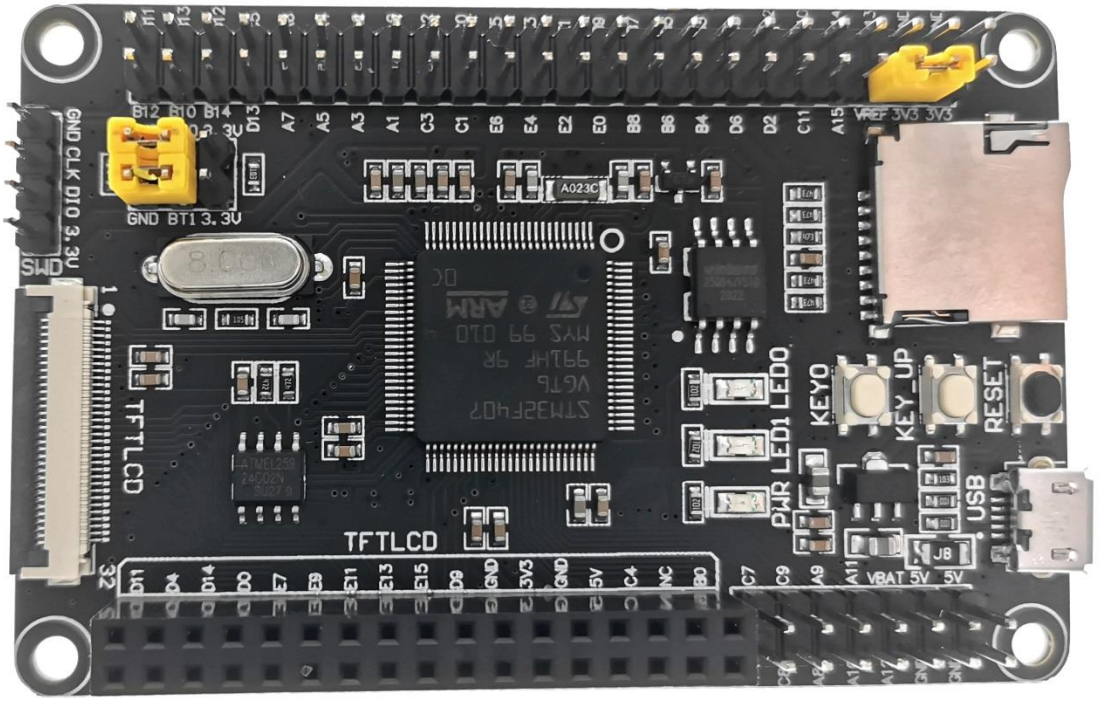
# STM32F407VxT6 最小系统板使用说明

--STM32F407VGT6

--STM32F407VET6




V1.0

2020-07-20

# 目录

# 1. 最小系统板简介

## 1.1 板载资源介绍

此 STM32F407VxT6 最小系统板推出的目的就是方便用户进行项目 DIY。它可以兼容 STM32F407VGT6 和 STM32F407VET6 两款 STM32 芯片，体积虽小，功能还是比较强大的。下面我们开始介绍此款最小系统板。

