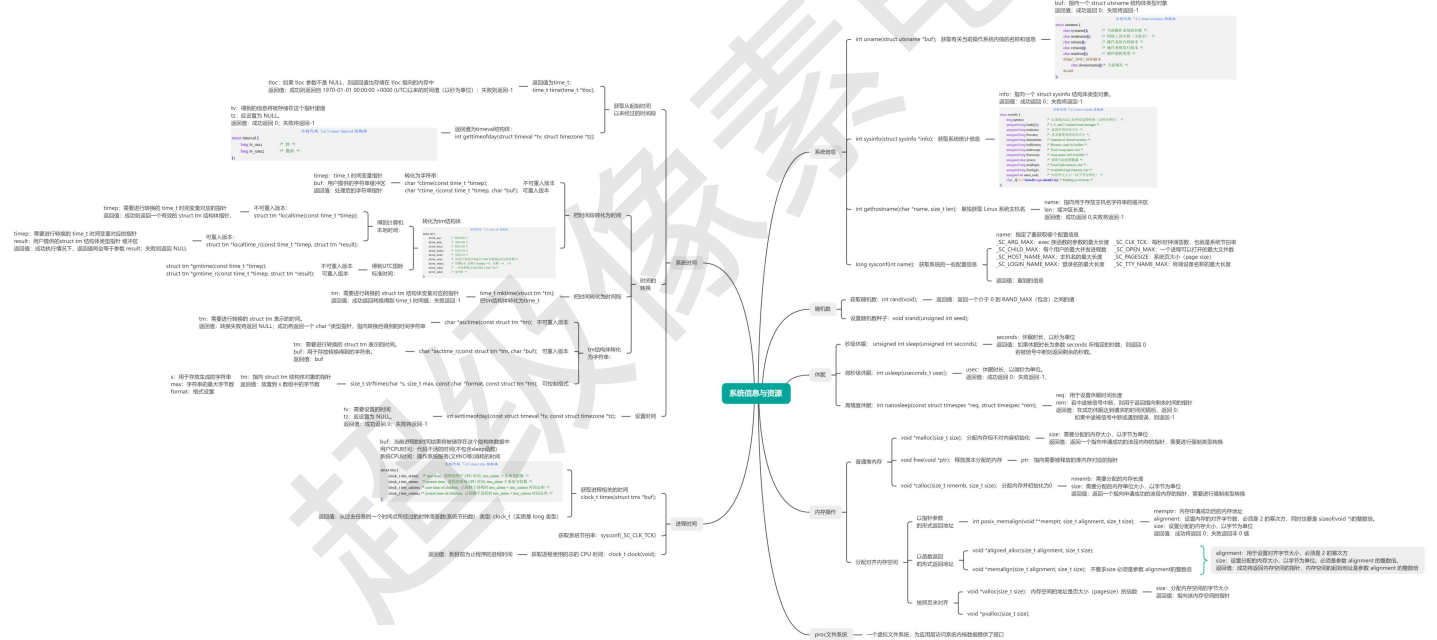


图 5.8.1 目录在文件系统中的存储形式

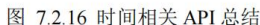
字符串处理



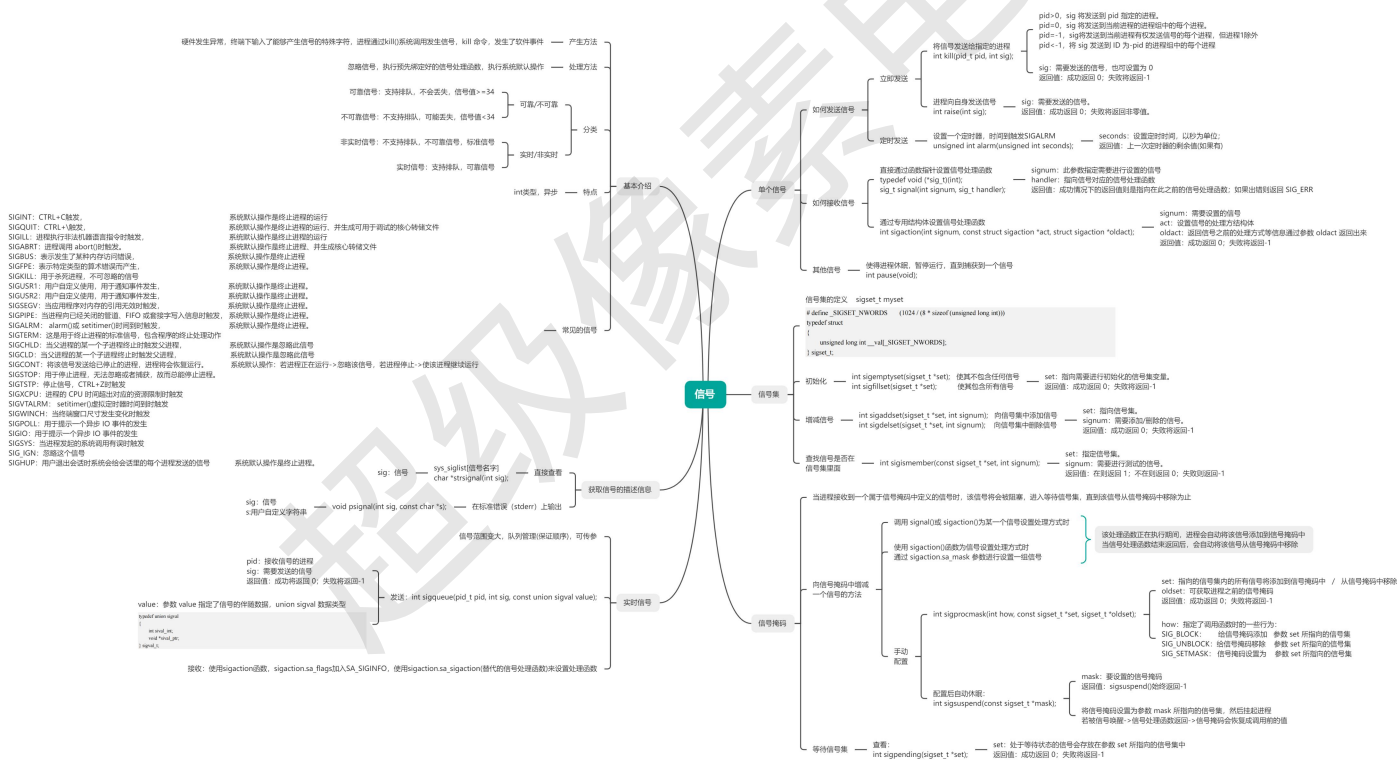
系统信息与资源



信号



信号



Sigation 结构体

示例代码 8.4.2 struct sigaction 结构体

```
struct sigaction {  
    void (*sa_handler)(int);  
    void (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t sa_mask;  
    int sa_flags;  
    void (*sa_restorer)(void);  
};
```

指定信号处理函数

替代的信号处理函数

信号信号掩码字段，决定信号竞争优先级

标志位，控制信号处理过程

不使用

SA_NOCLDSTOP: 屏蔽SIGCHLD 信号

SA_NOCLDWAIT: 子进程终止时阻止其变为僵尸进程

SA_NODEFER: 不要阻塞从某个信号自身的信号处理函数中接收此信号

SA_RESETHAND: 执行完信号处理函数之后，将信号的处理方式设置为系统默认操作

SA_RESTART: 被信号中断的系统调用，在信号处理完成之后将自动重新发起

SA_SIGINFO: 表示使用 sa_sigaction 作为信号处理函数

示例代码 8.4.3 siginfo_t 结构体

```
siginfo_t {  
    int si_signo; /* Signal number */  
    int si_errno; /* An errno value */  
    int si_code; /* Signal code */  
    int si_trapno; /* Trap number that caused hardware-generated signal(unused on most architectures) */  
    pid_t si_pid; /* Sending process ID */  
    uid_t si_uid; /* Real user ID of sending process */  
    int si_status; /* Exit value or signal */  
    clock_t si_utime; /* User time consumed */  
    clock_t si_stime; /* System time consumed */  
    sigval_t si_value; /* Signal value */  
    int si_int; /* POSIX.1b signal */  
    void *si_ptr; /* POSIX.1b signal */  
    int si_overrun; /* Timer overrun count; POSIX.1b timers */  
    int si_timerid; /* Timer ID; POSIX.1b timers */  
    void *si_addr; /* Memory location which caused fault */  
    long si_band; /* Band event (was int in glibc 2.3.2 and earlier) */  
    int si_fd; /* File descriptor */  
    short si_addr_lsb; /* Least significant bit of address(since Linux 2.6.32) */  
    void *si_call_addr; /* Address of system call instruction(since Linux 3.5) */  
    int si_syscall; /* Number of attempted system call(since Linux 3.5) */  
    unsigned int si_arch; /* Architecture of attempted system call(since Linux 3.5) */  
};
```

引发处理函数被调用的信号

表示文件描述符 si_fd 发生了什么事件

位掩码

发生异步 I/O 事件的文件描述符

表 13.4.1 siginfo_t 结构体中的 si_code 和 si_band 的可能值

si_code	si_band 掩码值	描述/说明
POLL_IN	POLLIN POLLRDNORM	可读取数据
POLL_OUT	POLLOUT POLLWRNORM POLLWRBAND	可写入数据
POLL_MSG	POLLIN POLLRDNORM POLLMSG	不使用
POLL_ERR	POLLERR	I/O 错误
POLL_PRI	POLLPRI POLLRDNORM	可读取高优先级数据
POLL_HUP	POLLHUP POLLERR	出现宕机

进程

正文(代码块): 机器语言指令, 只读, 可共享
初始化数据块(数据块): 包含了静态初始化的全局变量和静态变量
未初始化数据块(bss块): 包含了未进行静态初始化的全局变量和静态变量, 程序运行时, 由加载器来分配该段内存空间, 系统将其初始化为 0
栈: 包含函数内用的局部变量以及每次函数调用所需保存的函数、动态增长和堆、由栈帧组成
堆: 可在运行时动态进行内存分配的一块区域

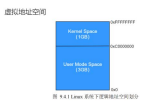
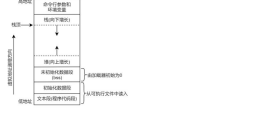


图 9.1.1 Linux 系统内存空间划分示意图

内存布局

名称-值-范围的组合

增加: export 名称=值
删除: export -n 名称=值

extern char **environ; 申明外部全局变量 environ

name: 指定要取的环境变量名称 ("name"不是基本)
只想要获取某个指定的环境变量
char *getenv(const char *name);
返回: 返回该环境变量的值(以字符串的形式)
string: 字符串指针, 指向 name=value 形式的字符串 ("string"不是基本)
返回: 成功返回 0, 失败则返回非 0 值

name: 需要添加或修改的环境变量名称 (不会有基本问题)
value: 环境变量的值 (不会有基本问题)
返回: 成功返回 0, 失败则返回 -1, 并设置 errno.

overwrite: 0, 则不会覆盖原本name表示已存在的环境变量
若 overwrite=0, 会覆盖原本name表示已存在的环境变量

创建基本缓冲区:
int setenv(const char *name, const char *value, int overwrite);

删除:
int unsetenv(const char *name);

执行程序时: 使用 NAME=value ./app 指令

清除环境变量:
int clearenv(void);
environ = NULL;

若fork前已打开文件, 则文件表项共享

父进程 fork 子进程, 共享代码段, 不共享内存空间

子进程独立使用new/free 函数或运行新程序或者使用 exit()退出

父进程调用 fork()退出, 子进程调用 exit()退出

对符合每实现父进程子进程的协调

父进程调用 wait()退出, 子进程调用 wait()退出

等待进程的任一子进程终止, 并获取子进程的终止状态信息, 同时可处理已终止子进程留下的垃圾

返回: 若成功则返回已终止的子进程对应的进程号; 失败则返回 -1

处理终止的子进程
pid_t wait(int *status);

status: 存储子进程终止时的状态信息

WIFEXITED(status): 如果子进程正常终止, 则返回 true

WEXITSTATUS(status): 返回子进程退出状态

WIFSIGNALED(status): 如果子进程被信号终止, 则返回 true

WTERMSIG(status): 返回导致子进程终止的信号编号

WCOREDUMP(status): 如果子进程终止时产生了核心转储文件, 则返回 true

pid: 表示需要等待的哪个子进程, 关于参数 pid 的取值范围如下:
pid=0, 表示等待任何子进程 (父进程) 的所有子进程;
pid=-1, 则等待任何子进程 (父进程) 的所有子进程;
pid=-1, 则等待任何子进程 (父进程) 的所有子进程;
pid=-1, 则等待任何子进程 (父进程) 的所有子进程;

options: 位掩码, 可以在 0 个或多个如下标志:
WNOHANG: 如果子进程有未读状态, 则立即返回
WUNTRACED: 除了返回已终止的子进程的状态信息外, 还返回因信号而停止 (暂停运行) 的子进程状态信息
WCONTINUED: 返回那些被 SIGCONT 信号唤醒后运行的子进程的状态信息

若成功则返回已终止的子进程对应的进程号; 失败则返回 -1

若 options=0, 则返回可以判断是否有子进程发生状态改变, 若返回等于 0 表示没有发生改变

waitid()函数

进程

基本定义: 一个可执行程序实例, 操作系统进行资源分配和调度的最小单位

开始与结束: 开始: 由父进程创建, 内核基所有进程的父进程
用户代码的执行: 引导代码和加载器 -> main()函数
结束: return/exit/abort
int atexit(void (*function)(void)); function: 指向注册的函数
返回: 成功返回 0; 失败返回非 0

进程号: 用于唯一标识系统中的某一个进程的正数
获取本进程的进程号: pid_t getpid(void);
获取父进程的进程号: pid_t getppid(void);

执行用户程序: 将新程序加载到某一进程的内存空间, 然后从头执行
int execve(const char *filename, char *const argv[], char *const envp[]); filename: 指向需要载入当前进程空间的程序的路径名
argv: 指定了传递给程序的命令行参数, 以 NULL 结束
envp: 指定了当前程序的环境变量列表, 以 NULL 结束
返回: 成功返回 0; 失败则返回非 0

通过双指针传递argv: int execl(const char *path, const char *arg..., /* (char *) NULL */);
int execlp(const char *path, const char *arg..., /* (char *) NULL */);
int execlpe(const char *path, const char *arg..., /* (char *) NULL */);
可在PATH中查找可执行文件

通过可变参传递argv: int execl(const char *path, const char *arg..., /* (char *) NULL */);
int execlp(const char *path, const char *arg..., /* (char *) NULL */);
int execlpe(const char *path, const char *arg..., /* (char *) NULL */);
可在PATH中查找可执行文件

执行shell命令: int system(const char *command);
command: 需要执行的 shell 命令
返回: 成功返回非 0; 失败则返回非 0

进程状态: 运行、就绪、阻塞、终止

用户进程和孤儿进程: 孤儿进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)
孤儿进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

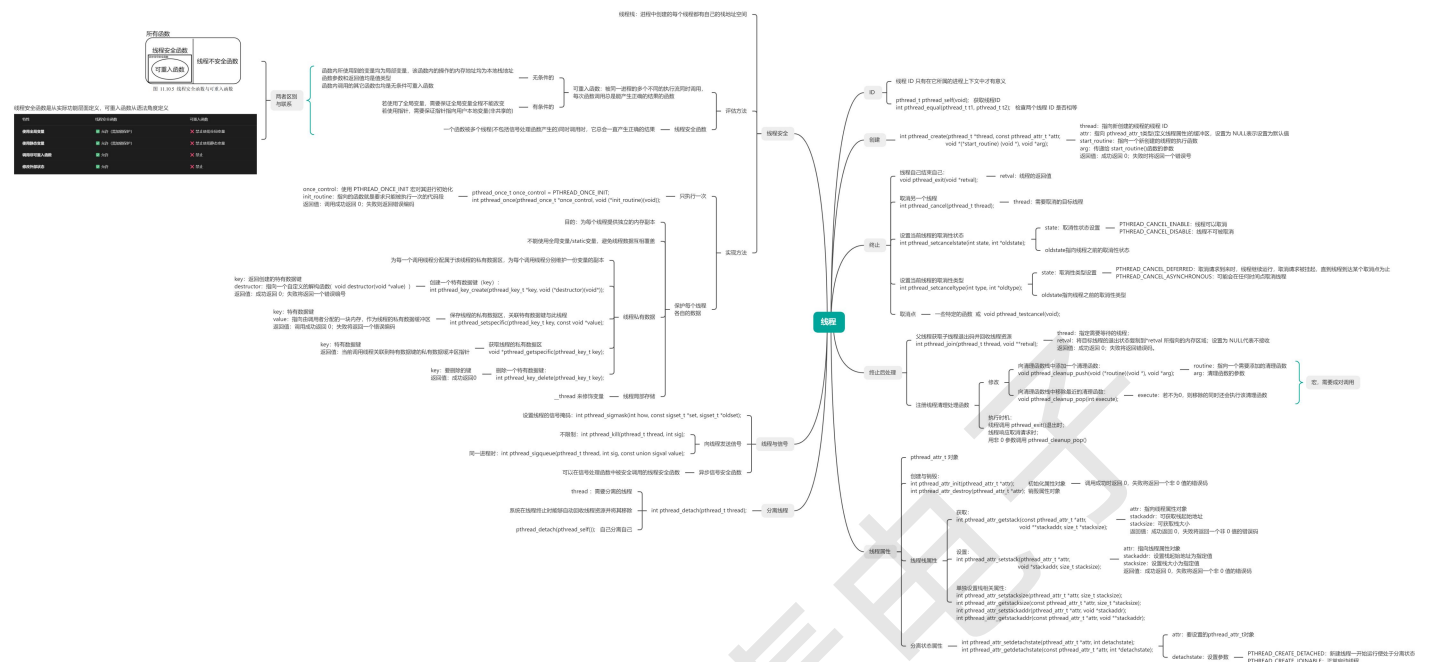
僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

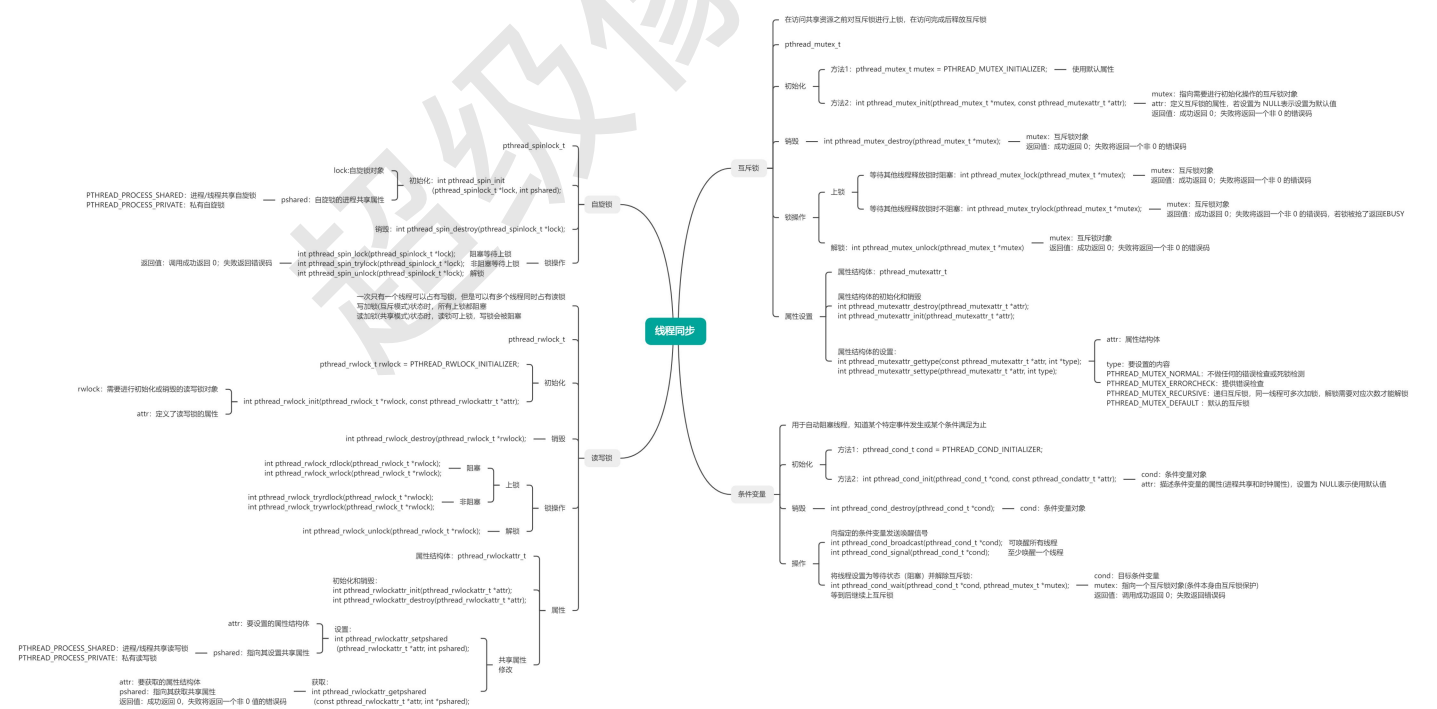
僵尸进程: 僵尸进程: 父进程已终止的子进程 (因为父进程还没有调用 wait()函数为其父)

线程

基本内容



线程同步



代码编写方法(韦东山)

创建线程

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine) (void *), void *arg);

Compile and link with -pthread.
```

```
ret = pthread_create(&tid, NULL, my_thread_func, NULL);
```

```
static void *my_thread_func (void *data)
{
```

信号量

```
// 初始化
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

等待/释放:

```
#include <pthread.h>
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

```
/* 2. 主线程读取标准输入, 发给"接收线程" */
while (1)
```

```
{
    fgets(g_buf, 1000, stdin);

    /* 通知接收线程 */
    sem_post(&g_sem);
}
```

```
static void *my_thread_func (void *data)
{
    while (1)
    {
        //sleep(1);
        /* 等待通知 */
        //while (g_hasData == 0);
        sem_wait(&g_sem);

        /* 打印 */
        printf("recv: %s\n", g_buf);
    }

    return NULL;
}
```

互斥量

```
static pthread_mutex_t g_tMutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_lock(&g_tMutex);
```

```
pthread_mutex_unlock(&g_tMutex);
```

```
/* 2. 主线程读取标准输入，发给"接收线程" */
while (1)
{
    fgets(buf, 1000, stdin);
    pthread_mutex_lock(&g_tMutex);
    memcpy(g_buf, buf, 1000);
    pthread_mutex_unlock(&g_tMutex);

    /* 通知接收线程 */
    sem_post(&g_sem);
}

static void *my_thread_func (void *data)
{
    while (1)
    {
        //sleep(1);
        /* 等待通知 */
        //while (g_hasData == 0);
        sem_wait(&g_sem);

        /* 打印 */
        pthread_mutex_lock(&g_tMutex);
        printf("recv: %s\n", g_buf);
        pthread_mutex_unlock(&g_tMutex);
    }

    return NULL;
}
```

信号+互斥

```
static pthread_cond_t g_tConVar = PTHREAD_COND_INITIALIZER;
```

```
static pthread_mutex_t g_tMutex = PTHREAD_MUTEX_INITIALIZER;
```

```
// 线程A: 等待条件成立
```

```
pthread_mutex_lock(&g_tMutex);
```

```
pthread_cond_wait(&g_tConVar, &g_tMutex);
```

```
/* 操作临界资源 */
```

```
pthread_mutex_unlock(&g_tMutex);
```

```
// 线程B: 唤醒等待g_tConVar的线程
```

```
pthread_cond_signal(&g_tConVar);
```

```

static void *my_thread_func (void *data)
{
    while (1)
    {
        //sleep(1);
        /* 等待通知 */
        //while (g_hasData == 0);
        pthread_mutex_lock(&g_tMutex);
        pthread_cond_wait(&g_tConVar, &g_tMutex);

