基于STM32F407的软件开发实践

> **够用的硬件**
> **能用的代码**
> **实用的教程**
> 官网：www.wujique.com
> github:https://github.com/wujique/stm32f407
> 资料下载：https://pan.baidu.com/s/1bHUVe6X6tymktUHk_z91cA

本文档说明对应版本软件。也就是github上master对应的tag。

0 概述

网络上开发板种类非常丰富，教程也很多，但是缺少总体设计的指导。
我们基于STM32F407，开发了一套完整的底层程序。
本文档就是是对最新的完整代码说明。
另外，整理了整个开发过程，作为每一个外设的教程说明。

1 设计理念

1 设备与驱动分离
2 每一种设备有一个单独链表管理。

2 系统支撑模块

系统依赖的底层软件模块。

2.1 内存管理

使用k&r内存管理方案，进行了一定优化。

2.2 链表管理

直接使用linux的list.h。

2.3 freeRTOS

待补充。
目前我们主要使用freeRTOS的任务切换功能。

2.4 fatfs

根据ST官方SD卡和USB例程移植，暂时不做过多解释。

3 接口

3.1 串口

待补充。
环形缓冲。

3.2 I2C

- 文件：mcu_i2c.c/mcu_i2c.h
- 接口：

```
/**
 *@brief:      mcu_i2c_register
 *@details:    初始化I2C接口， 相当于注册一个I2C设备
 *@param[in]   const DevI2c * dev I2C设备信息
 *@param[out]  无
 *@retval:
 */
extern s32 mcu_i2c_register(const DevI2c * dev);
/**
 *@brief:      mcu_i2c_transfer
 *@details:    中间无重新开始位的传输流程
 *@param[in]   DevI2cNode * node  I2C节点
                           u8 addr   7位地址
               u8 rw    0 写，1 读
               u8* data
               s32 datalen 发送数据长度
 *@param[out]  无
 *@retval:
 */
extern s32 mcu_i2c_transfer(DevI2cNode * node, u8 addr, u8 rw, u8* data, s32 datalen);
/**
 *@brief:      mcu_i2c_open
 *@details:    根据名字打开一个i2c接口
 *@param[in]   void
 *@param[out]  无
 *@retval:     返回设备节点
 */
extern DevI2cNode *mcu_i2c_open(char *name);
/**
 *@brief:      mcu_i2c_close
 *@details:    关闭I2C节点
 *@param[in]   DevI2cNode *node
 *@param[out]  无
 *@retval:     -1 关闭失败；0 关闭成功
 */
extern s32 mcu_i2c_close(DevI2cNode * node);
```

- 主要数据结构

```
/*
        i2c设备定义
*/

typedef struct
{
        /*设备名称*/
        char name[DEV_NAME_SIZE];

        /*设备需要的资源，模拟I2C只需要两根IO口*/
        MCU_PORT sclport;
        u16 sclpin;

        MCU_PORT sdaport;
        u16 sdapin;
}DevI2c;

/*

        I2C节点

*/
typedef struct
{

        s32 gd;
        DevI2c dev;
        struct list_head list;
}DevI2cNode;
```

- 使用说明：

1. 主要是用IO模拟I2C功能，没有用STM32的硬件I2C。
2. 依赖于mcu_bsp_stm32.c中的IO口操作。
3. 注册一个I2C设备。
   首先定义一个i2c设备：

```
const DevI2c DevVi2c1={
                .name = "VI2C1",

                .sclport = MCU_PORT_D,
                .sclpin = GPIO_Pin_6,

                .sdaport = MCU_PORT_D,
                .sdapin = GPIO_Pin_7,
                };
```

然后将其注册到I2C模块内（如系统存在相同名字的设备，则注册失败）：

```
/*注册I2C总线*/
mcu_i2c_register(&DevVi2c1);
```

4. 使用I2C设备。

首先打开I2C设备：

```
        DevI2cNode *node;

        node = mcu_i2c_open("VI2C1");
        if(node == NULL)
        {
                wjq_log(LOG_DEBUG, "open VI2C1 err!\r\n");
                while(1);
        }
```