STM32F407VxT6 最小系统板板载资源如图 1.1 所示：

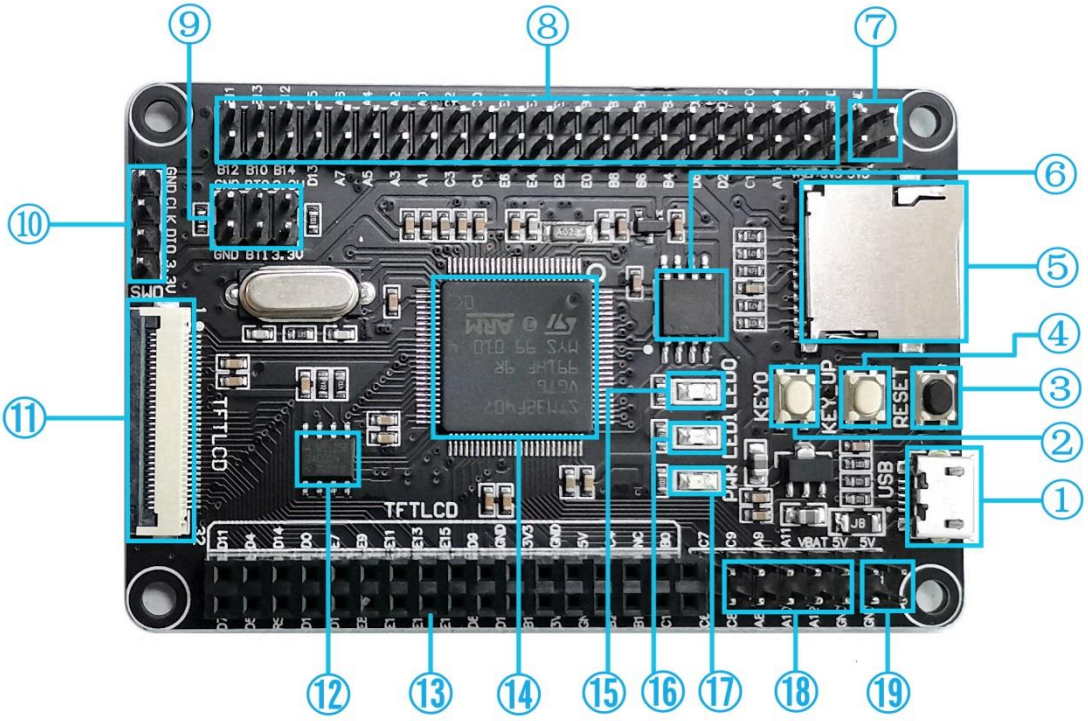



图 1.1 STM32F407VxT6 最小系统板板载资源

① USB 通信接口
② KEY0 测试按键
③ RESET 复位按键
④ KEY_UP 测试/唤醒按键
⑤ SD 卡卡槽
⑥ W25Q64 SPI Flash
⑦ 3.3V 电源输入/输出引脚
⑧ 扩展 IO
⑨ BT0/BT1 启动选择端口
⑩ SWD 下载仿真接口
⑪ TFT_LCD FPC 排线直插接口
⑫ 24C02 EEPROM

⑬ TFT_LCD 排针直插接口
⑭ STM32F407VGT6/STM32F407VET6 主芯片
⑮ LED0 测试灯（蓝色）
⑯ LED1 测试灯（蓝色）
⑰ 电源指示灯（红色）
⑱ 扩展 IO
⑲ 5V 电源输入/输出引脚

## 1.2 板载资源详细说明

### ① USB 通信接口

此接口为一个 Micro USB 口，用于最小系统板和 PC 机进行 USB 通信，包括 USB 从机（SLAVE）通信（最小系统板作为 U 盘）和 USB 主机（HOST）通信（最小系统板作为主机）。此接口还可以作为电源供电接口（作为 USB 主机时除外）。