        /* 打印 */
        printf("recv: %s\n", g_buf);
        pthread_mutex_unlock(&g_tMutex);
    }

    return NULL;
}

/* 2. 主线程读取标准输入，发给"接收线程" */
while (1)
{
    fgets(buf, 1000, stdin);
    pthread_mutex_lock(&g_tMutex);
    memcpy(g_buf, buf, 1000);
    pthread_cond_signal(&g_tConVar); /* 通知接收线程 */
    pthread_mutex_unlock(&g_tMutex);
}

```

一些讨论

1.关于 main 函数和代码段的讨论

- (1)一个 main 函数本身既不是线程也不是进程，但它是程序执行的起点
- (2)一段正在执行的普通 C 语言代码（未涉及 IO 和系统调用）是由线程执行的，而不是进程

2.关于进程和线程区别的讨论

- (1)一个程序运行时是一个进程，如果这个程序创建了多个线程，那么操作系统调度器会尽可能把这些线程分配到不同的 CPU 核上并行运行，从而加速程序执行
- (2)为什么同时有线程和进程的区分：为了解决不同维度的问题，进程解决"安全隔离"问题，线程解决"资源高效"问题。
- (3)程序运行时，进程为程序运行创建一个独立的、安全的沙箱环境，线程在进程环境中创建一条新的执行流

工作内容	进程	线程
内存管理	创建独立的地址空间	共享进程地址空间
资源分配	分配文件表、信号表等	共享进程资源
栈分配	主线程栈+可能的线程栈	仅分配私有栈
ID分配	分配新的PID	分配线程ID（进程内唯一）
上下文切换	切换地址空间+寄存器	仅切换寄存器
隔离建立	完整的沙箱隔离	有限的栈隔离

3.关于进程线程和 FreeRTOS 多任务的讨论

特性	FreeRTOS任务	线程	进程
内存隔离	无（或有限）	共享进程空间	完全隔离
切换开销	很小（几十周期）	较小	较大
通信方式	队列、信号量、全局变量	共享内存、锁	IPC机制
资源管理	简单，静态分配为主	进程管理	操作系统全面管理
适用场景	资源受限嵌入式系统	桌面/服务器应用	需要隔离的应用

特性	FreeRTOS 任务	线程	进程
内存空间	全局共享，无保护	共享进程空间	完全独立
硬件要求	无MMU，简单CPU	需要MMU支持	需要MMU支持
切换开销	很小（保存寄存器）	中等	很大（切换页表等）
通信方式	全局变量、队列等	共享内存+锁	IPC（管道、消息等）
错误影响	一个任务出错可能破坏整个系统	可能影响同进程的其他线程	隔离，不影响其他进程

1. 什么是“任务”（Task）？—— FreeRTOS 的层面

在 FreeRTOS 中，“任务”是调度器直接管理和调度的基本执行单元。它非常“裸”：

2. 为什么 FreeRTOS 没有实现“进程”概念？

因为“进程”的核心是内存保护和资源隔离，而这在单片机上要么不可能，要么代价太高。

3. 为什么 FreeRTOS 也没有实现“线程”概念？

因为“线程”是建立在“进程”概念之上的。

线程与进程的关系：

复制

```
操作系统
├─ 进程A（独立的地址空间）
│   └─ 线程1（共享进程A的资源）
│   └─ 线程2（共享进程A的资源）
│   └─ 线程3（共享进程A的资源）
└─ 进程B（独立的地址空间）
    └─ 线程4（共享进程B的资源）
    └─ 线程5（共享进程B的资源）
```

“线程”的存在前提是“进程”提供的隔离环境。FreeRTOS 既然没有实现进程概念，自然也就没有线程概念。

4.关于 MMU 的讨论

您的理解是对的：虚拟地址空间确实可以比物理内存大得多，而且如果所有虚拟地址都写满数据，物理内存肯定不够。但神奇之处在于，操作系统和MMU通过协作，确保这种情况永远不会发生。

您担心的“数据丢失”问题通过以下机制避免：

1. **虚拟地址 ≠ 物理内存占用**：虚拟地址空间只是潜在的可用的范围
2. **按需分页**：只有实际访问的页面才分配物理内存
3. **页面置换**：物理内存满时，不用的页面被换出到硬盘
4. **安全机制**：极端情况下终止进程而非丢失数据

MMU的魔法在于：它让每个程序都感觉自己拥有巨大的、连续的内存空间，而操作系统在背后智能地管理有限的物理资源，通过硬盘扩展有效内存容量。

5.关于使用局部变量使得函数变成线程安全函数的讨论

在使用特有数据键保护 buf 使得函数变成线程安全函数的代码(stderr 函数)中：

(1)为什么不直接让 buf 变成局部变量使得这个线程变成线程安全函数

核心答案：

不能使用局部变量 buf 来实现线程安全，因为 strerror 函数的返回值是一个指向字符串的指针(char *)。如果 buf 是局部变量，它将在函数返回时被销毁，其内存空间会被回收，导致调用者拿到的指针指向一个无效的（已被释放或即将被覆盖的）内存地址，从而引发程序崩溃或数据错误。

```
static char *strerror(int errnum)
{
    char buf[MAX_ERROR_LEN]; // 错误！这是栈上的局部变量

    if (errnum < 0 || errnum >= _sys_nerr || NULL == _sys_errlist[errnum])
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", errnum);
    else {
        strncpy(buf, _sys_errlist[errnum], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0';
    }
    return buf; // 致命错误！返回了一个指向即将失效的局部变量的指针
}
```

(2)为什么不能使用 static *buf

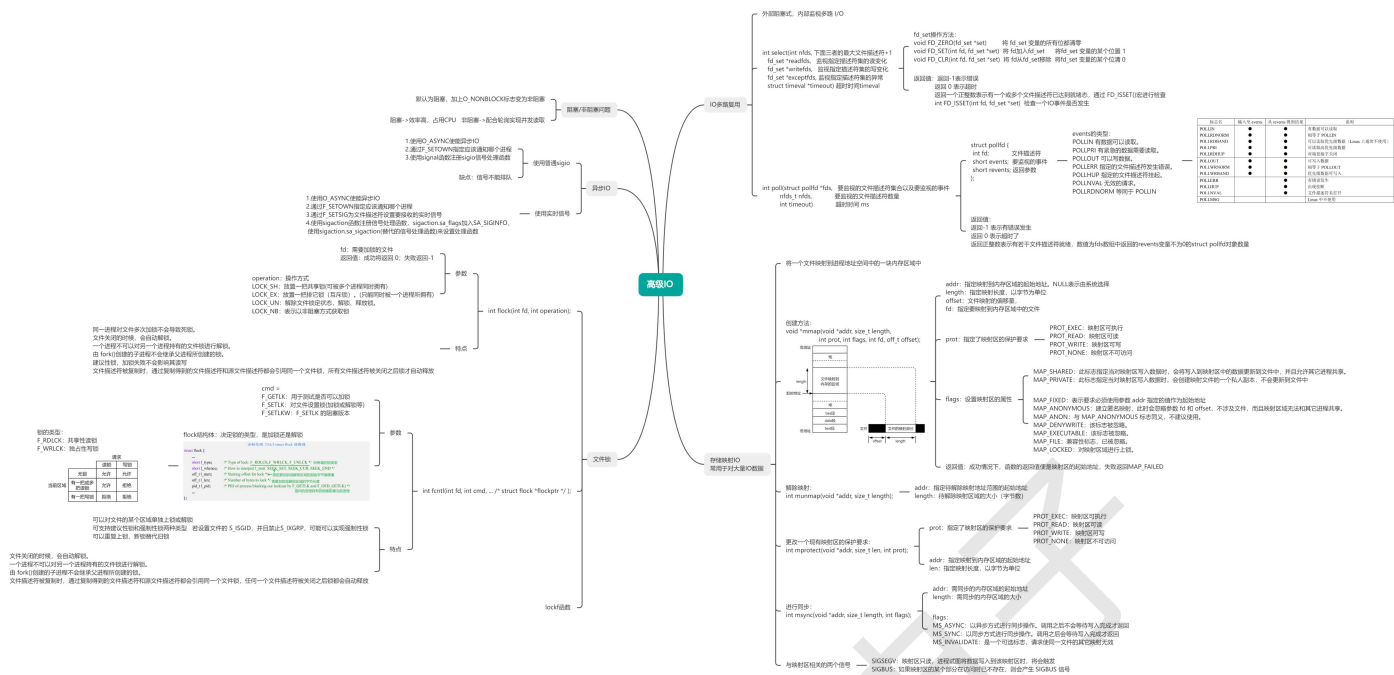
静态变量在进程的所有线程间共享。这意味着：

- 线程A调用 strerror(1)，开始向 buf 写入 "Operation not permitted"
- 同时，线程B调用 strerror(2)，开始向同一个 buf 写入 "No such file or directory"
- 两个线程同时读写同一块内存，结果不可预测：
 - 可能得到混合字符串："Operatio such file or directory"
 - 可能程序崩溃
 - 可能某个线程的写入被完全覆盖

线程特有数据的关键优势：

- 每个线程通过相同的键 (strerror_key) 访问数据
- 但每个线程获取到的是自己独有的数据指针
- 系统自动维护这种“键→线程私有数据”的映射关系

Mmap 的使用(韦东山)



Mmap 的使用(韦东山)

头文件：

```
#include <sys/mman.h>
```

函数原型:

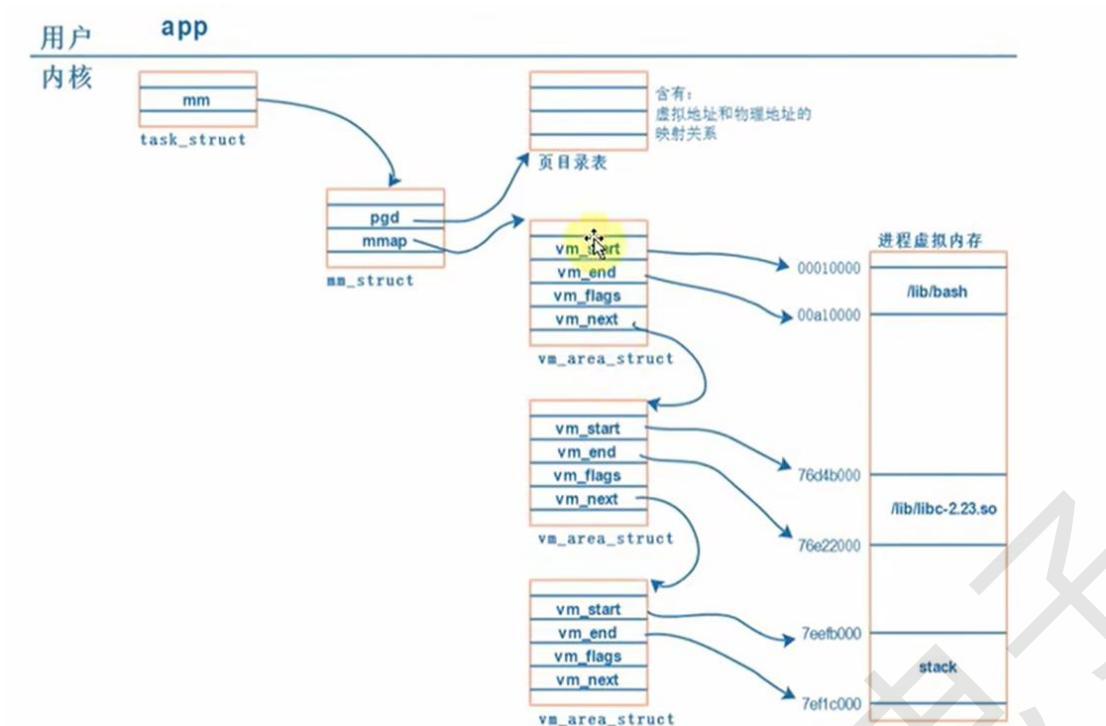
```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

函数说明:

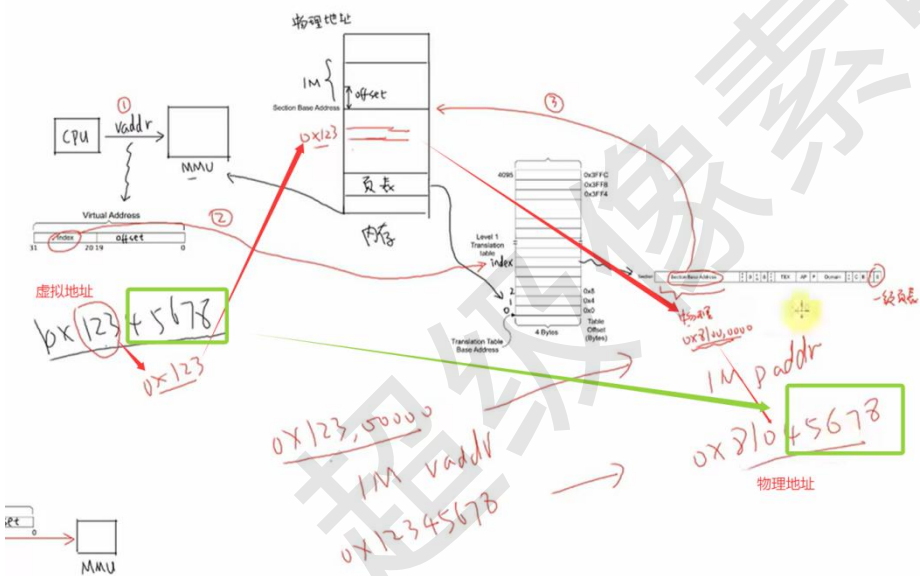
- ① **addr** 表示指定映射的内存起始地址，通常设为 **NULL** 表示让系统自动选定地址，并在成功映射后返回该地址；
- ② **length** 表示将文件中多大的内容映射到内存中；
- ③ **prot** 表示映射区域的保护方式，可以为以下 4 种方式的组合
 - **PROT_EXEC** 映射区域可被执行
 - **PROT_READ** 映射区域可被读出
 - **PROT_WRITE** 映射区域可被写入

- **PROT_NONE** 映射区域不能存取
- ④ **Flags** 表示影响映射区域的不同特性，常用的有以下两种
 - **MAP_SHARED** 表示对映射区域写入的数据会复制回文件内，原来的文件会改变。
 - **MAP_PRIVATE** 表示对映射区域的操作会产生一个映射文件的复制，对此区域的任何修改都不会写回原来的文件内容中。
- ⑤ **返回值**：若成功映射，将返回指向映射的区域的指针，失败将返回-1。

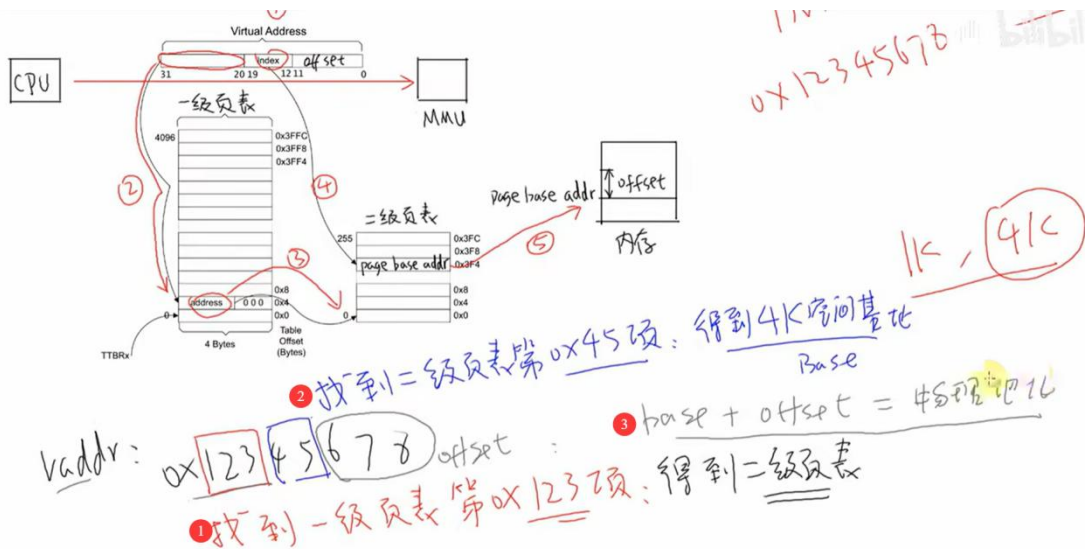
Mmap 的原理(韦东山)



一级页表，一次映射 1M



二级页表，在这里一次映射 4k



使用 mmap 时, 需要有 cache、buffer 的知识。下图是 CPU 和内存之间的关系, 有 cache、buffer(写缓冲器)。Cache 是一块高速内存; 写缓冲器相当于一个 FIFO, 可以把多个写操作集合起来一次写入内存。

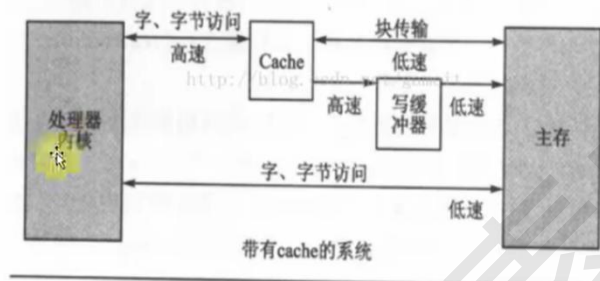
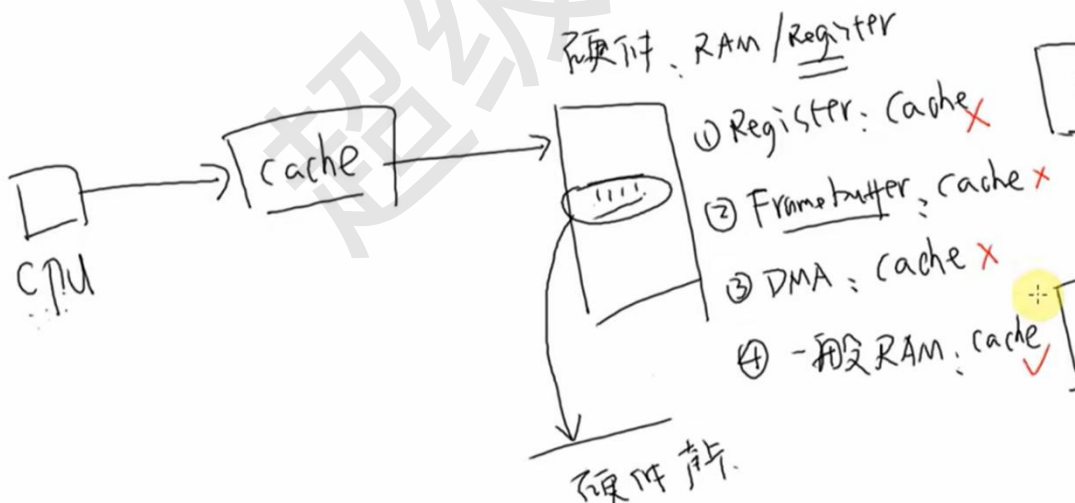


图 12.2 cache、处理器内核及主存之间的关系

程序运行时“局部性原理”, 这又分为时间局部性、空间局部性。

- ① 时间局部性:
在某个时间点访问了存储器的特定位置, 很可能在一小段时间里, 会反复地访问这个位置。
- ② 空间局部性:
访问了存储器的特定位置, 很可能在不久的将来访问它附近的位置。

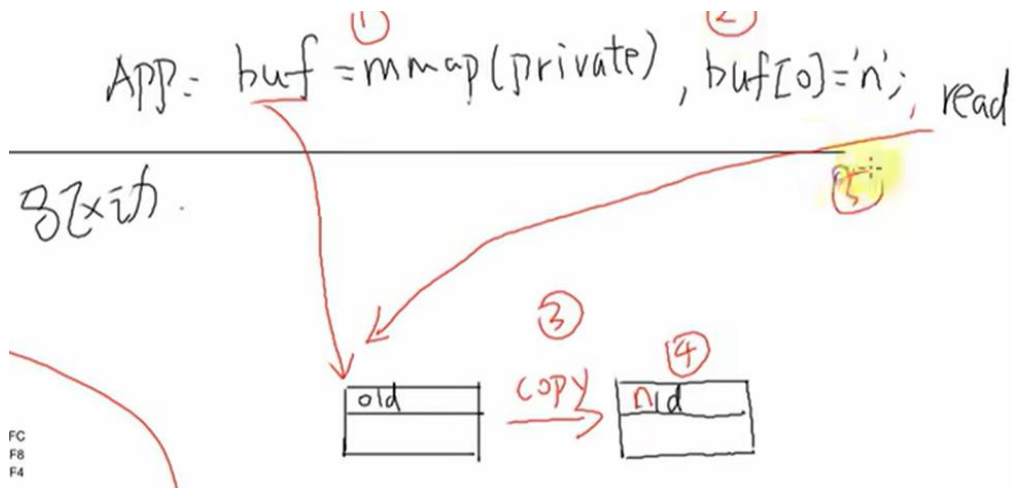


19.9.3.3 驱动程序要做的事

驱动程序要做的事情有 3 点:

- ① 确定物理地址
- ② 确定属性: 是否使用 cache、buffer
- ③ 建立映射关系

使用 private 属性，当 mmap 数值更改后会发生深拷贝



用户程序

```
/* 1. 打开文件 */
fd = open("/dev/hello", O_RDWR);

/* 2. mmap
 * MAP_SHARED : 多个APP都调用mmap映射同一块内存时，对内存的修改大家都可以看到。
 *              就是说多个APP、驱动程序实际上访问的都是同一块内存
 * MAP_PRIVATE : 创建一个copy on write的私有映射。
 *              当APP对该内存进行修改时，其他程序是看不到这些修改的。
 *              就是当APP写内存时，内核会先创建一个拷贝给这个APP，
 *              这个拷贝是这个APP私有的，其他APP、驱动无法访问。
 */
buf = mmap(NULL, 1024*8, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

```
/* 3. write */
strcpy(buf, "new");
```

```
/* 4. read & compare */
/* 对于MAP_SHARED映射: str = "new"
 * 对于MAP_PRIVATE映射: str = "old"
 */
read(fd, str, 1024);
```

```
munmap(buf, 1024*8);
close(fd);
```

驱动程序

```
static int __init hello_init(void)
{
    int err;

    kernel_buf = kmalloc(bufsiz, GFP_KERNEL);
    strcpy(kernel_buf, "old");
}
```

```
/* 2. 定义自己的file_operations结构体
static struct file_operations hello_drv = {
    .owner    = THIS_MODULE,
    .open     = hello_drv_open,
    .read     = hello_drv_read,
    .write    = hello_drv_write,
    .release  = hello_drv_close,
    .mmap     = hello_drv_mmap,
};
```


方法 1：放入工具链默认路径(韦东山)

1. 设置编译工具链

```
// 对于 IMX6ULL，命令如下
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-

export PATH=$PATH:/home/book/100ask_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabi-
ihf_sdk-buildroot/bin
```

2. 确定编译工具链是 `arm-buildroot-linux-gnueabi-gcc`，开始编译，先生成 `makefile`，注意下面的

`arm-buildroot-linux-gnueabi` 要改为实际的工具链

如果交叉编译工具链的前缀是 `arm-buildroot-linux-gnueabi-`，比如 `arm-buildroot-linux-gnueabi-gcc`，交叉编译开源软件时，如果它里面有 `configure`，万能命令如下：

```
./configure --host=arm-buildroot-linux-gnueabi --prefix=$PWD/tmp
```

使用 `makefile` 编译

```
make
make install
```

就可以在当前目录的 `tmp` 目录下看见 `bin`，`lib`，`include` 等目录，里面存有可执行程序、库、头文件。

3. 把编译好的头文件、库文件放到工具链目录里（可选），到时候编译自己的测试文件时可以很方便地直接调用源码的库函数

如果你编译的是一个库，请把得到的头文件、库文件放入工具链的 `include`、`lib` 目录里。别的程序要使用这些头文件、库时，会很方便。

工具链里可能有多个 `include`、`lib` 目录，放到哪里去？

执行下面命令来确定目录：

```
echo 'main(){}' | arm-buildroot-linux-gnueabi-gcc -E -v -
```

2. 命令分解解析	
部分	作用
<code>echo 'main(){}'</code>	生成一个最简单的C程序（空main函数）
<code> </code> （管道）	将程序代码传递给编译器
<code>arm-buildroot-linux-gnueabi-gcc</code>	交叉编译器（针对ARM架构）
<code>-E</code>	只进行预处理（不编译、不链接）
<code>-v</code>	详细模式（显示编译过程的详细信息）
<code>-</code>	从标准输入读取源代码（而不是文件）

结果如下：