然后进行通信，注意第二个参数地址是7bit模式，在传输函数内部会左移，并填充读写标志位。

```
u8 data[16];
mcu_i2c_transfer(node, 0x70, MCU_I2C_MODE_W, data, 8);
mcu_i2c_transfer(node, 0x70, MCU_I2C_MODE_R, data, 8);
```

使用结束后关闭设备

```
mcu_i2c_close(node);
```

5. 待改进
   有一下两点没有做：

> 1. 时钟配置
> 2. 跟STM32的硬件I2C统一。

6. 驱动实现细节
   待补充

3.3 SPI

设计SPI架构的一个主要理念是将SPI分为两部分：**SPI控制器**&**SPI通道**。

> SPI控制器，就是MISO/CLK/MOSI。
> SPI通道，就是控制器+CS。

这样划分的原因是：我们经常一个SPI配多个CS，链接多个外设。
上层操作的都是SPI通道，不直接操作SPI控制器。
SPI控制器也可以用软件模拟。

- 文件：mcu_spi.c/mcu_spi.h

- 接口

```
/**
 *@brief:          mcu_spi_register
 *@details:    注册SPI控制器设备
 *@param[in]   DevSpi *dev
 *@param[out]  无
 *@retval:
 */
extern s32 mcu_spi_register(const DevSpi *dev);
/**
 *@brief:      mcu_spich_register
 *@details:    注册SPI通道
 *@param[in]   DevSpiCh *dev
 *@param[out]  无
 *@retval:
 */
extern s32 mcu_spich_register(const DevSpiCh *dev);
/**
 *@brief:      mcu_spi_open
 *@details:    打开SPI通道
 *@param[in]   DevSpiChNode * node
               u8 mode      模式
               u16 pre      预分频
 *@param[out]  无
 *@retval:
                          打开一次SPI，在F407上大概要2us
 */
extern DevSpiChNode *mcu_spi_open(char *name, SPI_MODE mode, u16 pre);
/**
 *@brief:      mcu_spi_close
 *@details:    关闭SPI 通道
 *@param[in]   DevSpiChNode * node
 *@param[out]  无
 *@retval:
 */
extern s32 mcu_spi_close(DevSpiChNode * node);
/**
 *@brief:      mcu_spi_transfer
 *@details:    SPI 传输
 *@param[in]   DevSpiChNode * node
                          u8 *snd
               u8 *rsv
               s32 len
 *@param[out]  无
 *@retval:
 */
extern s32 mcu_spi_transfer(DevSpiChNode * node, u8 *snd, u8 *rsv, s32 len);
/**
 *@brief:      mcu_spi_cs
 *@details:    操控对应SPI的CS
 *@param[in]   DevSpiChNode * node
               u8 sta   1 高电平，0 低电平
 *@param[out]  无
 *@retval:
```

```
 */
extern s32 mcu_spi_cs(DevSpiChNode * node, u8 sta);
```

- 主要数据结构

```
/*
        SPI 分两层，
        1层是SPI控制器，不包含CS
        2层是SPI通道，由控制器+CS组成

*/
typedef enum{
        DEV_SPI_H = 1,//硬件SPI控制器
        DEV_SPI_V = 2,//IO模拟SPI
}DEV_SPI_TYPE;

/*

        SPI 控制器设备定义

*/
typedef struct
{
        /*设备名称*/
        char name[DEV_NAME_SIZE];

        /*设备类型，IO模拟 or 硬件控制器*/
        DEV_SPI_TYPE type;

        MCU_PORT clkport;
        u16 clkpin;

        MCU_PORT mosiport;
        u16 mosipin;

        MCU_PORT misoport;
        u16 misopin;

}DevSpi;

/*

        SPI控制器节点

*/
typedef struct
{
        /*句柄，空闲为-1，打开为0，spi控制器不能重复打开*/
        s32 gd;
        /* 控制器硬件信息，初始化控制器时拷贝设备树的信息到此 */
        DevSpi dev;

        /*模拟SPI的时钟分频设置*/
        u16 clk;

        /*链表*/
        struct list_head list;
}DevSpiNode;
```

```
/*
        SPI 通道定义
        一个SPI通道，有一个SPI控制器+一根CS引脚组成

*/
typedef struct
{
        /*通道名称，相当于设备名称*/
        char name[DEV_NAME_SIZE];
        /*SPI控制器名称*/
        char spi[DEV_NAME_SIZE];

        /*cs脚*/
        MCU_PORT csport;
        u16 cspin;
}DevSpiCh;

/*
        SPI通道节点

*/
typedef struct
{
        /**/
        s32 gd;

        DevSpiCh dev;

        DevSpiNode *spi;//控制器节点指针

        struct list_head list;
}DevSpiChNode;

/*

SPI模式

*/
typedef enum{
        SPI_MODE_0 =0,
        SPI_MODE_1,
        SPI_MODE_2,
        SPI_MODE_3,
        SPI_MODE_MAX
}SPI_MODE;
```