### ② KEY0 测试按键

此按键和主芯片的 PE4 引脚相连，作为普通的测试按键使用，可用于程序中按键输入功能。

### ③ RESET 复位按键

此按键和主芯片的复位引脚相连，用于复位主芯片，另外当 LCD TFT 直插或者用 FPC 软排线直连时，它还可以复位 LCD TFT。

### ④ KEY_UP 测试/唤醒按键

此按键和主芯片的 PA0（WAKE_UP）引脚相连，用于在待机状态下唤醒主芯片。如果不使用唤醒功能，则可以作为普通按键输入使用，例如控制 LED 亮灭等等。

### ⑤ SD 卡卡槽

此卡槽为 Micro SD 卡卡槽，采用 SDIO 方式驱动，用于插入 SD 卡，扩大最小系统板的数据存储空间。

### ⑥ W25Q64 SPI Flash

此设备为最小系统板外扩的 SPI Flash 芯片，容量为 64M bits（8M Bytes），可用于存储字库、图片和其他的用户数据，扩大数据存储空间。

### ⑦ 3.3V 电源输入/输出引脚

此引脚为两组 3.3V 电源输入和输出排针（包括正极 3.3V 和 GND），可用于对外部提供 3.3V 电源，也可从外部接入 3.3V 电源给最小系统板供电。

⑧⑱ 扩展 IO

此扩展 IO 口包括三处：2x25 双排针接口、2x5 双排针接口以及 TFT_LCD 排针直插接口。它们都和主芯片的 GPIO 引脚相连，为最小系统板 IO 引出端口，用于连接外部设备。除了 RTC 晶振占用的两个 IO(PC14 和 PC15),其他 IO 都引出来了，另外 RST、BAT、VREF 也都引出来了。需要注意的是如果 TFT_LCD 排针直插接口接了 LCD 模块，则其不能当做扩展 IO 口使用。

⑨ BT0/BT1 启动选择端口

此端口为一组 2x3 的排针，其中 BT0 和 BT1 引脚位于中间，两边引脚分别为 3.3V 和 GND。它用于选择主芯片复位后的启动模式，通过跳帽来选择连接 3.3V（高电平）还是 GND（低电平）。关于启动模式说明，请看 2.1 节。

⑩ SWD 下载仿真接口

此接口为四针单排接口，可提供 SWD 方式对最小系统板进行程序下载和仿真。如果不使用 SWD 功能，其中 DIO（PA13）和 CLK（PA14）可做普通 IO 口使用。

⑪ TFT_LCD FPC 排线直插接口

此接口为一个 32PIN FPC 座子，LCD 模块通过 FPC 软排线和它相连（不带 FPC 座子的 LCD 模块需要通过转接板连接）。该接口兼容所有的 LCD 并口模块，包括 2.4 寸、2.8 寸、3.2 寸、3.5 寸、3.97 寸、4.0 寸、5 寸和 7 寸等等，同时支持电阻触摸屏和电容触摸屏。

⑫ 24C02 EEPROM

此设备为最小系统板外扩的 EEPROM 芯片，容量为 2K bits（256 Bytes），用于存取一些掉电不能丢失、数量较少且读取速度和次数要求不高的数据，例如电阻触摸屏校准参数和一些系统设置参数等等。

⑬ TFT_LCD 排针直插接口

此接口为一个 2x17 的双排母接口，LCD 模块通过排针直插和它相连（不带排针的 LCD 模块需要通过转接板连接）。该接口兼容所有的 LCD 并口模块，包括 2.4 寸、2.8 寸、3.2 寸、3.5 寸、3.97 寸、4.0 寸、5 寸和 7 寸等等，同时支持电阻触摸屏和电容触摸屏。另外如果不使用 LCD 模块，该接口可以当做扩展 IO 使用。

⑭ STM32F407VGT6/STM32F407VET6 主芯片

此设备为最小系统板的主芯片，此最小系统板兼容 2 款主芯片：STM32F407VGT6 和 STM32F407VET6，它们只是内部 flash 容量不一样（F407VGT6 为 1024KB，F407VET6

为 512KB)。此两款主芯片都集成了 FPU 和 DSP 指令，并具有 192KB SRAM、12 个 16 位定时器、2 个 32 位定时器、2 个 DMA 控制器(共 16 个通道)、3 个 SPI、2 个全双工 I2S、3 个 IIC、6 个串口、2 个 USB（支持 HOST /SLAVE)、2 个 CAN、3 个 12 位 ADC、2 个 12 位 DAC、1 个 RTC（带日历功能)、1 个 SDIO 接口、1 个 FSMC 接口、1 个 10/100M 以太网 MAC 控制器、1 个摄像头接口、1 个硬件随机数生成器、以及 80 个通用 IO 口等。