```

$ testsh
1 book@100ask:~/01_all_series_quickstart/04_嵌入式Linux应用开发基础知识/source/11_input/02_tslib/tslib-1.21$ echo 'main()' | arm-buildroot-linux-gnueabi-gcc -t -v -
2
3 using built-in specs.
4 COLLECT_GCC=/home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/arm-buildroot-linux-gnueabi-hf-gcc.br_real
5 Target: arm-buildroot-linux-gnueabi-hf // 目标架构为 ARM 使用 GNU/Linux 和硬浮点 ABI
6
7 // 编译器配置信息, 指定了工具链的各种选项和参数
8 configured with: ./configure --prefix=/home/book/100ask_imgtool/sdk/buildroot_2020.02.x/output/host ...
9 Thread model: posix // 使用 POSIX 线程模型
10 gcc version 7.5.0 (buildroot 2020.02-geeslab) // gcc 版本信息
11
12 // 下面是编译选项的命令行选项, 包括目标 CPU、浮点 ABI 类型等
13 COLLECT_GCC_OPTIONS='-t -v' '-mcpu=cortex-a7' '-mfloat-abi=hard' '-mfpu=neon-vfpv4' '-mabi=aapcs-linux' '-marm' '-mls-dialect=gnu'
14
15 // 预处理 "cc1" 的实际执行路径
16 /home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./libexec/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/cc1 -t -quiet -v ...
17
18 // 忽略重复或不存在的目录
19 ignoring duplicate directory "/home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./lib/gcc/././lib/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/include"
20 ignoring nonexistent directory "/home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/arm-buildroot-linux-gnueabi-hf/sysroot/usr/local/include"
21 ignoring duplicate directory "/home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./lib/gcc/././lib/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/include-fixed"
22 ignoring duplicate directory "/home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./lib/gcc/././lib/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/././././arm-buildroot-linux-gnueabi-hf/include"
23
24 #include "... 搜索路径开始的地方 (用户指定的头文件路径)"
25 #include "... search starts here:"
26
27 // #include <...> 搜索路径开始的地方 (系统头文件路径)
28 #include <...> search starts here:
29 /home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./lib/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/include // gcc 内置头文件路径
30 /home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./lib/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/include-fixed // 固定包含路径
31 /home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./lib/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/././././arm-buildroot-linux-gnueabi-hf/include // ARM 架构特定头文件路径
32 /home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/arm-buildroot-linux-gnueabi-hf/sysroot/usr/include // 系统头文件路径, 通常用于标准 C/C++ 库
33
34 End of search list.
35
36 // 预处理后的代码输出
37 # 1 "cxdemo.h"
38 # 1 "build-in.h"
39 # 1 "command-line"
40 # 1 "command-line"
41 # 1 "/home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/arm-buildroot-linux-gnueabi-hf/sysroot/usr/include/stdc-predef.h" 1 3 4
42 # 32 "command-line" 2
43 # 1 "cxdemo.h"
44 main()
45
46 // 编译器搜索其他工具的路径
47 COMPILER_PATH=/home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./libexec/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/:...
48
49 // 库文件路径: 编译器链接阶段搜索库文件的位置
50 LIBRARY_PATH=/home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./lib/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/: // gcc 库路径
51 /home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./lib/gcc/ // 其他 gcc 库路径
52 /home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/bin/./lib/gcc/arm-buildroot-linux-gnueabi-hf/7.5.0/././././arm-buildroot-linux-gnueabi-hf/lib/ // ARM 架构库路径
53 /home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/arm-buildroot-linux-gnueabi-hf/sysroot/lib/ // 系统库路径
54 /home/book/100ask_imgtool/sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/arm-buildroot-linux-gnueabi-hf/sysroot/usr/lib/ // 用户库路径

```

它会列出头文件目录、库目录(LIBRARY_PATH)。

复制头文件和库文件:

```

book@100ask:~/11_tslib/tslib-1.21/tmp$ cp include/* /home/book/100ask_stm32mp157_pro-sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/arm-buildroot-linux-gnueabi-hf/sysroot/usr/include
book@100ask:~/11_tslib/tslib-1.21/tmp$
book@100ask:~/11_tslib/tslib-1.21/tmp$
book@100ask:~/11_tslib/tslib-1.21/tmp$ cp lib/*so* -d /home/book/100ask_stm32mp157_pro-sdk/toolchain/arm-buildroot-linux-gnueabi-hf_sdk-buildroot/arm-buildroot-linux-gnueabi-hf/sysroot/usr/lib/

```

4.把库文件放到板子上的/lib 或/usr/lib 目录下

程序在板上运行时, 需要用到板上/lib 或/usr/lib 下的库文件; 程序运行时不需要头文件。

给开发板拷贝可执行文件 bin 和库文件 lib

```

[root@imx6ull:/mnt/11_input/02_tslib/tslib-1.21/tmp]# cp bin/* /bin
[root@imx6ull:/mnt/11_input/02_tslib/tslib-1.21/tmp]# cp lib/*so* -d /lib
[root@imx6ull:/mnt/11_input/02_tslib/tslib-1.21/tmp/lib]# cp ts /lib -rf

```

拷贝配置文件

```

[root@imx6ull:/mnt/11_input/02_tslib/tslib-1.21/tmp/etc]# cp ts.conf /etc/

```

5.运行代码

```

[root@imx6ull:/bin]# ts test mt

```

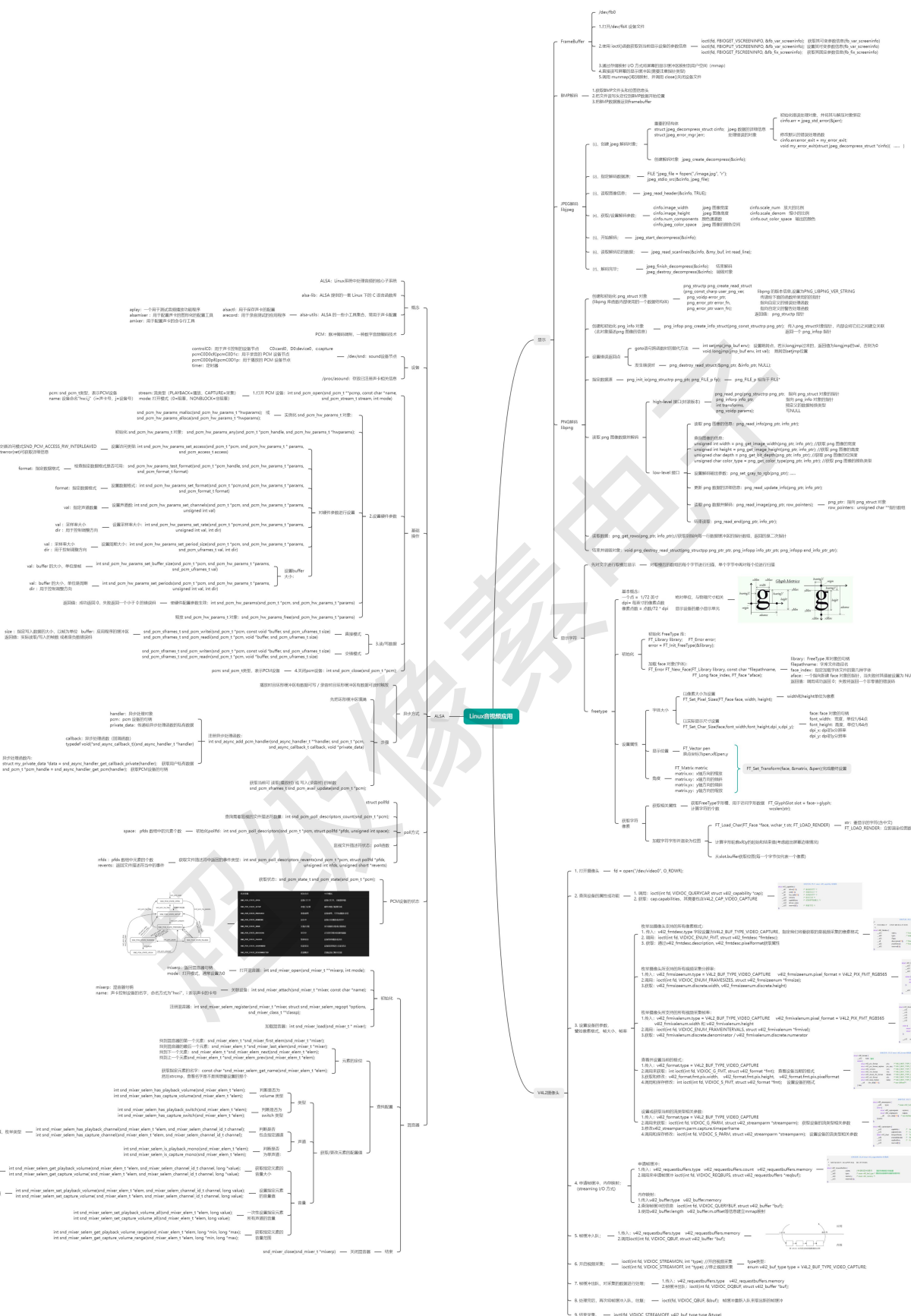
方法 2.自定义路径+编译指定

Eg:

```

freetype_test_rgb888App: freetype_test_rgb888.c
arm-linux-gnueabi-hf-gcc -o freetype_test_rgb888App freetype_test_rgb888.c \
-I/home/alientek/linux/IMX6ULL/c_linux/tools/freetype-2.8_build/include/freetype2 \
-L/home/alientek/linux/IMX6ULL/c_linux/tools/freetype-2.8_build/lib -lfreetype \
-L/home/alientek/linux/IMX6ULL/c_linux/tools/zlib-1.2.10_build/lib -lz \
-L/home/alientek/linux/IMX6ULL/c_linux/tools/libpng-1.6.35_build/lib -lpng -lm

```



1.关于指针变量和数组的讨论
char *a="123" 和 char b[]="123"的区别
char *a="123":

复制

栈区: 常量区:
[a:0x1000] → [0x2000: '1']['2']['3']['\0']
(地址值) (字符串内容)

char b[]="123":

复制

栈区:
[b: '1']['2']['3']['\0']
↑
b直接指向这里，b就是数组的首地址

操作	char *a(指针)	char b[](数组)
声明含义	创建指针变量	分配数组空间
内存占用	指针大小 + 字符串大小	仅字符串大小
地址层级	二级访问 (指针→数据)	一级直接访问
可赋值性	可以改变指向	不能改变

2.原本驱动实验中，编译进内核的 LED 驱动无法被卸载

被编译进内核的驱动不能在内核启动后卸载。

核心区别：内置驱动 vs 模块驱动

内置驱动 (Built-in)

- 编译方式: 直接编译进内核镜像 (vmlinuz)
- 加载时机: 内核启动时自动加载
- 卸载能力: ❌ 无法卸载
- 内存占用: 永久占用内存

模块驱动 (Module)

- 编译方式: 编译成.ko文件
- 加载时机: 可以动态加载/卸载
- 卸载能力: ✅ 可以卸载
- 内存占用: 需要时加载，不需要时可释放

3.在引脚输出实验中，设置 value 要先设置 active_low，否则 value 无法设置

```
/sys/devices/platform/soc/20000000.aips-bus/20ac000.gpio/gpio/gpio129 # echo "low"
" > value
sh: write error: Invalid argument
/sys/devices/platform/soc/20000000.aips-bus/20ac000.gpio/gpio/gpio129 # echo "high"
h" > value
sh: write error: Invalid argument
```

4.为什么不用编写 gpio 驱动

架构演进:

复制

传统方式:

应用层 → 自定义驱动 → GPIO硬件

现代方式:

应用层 → 通用GPIO子系统(sysfs) → GPIO硬件

↑

内核已内置通用驱动

关键技术:

1. 设备树(Device Tree): 描述硬件资源
2. GPIO子系统: 统一管理所有GPIO
3. sysfs虚拟文件系统: 用户空间与内核的桥梁

6. 完整的工作流程

复制

设备树定义GPIO控制器 → 内核注册控制器 → 自动创建所有GPIO引脚 → sysfs接口可用

↓

(可选) 设备树定义具体引脚用途 → 内核保留这些引脚 → 防止其他驱动冲突

7. 为什么你的GPIO129能用?

因为:

1. 设备树定义了GPIO5控制器 (基地址128)
2. 内核自动注册了GPIO5的所有引脚 (128-159)
3. GPIO5_IO01没有被设备树保留 (编号=128+1=129)
4. 所以可以通过sysfs动态申请使用

Tslib

源码分析(韦东山)

```
ts = ts_setup(NULL, 0);

struct tsdev *ts_setup(const char *dev_name, int nonblock)
{
    dev_name = dev_name ? dev_name : getenv("TSLIB_TSDEVICE");
    defname = &ts_name_default[0];
    fname = scan_devices();
    ts = ts_open(fname, nonblock);

    struct tsdev *ts_open(const char *name, int nonblock)
    {
        struct tsdev *ts;
        ts->fd = open(name, flags);
    }

    if (ts && ts_config(ts) != 0) {
        return __ts_config(ts, NULL, NULL, NULL);
    }

    static int ts_config(struct tsdev *ts, char **conf_modules,
                        char **conf_params, int raw)
    {
        conf_modules = strdup(TS_CONF);
        if (strcmp(tok, "module") == 0) {
            ret = ts_load_module(ts, module_name, p);
        } else if (strcmp(tok, "module_raw") == 0) {
            ret = ts_load_module_raw(ts, module_name, p);
        }

        int ts_load_module(struct tsdev *ts, const char *module, const char *params)
        {
            return __ts_load_module(ts, module, params, 0);
        }

        int ts_load_module_raw(struct tsdev *ts, const char *module, const char *params)
        {
            return __ts_load_module(ts, module, params, 1);
        }

        static int __ts_load_module(struct tsdev *ts, const char *module,
                                    const char *params, int raw)
        {
            info = __ts_load_module_shared(ts, module, params);
            ret = __ts_attach(ts, info);
            ret = __ts_attach_raw(ts, info);

            int __ts_attach(struct tsdev *ts, struct tslib_module_info *info)
            {
                info->dev = ts;
                info->next = ts->list;
                ts->list = info;
                return 0;
            }

            ret = ts_read(ts, &samp, 1);

            int ts_read(struct tsdev *ts, struct ts_sample *samp, int nr)
            {
                result = ts->list->ops->read(ts->list, samp, nr);

                static const struct tslib_ops linear_ops = {
                    .read = linear_read,
                    .read_mt = linear_read_mt,
                    .fini = linear_fini
                };

                static int linear_read(struct tslib_module_info *info, struct ts_sample *samp,
                                       int nr_samples)
                {
                    ret = info->next->ops->read(info->next, samp, nr_samples);
                    if (ret < 0) {
                        // 先调用下一级，再调用本
                        // 级，造成了递归的现象
                    }
                }
            }
        }
    }
}
```

/etc/ts.conf示例:

```
module_raw input
module pthres pmin=1
module dejitter delta=100
module linear
```

ts_sample 结构体

示例代码 18.3.1 struct ts_sample 结构体

```
struct ts_sample {
    int x; //X 坐标
    int y; //Y 坐标
    unsigned int pressure; //按压力大小
    struct timeval tv; //时间
};
```

ts_sample_mt 结构体

示例代码 18.3.2 struct ts_sample_mt 结构体

```
struct ts_sample_mt {  
    /* ABS_MT_* event codes. linux/include/uapi/linux/input-event-codes.h  
     * has the definitions.  
     */  
    int      x;          //X 坐标  
    int      y;          //Y 坐标  
    unsigned int pressure; //按压力大小  
    int      slot;       //触摸点 slot  
    int      tracking_id; //ID  
  
    int      tool_type;  
    int      tool_x;  
    int      tool_y;  
    unsigned int touch_major;  
    unsigned int width_major;  
    unsigned int touch_minor;  
    unsigned int width_minor;  
    int      orientation;  
    int      distance;  
    int      blob_id;  
  
    struct timeval tv;          //时间  
  
    /* BTN_TOUCH state */  
    short      pen_down;       //BTN_TOUCH 的状态  
  
    /* valid is set != 0 if this sample  
     * contains new data; see below for the  
     * bits that get set.  
     * valid is set to 0 otherwise  
     */  
    short      valid;          //此次样本是否有效标志 触摸点数据是否发生更新  
};
```

怎么抄官方测试代码写自己的代码

```
ts_test_mtc.c
source > 11_input > 02_tslib > tslib-121 > tests > C ts_test_mtc > @main(int, char **)
136 int main(int argc, char **argv)
162 while (1) {
180 switch (c) {
224 return 0;
225 }
226
227 if (errno) {
228 char str[9];
229 sprintf(str, "option ?");
230 str[7] = c & 0xff;
231 perror(str);
232 }
233
234
235
236 ts = ts_setup(tsdevice, 0);
237 if (!ts) {
238 perror("ts_setup");
239 return errno;
240 }
241 if (verbose && tsdevice)
242 printf("ts_test_mtc: using input device " GREEN "%s" RESET "\n", tsdevice);
243 #ifdef TS_HAVE_EVDEV
244 if (!ioctl(ts_fd(ts), EVIOCGABS(ABS_MT_SLOT), &slot) < 0) {
245 perror("ioctl EVIOCGABS");
246 ts_close(ts);
247 return errno;
248 }
249
250 max_slots = slot.maximum + 1 - slot.minimum;
```

```
mt_cal_distance.c
source > 11_input > 02_tslib > C mt_cal_distance.c > @main(int, char **)
27 int main(int argc, char **argv)
32 struct ts_sample_mt **smp_mt;
33 struct ts_sample_mt **pre_smp_mt;
34 int max_slots;
35 int point_pressed[20];
36 struct input_absinfo slot;
37 int touch_cnt = 0;
38
39
40 ts = ts_setup(NULL, 0);
41 if (!ts) {
42 printf("ts_setup err\n");
43 return -1;
44 }
45
46 if (!ioctl(ts_fd(ts), EVIOCGABS(ABS_MT_SLOT), &slot) < 0) {
47 perror("ioctl EVIOCGABS");
48 ts_close(ts);
49 return errno;
50 }
51
52 max_slots = slot.maximum + 1 - slot.minimum;
53
54 smp_mt = malloc(sizeof(struct ts_sample_mt *));
55 if (!smp_mt) {
56 ts_close(ts);
57 return -ENOMEM;
58 }
59 smp_mt[0] = calloc(max_slots, sizeof(struct ts_sample_mt));
60 if (!smp_mt[0]) {
```

```
ts_test_mtc.c
248 }
249
250 max_slots = slot.maximum + 1 - slot.minimum;
251 #endif
252 if (user_slots > 0)
253 max_slots = user_slots;
254
255 smp_mt = malloc(sizeof(struct ts_sample_mt *));
256 if (!smp_mt) {
257 ts_close(ts);
258 return -ENOMEM;
259 }
260 smp_mt[0] = calloc(max_slots, sizeof(struct ts_sample_mt));
261 if (!smp_mt[0]) {
262 free(smp_mt);
263 ts_close(ts);
264 return -ENOMEM;
265 }
266
267 if (open framebuffer()) {
268 close framebuffer();
269 free(smp_mt[0]);
270 free(smp_mt);
271 ts_close(ts);
272 exit(1);
273 }
```

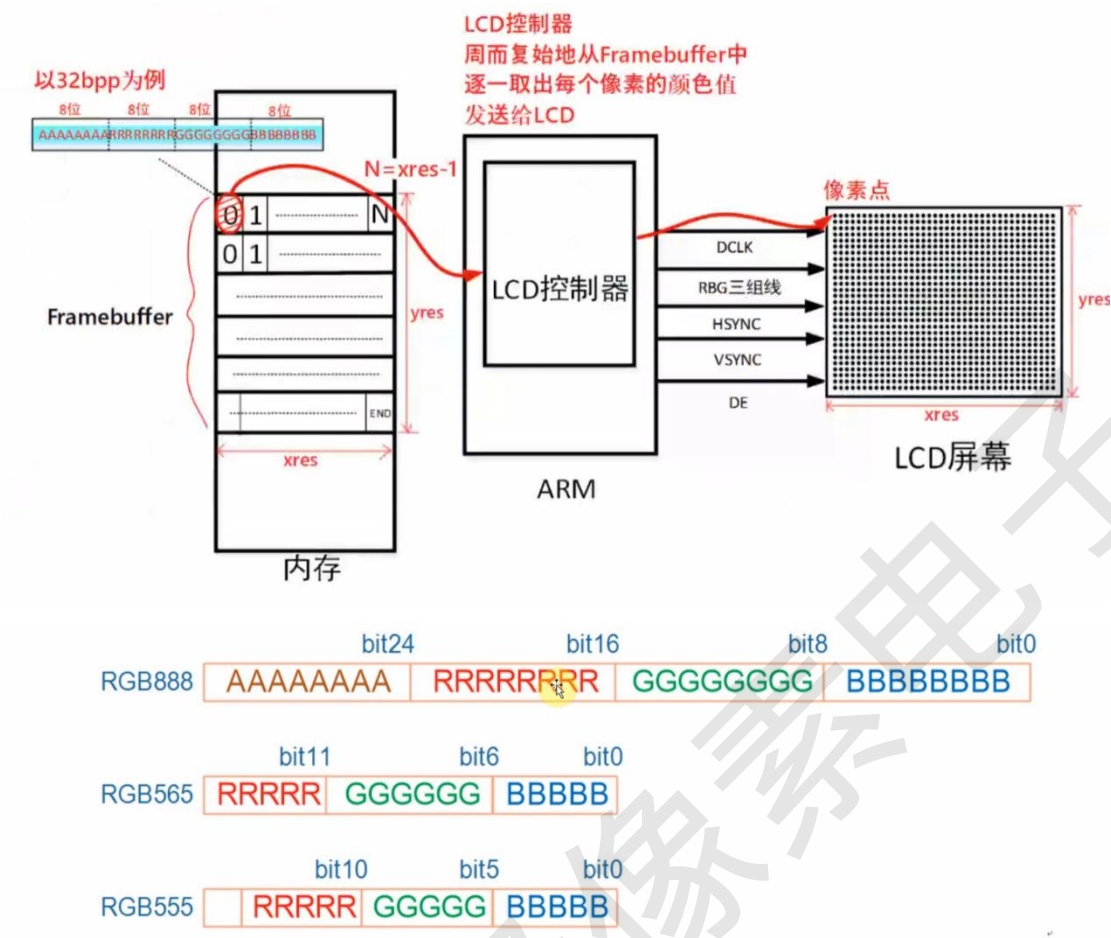
```
mt_cal_distance.c
52 max_slots = slot.maximum + 1 - slot.minimum;
53
54 smp_mt = malloc(sizeof(struct ts_sample_mt *));
55 if (!smp_mt) {
56 ts_close(ts);
57 return -ENOMEM;
58 }
59 smp_mt[0] = calloc(max_slots, sizeof(struct ts_sample_mt));
60 if (!smp_mt[0]) {
61 free(smp_mt);
62 ts_close(ts);
63 return -ENOMEM;
64 }
65
66 pre_smp_mt = malloc(sizeof(struct ts_sample_mt *));
67 if (!pre_smp_mt) {
68 ts_close(ts);
69 return -ENOMEM;
70 }
71 pre_smp_mt[0] = calloc(max_slots, sizeof(struct ts_sample_mt));
72 if (!pre_smp_mt[0]) {
73 free(pre_smp_mt);
74 ts_close(ts);
75 return -ENOMEM;
76 }
77
78 for (i = 0; i < max_slots; i++)
79 pre_smp_mt[0][i].valid = 0;
```

```
ts_test_mtc.c
321 for (j = 0; j < max_slots; j++) {
322 if ((mode & 15) != 1) { /* not in draw mode */
323 continue;
324 }
325
326 ret = ts_read_mt(ts, smp_mt, max_slots, 1);
327
328 /* Hide it */
329 for (j = 0; j < max_slots; j++) {
330 if ((mode & 15) != 1) { /* not in draw mode */
331 if (j > 0 && !(mode_mt[j] & CROSS_SHOW) &&
332 !(mode_mt[j] & CROSS_VISIBLE))
333 continue;
334
335 put_cross(x[j], y[j], 2 | XORMODE);
336 mode_mt[j] &= ~CROSS_VISIBLE;
337 }
338 }
339
340 if (ret < 0) {
341 perror("ts_read_mt");
342 close framebuffer();
343 free(smp_mt);
344 ts_close(ts);
345 exit(1);
346 }
347
348 No Ports Forwarded if (ret != 1)
```

```
mt_cal_distance.c
72 if (!pre_smp_mt[0]) {
73 ts_close(ts);
74 return -ENOMEM;
75 }
76
77
78 for (i = 0; i < max_slots; i++)
79 pre_smp_mt[0][i].valid = 0;
80
81 while (1)
82 {
83 ret = ts_read_mt(ts, smp_mt, max_slots, 1);
84
85 if (ret < 0) {
86 printf("ts_read_mt err\n");
87 ts_close(ts);
88 return -1;
89 }
90
91 for (i = 0; i < max_slots; i++)
92 {
93 if (smp_mt[0][i].valid)
94 memcpy(&pre_smp_mt[0][i], &smp_mt[0][i], sizeof(struct ts_sample_mt));
95 }
96
97 touch_cnt = 0;
```


Framebuffer

原理(韦东山)



操作方法(韦东山)

打开设备