- 使用说明

1. 接口依赖于更低一层的SPI控制器，可以使是IO口模拟的SPI（VSPI），也可以是硬件SPI。
2. 依赖BSP中IO口操作函数。
3. 定义一个IO模拟的SPI控制器（VSPI）,控制器名称叫做"VSPI1",

```
const DevSpi DevVSpi1IO={
                .name = "VSPI1",
                .type = DEV_SPI_V,

                /*clk*/
                .clkport = MCU_PORT_B,
                .clkpin = GPIO_Pin_0,
                /*mosi*/
                .mosiport = MCU_PORT_D,
                .mosipin = GPIO_Pin_11,
                /*miso*/
                .misoport = MCU_PORT_D,
                .misopin = GPIO_Pin_12,
        };
```

4. 定义一个硬件SPI控制器（STM32）,名字叫"SPI3"

```
const DevSpi DevSpi3IO={
                .name = "SPI3",
                .type = DEV_SPI_H,

                /*clk*/
                .clkport = MCU_PORT_B,
                .clkpin = GPIO_Pin_3,

                /*mosi*/
                .mosiport = MCU_PORT_B,
                .mosipin = GPIO_Pin_5,

                /*miso*/
                .misoport = MCU_PORT_B,
                .misopin = GPIO_Pin_4,
        };
```

因为硬件SPI控制器依赖于不同芯片，因此请修改下面函数
如果是STM32平台，比较容易修改。

```
mcu_hspi_init
mcu_hspi_open
mcu_hspi_close
mcu_hspi_transfer
```

5. 定义SPI通道，一共定义了5个通道，基于前面的两个SPI控制器

```
/* FLASH 1*/
const DevSpiCh DevSpi3CH1={
                .name = "SPI3_CH1",
                .spi = "SPI3",

                .csport = MCU_PORT_B,
                .cspin = GPIO_Pin_14,

        };
/* flash 2*/
const DevSpiCh DevSpi3CH2={
                .name = "SPI3_CH2",
                .spi = "SPI3",

                .csport = MCU_PORT_G,
                .cspin = GPIO_Pin_15,

        };
/*外扩SPI，可接COG、OLED、SPI TFT、RF24L01*/
const DevSpiCh DevSpi3CH3={
                .name = "SPI3_CH3",
                .spi = "SPI3",

                .csport = MCU_PORT_G,
                .cspin = GPIO_Pin_6,

        };


/* 触摸屏，IO模拟SPI*/
const DevSpiCh DevVSpi1CH1={
                .name = "VSPI1_CH1",
                .spi = "VSPI1",

                .csport = MCU_PORT_B,
                .cspin = GPIO_Pin_1,

        };
/* SPI彩屏，跟触摸屏用相同的控制器*/
const DevSpiCh DevVSpi1CH2={
                .name = "VSPI1_CH2",
                .spi = "VSPI1",

                .csport = MCU_PORT_D,
                .cspin = GPIO_Pin_14,
        };
```