⑮ **LED0 测试灯（蓝色）**

此设备为一颗蓝色的 LED 测试灯。可用于程序中测试 GPIO 输出、指示程序运行状态或者实现某些灯光效果等功能。

⑯ **LED1 测试灯（蓝色）**

此设备为一颗蓝色的 LED 测试灯。可用于程序中测试 GPIO 输出、指示程序运行状态或者实现某些灯光效果等功能。

⑰ **电源指示灯（红色）**

此设备为一颗红色的 LED 指示灯。用于指示最小系统板上电情况，如果电源正常接通，该指示灯会点亮，否则不亮。

⑲ **5V 电源输入/输出引脚**

此引脚为两组 5V 电源输入和输出排针（包括正极 5V 和 GND)，可用于对外部提供 5V 电源，也可从外部接入 5V 电源给最小系统板供电。

# 2. 硬件资源详解

## 2.1 硬件原理图详解

STM32F407VxT6 最小系统板硬件原理图包含如下 14 个部分：

- ✧ 主芯片
- ✧ USB 通信接口
- ✧ 电源控制
- ✧ 启动模式选择
- ✧ SWD 接口
- ✧ TFT_LCD FPC 接口
- ✧ 外部扩展 IO 接口
- ✧ LED 灯控制
- ✧ 按键控制
- ✧ RTC 电池控制
- ✧ 复位控制
- ✧ SD 卡控制

- ✧ EEPROM 控制
- ✧ SPI Flash 控制

各部分原理图的详细说明如下所示：

### 2.1.1 主芯片

STM32F407VxT6 最小系统板兼容两款主 IC：STM32F407VGT6 和 STM32F407VET6。此两款主 IC 除了内部 Flash 容量不一样（F407VGT6 为 1024KB，F407VET6 为 512KB），其他的功能都是一样的，所以原理图完全通用。