```
fd fb = open("/dev/fb0", 0_RDWR);
```

获得信息

```
if (ioctl(fd_fb, FBIOGET_VSCREENINFO, &var))
// ioctl(input / output control) 是一个用于设备驱动程序的系统调用函数，
// 用来对设备进行各种控制操作。它通常用于在用户空间和设备驱动程序之间进行一些特定的控制或配置操作，
// 适合那些无法通过标准的读写接口完成的操作。
```

映射内存

```
fb_base = (unsigned char *)mmap(NULL, screen_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd_fb, 0);
```

清屏

```
// s: 指向要操作的内存块的指针。
// c: 要设置的值，传入时会被转换为 unsigned char 类型，并填充到内存块中。
// n: 要填充的字节数。
memset(fb_base, 0x00, screen_size);
```

写入数据

```
*pen_8 = color;
```

取消内存映射

```
munmap(fb_base, screen_size);
// munmap 是 Linux / Unix 系统中用于解除之前通过 mmap 函数映射的内存区域的函数。通过 mmap，用户可以将文件或设备映射到内存中，
// 以便直接通过指针访问文件内容，而 munmap 用于取消这种映射，释放资源。
// munmap(void *addr, size_t length);
// 参数说明：
// addr：要解除映射的内存区域的起始地址，通常是通过 mmap 返回的指针。
// length：要解除映射的内存区域的长度，通常与 mmap 时的长度相同。
```

关闭文件

```
close(fd_fb);
```

相关结构体

FrameBuffer 设备的可变参数信息 示例代码 19.3.1 struct fb_var_screeninfo 结构体

```
struct fb_var_screeninfo {
    __u32 xres;          /* 可视区域，一行有多少个像素点，X 分辨率 */
    __u32 yres;          /* 可视区域，一列有多少个像素点，Y 分辨率 */
    __u32 xres_virtual;  /* 虚拟区域，一行有多少个像素点 */
    __u32 yres_virtual;  /* 虚拟区域，一列有多少个像素点 */
    __u32 xoffset;       /* 虚拟到可见屏幕之间的行偏移 */
    __u32 yoffset;       /* 虚拟到可见屏幕之间的列偏移 */

    __u32 bits_per_pixel; /* 每个像素点使用多少个 bit 来描述，也就是像素深度 bpp */
    __u32 grayscale;      /* =0 表示彩色，=1 表示灰度，>1 表示 FOURCC 颜色 */

    /* 用于描述 R、G、B 三种颜色分量分别用多少位来表示以及它们各自的偏移量 */
    struct fb_bitfield red;   /* Red 颜色分量色域偏移 */
    struct fb_bitfield green; /* Green 颜色分量色域偏移 */
    struct fb_bitfield blue;  /* Blue 颜色分量色域偏移 */
    struct fb_bitfield transp; /* 透明度分量色域偏移 */

    __u32 nonstd; /* nonstd 等于 0，表示标准像素格式；不等于 0 则表示非标准像素格式 */
    __u32 activate;

    __u32 height; /* 用来描述 LCD 屏显示图像的高度（以毫米为单位） */
    __u32 width; /* 用来描述 LCD 屏显示图像的宽度（以毫米为单位） */

    __u32 accel_flags;

    /* 以下这些变量表示时序参数 */
    __u32 pixclock; /* pixel clock in ps (pico seconds) */
    __u32 left_margin; /* time from sync to picture */
    __u32 right_margin; /* time from picture to sync */
    __u32 upper_margin; /* time from sync to picture */
    __u32 lower_margin;
    __u32 hsync_len; /* length of horizontal sync */
    __u32 vsync_len; /* length of vertical sync */
    __u32 sync; /* see FB_SYNC_* */
    __u32 vmode; /* see FB_VMODE_* */
    __u32 rotate; /* angle we rotate counter clockwise */
    __u32 colorspace; /* colorspace for FOURCC-based modes */
    __u32 reserved[4]; /* Reserved for future compatibility */
};
```

LCD 的 RGB 像素格式 示例代码 19.3.2 struct fb_bitfield 结构体

```
struct fb_bitfield {
    __u32 offset; /* 偏移量 */
    __u32 length; /* 长度 */
    __u32 msb_right; /* != 0 : Most significant bit is right */
};
```

FrameBuffer 设备的固定参数信息 示例代码 19.3.3 struct fb_fix_screeninfo 结构体

```
struct fb_fix_screeninfo {
    char id[16]; /* 字符串形式的标识符 */
    unsigned long smem_start; /* 显存的起始地址（物理地址） */

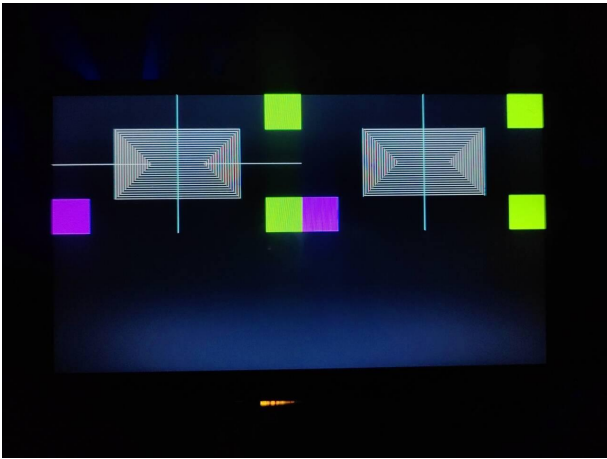
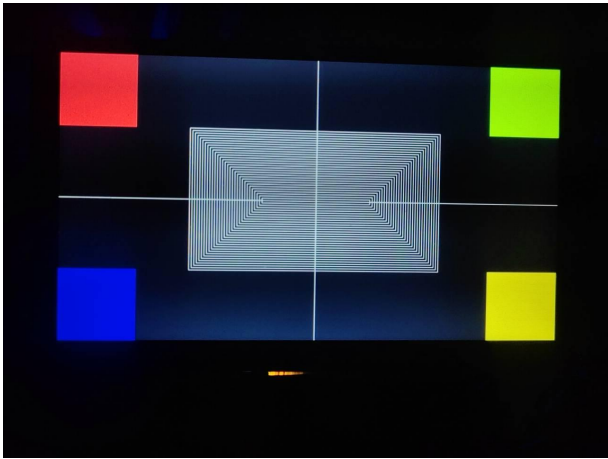
    __u32 smem_len; /* 显存的长度 */
    __u32 type;
    __u32 type_aux;
    __u32 visual;
    __u16 xpanstep;
    __u16 ypanstep;
    __u16 ywrapstep;
    __u32 line_length; /* 一行的字节数 */
    unsigned long mmio_start; /* Start of Memory Mapped I/O(physical address) */
    __u32 mmio_len; /* Length of Memory Mapped I/O */
    __u32 accel; /* Indicate to driver which specific chip/card we have */
    __u16 capabilities;
    __u16 reserved[2];
};
```

一些讨论

1.原本的代码是驱动 800*480 屏幕的，为 RGB565，要改成 1024*600 的，为 ARGB8888

(1)需要修改颜色格式从 RGB565->RGB888

(2)而且还要修改 static unsigned short *screen_base = NULL，把 short 变成 int，否则只能显示一半，并且颜色出错(左图正确，右图错误)



为什么？

①.32 位指针和 16 位指针的区别:

1. 指针类型的关键区别

不是指针本身的大小，而是指针指向的数据类型的大小！

```
c
unsigned int *screen_base; // 指向32位整数 (4字节)
unsigned short *screen_base; // 指向16位整数 (2字节)
```

这两种指针本身都是32位（在32位系统）或64位（在64位系统），区别在于它们指向的数据类型。

1. 内存的本质：字节数组

mmap映射的确实是一个连续的字节数组：

复制

地址：	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
数据：	[00]	[FF]	[00]	[00]	[00]	[00]	[FF]	[00]	[00]	[00]	[00]	[FF]	[FF]	[FF]	[FF]	[FF]
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
	字节	字节	字节	字节	字节	字节	字节	字节	字节	字节	字节	字节	字节	字节	字节	字节

2. 指针类型决定了“读取粒度”

使用32位指针访问：

c 下载 复制 运行

```
unsigned int *p = (unsigned int*)addr;
```

p[0] → 读取地址0-3的4个字节：[00][FF][00][00] → 组成32位值 0x00FF0000
p[1] → 读取地址4-7的4个字节：[00][00][FF][00] → 组成32位值 0x0000FF00
p[2] → 读取地址8-11的4个字节：[00][00][00][FF] → 组成32位值 0x000000FF

使用16位指针访问：

c 下载 复制 运行

```
unsigned short *p = (unsigned short*)addr;
```

p[0] → 读取地址0-1的2个字节：[00][FF] → 组成16位值 0xFF00
p[1] → 读取地址2-3的2个字节：[00][00] → 组成16位值 0x0000
p[2] → 读取地址4-5的2个字节：[00][00] → 组成16位值 0x0000
p[3] → 读取地址6-7的2个字节：[FF][00] → 组成16位值 0x00FF

②.颜色 and 尺寸破坏原因

2. 错误原因与图片的对应关系

问题1：颜色错乱（32位颜色被16位指针拆分）

c

下载

复制

运行

```
// 你的颜色值（32位ARGB8888格式）：  
红色：0x00FF0000 → 被拆成两个16位值  
绿色：0x0000FF00 → 结构破坏  
蓝色：0x000000FF → 数据丢失  
黄色：0x00FFFF00 → 完全错位  
  
// 16位指针的破坏性访问：  
32位像素被拆成2个16位写入，颜色编码完全混乱
```

在第二张图中的体现：

- 红色方块变成了紫色（红色+蓝色的混合效果）
- 黄色方块变成了绿色（黄色中的红色成分丢失）

问题2：内存映射错位（只显示部分屏幕）

c

下载

复制

运行

```
// 32bpp: 1024×600×4 = 2,457,600 字节  
// 16位指针访问：只能正确映射一半的内存区域
```

在第二张图中的体现：

- 中央网格只显示上半部分
- 四个角方块位置偏移、大小异常

问题3：像素结构破坏

每个32位像素被拆分成两个不相关的16位值，导致：

- 颜色分量错位（R、G、B通道混乱）
- 相邻像素数据互相覆盖

2.memset 传参是字节，不是整个 int

memset的函数原型：

c

下载

复制

运行

```
void *memset(void *s, int c, size_t n);
```

关键点：memset是按字节设置内存，不是按像素！

Bmp 显示

Bmp 解码用到的结构体

数据段名称	大小 (Byte)	说明
bmp 文件头 (bmp file header)	14	包含 BMP 文件的格式、大小、到位图数据的偏移量等信息
位图信息头 (bitmap information)	通常为 40 或 56 字节	包含位图信息头大小、图像的尺寸、图像大小、位平面数、压缩方式以及颜色索引等信息
调色板 (color palette)	由颜色索引数决定	可选，如果使用索引来表示图像的颜色，则调色板就是索引与其对应颜色的映射表
位图数据 (bitmap data)	由图像尺寸决定	图像数据

```
typedef struct tagBITMAPFILEHEADER
{
    UINT16 bFileType;
    DWORD bSize;
    UINT16 bReserved1;
    UINT16 bReserved2;
    DWORD biOffset;
} BITMAPFILEHEADER;
```

变量名	地址偏移	大小	作用
bFileType	00H	2 bytes	说明 bmp 文件的类型，可取值为： ①BM - Windows ②BA - OS/2 Bitmap Array ③CI - OS/2 Color Icon ④CP - OS/2 Color Pointer ⑤IC - OS/2 Icon ⑥PT - OS/2 Pointer
bSize	02H	4 bytes	说明该文件的大小，以字节为单位。
bReserved1	06H	2 bytes	保留字段，必须设置为 0。
bReserved2	08H	2 bytes	保留字段，必须设置为 0。
biOffset	0AH	4 bytes	说明从文件起始位置到图像数据之间的字节偏移量。这个参数非常有用，因为位图信息头和调色板的长度会根据不同的情况而变化，所以我们可以用这个偏移量迅速从文件中找到图像数据的偏移地址。

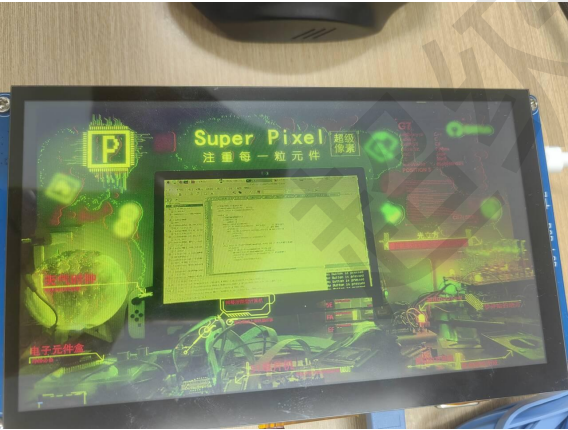
```
typedef struct tagBITMAPINFOHEADER {
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG biXPelsPerMeter;
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
```

变量名	地址偏移	大小	作用
biSize	0EH	4 bytes	位图信息头大小。
biWidth	12H	4 bytes	图像的宽度，以像素为单位。
biHeight	16H	4 bytes	图像的高度，以像素为单位。 注意，这个值除了用于描述图像的高度之外，它还有另外一个用途，用于指明该图像是正向的位图，还是正向的位图。 如果该值是一个正数，说明是正向的位图；如果该值是一个负数，则说明是正向的位图。一般情况下，BMP 图像都是正向的位图，也就是该值是一个正数。
biPlanes	1AH	2 bytes	色彩平面数，该值总被设置为 1。
biBitCount	1CH	2 bytes	像素深度，指明一个像素点需要多少个 bit 数据来描述。其值可为 1、4、8、16、24、32。
biCompression	1EH	4 bytes	说明图像数据的压缩类型，取值范围如下： ①0 - RGB 方式 ②1 - 8bpp 的 RLE 方式，只用于 8bit 位图 ③3 - 4bpp 的 RLE 方式，只用于 4bit 位图 ④5 - Bit-fields 方式 ⑤5 - 仅用于打印机
biSizeImage	22H	4 bytes	说明图像的大小，以字节为单位，当压缩类型为 BI_RGB 时，可设置为 0。
biXPelsPerMeter	26H	4 bytes	水平分辨率，用像素/米来表示，有符号整数。
biYPelsPerMeter	2AH	4 bytes	垂直分辨率，用像素/米来表示，有符号整数。
biClrUsed	2EH	4 bytes	说明位图实际使用的颜色表中的颜色索引数。
biClrImportant	32H	4 bytes	说明对图像显示有重要影响的颜色索引的数目，如果是 0，则表示都重要。

图 19.5.6 正向位图和倒向位图

主要问题

1.lcd 偏色问题，因为 bmp 导出的时候设置错误，导致原本应该显示 ARGB 的图片，显示成了 RGBX，显示效果为红->黑 绿->红 蓝->绿



JPEG 解码

主要问题

1.显示图像的时候改变了 framebuffer 起始地址变量，而 munmap 时使用了修改后的变量，于是段错误

```
memcpy(screen_base, fb_line_buf, valid_bytes);
// printf("line %d success!\n", current_line);
current_line++;
screen_base += width; //+width 定位到LCD下一行显存地址的起点
```

```
/home/apps # ./show_jpeg_image_rgb888App room1.jpg
line_length=4096,bpp=32,screen_size=2457600,width=1024,height=600
mmap succrss,screen_base=1992314880
jpeg图像大小: 1024*600
valid_bytes=4096
buffer show end!
show image end
out of image function
screen_base=1994772480,screen_size=2457600
Segmentation fault
```

2.但是之前的 bmp 实现中，munmap 时也使用了修改后的变量，但不会报错，原因是指针是倒着从结束位置到起始位置附近，才 munmap 的

```
/home/apps # ./bmp_show_rgb888App ./19_lcd/room2b_reverse.bmp
width=1024,height=600
mmap succrss,screen_base=1992847360
文件大小: 2457656
位图数据的偏移量: 54
位图信息头大小: 40
图像分辨率: 1024*600
像素深度: 32
reverse order bmp detected
screen_base=1992843264,screen_size=2457600
```

PNG 解码

主要问题

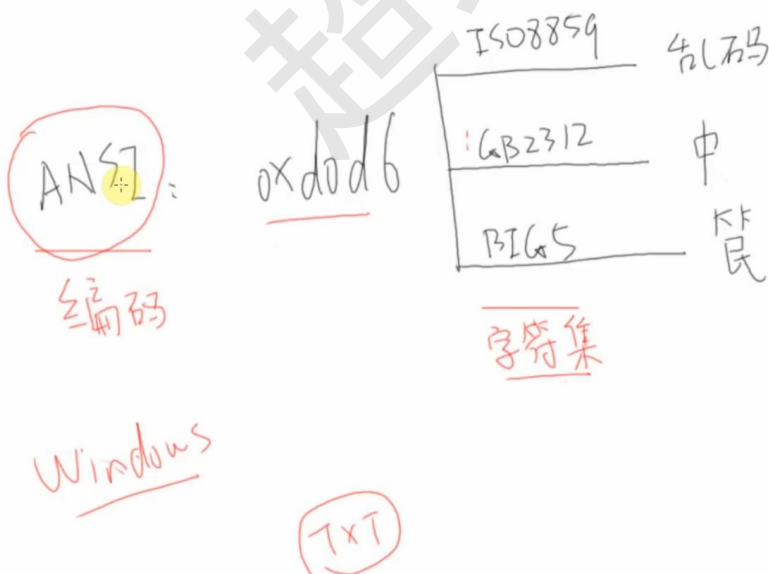
Makefile 中在 -I、-L 和路径之间不能有空格，否则会报错

Freetype

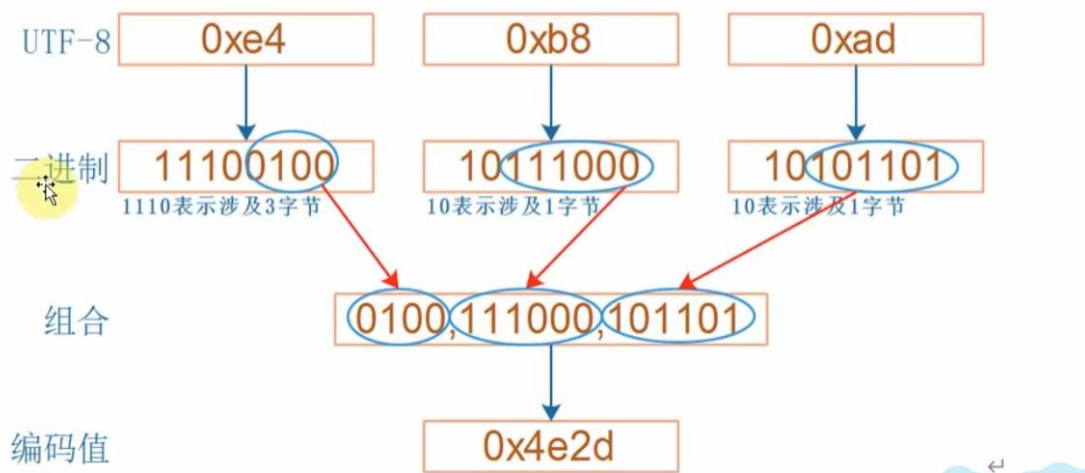
有关字符编码的讨论(韦东山)

1.ASCII: 一个字节的 7 位就可以表示 128 个数值，在 ASCII 码中最高位永远是 0。<

2.ANSI+字符集



3.Unicode



源码框图(韦东山)

```

FT_Library library; // 对应 freetype 库
error = FT_Init_FreeType(&library); // 初始化库

error = FT_New_Face(library, argv[1], 0, &face); // 打开字体文件 argv[1]为字体文件

FT_Face face; // 矢量字体文件
FT_Set_Pixel_Sizes(face, font_size, 0); // 设置字体大小

error = FT_Load_Char(face, wstr[i], FT_LOAD_RENDER);

FT_Glyph glyph; // 有字符的原始关键点信息
slot = face->glyph;
FT_GlyphSlot slot // 一个字符的点阵位图
FT_Set_Transform(face, 0, &pen);
FT_Vector pen; // 坐标点

/* 计算外框 */
compute_string_bbox(face, wstr, &bbox);
FT_BBox bbox; // 表示一个字符的外框

typedef struct FT_BBox_
{
    FT_Pos xMin, yMin;
    FT_Pos xMax, yMax;
} FT_BBox;

/* 在LCD上绘制：使用LCD坐标 */
draw_bitmap( &slot->bitmap,
             slot->bitmap_left,
             var.yres - slot->bitmap_top);

```

主要问题

命令行忘记加参数导致段错误，之后因为把背景色调成黑色而没有把黑色文字调成白色导致字体无法显示

主要配置的相关结构体



```

/home/apps # ./v4l2_cameraApp /dev/video0
Error: /dev/video0: No capture video device!
/home/apps # ./v4l2_cameraApp /dev/video0
Error: /dev/video0: No capture video device!
/home/apps # ls -l /dev/video*
crw-rw----  1 root  0           81,  0 Jan  1 00:00 /dev/video0

```

这个 `/dev/video0` 设备确实存在，但它不是摄像头设备！让我来详细解释它是什么以及如何确认。

1. 这个 `/dev/video0` 是什么？

在你的启动日志中，我找到了关键线索：

```
log
mxsfb 21c8000.lcdif: initialized
Console: switching to colour frame buffer device 128x37
```

这个 `/dev/video0` 是 LCD 显示器的帧缓冲设备 (framebuffer)，不是摄像头！

我用的是我自己移植的根文件系统和 Linux kernel，而正点原子实验用的是出厂根文件系统，我认为有可能是因为我没有编译和安装摄像头驱动导致的，于是照着正点原子出厂的内核来修改我的设备树重新编译，但是还是不行，启动日志无 `ov5640` 相关信息，并且 `/dev` 仅有 `video0`，没有 `video1`

让我仔细检查你的新启动日志，查找 `OV5640` 相关信息。

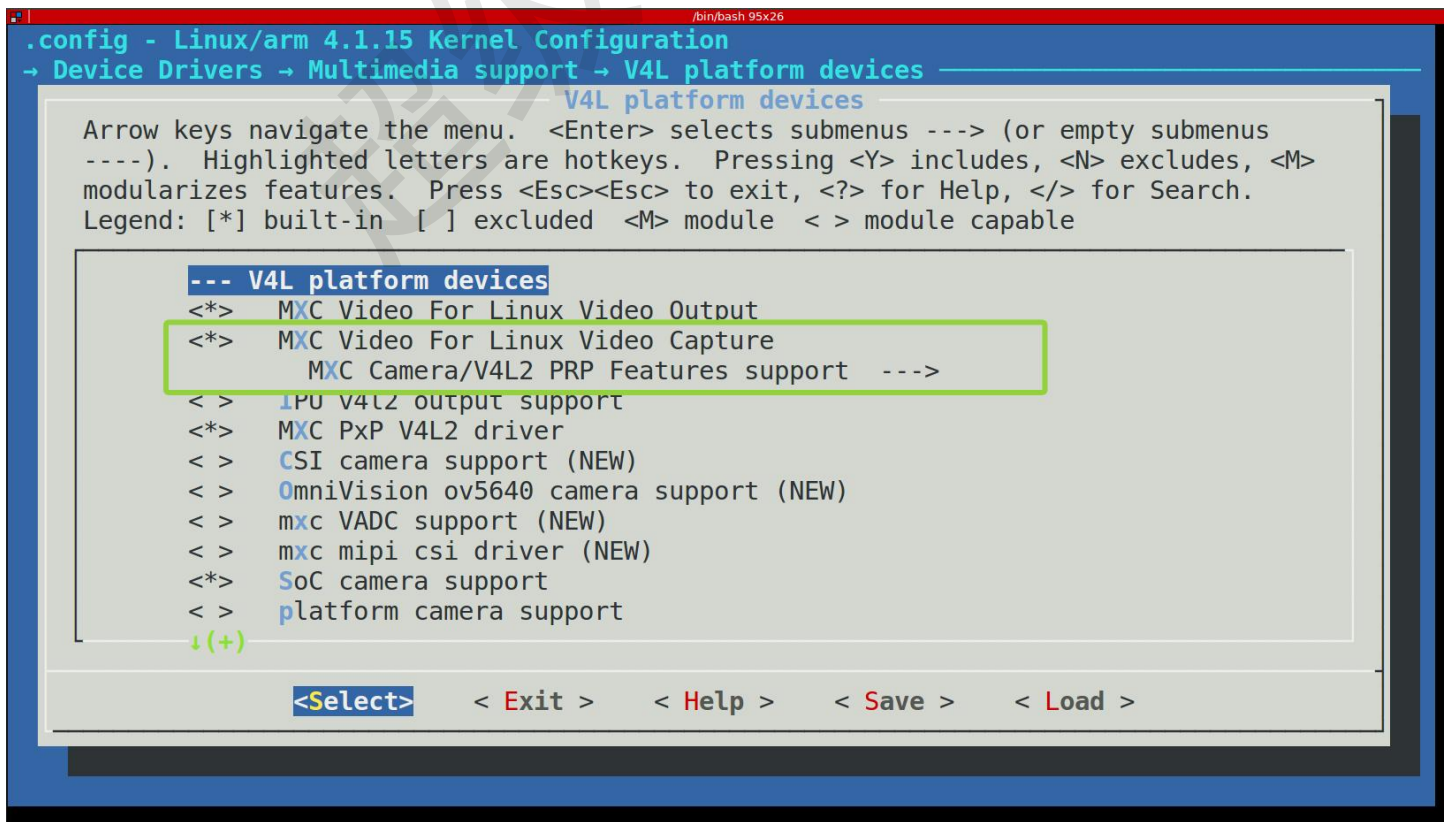
1. 仔细搜索日志

我仔细检查了你的新启动日志，仍然没有发现任何 `OV5640` 相关的信息。以下是详细分析：

搜索关键词结果：

- "ov5640": 0次出现
- "5640": 0次出现
- "camera": 0次出现 (除了 video capture interface 的通用声明)
- "sensor": 0次出现
- "mxc_capture": 0次出现
- "v4l2": 2次出现 (都是通用接口，不是摄像头驱动)

我觉得除了设备树之外，可能还需要处理摄像头驱动问题，于是在网上搜索“正点原子阿尔法 `ov5640` 摄像头”，找到了这个教程 <https://blog.csdn.net/u014754841/article/details/135548242>，开始修改 Linux 内核配置