6. 将控制器和通道注册到SPI系统

```
/*注册SPI控制器*/
        mcu_spi_register(&DevSpi3IO);
        mcu_spi_register(&DevVSpi1IO);

        /*注册SPI 通道*/
        mcu_spich_register(&DevSpi3CH1);
        mcu_spich_register(&DevSpi3CH2);
        mcu_spich_register(&DevSpi3CH3);

        mcu_spich_register(&DevVSpi1CH1);
        mcu_spich_register(&DevVSpi1CH2);
```

7. 使用SPI通道

```
void spi_example(void)
{
        DevSpiChNode *spichnode;
        u8 src[16];
        u8 rsv[16];

        /*打开SPI通道*/
        spichnode = mcu_spi_open("VSPI1_CH1", SPI_MODE_1, 4);
        if(spichnode == NULL)
        {
                while(1);
        }
        /*读10个数据*/
        mcu_spi_transfer(spichnode, NULL, rsv, 10);
        /*写10个数据*/
        mcu_spi_transfer(spichnode, src, NULL, 10);
        /*读写10个数据*/
        mcu_spi_transfer(spichnode, src, rsv, 10);

        mcu_spi_close(spichnode);
}
```

8. cs控制

   打开SPI通道时，默认将CS拉低，关闭SPI通道，则拉高CS。

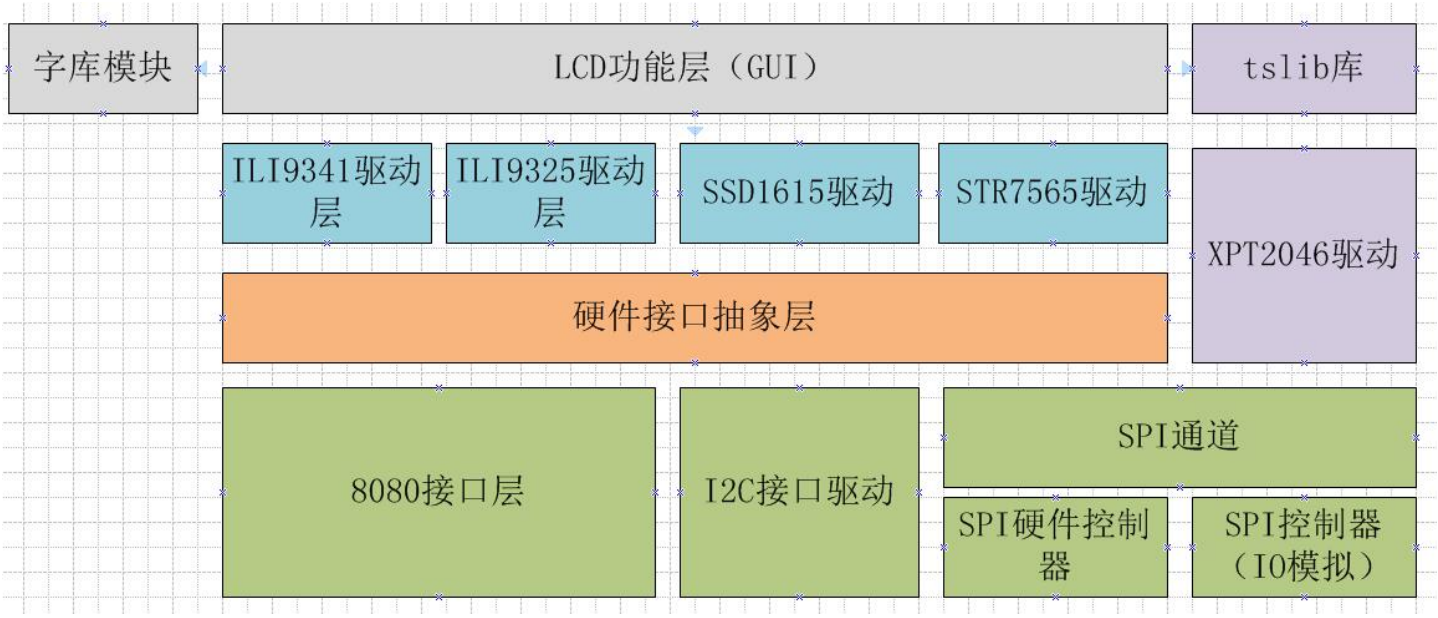
   像SPI FLASH设备，需要在通信过程中通过CS下降沿表明传输命令，因此提供了mcu_spi_cs，可以在打开Spi通道后控制CS状态。