主芯片的原理图如图 2.1 所示。

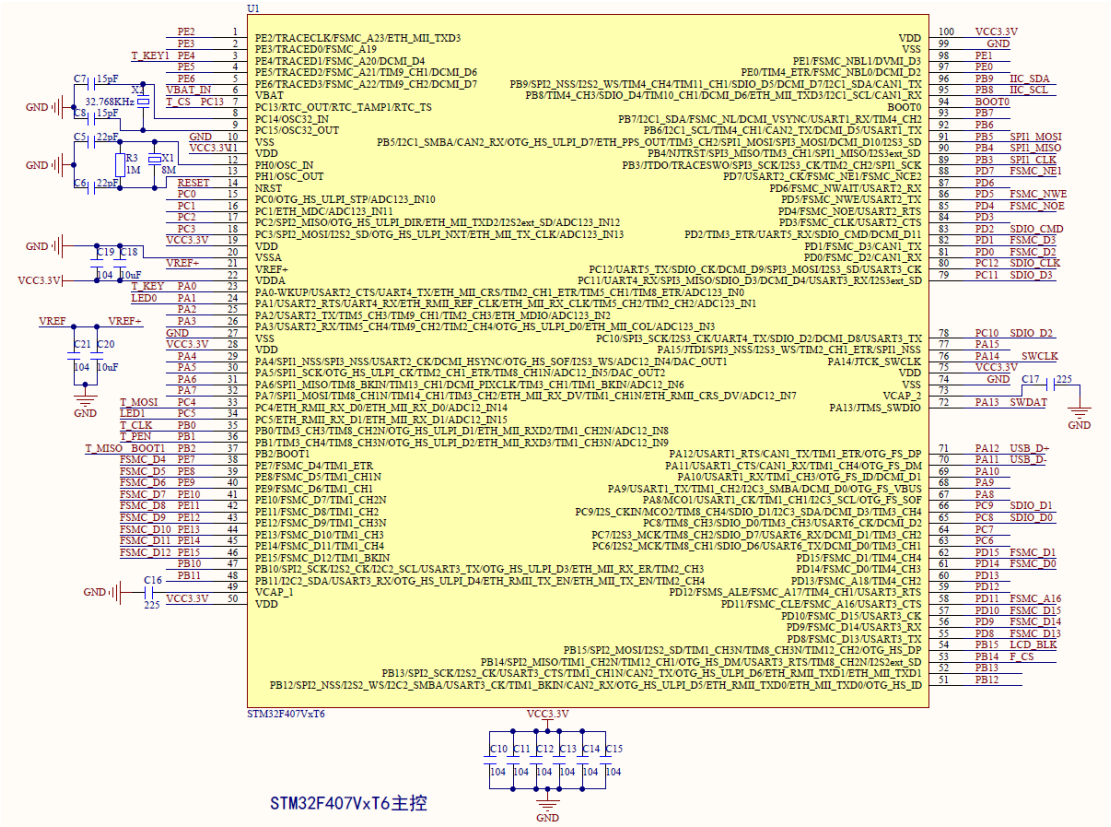



图 2.1 主芯片原理图

从原理图中可以看到 U1 为主芯片 STM32F407VGT6 或 STM32F407VET6。还包括如下几部分电路：

A、 RTC 实时时钟晶振（32.768KHz）电路，用于给 RTC 模块提供计时时钟源；

B、 主芯片晶振（8M）电路，用于给主芯片提供时钟源；

C、 主芯片输入电源滤波电路，通过并联 104 和 106 电容，使主芯片输入电压保存文档；

D、 VREF 参考参考电压输入电路，其中 VREF+接入主芯片的 IO，VREF 接外部输入的参考电压。如果进行 ADC 采集测试时，VREF+引脚必须给一个参考电压，可

以通过跳帽连接到旁边的 3.3V 电源上。

### 2.1.2 USB 通信接口

USB 通信接口原理图如图 2.2 所示：

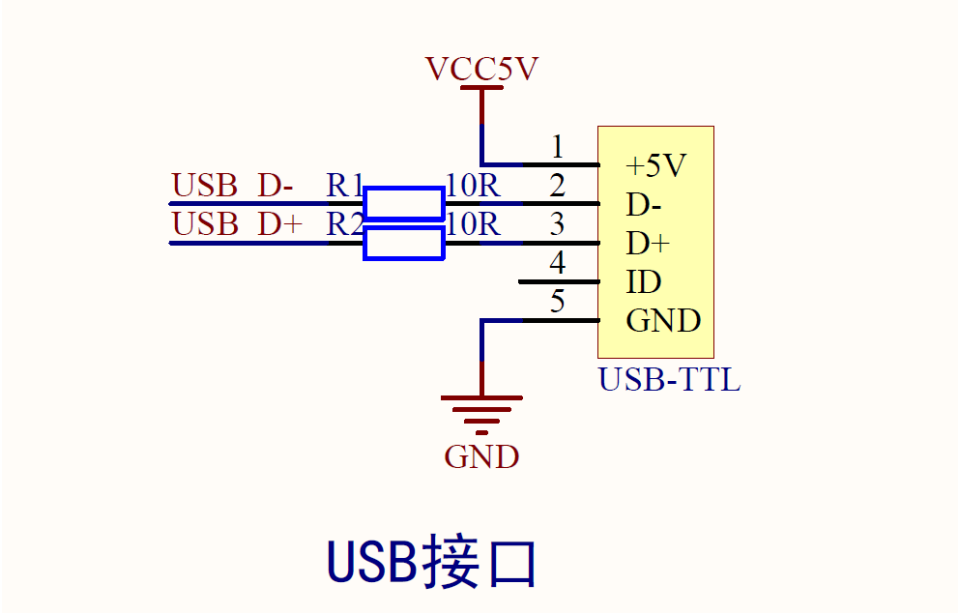



图 2.2 USB 接口原理图

图中 USB_D-和 USB_D+分别连接在主芯片的 PA11 和 PA12 引脚上。R1 和 R2 串联到电路用于平衡阻抗。VCC5V 用于接入 5V 电源，GND 接地。