```
.config - Linux/arm 4.1.15 Kernel Configuration
[...] ers → Multimedia support → V4L platform devices → MXC Camera/V4L2 PRP Features support
MXC Camera/V4L2 PRP Features support
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus
----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M>
modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search.
Legend: [*] built-in [ ] excluded <M> module < > module capable

<*> OmniVision ov5640 camera support
< > OmniVision ov5642 camera support
< > OmniVision ov5640 camera support using mipi
< > Analog Device adv7180 TV Decoder Input support
<*> Select Overlay Routing (Queue ipu device for overlay library) --->
<*> Pre-processor Encoder library
<*> IPU CSI Encoder library

<Select> < Exit > < Help > < Save > < Load >
```

原本为<M>

然后编译内核源码

```
alientek@zdyz-imx6ull: ~/linux/IMX6ULL/my_linux/linux-imx-rel_imx_4.1.15_2.1.0_ga$ make -j4
scripts/kconfig/conf --silentoldconfig Kconfig
CHK include/config/kernel.release
CHK include/generated/uapi/linux/version.h
CHK include/generated/utsrelease.h
make[1]: 'include/generated/mach-types.h' is up to date.
CHK include/generated/bounds.h
CHK include/generated/asm-offsets.h
CALL scripts/checksyscalls.sh
CHK include/generated/compile.h
CC drivers/media/platform/mxc/capture/ipu_fg_overlay_sdc.o
CC drivers/media/platform/mxc/capture/ipu_bg_overlay_sdc.o
LD drivers/media/platform/mxc/capture/built-in.o
LD drivers/media/platform/built-in.o
LD drivers/media/built-in.o
LD drivers/built-in.o
LINK vmlinux
LD vmlinux.o
MODPOST vmlinux.o
GEN .version
CHK include/generated/compile.h
UPD include/generated/compile.h
CC init/version.o
LD init/built-in.o
```

启动后出现如下信息，发现是 GPIO 驱动出现冲突

```
imx6ul-pinctrl 20e0000.iomuxc: could not request pin 27 (MX6UL_PAD_GPIO1_I004) from group csi_pwn_rstgrp on device 20e0000.iomuxc
ov5640 1-003c: Error applying setting, reverse things back
imx6ul-pinctrl 20e0000.iomuxc: pin MX6UL_PAD_GPIO1_I004 already requested by 2088000.pwm; cannot claim for 1-003c
imx6ul-pinctrl 20e0000.iomuxc: pin-27 (1-003c) status -22
imx6ul-pinctrl 20e0000.iomuxc: could not request pin 27 (MX6UL_PAD_GPIO1_I004) from group csi_pwn_rstgrp on device 20e0000.iomuxc
ov5640 1-003c: Error applying setting, reverse things back
ov5640 1-003c: setup pinctrl failed
ov5640: probe of 1-003c failed with error -22
```


太好了！你的内核启动日志中确实出现了 OV5640 的相关信息，但是有一个关键的错误！

1. 关键信息分析

从你的启动日志中，我找到了这些关键行：

log

复制

```
imx6ul-pinctrl 20e0000.iomuxc: pin MX6UL_PAD_GPIO1_I004 already requested by 2088000.pwm; cannot claim for 1-003c
imx6ul-pinctrl 20e0000.iomuxc: pin-27 (1-003c) status -22
imx6ul-pinctrl 20e0000.iomuxc: could not request pin 27 (MX6UL_PAD_GPIO1_I004) from group csi_pwn_rstgrp on device 20e0000.io
ov5640 1-003c: Error applying setting, reverse things back
ov5640 1-003c: setup pinctrl failed
ov5640: probe of 1-003c failed with error -22
```

2. 问题诊断

核心问题：GPIO 引脚冲突

- 冲突引脚：MX6UL_PAD_GPIO1_I004 (GPIO1_4)
- 冲突设备：2088000.pwm (PWM 设备) 和 1-003c (OV5640 摄像头)
- 错误原因：同一个 GPIO 引脚被两个设备同时使用

于是修改设备树，禁用了冲突的 pwm 驱动

```
arch > arm > boot > dts > my_device.dts
757     MX6UL_PAD_UART2_CTS_B_ECSP13_MOSI 0x10b1 /* MOSI */
758     >;
759 };
760
761 pinctrl_uart3: uart3grp {
762     fsl,pins = <
763         MX6UL_PAD_UART3_RX_DATA_UART3_DCE_RX 0x1b0b1
764         MX6UL_PAD_UART3_TX_DATA_UART3_DCE_TX 0x1b0b1
765     >;
766 };
767
768 pinctrl_tsc: tscgrp {
769     fsl,pins = <
770         /* MX6UL_PAD_GPIO1_I009_GPIO1_I009 0xf080 /* TSC_INT for FT5426 */
771         MX6UL_PAD_GPIO1_I009_GPIO1_I009 0x79 /* TSC_INT for GT911 */
772     >;
773 };
774
775 // not enable bucause for ov5640
776 // pinctrl_pwm3: pwm3grp {
777 //     fsl,pins = <
778 //         MX6UL_PAD_GPIO1_I004_PWM3_OUT 0x110b0
779 //     >;
780 // };
781
782 //zdyz for ov5640
783 csi_pwn_rst: csi_pwn_rstgrp {
784     fsl,pins = <
785         MX6UL_PAD_GPIO1_I002_GPIO1_I002 0x10b0
786         MX6UL_PAD_GPIO1_I004_GPIO1_I004 0x10b0
787     >;
788 };
789
790 };
791
792
793
```



```
arch > arm > boot > dts > my_device.dts
1011 };
1012
1013 &ecspi3 {
1014     fsl,spi-num-chipselects = <1>;
1015     cs-gpios = <&gpio1 20 GPIO_ACTIVE_LOW>;
1016     pinctrl-names = "default";
1017     pinctrl-0 = <&pinctrl_ecspi3>;
1018     status = "okay";
1019
1020     spidev: icm20608@0 {
1021         compatible = "alientek,icm20608";
1022         spi-max-frequency = <8000000>;
1023         reg = <0>;
1024     };
1025 };
1026 // not enable because for ov5640
1027 // &pwm3 {
1028 //     pinctrl-names = "default";
1029 //     pinctrl-0 = <&pinctrl_pwm3>;
1030 //     clocks = <&clks IMX6UL_CLK_PWM3>,
1031 //             <&clks IMX6UL_CLK_PWM3>;
1032 //     status = "okay";
1033 // };
1034
1035 &adc1 {
1036     pinctrl-names = "default";
1037     pinctrl-0 = <&pinctrl_adc1>;
1038     num-channels = <2>;
1039     vref-supply = <&reg_vref_adc>;
1040     status = "okay";
1041 };
```

出现下面信息

```
camera ov5640, is found
```

```
/dev # ls |grep video
video0
```

从你的启动日志中，我看到了好消息和坏消息：

好消息 ✓

OV5640 摄像头已经被成功识别了！

log

复制

```
camera ov5640, is found # 摄像头被成功检测到！
```

坏消息 ✗

但是摄像头没有创建 /dev/video 设备节点！

于是我问正点原子客服

2025-10-16 10:23:15 tb197729344



我尝试着往我自己的根文件系统里面移植ov 5640驱动，但是一直都没有成功，你们这边能提供这类的技术支持吗？需要额外付费的也可以

已读



正点原子旗舰店:linux技术小志 2025-10-16 10:23:50

网上有教程的

更换关键词“正点原子 Linux ov5640 摄像头”，找到这个教程 https://blog.csdn.net/Liang_z_p/article/details/143766413，编译了驱动，为了防止和之前我在内核启用的驱动冲突，我把内核 menuconfig 改成了原本的设置，modprobe 加载新编译的驱动后，dev 底下终于出现了 video1，可是加载应用程序时，出现了

```
/home/apps # ./v4l2_cameraApp /dev/video1
format<0x56595559>, description<Yunknown pixelformat:'RGBP'
UYV-16>
size<640*480> <15fps><30mx6s-csi 21c4000.csi: Fourcc format (0x50424752) invalid.
fps>
size<320*240> <15fps><30fps>
size<720*480> <15fps><30fps>
size<720*576> <15fps><30fps>
size<1280*720> <15fps><30fps>
size<1920*1080> <15fps>
size<2592*1944> <15fps>
size<176*144> <15fps><30fps>
size<1024*768> <15fps><30fps>

ioctl error: VIDIOC_S_FMT: Invalid argument
```

从你的输出和代码来看，问题很明确：**摄像头不支持 RGB565 格式**，但你的代码强制设置了 RGB565 格式。

1. 问题分析

从你的输出可以看到：

```
format<0x56595559>, description<Yunknown pixelformat:'RGBP'
UYV-16>
```

摄像头实际支持的格式是 V4L2_PIX_FMT_UYVY (YUYV 4:2:2)，但你的代码强制设置：

```
c
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_RGB565; // 强制设置为RGB565
```

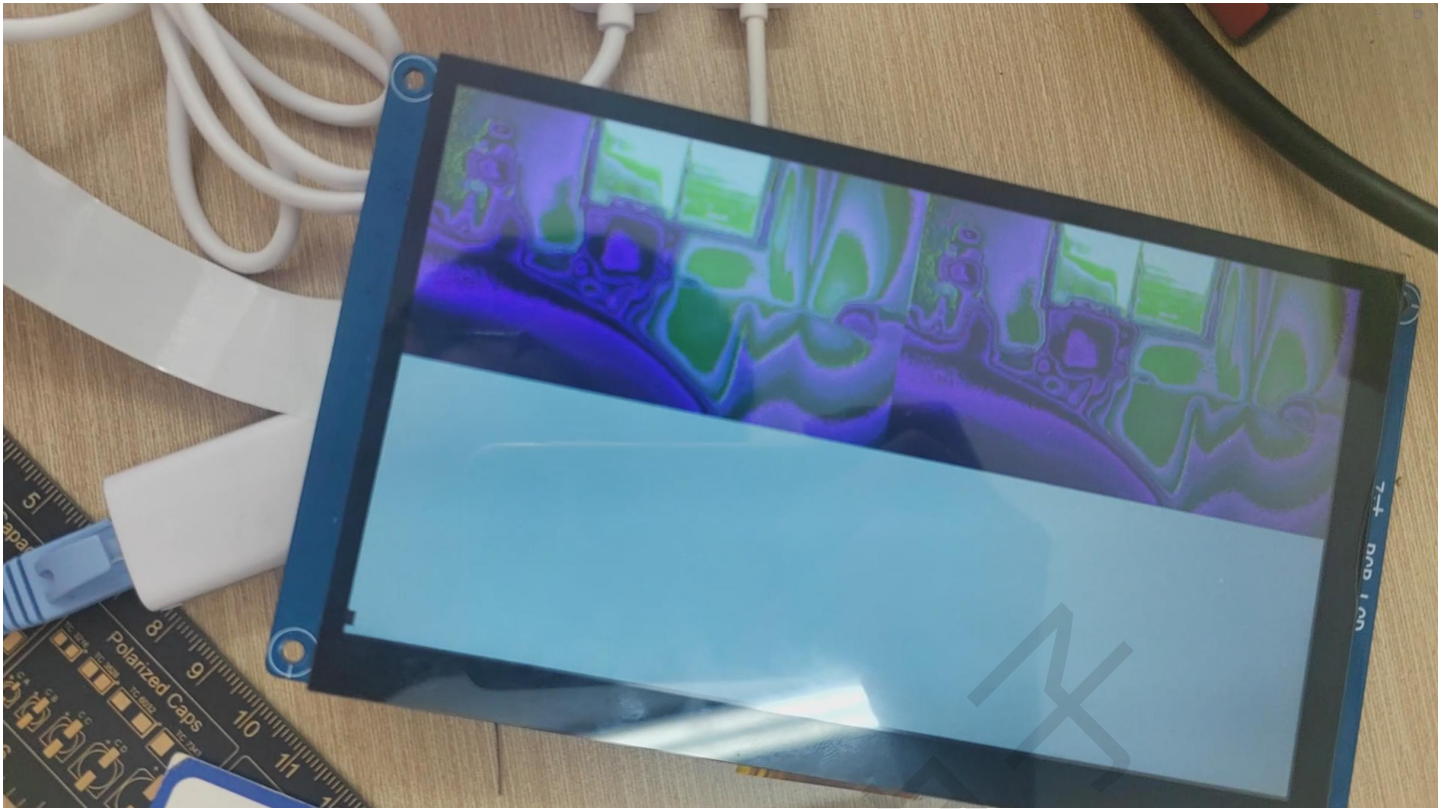
可是在新教程里面，笔者运行的结果含有 RGB565，我忽然想起来我只是按照教程的文字说明找到对应的内核摄像头相关文件去编译，其文件内容可能不一样，于是我找到了教程里面提到的开源链接 <https://gitee.com/peng-lzp/ov5640>，重新编译代码，结果如下

```
/home/apps # ./v4l2_camera_internetApp /dev/video1
format<0x50424752>, description<RGB565_LE>
size<640*480> <15fps><30fps>
size<320*240> <15fps><30fps>
size<720*480> <15fps><30fps>
size<720*576> <15fps><30fps>
size<1280*720> <15fps><30fps>
size<1920*1080> <15fps>
size<2592*1944> <15fps>
size<176*144> <15fps><30fps>
size<1024*768> <15fps><30fps>
size<480*272> <15fps><30fps>
size<800*480> <15fps><30fps>
size<1024*600> <15fps><30fps>
size<1280*800> <15fps><30fps>

format<0x4745504a>, description<JPEG>
size<640*480> <15fps><30fps>
size<320*240> <15fps><30fps>
size<720*480> <15fps><30fps>
size<720*576> <15fps><30fps>
size<1280*720> <15fps><30fps>
size<1920*1080> <15fps>
size<2592*1944> <15fps>
size<176*144> <15fps><30fps>
size<1024*768> <15fps><30fps>
size<480*272> <15fps><30fps>
size<800*480> <15fps><30fps>
size<1024*600> <15fps><30fps>
size<1280*800> <15fps><30fps>

format<0x56595559>, description<YUYV-16>
size<640*480> <15fps><30fps>
size<320*240> <15fps><30fps>
size<720*480> <15fps><30fps>
size<720*576> <15fps><30fps>
size<1280*720> <15fps><30fps>
size<1920*1080> <15fps>
size<2592*1944> <15fps>
size<176*144> <15fps><30fps>
size<1024*768> <15fps><30fps>
size<480*272> <15fps><30fps>
size<800*480> <15fps><30fps>
size<1024*600> <15fps><30fps>
size<1280*800> <15fps><30fps>

视频帧大小<1024 * 600>
^C
```



现在就差摄像头输出的 RGB565 和屏幕支持的 ARGB8888 不支持的问题了，但是摄像头驱动不支持 ARGB8888

```
设置格式...  
ioctl error: VIDIOC_S_FMT: Invalid argument
```

我又不想改驱动，于是修改应用层，在应用层里完成了 RGB565 到 ARGB8888 的转换

从你的输出可以看到，摄像头**确实支持 RGB565 格式**（0x50424752就是 V4L2_PIX_FMT_RGB565），但你想改成 ARGB8888。让我来帮你修改代码。



串口

termios 结构体

示例代码 26.1.1 struct termios 结构体

```
struct termios
{
    tcflag_t c_iflag; /* input mode flags */ 输入模式
    tcflag_t c_oflag; /* output mode flags */ 输出模式
    tcflag_t c_cflag; /* control mode flags */ 控制模式
    tcflag_t l_cflag; /* local mode flags */ 本地模式
    cc_t c_cc[NCCS]; /* control characters */ 特殊控制字符
    speed_t c_ispeed; /* input speed */ 输入速率
    speed_t c_ospeed; /* output speed */ 输出速率
};
```

IGNBRK	忽略输入终止条件
BRKINT	当检测到输入终止条件时发送 SIGINT 信号
IGNPAR	忽略制错误和奇偶校验错误
PARMRK	对奇偶校验错误输出标记
INPCK	对接收到的数据执行奇偶校验
ISTRIP	将所有接收到的数据截断为 7 比特位，也就是去除第八位
DLCR	将接收到的 NL（换行符）转换为 CR（回车符）
KONCR	忽略接收到的 CR（回车符）
KERNL	将接收到的 CR（回车符）转换为 NL（换行符）
LCULC	将接收到的大写字母转换为小写字母
IXON	启动输出软件流控
IXOFF	启动输入软件流控

OPOST	启用输出处理功能，如果不设置该标志则其他标志都被忽略
OLCUC	将输出字符中的大写字母转换成小写字母
ONLCR	将输出中的换行符（NL '\n'）转换成换行符（CR '\r'）
OCRLN	将输出中的回车符（CR '\r'）转换成换行符（NL '\n'）
ONOCR	在第 0 列不输出回车符（CR）
ONLRET	不输出回车符
OFILL	发送填充字符以提供延时
OFDEL	如果设置该标志，则表示填充字符为 DEL 字符，否则为 NULL 字符

CBAUD	波特率的位掩码
B0	波特率为 0
B1200	1200 波特率
B1800	1800 波特率
B2400	2400 波特率
B4800	4800 波特率
B9600	9600 波特率
B19200	19200 波特率
B38400	38400 波特率
B57600	57600 波特率
B115200	115200 波特率
B230400	230400 波特率
B460800	460800 波特率
B500000	500000 波特率
B576000	576000 波特率
B921600	921600 波特率
B1000000	1000000 波特率
B1152000	1152000 波特率
B1500000	1500000 波特率
B2000000	2000000 波特率
B2500000	2500000 波特率
B3000000	3000000 波特率
-----	-----
CSIZE	数据位的位掩码
CS5	5 个数据位
CS6	6 个数据位
CS7	7 个数据位
CS8	8 个数据位
CSTOPB	2 个停止位，如果不设置该标志则默认是一个停止位
CREAD	接收使能
PARENB	使能奇偶校验
PARODD	使用奇校验，而不是偶校验
HUPCL	无延时挂断调制解调器
CLOCAL	忽略调制解调器控制线
CRTSCTS	使能硬件流控

ISIG	若收到信号字符（INTR、QUIT 等），则会产生相应的信号
ICANON	启用规范模式
ECHO	启用输入字符的本地回显功能。当我们在终端输入字符的时候，字符会显示出来，这就是回显功能
ECHOE	若设置 ICANON，则允许退格操作
ECHOK	若设置 ICANON，则 KILL 字符会删除当前行
ECHONL	若设置 ICANON，则允许回显换行符
ECHOCTL	若设置 ECHO，则制表符（制表符、换行符等）会显示为“^X”，其中 X 的 ASCII 码等于给相应控制字符的 ASCII 码加上 0x40。例如，退格符（0x08）会显示为“^H”（H 的 ASCII 码为 0x48）
ECHOPRT	若设置 ICANON 和 ECHO，则删除字符（退格符等）和被删除的字符都会显示
ECHOKE	若设置 ICANON，则允许回显在 ECHOE 和 ECHOPRT 中设置的 KILL 字符
NOFLSH	在通常情况下，当接收到 INTR、QUIT 和 SUSP 控制字符时，会清空输入和输出队列。如果设置该标志，则所有的队列都不会被清空
TOSTOP	若一个后台进程试图向它的控制终端进行写操作，则系统向该后台进程的进程组发送 SIGTTOU 信号。该信号通常终止进程的执行
IXEXTN	启用输入处理功能

- VEOL: 文件结尾符 EOF，对应键为 Ctrl+D。该字符使终端驱动程序将输入行中的所有字符传递正在读取输入的应用程序。如果文件结尾符是该行的第一个字符，则用户程序中的 read 返回 0，表示文件结束。
- VEOL2: 第二行结尾符 EOL2，对应键为 Line feed（LF）。
- VERASE: 删除操作符 ERASE，对应键为 Backspace（BS）；该字符使终端驱动程序删除输入行的最后一个字符。
- VINTR: 中断控制字符 INTR，对应键为 Ctrl+C；该字符使终端驱动程序向与终端相连的进程发送 SIGINT 信号。
- VKILL: 删除行符 KILL，对应键为 Ctrl+U。该字符使终端驱动程序删除整个输入行。
- VMIN: 在非规范模式下，指定最少读取的字节数 MIN。
- VQUIT: 退出操作符 QUIT，对应键为 Ctrl+Z。该字符使终端驱动程序向与终端相连的进程发送 SIGQUIT 信号。
- VSTART: 开始字符 START，对应键为 Ctrl+Q；重新启动被 STOP 暂停的输出。
- VSTOP: 停止字符 STOP，对应键为 Ctrl+S；该字符作用“截流”，即阻止向终端的进一步输出。用于支持 NONCANON 流控。
- VSUSP: 挂起字符 SUSP，对应键为 Ctrl+Z。该字符使终端驱动程序向与终端相连的进程发送 SIGSUSP 信号，用于挂起当前应用程序。
- VTIME: 非规范模式下，指定读取的每个字符之间的超时时间（以分秒为单位）TIME。

串口设备(韦东山)

Tty：终端

dev/tty0 前台终端，在程序运行时如果切换终端且不重启原理程序，会自动更新为新的终端

dev/tty 当前终端，在程序运行时如果切换终端且不重启原理程序，仍然为原来的终端

dev/tty3 和 dev/tty4 两个不同的特定终端

Console：控制台，功能更强大，可打印内核信息

选哪个？内核的打印信息从哪个设备上显示出来？

可以通过内核的cmdline来指定，

比如: console=ttyS0 console=tty

我不想去分辨这个设备是串口还是虚拟终端，

有没有办法得到这个设备？

有！通过/dev/console！

console=ttyS0时：/dev/console就是ttyS0

console=tty时：/dev/console就是前台程序的虚拟终端

console=tty0时：/dev/console就是前台程序的虚拟终端

console=ttyN时：/dev/console就是/dev/ttyN

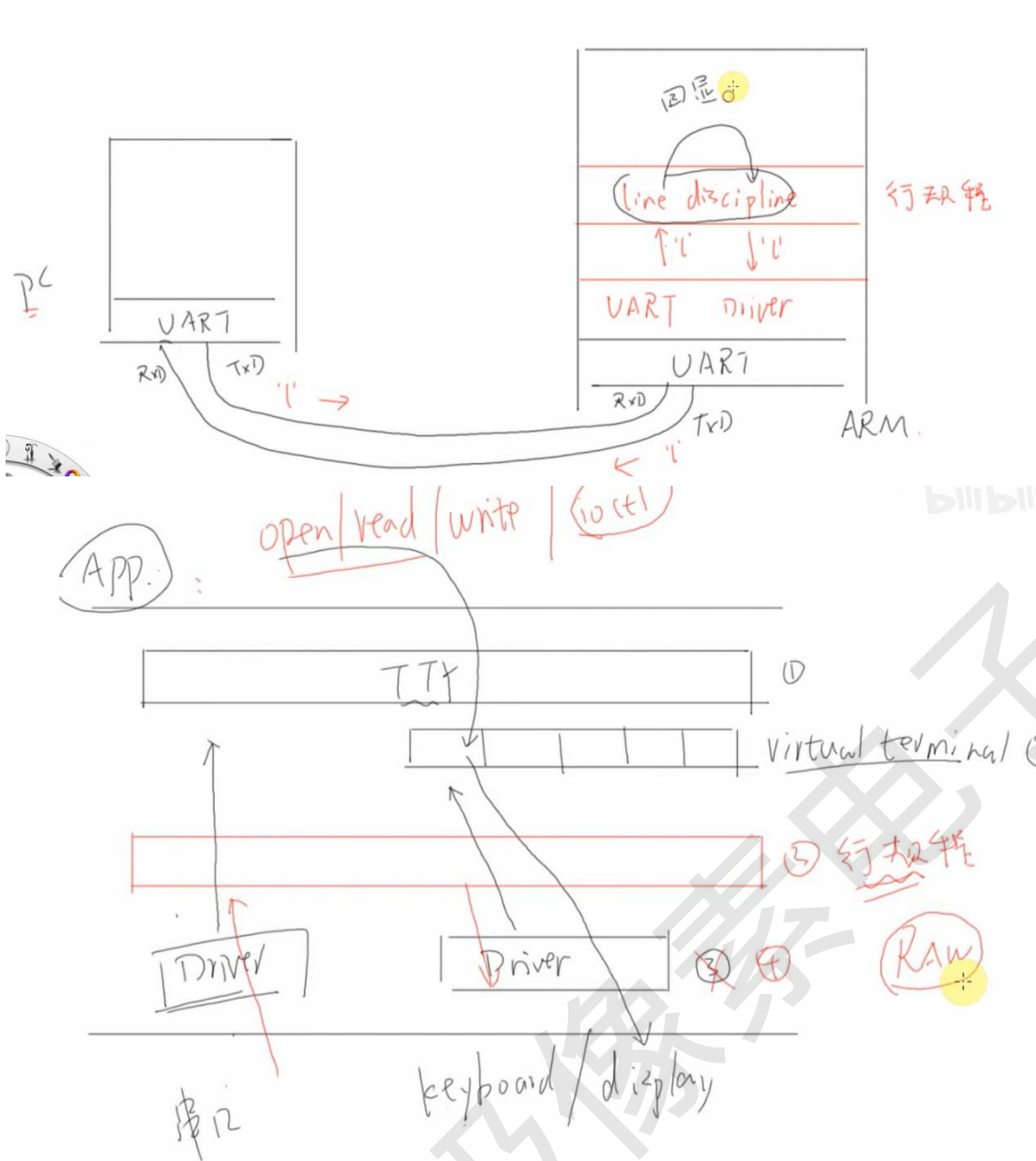
console有多个取值时，使用最后一个取值来判断

下面两个是串口

/dev/ttyS0、/dev/ttySAC0、

行规层(韦东山)

可用来缓存用户输入字符，实现退格输入回显等操作



代码编写方法(韦东山)