4 显示

LCD显示是一个常用设备，种类繁多，接口多样化，如果程序没有设计架构，将会非常混乱。
大部分开发板提供的初始化代码，在一个函数内包含所有种类LCD的初始化，就像裹脚布一样，又长又乱。

非常不利于维护，也不利于扩展，更加不利于程序员健康。
根据经验，设计了如下的LCD架构，希望对大家有用。

> 最低层绿色，是接口设备，不属于显示模块。
> 右边浅紫色是触摸模块。
> 橙色，是我们抽象统一的LCD硬件接口层。
> 浅蓝色则是不同的LCD驱动IC。
> 最顶则是LCD接口和基本的GUI功能。



4.1 硬件接口抽象层

设计这一层的理念是：

> 同一个屏幕，通常可以支持多种通信接口，例如SSD1615的OLED屏幕，就可以用在SPI或者I2C接口上。
> 不同的LCD，通常都有下面硬件：通信接口、A0（DC）、复位脚、背光脚。如果我们将这些标准化，那么蓝色层，就可以用一套接口实现不同的硬件连接。

- 文件：dev_lcdbus.c、dev_lcdbus.h
- 接口

```
extern s32 bus_lcd_bl(DevLcdBusNode *node, u8 sta);
extern s32 bus_lcd_rst(DevLcdBusNode *node, u8 sta);
extern DevLcdBusNode *bus_lcd_open(char *name);
extern s32 bus_lcd_close(DevLcdBusNode *node);
extern s32 bus_lcd_write_data(DevLcdBusNode *node, u8 *data, u32 len);
extern s32 bus_lcd_flush_data(DevLcdBusNode *node, u8 *data, u32 len);
extern s32 bus_lcd_flush_wait(DevLcdBusNode *node);
extern s32 bus_lcd_read_data(DevLcdBusNode *node, u8 *data, u32 len);
extern s32 bus_lcd_write_cmd(DevLcdBusNode *node, u8 cmd);
extern s32 dev_lcdbus_register(const DevLcdBus *dev);
```

- 主要数据结构

```
/*
        系统总共有三种LCD总线, SPI,I2C,8080
        */
typedef enum{
        LCD_BUS_NULL = 0,
        LCD_BUS_SPI,
        LCD_BUS_I2C,
        LCD_BUS_8080,
        LCD_BUS_MAX
}LcdBusType;
/*

        lcdbus设备定义

*/
typedef struct
{
        /*设备名称*/
        char name[DEV_NAME_SIZE];
        /*总线类型: SPI or I2C or 8080*/
        LcdBusType type;
        /*总线名字*/
        char basebus[DEV_NAME_SIZE];

        /*
                3根线: A0-命令数据, rst-复位, bl-背光
                I2C总线的LCD不需要这三根线
        */

        MCU_PORT A0port;
        u16 A0pin;

        MCU_PORT rstport;
        u16 rstpin;

        MCU_PORT blport;
        u16 blpin;
}DevLcdBus;
/*

        lcdbus节点

*/
typedef struct
{
        s32 gd;

        DevLcdBus dev;

        void *basenode;

        struct list_head list;
}DevLcdBusNode;
```