```
// 1.打开串口|
fd = open(com, O_RDWR|O_NOCTTY);
```

```
// 2.设置行规程,比如波特率、数据位、停止位、检验位、RAW 模式、一有数据就返回;
// 下面为自己实现的函数
iRet = set_opt(fd, 115200, 8, 'N', 1);
```

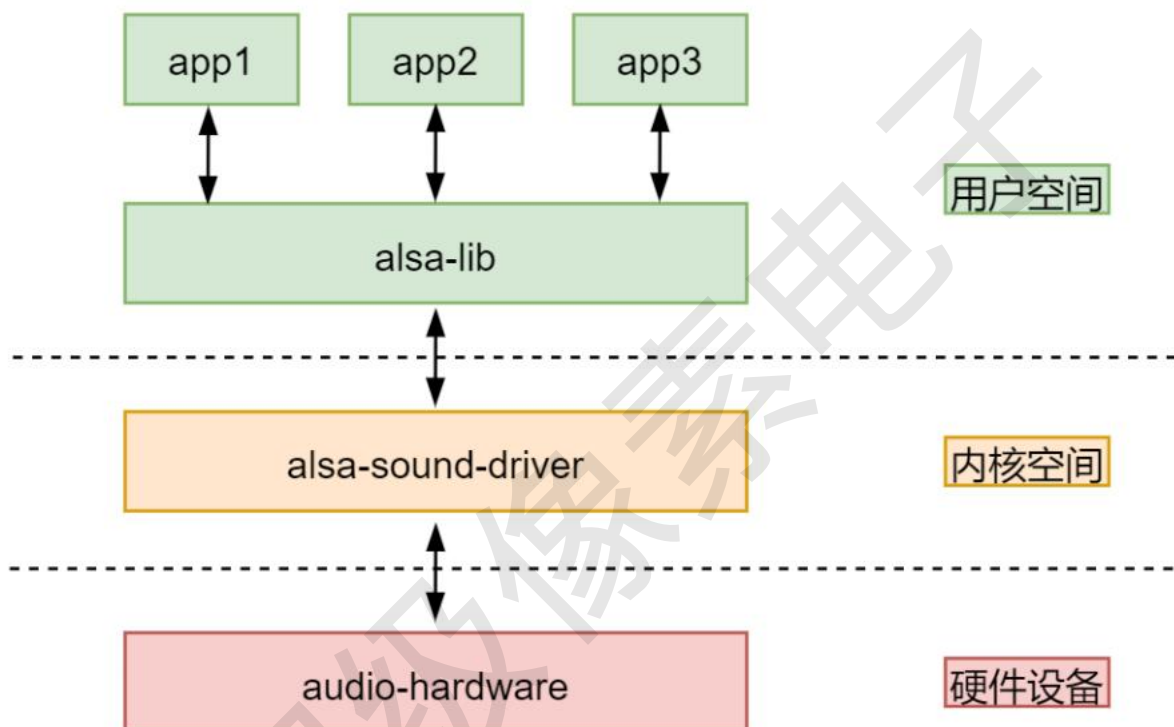
```
// 3.开始串口收发
iRet = write(fd, &c, 1);
iRet = read(fd, &c, 1);
```

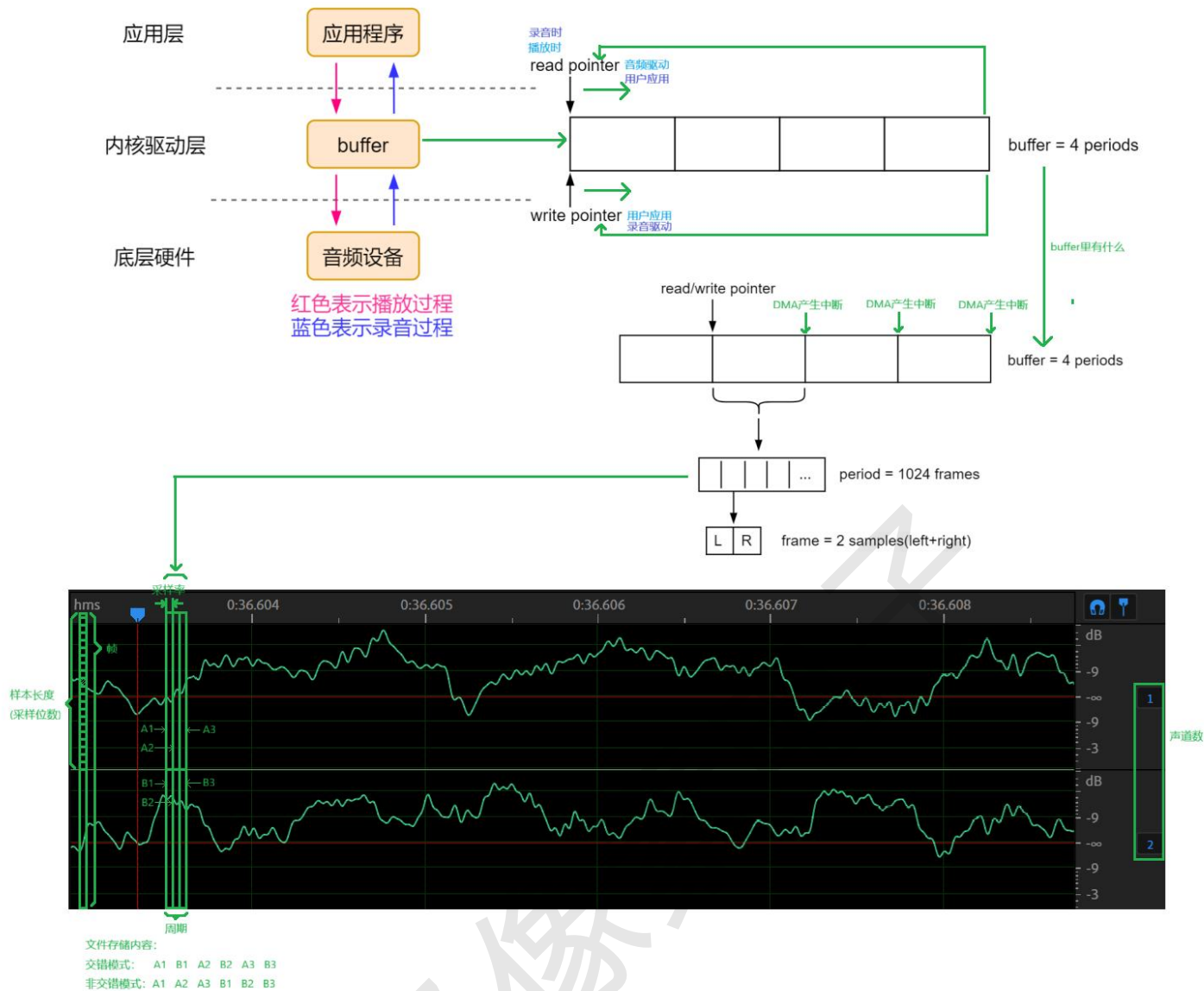
通过设置合适的等待条件,使得串口可以接收到完整数据,而不是还没收到数据就在极短的时间就退出了

```
newtio.c_cc[VMIN] = 1; /* 读数据时的最小字节数：没读到这些数据我就不返回！ */
newtio.c_cc[VTIME] = 0; /* 等待第1个数据的时间：
    * 比如VMIN设为10表示至少读到10个数据才返回，
    * 但是没有数据总不能一直等吧？可以设置VTIME(单位是10秒)
    * 假设VTIME=1，表示：
    *     10秒内一个数据都没有的话就返回
    *     如果10秒内至少读到了1个字节，那就继续等待，完全读到VMIN个数据再返回
    */
```

ALSA

结构





Wav 音频格式

1. RIFF 头 (RIFF_t)

字段	值	说明
ChunkID	"RIFF"	固定标识, 表示这是RIFF格式文件
ChunkSize	计算值	文件总大小 - 8字节 (不包括ChunkID和ChunkSize自身)
Format	"WAVE"	固定标识, 表示这是WAVE音频文件

2. FMT 子块 (FMT_t)

字段名	数据类型	字节偏移	含义与作用	示例值 (44.1kHz 立体声 16bit)
Subchunk1ID	char[4]	0	格式块标识符 固定为字符串 "fmt " (注意末尾有一个空格)	'f', 'm', 't', ' '
Subchunk1Size	uint32_t	4	格式块数据大小 表示后续AudioFormat到BitsPerSample这些字段的总字节数。 对于最常见的PCM格式，该值固定为16。	16
AudioFormat	uint16_t	8	音频格式代码 定义音频数据的编码格式。 1 = PCM (脉冲编码调制，即未压缩的原始音频) 其他值表示各种压缩格式 (如ADPCM, μ-law等)。	1(PCM)
NumChannels	uint16_t	10	声道数量 定义音频的声道数。 1 = 单声道 (Mono) 2 = 立体声 (Stereo) 大于2的值用于多声道环绕声。	2(立体声)
SampleRate	uint32_t	12	采样率 定义每秒对声音波形采样的次数，单位为赫兹 (Hz)。 该值越高，音频可表现的频率范围 (高音) 越广。	44100(CD音质)
ByteRate	uint32_t	16	字节率 表示每秒播放音频所需的数据字节数。 计算公式: $\text{SampleRate} * \text{NumChannels} * \text{BitsPerSample} / 8$	$44100 * 2 * 16 / 8 = 176400$
BlockAlign	uint16_t	20	数据块对齐 表示一个完整的、包含所有声道的“采样帧”的字节大小。 计算公式: $\text{NumChannels} * \text{BitsPerSample} / 8$ 这是数据读写和处理时的基本单位。	$2 * 16 / 8 = 4$
BitsPerSample	uint16_t	22	位深度 定义每个声道的每个采样点用多少位 (bit) 来表示。 该值越高，音频的动态范围和精度越高 (音量细节更丰富)。 常见值: 8、16、24、32。	16(CD音质)

3.可选的保留块(格式和 data 子块一样，但是标识不是“DATA”)

3. DATA 子块 (DATA_t)

字段	值	说明
Subchunk2ID	"data"	数据块标识
Subchunk2Size	计算值	data子块 实际的音频数据大小 (字节)

主要问题

1.编译虽然指定库文件，但是提示找不到库


```
alientek@zdyz-imx6ull: ~/linux/IMX6ULL/c_linux/code/28_alsa-lib$ make
arm-linux-gnueabihf-gcc -o pcm_playbackApp pcm_playback.c \
-I/home/alientek/linux/IMX6ULL/tool/alsa-lib/include \
-L/home/alientek/linux/IMX6ULL/tool/alsa-lib/lib
/tmp/ccNS0V9K.o: In function `snd_pcm_init':
pcm_playback.c:(.text+0x1e): undefined reference to `snd_pcm_open'
pcm_playback.c:(.text+0x36): undefined reference to `snd_strerror'
pcm_playback.c:(.text+0x5c): undefined reference to `snd_pcm_hw_params_malloc'
pcm_playback.c:(.text+0x70): undefined reference to `snd_pcm_hw_params_any'
pcm_playback.c:(.text+0x88): undefined reference to `snd_strerror'
pcm_playback.c:(.text+0xb0): undefined reference to `snd_pcm_hw_params_set_access'
pcm_playback.c:(.text+0xc8): undefined reference to `snd_strerror'
pcm_playback.c:(.text+0xf0): undefined reference to `snd_pcm_hw_params_set_format'
```

为什么仍然报错 "undefined reference to snd_pcm_open"?

即使 libasound.so 存在，编译时仍然报错，可能是因为：

- 1. -lasound 没有正确链接（最常见的问题）
- 2. 交叉编译工具链的库路径不正确
- 3. 动态库和静态库混淆

• -lasound：告诉链接器使用 libasound.so

既然是so是动态库，那不是要运行时才会用到吗，为什么编译时要指定

你提出了一个很好的问题！确实，动态库（.so）在运行时才会被加载，但为什么编译时仍然需要指定 -lasound？

这是因为 编译和链接是两个不同的阶段，而 -lasound 的作用是告诉链接器（ld）：

- 1. 这个程序依赖 libasound.so（记录在可执行文件的动态段中）。
- 2. 在链接阶段检查这些函数是否存在（但不直接嵌入库代码）。

对比：动态库 vs. 静态库

行为	动态库（.so）	静态库（.a）
编译时	需要 -lxxx（记录依赖关系）	需要 -lxxx（直接嵌入代码）
运行时	需要 .so 文件在 LD_LIBRARY_PATH	不需要额外库（代码已嵌入）
文件大小	可执行文件较小	可执行文件较大
更新	替换 .so 即可升级	需重新编译链接

2. 为什么使用异步/poll 方式且功能为录音的时候，需要显示调用 snd_pcm_start，而播放却不用

1. 播放模式 (PLAYBACK) 为何无需显式 `snd_pcm_start`?

关键原因

- **驱动自动管理启动:**
当首次调用 `snd_pcm_writei` 时, ALSA驱动会隐式将设备状态从 `PREPARED` 切换到 `RUNNING`, 自动启动音频硬件 (如DAC)。
- **缓冲区初始为空:**
播放模式下, 驱动缓冲区初始为空, `poll` 会立即返回 `POLLOUT` (可写), 因此不会阻塞。
- **设计逻辑:**
播放是“主动推送”模型, 用户写入数据即触发硬件工作, 无需显式启动。

你的理解修正

- ❌ 不完全是因为“音频文件可以直接写入”。
- ✅ **真正原因:** ALSA驱动在首次写入时自动启动硬件, 保证播放连续性。

2. 录音模式 (CAPTURE) 为何必须显式 `snd_pcm_start`?

关键原因

- **硬件需显式激活:**
录音硬件 (如ADC、麦克风) 需要明确指令才开始采集数据。调用 `snd_pcm_start` 会发送信号启动硬件。
- **缓冲区初始无数据:**
录音模式下, 驱动缓冲区初始为空, 若不启动硬件, `poll` 会无限阻塞 (无数据可读)。
- **设计逻辑:**
录音是“被动拉取”模型, 需用户明确控制采集时机 (避免无意义耗电或资源占用)。

你的理解修正

- ❌ 不完全是因为“先有数据缓冲区才能可读”。
- ✅ **真正原因:** 硬件必须被显式激活才能生成数据, 否则 `poll` 因无数据而阻塞。




```

iRet = listen(iSocketServer, BACKLOG);
// 此函数宣告服务器可以接受连接请求。
// iSocketServer 是 bind 后的文件描述符。
// BACKLOG 设置请求排队的最大长度。当有多个客户端程序和服务端相连时，使用这个表示可以介绍的排队长度。

iSocketClient = accept(iSocketServer, (struct sockaddr *)&tSocketClientAddr, &iAddrLen);
// 服务器使用此函数获得连接请求，并且建立连接。
// iSocketServer 是 listen 后的文件描述符。
// tSocketClientAddr, iAddrLen 是用来给客户端的程序填写的，服务器端只要传递指针就可以了，bind, listen 和 accept 是服务器端用的函数。
// accept 调用时，服务器端的程序会一直阻塞到有一个客户端发出了连接。accept 成功时返回最后的服务器端的文件描述符，这个时候服务器端可以向该描述符写信息了，失败时返回 - 1。

/* 接收客户端发来的数据并显示出来 */
iRecvLen = recv(iSocketClient, ucRecvBuf, 999, 0);
printf("Get Msg From Client %d: %s\n", iClientNum, ucRecvBuf);
close(iSocketServer);

```

客户端

```

iSocketClient = socket(AF_INET, SOCK_STREAM, 0);

if (0 == inet_aton(argv[1], &tSocketServerAddr.sin_addr)) // 把用户IP地址转换成可以被识别的形式
{
    // 处理错误
}

iRet = connect(iSocketClient, (const struct sockaddr *)&tSocketServerAddr, sizeof(struct sockaddr));
iSendLen = send(iSocketClient, ucSendBuf, strlen(ucSendBuf), 0);

close(iSocketClient);

```

UDP

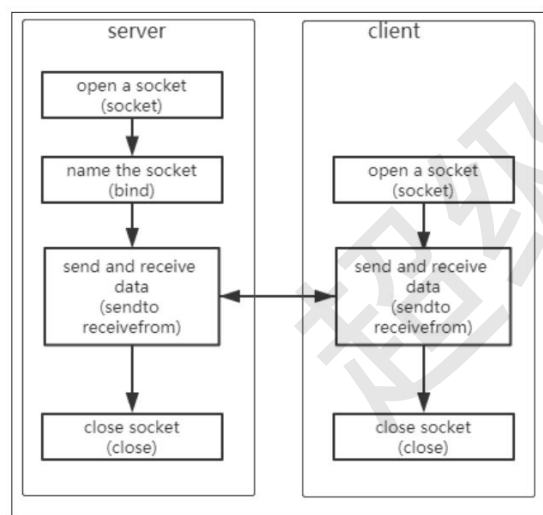


图 8.5UDP 用户数据包模式

讨论

1.为什么新房墙壁已经有网口仍然无法上网

一、关于"新房网络接口与运营商选择"

1. 物理接口是"空白画布"

- 新房子的网络接口就像未开通的电话线或没通水的管道
- 接口本身只是物理连接点，不包含任何网络服务

2. 运营商是"服务提供商"

- 运营商竞争：电信、联通、移动等会提前在小区铺设光纤基础设施
- 你的选择权：只能选择已入驻小区的运营商
- 工程师上门的作用：将你家的接口连接到选定的运营商网络

2.为什么 IPV4 地址分完了但是网络还能正常用

2. 运营商的"地址魔术" (NAT技术)

(1) 核心原理：地址复用

bash

复制

```
# 就像酒店房间号复用：
实际房间只有100间，但通过"楼层+房号"体系服务更多客人

# 网络地址复用：
1个公网IP → 通过NAT技术 → 服务成百上千个用户
```

(2) 具体技术实现

bash

复制

```
# 运营商级NAT (CGNAT)：
你的设备（内网IP） → 运营商（共享公网IP） → 互联网
示例：
你的真实IP：100.64.23.45（运营商内网）
对外显示IP：118.112.78.90（与邻居共享）
```

IIC 和 SMBus(韦东山)

协议

1. SMBus是I2C协议的一个子集

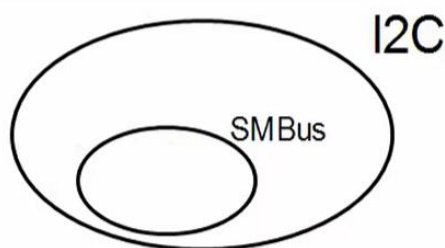
SMBus: System Management Bus, 系统管理总线。

SMBus最初的目的是为智能电池、充电电池、其他微控制器之间的通信链路而定义的。

SMBus也被用来连接各种设备, 包括电源相关设备, 系统传感器, EEPROM通讯设备等等

SMBus 为系统和电源管理这样的任务提供了一条控制总线, 使用 SMBus 的系统, 设备之间发送和接收消息都是通过 SMBus, 而不是使用单独的控制线, 这样可以节省设备的管脚数。

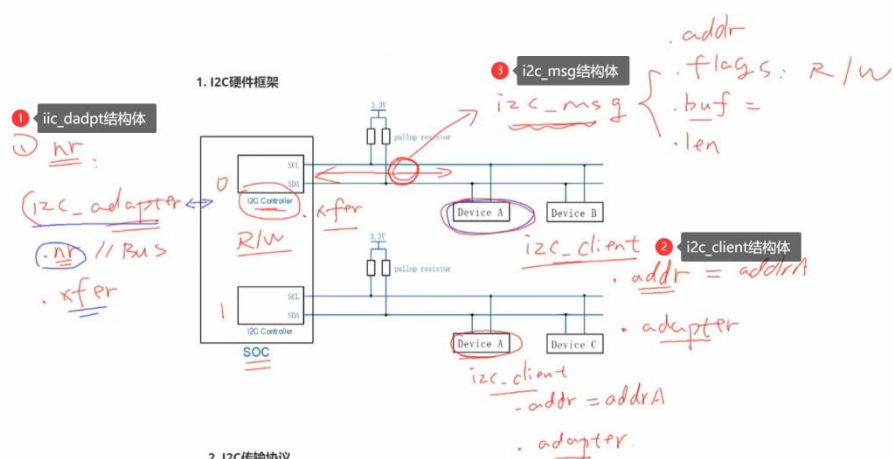
SMBus是基于I2C协议的, SMBus要求更严格, SMBus是I2C协议的子集。



SMBus有哪些更严格的要求? 跟一般的I2C协议有哪些差别?

- VDD的极限值不一样
 - I2C协议: 范围很广, 甚至讨论了高达12V的情况
 - SMBus: 1.8V~5V
 - 最小时钟频率、最大的Clock Stretching
 - Clock Stretching含义: 某个设备需要更多时间进行内部的处理时, 它可以把SCL拉低占住I2C总线
 - I2C协议: 时钟频率最小值无限制, Clock Stretching时长也没有限制
 - SMBus: 时钟频率最小值是10KHz, Clock Stretching的最大时间值也有限制
 - 地址回应(Address Acknowledge)
 - 一个I2C设备接收到它的设备地址后, 是否必须发出回应信号?
 - I2C协议: 没有强制要求必须发出回应信号
 - SMBus: 强制要求必须发出回应信号, 这样对方才知道该设备的状态: busy, failed, 或是被移除了
 - SMBus协议明确了数据的传输格式
 - I2C协议: 它只定义了怎么传输数据, 但是并没有定义数据的格式, 这完全由设备来定义
 - SMBus: 定义了几种数据格式(后面分析)
 - REPEATED START Condition(重复发出S信号)
 - 比如读EEPROM时, 涉及2个操作:
 - 把存储地址发给设备
 - 读数据
 - 在写、读之间, 可以不发出P信号, 而是直接发出S信号: 这个S信号就是 REPEATED START
 - 如下图所示
- | | | | | | | |
|----------------|---------|----|---|--------------|---|-----|
| S | Address | Wr | A | Command Code | A | ... |
| 1 | | 1 | | 8 | | 1 |
| REPEATED START | | | | | | |
| Sr | Address | Rd | A | Data Byte | N | P |
| 7 | | 1 | | 8 | | 1 |
- SMBus Low Power Version
 - SMBus也有低功耗的版本

表示方式



● 使用 `i2c_adapter` 表示一个 I2C BUS，或称为 I2C Controller，里面有 2 个重要的成员：

- `nr`：第几个 I2C BUS(I2C Controller)
- `i2c_algorithm`，里面有该 I2C BUS 的传输函数，用来收发 I2C 数据

1 怎么表示 I2C Device

- 一个 I2C Device，一定有设备地址
 - 它连接在哪个 I2C Controller 上，即对应的 `i2c_adapter` 是什么
- 使用 `i2c_client` 来表示一个 I2C Device

2 怎么表示要传输的数据

- `i2c_msg` 中的 `flags` 用来表示传输方向：bit 0 等于 `I2C_M_RD` 表示读，bit 0 等于 0 表示写
- 一个 `i2c_msg` 要么是读，要么是写

○ 一个 `i2c_msg` 要么是读，要么是写

○ 举例：设备地址为 `0x50` 的 EEPROM，要读取它里面存储地址为 `0x10` 的一个字节，应该构造几个 `i2c_msg`？

- 要构造 2 个 `i2c_msg`
- 第一个 `i2c_msg` 表示写操作，把要访问的存储地址 `0x10` 发给设备
- 第二个 `i2c_msg` 表示读操作
- 代码如下

```
u8 data_addr = 0x10;
i8 data;
struct i2c_msg msgs[2];
```

```
msgs[0].addr = 0x50;
msgs[0].flags = 0;
msgs[0].len = 1;
msgs[0].buf = &data_addr;
```

```
msgs[1].addr = 0x50;
msgs[1].flags = I2C_M_RD;
msgs[1].len = 1;
msgs[1].buf = &data;
```


使用 i2ctools 指令操作 smbus

查看所有 iic 总线

```
[root@imx6ull:/usr/sbin]# i2cdetect -l
i2c-1    i2c        21a4000.i2c    I2C adapter
i2c-0    i2c        21a0000.i2c    I2C adapter
```

查看指定总线上设备

```
[root@imx6ull:/usr/sbin]# i2cdetect 0
i2cdetect: WARNING! This program can confuse your I2C bus
Continue? [y/N] y
  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
```

U: 该地址有 iic 设备, 有驱动程序

其他非--数值: 该地址有 iic 设备, 无驱动程序

AP3216C是红外、光强、距离三合一的传感器, 以读出光强、距离值为例, 步骤如下:

- 复位: 往寄存器0写入0x4

```
[root@imx6ull:/usr/sbin]# i2cset
BusyBox v1.31.1 (2020-06-23 10:29:55 CST) multi-call binary.

Usage: i2cset [-fy] [-m MASK] BUS CHIP-ADDRESS DATA-ADDRESS [VALUE] ... [MODE]

Set I2C registers
I2CBUS I2C bus number
ADDRESS 0x03-0x77
MODE is:
c      Byte, no value
b      Byte data (default)
w      Word data
i      I2C block data
s      SMBus block data
Append p for SMBus PEC
-f      Force access
-y      Disable interactive mode
-r      Read back and compare the result
-m MASK Mask specifying which bits to write

[root@imx6ull:/usr/sbin]# i2cset -f -y 0 0x1e 0 0x4
```

- 使能: 往寄存器0写入0x3

```
[root@imx6ull:/usr/sbin]# i2cset -f -y 0 0x1e 0 0x3
```

- 读光强: 读寄存器0xC、0xD得到2字节的光强

```
[root@imx6ull:/usr/sbin]# i2cget
BusyBox v1.31.1 (2020-06-23 10:29:55 CST) multi-call binary.

Usage: i2cget [-fy] BUS CHIP-ADDRESS [DATA-ADDRESS [MODE]]

Read from I2C/SMBus chip register:
I2CBUS I2C bus number
ADDRESS 0x03-0x77
MODE is:
b      Read byte data (default)
w      Read word data
c      Write byte/read byte
Append p for SMBus PEC
-f      Force access
-y      Disable interactive mode

[root@imx6ull:/usr/sbin]# i2cget -f -y 0 0x1e 0xc w
0x002f
```

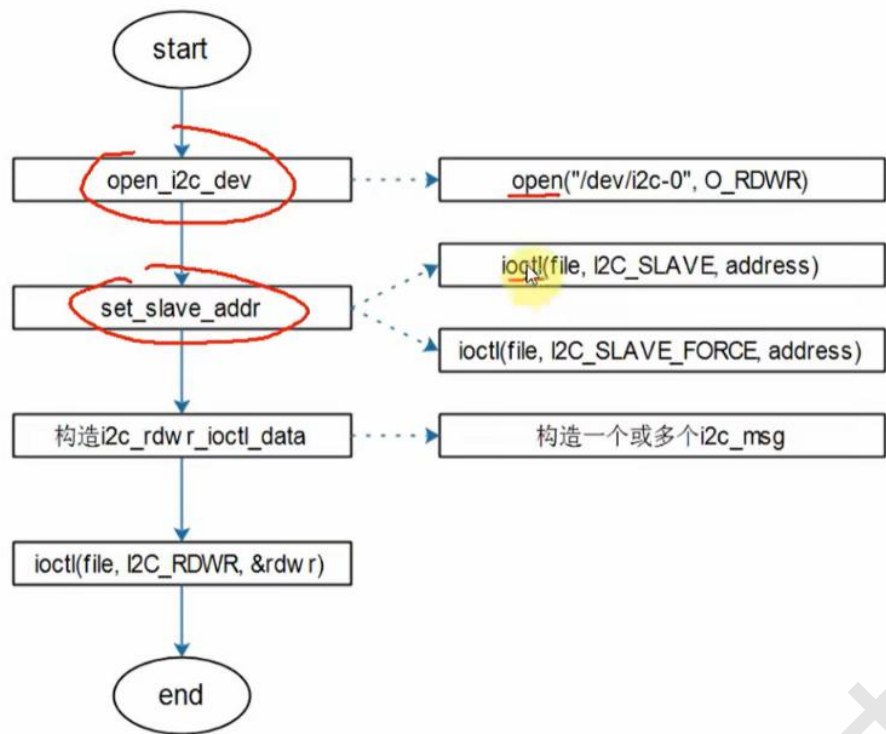
使用 i2ctools 指令操作 iic

```
[root@100ask:~]# i2ctransfer
Usage: i2ctransfer [-f] [-y] [-v] [-V] [-a] I2CBUS DESC [DATA] [DESC [DATA]]...
I2CBUS is an integer or an I2C bus name
DESC describes the transfer in the form: {r|w}LENGTH[@address]
1) read/write-flag 2) LENGTH (range 0-65535) 3) I2C address (use last one if omitted)
DATA are LENGTH bytes for a write message. They can be shortened by a suffix:
= (keep value constant until LENGTH)
+ (increase value by 1 until LENGTH)
- (decrease value by 1 until LENGTH)
p (use pseudo random generator until LENGTH with value as seed)

Example (bus 0, read 8 byte at offset 0x64 from EEPROM at 0x50):
# i2ctransfer 0 w1@0x50 0x64 r8
Example (same EEPROM, at offset 0x42 write 0xff 0xfe ... 0xf0):
# i2ctransfer 0 w17@0x50 0x42 0xff-
[root@100ask:~]# i2ctransfer -f -y 0 w2@0x1e 0 0x4
```

5.1 使用I2C方式

示例代码: i2ctransfer.c



2 使用 SMBus 方式

示例代码: i2cget.c、i2cset.c

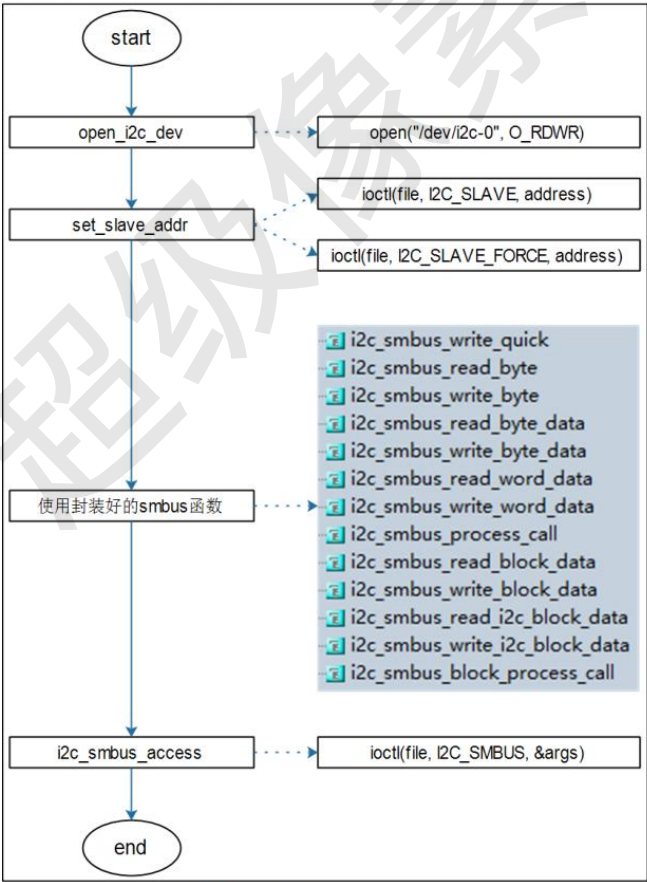
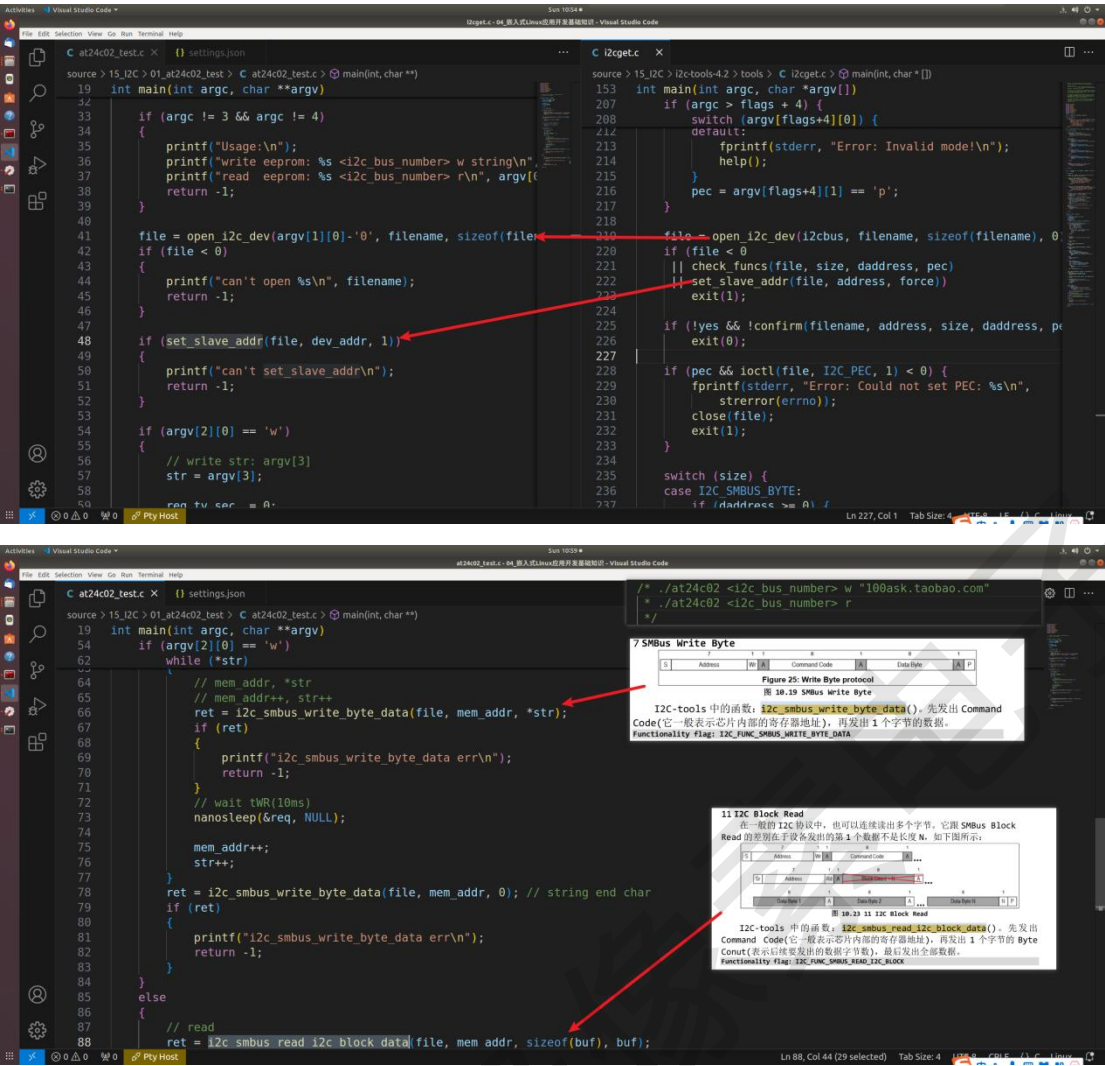
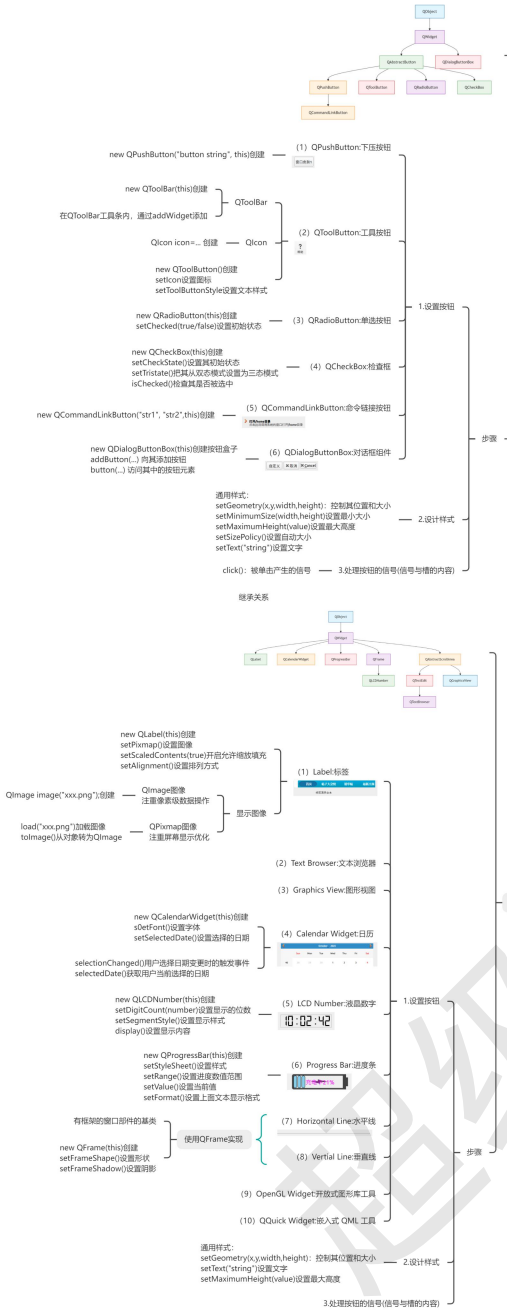
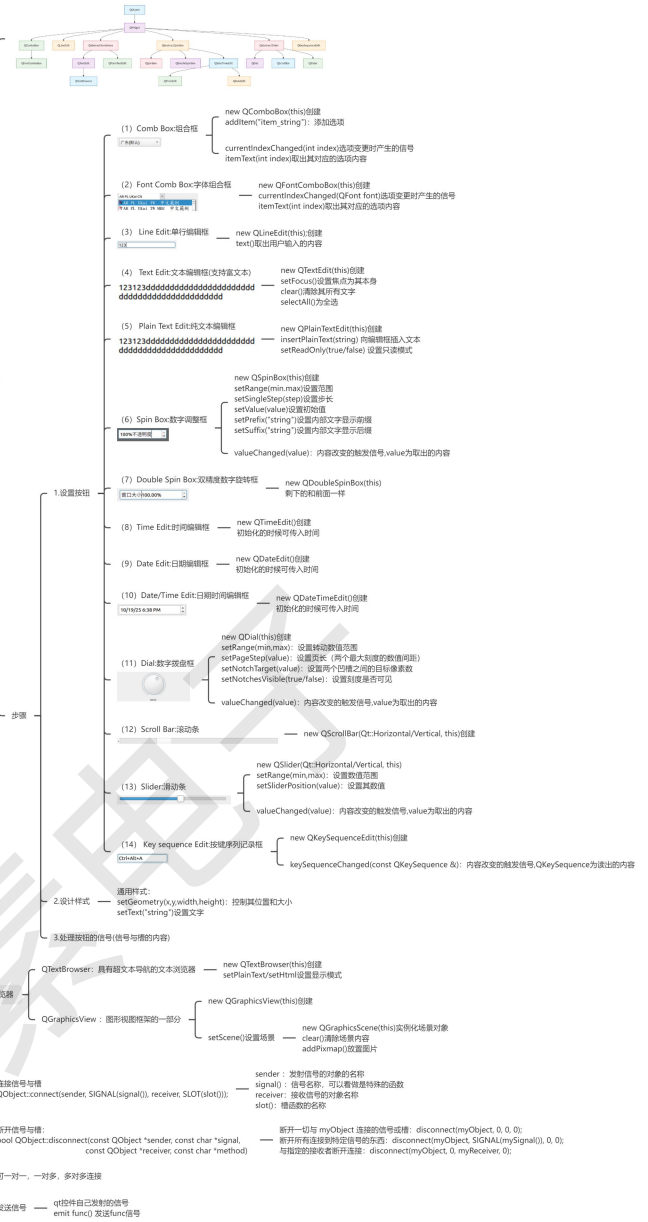


图 10.34 SMBus 读写源码流程

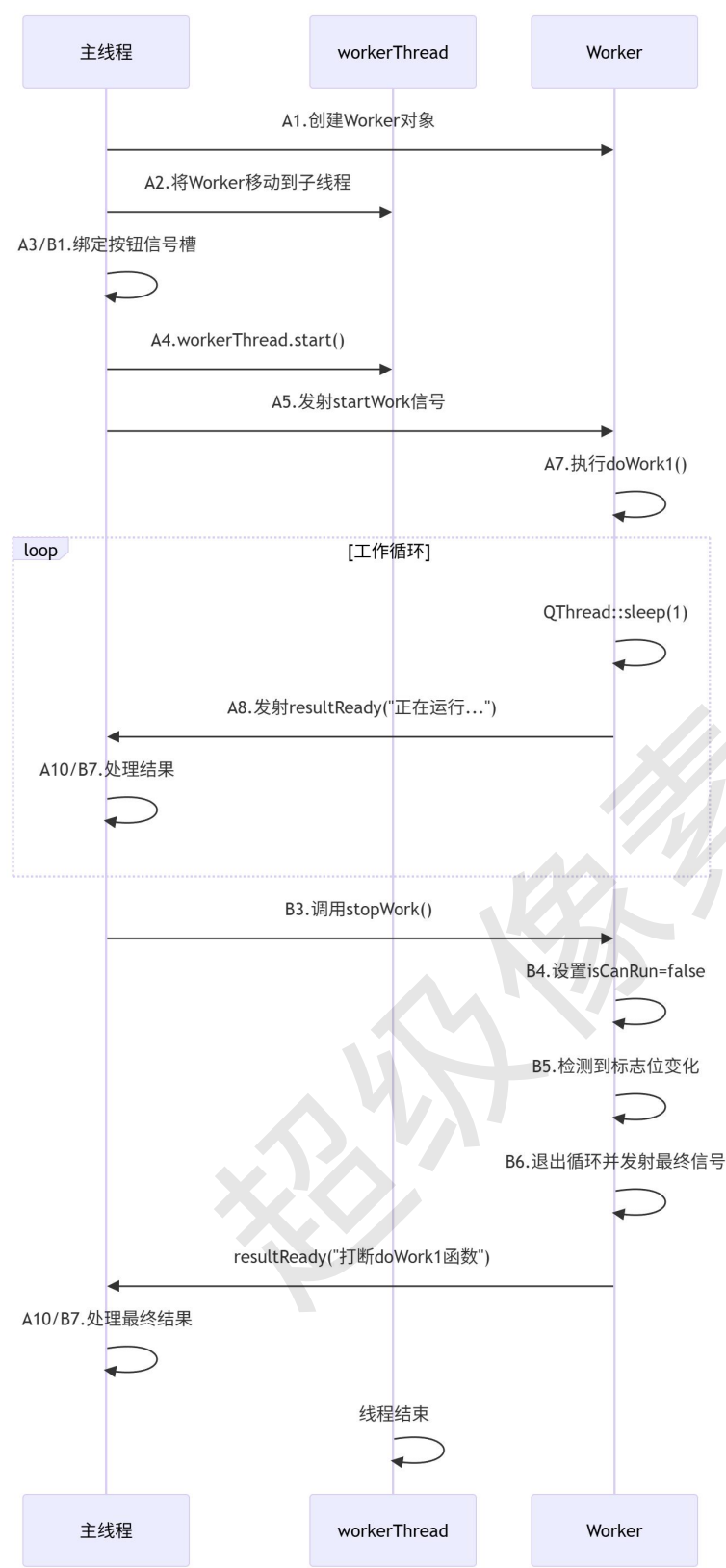
使用代码操作 iic



QT 应用程序开发

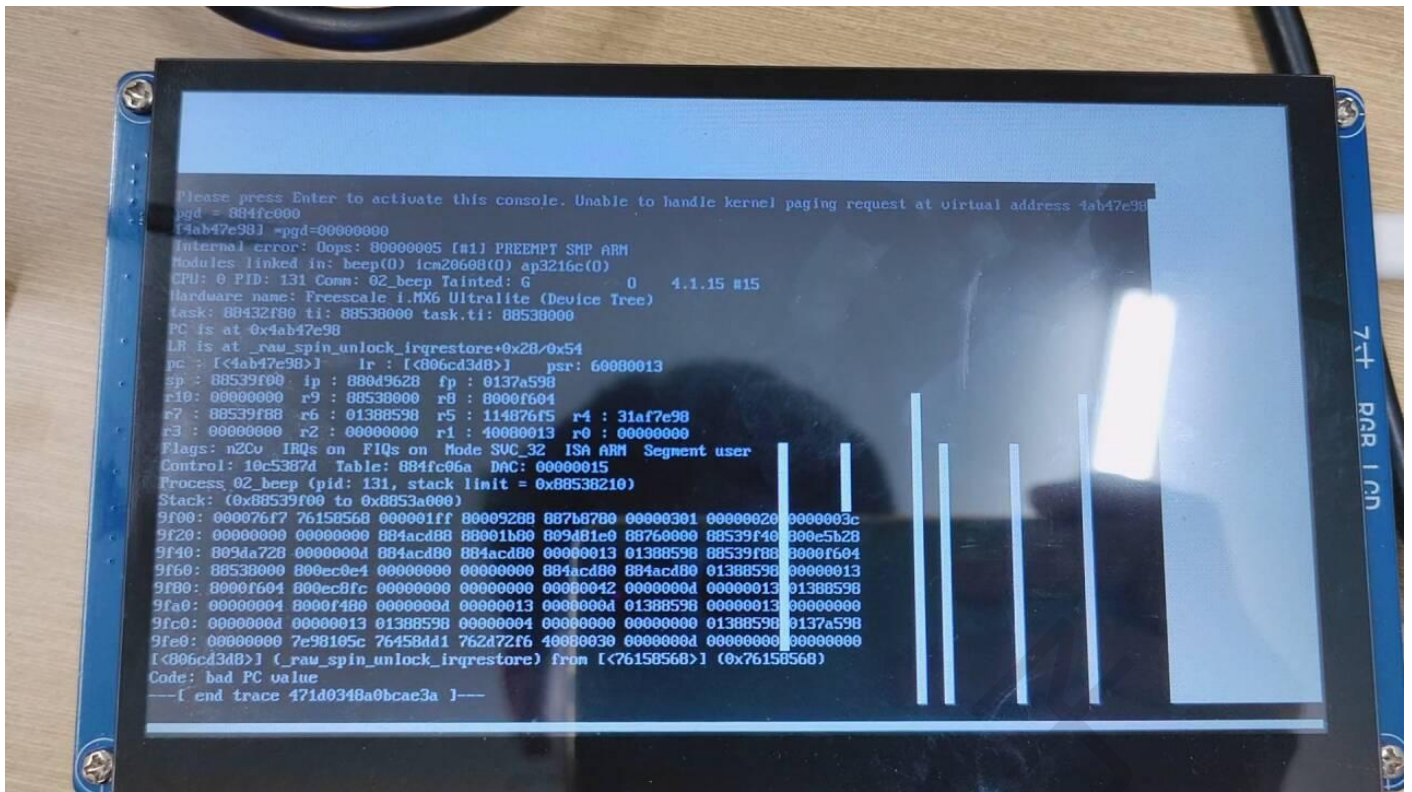


线程的继承 QObject 方法的一个使用例子



蜂鸣器问题

按下按钮后出现内核错误