- 定义3个LCDBUS接口

```
/*
        串行LCD接口，使用真正的SPI控制
        外扩SPI
*/
const DevLcdBus BusLcdSpi3={
        .name = "BusLcdSpi3",
        .type = LCD_BUS_SPI,
        .basebus = "SPI3_CH3",

        .A0port = MCU_PORT_G,
        .A0pin = GPIO_Pin_4,

        .rstport = MCU_PORT_G,
        .rstpin = GPIO_Pin_7,

        .blport = MCU_PORT_G,
        .blpin = GPIO_Pin_9,
};


const DevLcdBus BusLcdI2C1={
        .name = "BusLcdI2C1",
        .type = LCD_BUS_I2C,
        .basebus = "VI2C1",

        /*I2C接口的LCD总线，不需要其他IO*/

};

const DevLcdBus BusLcd8080={
        .name = "BusLcd8080",
        .type = LCD_BUS_8080,
        .basebus = "8080",//不使用用，8080操作直接嵌入在LCD BUS代码内

        /*8080 不用A0脚，填背光进去*/
        .A0port = MCU_PORT_B,
        .A0pin = GPIO_Pin_15,

        .rstport = MCU_PORT_A,
        .rstpin = GPIO_Pin_15,

        .blport = MCU_PORT_B,
        .blpin = GPIO_Pin_15,

};
```

将他们注册到LCDBUS系统

```
/*注册LCD总线*/
dev_lcdbus_register(&BusLcdSpi3);
dev_lcdbus_register(&BusLcdI2C1);
dev_lcdbus_register(&BusLcd8080);
```

4.2 显示IC驱动

就是实现不同的LCD驱动。
如何实现不同的显示IC驱动？那就要知道LCD到底是什么，能做什么，功能是什么？
我们经过讨论，一致认为，**无论哪种LCD，就是一个显示点的设备**，根据这个结论定义了一套驱动接口。

- 文件：dev_ILI9341.c、dev_ILI9341.h和dev_str7565.c、dev_str7565.h
- 接口

```
/*
        LCD驱动定义
*/
typedef struct
{
        u16 id;

        s32 (*init)(DevLcdNode *lcd);

        s32 (*draw_point)(DevLcdNode *lcd, u16 x, u16 y, u16 color);
        s32 (*color_fill)(DevLcdNode *lcd, u16 sx,u16 ex,u16 sy,u16 ey, u16 color);
        s32 (*fill)(DevLcdNode *lcd, u16 sx,u16 ex,u16 sy,u16 ey,u16 *color);

        s32 (*prepare_display)(DevLcdNode *lcd, u16 sx, u16 ex, u16 sy, u16 ey);
        s32 (*flush)(DevLcdNode *lcd, u16 *color, u32 len);

        s32 (*onoff)(DevLcdNode *lcd, u8 sta);
        void (*set_dir)(DevLcdNode *lcd, u8 scan_dir);
        void (*backlight)(DevLcdNode *lcd, u8 sta);
}_lcd_drv;
```

- 原理
  LCD驱动，是一个很好的抽象，与LCD设备和LCDBUS完全解耦合。
  所有需要的硬件信息，全部通过lcd指针传入。
  这也就意味着，同一个驱动，可以支持多个LCD设备，或者是多种LCD接口。
  例如：
  STR7565驱动，可以同时支持两个COG LCD，一个是12864，挂载SPI1_ch1，一个是12832，挂在SPI1_CH2。
  SSD1565驱动，可以同时支持两个OLED LCD，一个是SPI接口，一个是I2C接口。

如果你只有一个LCD，用了这套代码，也很容易修改他的链接方式，只要挂到不同的LCDBUS即可。

- 添加驱动
  1 按照范例添加驱动。
  2 把驱动添加到驱动列表。

```
/*
        所有驱动列表
*/
_lcd_drv *LcdDrvList[] = {
                                        &TftLcdILI9341Drv,
                                        &TftLcdILI9325Drv,
                                        &CogLcdST7565Drv,
                                        &OledLcdSSD1615rv,
                                        &TftLcdILI9341_8_Drv,
                                        &TftLcdST7735R_Drv,
};
```

4.3 LCD设备层

驱动层将LCD跟LCDBUS链接起来，那么就需要LCD层提供设备信息。
就像上面的架构图一样，一个LCD跟那些东西有关系，需要哪些信息呢？
1 设备定义：一个叫什么名字的LCD挂在哪个LCDBUS上，LCD是什么类型（ID）？
2 知道了ID，就可以通过设备参数查找LCD信息，长宽等。
3 根据ID，也就能匹配驱动。
4 根据挂载信息，驱动就能找到接口操作函数。

- 文件：dev_lcd.c、dev_lcd.h
- 接口
  这些接口提供给APP或者是GUI使用

```
extern s32 dev_lcd_register(const DevLcd *dev);
extern DevLcdNode *dev_lcd_open(char *name);
extern s32 dev_lcd_close(DevLcdNode *node);
extern s32 dev_lcd_drawpoint(DevLcdNode *lcd, u16 x, u16 y, u16 color);
extern s32 dev_lcd_prepare_display(DevLcdNode *lcd, u16 sx, u16 ex, u16 sy, u16 ey);
extern s32 dev_lcd_fill(DevLcdNode *lcd, u16 sx,u16 ex,u16 sy,u16 ey,u16 *color);
extern s32 dev_lcd_color_fill(DevLcdNode *lcd, u16 sx,u16 ex,u16 sy,u16 ey,u16 color);
extern s32 dev_lcd_backlight(DevLcdNode *lcd, u8 sta);
extern s32 dev_lcd_display_onoff(DevLcdNode *lcd, u8 sta);
extern s32 dev_lcd_setdir(DevLcdNode *node, u8 dir, u8 scan_dir);
```

- 主要数据结构

```
/*
        设备定义
        包含挂载方式定义
        也就是说明有一什么ID的设备挂载什么地方
        例如定义一个COG LCD挂载在SPI3上
        用什么驱动？LCD具体参数是什么？通过ID匹配
        同一个类型的LCD，驱动相同，只是像素大小不一样，如何处理？
        可以重生一个驱动结构体，函数一样，ID不一样。
*/
typedef struct
{
        char name[DEV_NAME_SIZE]; //设备名字

        char buslcd[DEV_NAME_SIZE]; //挂在那条LCD总线上

        u16 id;

        u16 width;      //LCD 宽度    竖屏
        u16 height;     //LCD 高度     竖屏
}DevLcd;
/*
        初始化的时候会根据设备数定义，
        并且匹配驱动跟参数，并初始化变量。
        打开的时候只是获取了一个指针
*/
struct _strDevLcdNode
{
        s32 gd;//句柄，控制是否可以打开

        DevLcd  dev;

        /* LCD驱动 */
        _lcd_drv *drv;

        /*驱动需要的变量*/
        u8  dir;        //横屏还是竖屏控制：0，竖屏；1，横屏。
        u8  scandir;//扫描方向
        u16 width;      //LCD 宽度
        u16 height;     //LCD 高度

        void *pri;//私有数据，黑白屏跟OLED屏在初始化的时候会开辟显存

        DevLcdBusNode *busnode;

        struct list_head list;
};
```

- 添加LCD

> 添加LCD之前，请已经实现LCDBUS和LCD驱动。
> 先定义一个LCD设备，然后调用dev_lcd_register将其注册到LCD模块。

```
/*I2C接口的 OLED*/
const DevLcd DevLcdOled1={
        .name = "i2coledlcd",
        .buslcd = "BusLcdI2C1",
        .id = 0X1315,
        .width = 64,
        .height = 128
        };

        ```
        /*注册LCD设备*/
        dev_lcd_register(&DevLcdOled1);
```

- 使用LCD
  使用前先执行dev_lcd_open，获取LCD节点。
  然后就可以使用LCD提供的函数了。

4.4 显示

所谓的显示是属于GUI层，也就是基于LCD接口实现一些简单的功能。
例如划线，显示字符等。
目前还不是很完善，有待补充。

4.5 字库

最新代码支持汉字全字库，字库放在SD卡中。

4.6 BMP图片显示

最大代码已经实现BMP图片解码，速度已经过优化。

5 其他模块

5.1 触摸屏检测

实现XPT2046、STM32内部ADC另种检测方案

5.2 tslib

LINUX下的触摸屏校准库，成功移植到STM32。

5.3 矩阵按键检测

end
```