```
/home/qt # ./02_beep
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
buf: "" ,beep_state: 0
buf: "" ,beep_state: 0Unable to handle kernel paging request at virtual address 4ab47e98

pgd = 884fc000
[4ab47e98] *pgd=00000000
Internal error: Oops: 80000005 [1] PREEMPT SMP ARM
Modules linked in: beep(0) icm20608(0) ap3216c(0)
CPU: 0 PID: 131 Comm: 02_beep Tainted: G 0 4.1.15 #15
Hardware name: Freescale i.MX6 Ultralite (Device Tree)
task: 88432f80 ti: 88538000 task.ti: 88538000
PC is at 0x4ab47e98
LR is at _raw_spin_unlock_irqrestore+0x28/0x54
pc : [<4ab47e98>] lr : [<806cd3d8>] psr: 60080013
sp : 88539f00 ip : 880d9628 fp : 0137a598
r10: 00000000 r9 : 88538000 r8 : 8000f604
r7 : 88539f88 r6 : 01388598 r5 : 114876f5 r4 : 31af7e98
r3 : 00000000 r2 : 00000000 r1 : 40080013 r0 : 00000000
Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user
Control: 10c5387d Table: 884fc06a DAC: 00000015
Process 02_beep (pid: 131, stack limit = 0x88538210)
Stack: (0x88539f00 to 0x8853a000)
9f00: 000076f7 76158568 000001ff 80009288 887b8780 00000301 00000020 0000003c
9f20: 00000000 00000000 884acd88 88001b80 809d81e0 88760000 88539f40 800e5b28
9f40: 809da728 0000000d 884acd80 884acd80 00000013 01388598 88539f88 8000f604
9f60: 88538000 800ec0e4 00000000 00000000 884acd80 884acd80 01388598 00000013
9f80: 8000f604 800ec8fc 00000000 00000000 00080042 0000000d 00000013 01388598
9fa0: 00000004 8000f480 0000000d 00000013 0000000d 01388598 00000013 00000000
9fc0: 0000000d 00000013 01388598 00000004 00000000 00000000 01388598 0137a598
9fe0: 00000000 7e98105c 76458dd1 762d72f6 40080030 0000000d 00000000 00000000
[<806cd3d8>] (_raw_spin_unlock_irqrestore) from [<76158568>] (0x76158568)
Code: bad PC value
---[ end trace 471d0348a0bcae3a ]---
Segmentation fault
/home/qt #
```

原因:

在 getstate 调用完之后, 已经是不稳定状态了, 我发现加一个 qdebug() 都会影响程序是否崩溃的结果


```

bool MainWindow::getBeepState()
{
    /* 如果文件不存在，则返回 */
    if (!file.exists())
        return false;

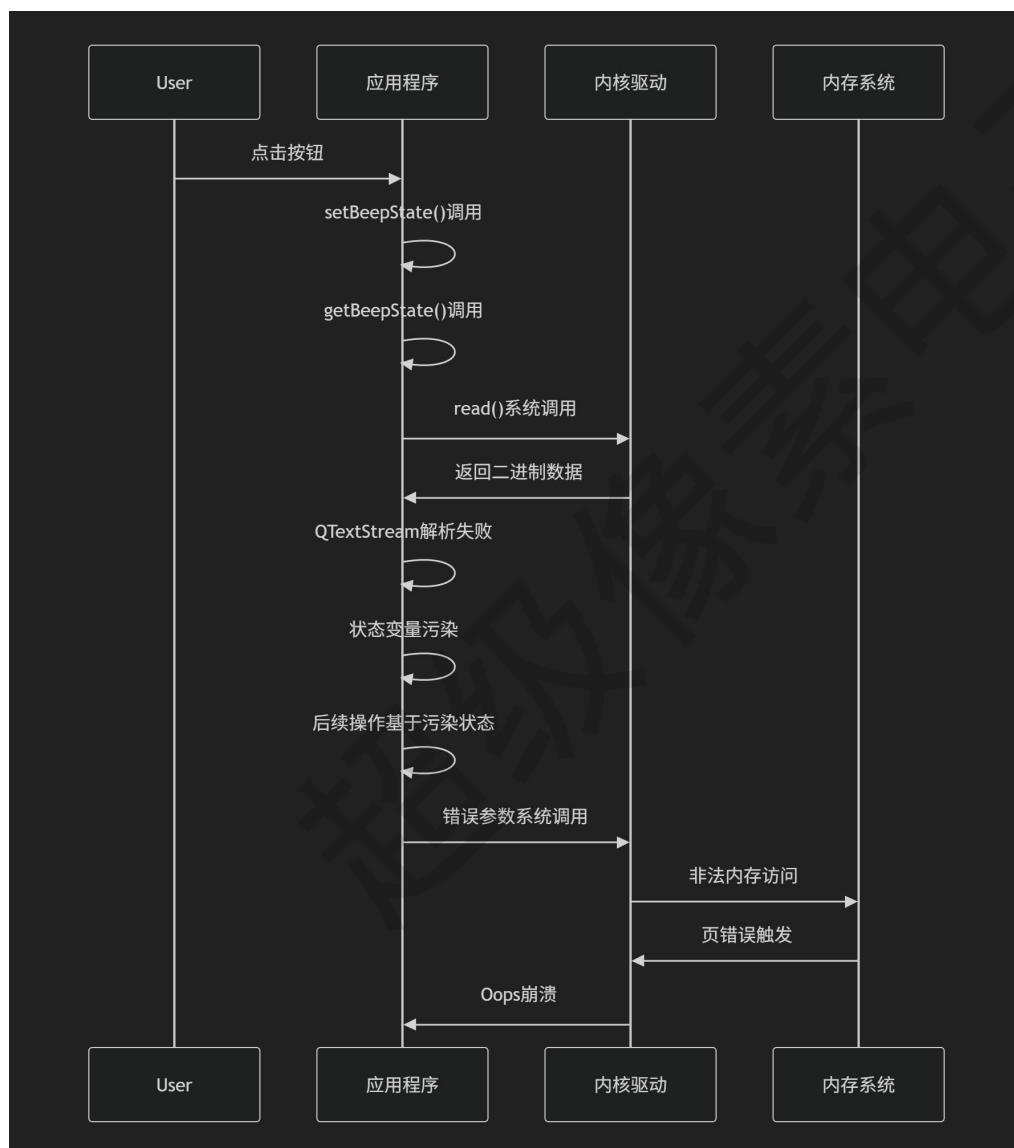
    if(!file.open(QIODevice::ReadWrite))
        qDebug()<<file.errorString();

    QTextStream in(&file);

    /* 读取文件所有数据 */
    QString buf = in.readLine(); ←

    /* 打印出读出的值 */
    qDebug()<<"buf: "<<buf<<" ,beep_state: "<<beep_state;
    file.close();

```



CAN 问题

主要问题：makefile 没写好，导致一个文件被同时编译出了多个目标

```

1  # 定义编译器和标志
2  CC = arm-linux-gnueabi-gcc
3
4  # 目标文件
5  TARGET = can_readApp can_writeApp can_read_loopbackApp can_write_loopbackApp can_loopbackApp
6  SRC = can_read.c can_write.c can_read_loopback.c can_write_loopback.c can_loopback.c
7
8  # 默认目标
9  all: $(TARGET)
10
11 # 编译规则
12 $(TARGET): $(SRC)
13 |     $(CC) -o $@ $<
14
15 # 清理
16 clean:
17 |     rm -f $(TARGET)
18
19 .PHONY: all clean

```

1. 当前Makefile的问题

(1) 主要问题

1. 依赖关系错误: $$(TARGET): (SRC) 表示每个目标都依赖所有源文件, 这会导致:

- 编译 can_readApp 时会尝试用所有源文件 (can_read.c can_write.c...)
- 实际应该是一个目标对应一个源文件

2. 缺少编译标志: 没有添加常见的警告和优化选项

3. 隐式规则未利用: 没有利用Makefile的隐式规则简化编写

摄像头(opencv)问题

主要问题: 之前移植的摄像头驱动不兼容 opencv 的部分 ioctl 操作指令, 所以不管怎么改, 只要遇到 capture->open(index);, 必定报 VIDIOC_QUERYCTRL: Inappropriate ioctl for device, 并且界面卡死

```

/home/qt # ./05_opencv_camera
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
Camera::selectCameraDevice()
going to open camera!
Unable to stop the stream: Invalid argument
Camera::selectCameraDevice()
going to open camera!
VIDIOC_QUERYCTRL: Inappropriate ioctl for device
VIDIOC_QUERYCTRL: Inappropriate ioctl for device

```

```

/home/qt # ./05_opencv_camera
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
going to get camera open status!
going to selectCameraDevice() !
Camera::selectCameraDevice()
going to open camera!
VIDIOC_QUERYCTRL: Inappropriate ioctl for device
VIDIOC_QUERYCTRL: Inappropriate ioctl for device

```

按键问题

主要问题: 开发板上的用户按键原本被我在 Linux 内核中设置为回车键, 并且在 event 3 上面进行输出, 可是在 QT 里面检测不到

```

/dev/input # hexdump event3
00000000 2b23 0000 c890 0003 0001 001c 0001 0000
00000010 2b23 0000 c890 0003 0000 0000 0000 0000
00000020 2b23 0000 0123 0005 0001 001c 0000 0000
00000030 2b23 0000 0123 0005 0000 0000 0000 0000
^C
/dev/input # hexdump event2
00000000 2b2e 0000 7436 0003 0004 0004 0028 0007
00000010 2b2e 0000 7436 0003 0001 001c 0001 0000
00000020 2b2e 0000 7436 0003 0000 0000 0000 0000
00000030 2b2e 0000 eb4a 0004 0004 0004 0028 0007
00000040 2b2e 0000 eb4a 0004 0001 001c 0000 0000
00000050 2b2e 0000 eb4a 0004 0000 0000 0000 0000

```

开发板按键

USB键盘

开始 AI 认为是因为屏幕上的终端拦截了按键的输入导致的, 但是我发现无法关闭屏幕终端

```

/home/qt # echo 0 > /sys/class/tty/tty0/active
-/bin/sh: can't create /sys/class/tty/tty0/active: Permission denied
/home/qt # echo 0 > /sys/class/tty/tty0/
active      dev          power/      subsystem/ uevent
/home/qt # echo 0 > /sys/class/tty/tty0/
-/bin/sh: can't create /sys/class/tty/tty0/: Is a directory
/home/qt # echo 0 > /sys/class/tty/tty0/active
-/bin/sh: can't create /sys/class/tty/tty0/active: Permission denied
/home/qt # killall getty # 结束所有终端进程
killall: getty: no process killed
/home/qt # su
/home/qt # su -
~ # ls
~ # echo 0 > /sys/class/tty/tty0/active
-sh: can't create /sys/class/tty/tty0/active: Permission denied

```

之后我将 event 2 和 event 3 输出 hexdump 的详细内容告诉 AI，AI 认为可能是 QT 的输入子系统监视配置发生错误，于是让我排查

```

/home/qt # ./07_key
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
qt.qpa.input: Initializing tslib plugin "TsLib" ""
qt.qpa.input: tslib device is "/dev/input/event1"
qt.qpa.input: evdevkeyboard: Using device discovery
qt.qpa.input: static device discovery for type QFlags<QDeviceDiscovery::QDeviceType>(Device_Keyboard)
qt.qpa.input: doing static device discovery for "/dev/input/event0"
qt.qpa.input: doing static device discovery for "/dev/input/event1"
qt.qpa.input: doing static device discovery for "/dev/input/event2"
qt.qpa.input: Found keyboard at "/dev/input/event2"
qt.qpa.input: doing static device discovery for "/dev/input/event3"
qt.qpa.input: doing static device discovery for "/dev/input/event4"
qt.qpa.input: doing static device discovery for "/dev/input/mice"
qt.qpa.input: doing static device discovery for "/dev/input/mouse0"
qt.qpa.input: doing static device discovery for "/dev/input/mouse1"
qt.qpa.input: Found matching devices ("/dev/input/event2")
qt.qpa.input: Adding keyboard at "/dev/input/event2"
qt.qpa.input: Try to create keyboard handler for "/dev/input/event2" ""
qt.qpa.input: Opening keyboard at "/dev/input/event2"
qt.qpa.input: Create keyboard handler with for device "/dev/input/event2"
qt.qpa.input: Unload current keymap and restore built-in
qt.qpa.input: numlock=0 , capslock=0, scrolllock=0
qt.qpa.input: evdevmouse: Using device discovery
qt.qpa.input: static device discovery for type QFlags<QDeviceDiscovery::QDeviceType>(Device_Mouse|Device_Touchpad)
qt.qpa.input: doing static device discovery for "/dev/input/event0"
qt.qpa.input: doing static device discovery for "/dev/input/event1"
qt.qpa.input: doing static device discovery for "/dev/input/event2"
qt.qpa.input: doing static device discovery for "/dev/input/event3"
qt.qpa.input: doing static device discovery for "/dev/input/event4"
qt.qpa.input: Found mouse at "/dev/input/event4"
qt.qpa.input: doing static device discovery for "/dev/input/mice"
qt.qpa.input: doing static device discovery for "/dev/input/mouse0"
qt.qpa.input: doing static device discovery for "/dev/input/mouse1"
qt.qpa.input: Found matching devices ("/dev/input/event4")
qt.qpa.input: Adding mouse at "/dev/input/event4"
qt.qpa.input: create mouse handler for "/dev/input/event4" ""
qt.qpa.input: Initializing tslib plugin "tslib" "/dev/input/event1"
qt.qpa.input: tslib device is "/dev/input/event1"

```

根据你的调试输出，可以明确以下关键信息：

问题核心诊断

1. Qt输入设备绑定错误

- 日志显示Qt仅绑定了event2（USB键盘）和event4（鼠标）
- 开发板按键对应的event3未被Qt捕获

输入指令重新配置后解决

```

/home/qt # export QT_QPA_EVDEV_KEYBOARD_PARAMETERS="/dev/input/event3:grab=1"
/home/qt # ./07_key
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
qt.qpa.input: Initializing tslib plugin "TsLib" ""
qt.qpa.input: tslib device is "/dev/input/event1"
qt.qpa.input: Adding keyboard at "/dev/input/event3"
qt.qpa.input: Try to create keyboard handler for "/dev/input/event3" "grab=1"
qt.qpa.input: Opening keyboard at "/dev/input/event3"
qt.qpa.input: Create keyboard handler with for device "/dev/input/event3"
qt.qpa.input: Unload current keymap and restore built-in
qt.qpa.input: numlock=0 , capslock=0, scrolllock=0
qt.qpa.input: evdevmouse: Using device discovery
qt.qpa.input: static device discovery for type QFlags<QDeviceDiscovery::QDeviceType>(Device_Mouse|Device_Touchpad)
qt.qpa.input: doing static device discovery for "/dev/input/event0"
qt.qpa.input: doing static device discovery for "/dev/input/event1"
qt.qpa.input: doing static device discovery for "/dev/input/event2"
qt.qpa.input: doing static device discovery for "/dev/input/event3"
qt.qpa.input: doing static device discovery for "/dev/input/event4"
qt.qpa.input: Found mouse at "/dev/input/event4"
qt.qpa.input: doing static device discovery for "/dev/input/mice"
qt.qpa.input: doing static device discovery for "/dev/input/mouse0"
qt.qpa.input: doing static device discovery for "/dev/input/mouse1"
qt.qpa.input: Found matching devices ("/dev/input/event4")
qt.qpa.input: Adding mouse at "/dev/input/event4"
qt.qpa.input: create mouse handler for "/dev/input/event4" ""
qt.qpa.input: Initializing tslib plugin "tslib" "/dev/input/event1"
qt.qpa.input: tslib device is "/dev/input/event1"

```


QT 旋转屏幕问题

最开始按照 GPT 方法在 mainwindow 添加代码，但是会报错

```
DQT_CORE_LIB -I. -I../qt5.12.9/build/include -I../qt5.12.9/build/include/QtWidgets -
I../qt5.12.9/build/include/QtGui -I../qt5.12.9/build/include/QtCore -I. -I../qt5.12.9/build/mkspecs/linux-
arm-gnueabi-g++ -o moc_mainwindow.o moc_mainwindow.cpp
mainwindow.cpp: In constructor 'MainWindow::MainWindow(QWidget*)':
mainwindow.cpp:23:28: error: 'rotateScreenToPortrait' was not declared in this scope
    rotateScreenToPortrait();
    ^
Makefile:779: recipe for target 'mainwindow.o' failed
make: *** [mainwindow.o] Error 1
```

❌ 编译失败，请检查错误

我看到网上(<https://blog.csdn.net/a3121772305/article/details/90116793>)有人修改了 main 函数，我先尝试了方法一，但是在我这边不生效，后面我发现是因为方法 1 不适用我的 linuxfb，只能用方法 2，但是方法 2 需要修改源码，并且我看到评论说修改源码后会出现性能和触摸不准的问题，于是不考虑使用方法 2

QT 屏幕旋转的两种方式

转载 于 2019-05-11 23:48:34 发布 · 1.2w 阅读 · 4 · 35

2048 AI社区 加入社区

QT 专栏收录该内容

6 篇文章

订阅专栏

摘要 本文介绍在Qt中使用两种方法实现图形界面的旋转。第一种方法通过QGraphicsProxyWidget旋转QWidget，第二种方法修改linuxfb源码支持旋转。文章详细展示了代码实现，并提供了环境变量配置...

1、方式一：

```
cpp
1 #include "mainwindow.h"
2 #include <QApplication>
3 #include <QGraphicsView>
4 #include <QGraphicsProxyWidget>
5
6 int main(int argc, char *argv[])
7 {
8     QApplication a(argc, argv);
9     MainWindow w_ui;
```

2、方式二：针对 linuxfb

(1) 修改linuxfb/qlinuxfbscreen.h，如下所示：

我尝试让 gpt 修改 mian 代码，gpt 想出修改环境变量，
qputenv("QT_QPA_PLATFORM", "linuxfb:fb=/dev/fb0");
qputenv("QT_QPA_PLATFORM", "linuxfb:fb=/dev/fb0:rotation=90");
但是还是不行

尝试输入

```
export QT_QPA_EGLFS_ROTATION=90 export
export QT_QPA_PLATFORM=linuxfb:fb=/dev/fb0:rotation=90
export QT_QPA_FB_ROTATION=90
```

也没有效果

老是要密码 2021.07.12

老哥，可以转发不

lsq8178 2018.07.09

我照着这个方法设置了，可以翻转屏幕，和1楼的同学同样的问题，不知道楼主有没有在深入研究，怎么解决的

复制品 2018.05.22

你好，我现在按照这个改了后是旋转了，但是程序的原点和屏幕的原点不一致，应用程序的图形界面只占了屏幕的一部分，而且ts的触摸响应位置也乱了。请问楼主有没有遇到这种问题啊，怎么解决的，不胜感激！

z526882183 回复 lsq8178 2019.08.09

export

QT_QPA_EVDEV_TOUCHSCREEN_PARAMETERS=/dev/input/e
vent1:rotate=270 加上这环境变量，触摸位置也会旋转，
event1需要根据自己的触摸设备而定

查看全部 2 条回复

QT 控制终端问题

1. Shell（命令行解释器）

Shell是软件层面的**命令解释器**，负责接收用户输入的命令并调用系统执行。常见的Shell类型：

Shell名称	全称/来源	特点	典型路径
sh	Bourne Shell	最古老的Unix Shell，所有Shell的祖先，功能基础	/bin/sh
ash	Almquist Shell	轻量级sh兼容实现，嵌入式系统常用（如OpenWRT、Alpine）	/bin/ash
bash	Bourne-Again Shell	sh的增强版，支持高级功能（自动补全、历史记录等）	/bin/bash
dash	Debian Almquist Shell	Debian系的sh替代品，比ash更快，但功能更少	/bin/dash
busybox	嵌入式工具集	集成多个精简工具（包括一个微型Shell），常见于路由器等嵌入式设备	/bin/busybox

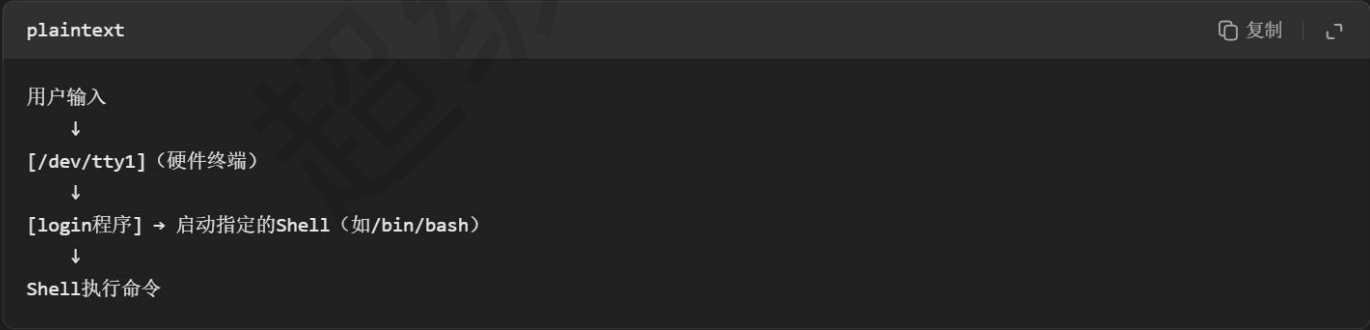
2. /dev/tty1（硬件终端设备）

这是Linux系统的**物理终端设备**，属于硬件/内核层面的概念：

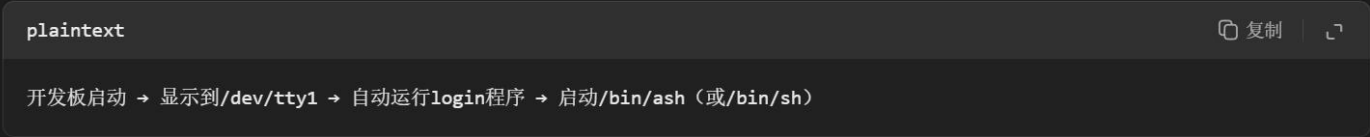
设备路径	类型	作用
/dev/tty1	虚拟控制台（VC）	对应键盘和显示器连接的第一个文本控制台（按Ctrl+Alt+F1切换）
/dev/ttyS0	串口终端	通过串口（如RS-232）连接的物理终端（常见于嵌入式开发板）
/dev/pts/0	伪终端（SSH/Xterm）	远程登录或图形终端模拟器创建的虚拟终端（如SSH会话或GNOME Terminal）

3. Shell与/dev/tty1的关系

- **Shell运行在终端上：**
无论物理终端（tty1）还是伪终端（pts/0），都需要一个Shell（如bash）来解释用户输入的命令。



- **嵌入式系统的典型组合：**



在 ps 列表里面发现了终端进程，杀掉后当前终端被重新启动

/ # ps			
PID	USER	TIME	COMMAND

```
61 root      0:00 [kworker/0:3]
63 root      0:00 [kworker/0:1H]
136 root      0:00 ~/bin/sh
137 root      0:00 ~/bin/sh
164 root      0:00 ps
/ # kill -9 136
```

Please press Enter to activate this console.

1. 内核启动最后的init进程
2. init读取/etc/inittab
↓
3. 执行`::sysinit:/etc/init.d/rcS`
↓
4. /etc/init.d/rcS完成:
 - 挂载文件系统
 - 配置网络
 - 启动基础服务↓
5. init继续执行inittab中的其他条目:
 - 启动getty登录终端
 - 定义Ctrl+Alt+Del行为↓
6. 系统进入指定运行级别