该接口可以做 USB 从机通信、USB 主机通信，还可以通过 VCC5V 输入 5V 电源给最小系统板供电。

### 2.1.3 电源控制

电源控制原理图如图 2.3 所示：

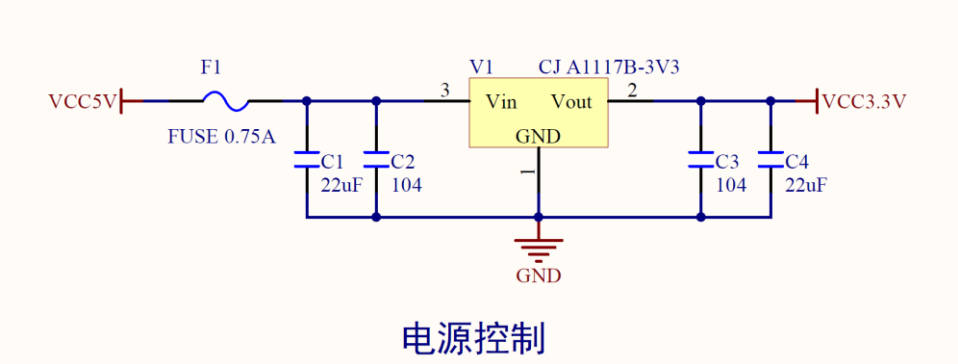



图 2.3 电源控制原理图

图中 VCC5V 为外部输入的 5V 电源，经过 F1 保险丝后，输入 CJA1117B-3V3 稳压芯片，经过转换后输出 VCC3.3V。其中：

F1 为 750mA 自恢复保险丝，一旦负载电流超过 750mA,F1 就自动断开，切断供电，

从而保护最小系统板器件不因电流过大而烧毁。

C1 和 C2 为 5V 输入端旁路滤波电容，用来维持输入电压稳定。

C3 和 C4 为 3.3V 输出端旁路滤波电容，用来维持输出电压稳定。

CJA1117B-3V3 为 3.3V 输出稳压芯片，用来给最小系统板提供稳定的 3.3V 电源。

### 2.1.4 启动模式选择

启动模式选择电路如图 2.4 所示：

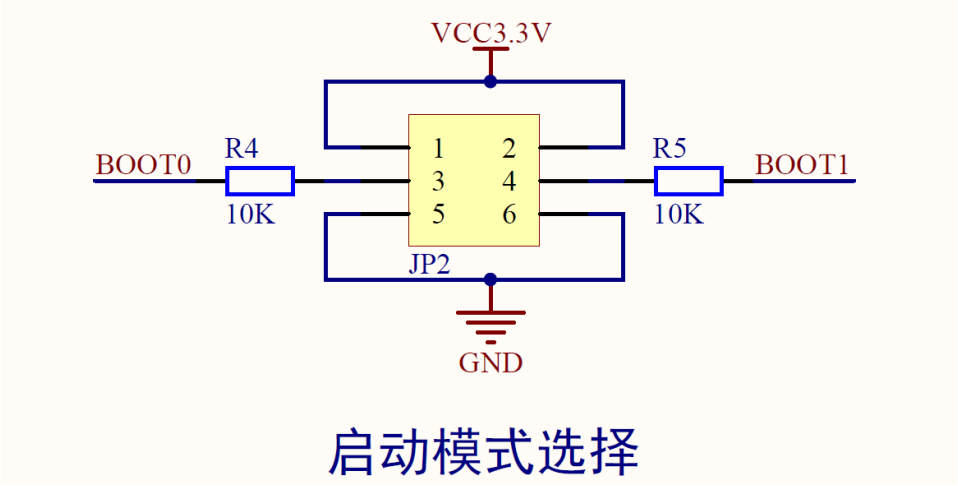



图 2.4 启动模式选择原理图

图中 BOOT0 和 BOOT1 分别接在主芯片的 BOOT0 和 PB2 引脚上。我们使用跳帽来选择 BOOT0 和 BOOT1 是接 VCC3.3V（高电平），还是 GND（低电平）。R4 和 R5 的作用取决于 BOOT0 和 BOOT1 所接的电平，如果接高电平，它们就起上拉电阻的作用，如果接低电平，它们就起下拉电阻的作用。BOOT0 和 BOOT1 对应的启动模式说明如表 2.1 所示：

| BOOT0 | BOOT1 | 启动模式 | 说明 |
|---|---|---|---|
| 0 | X | 用户闪存存储器 | 用户闪存存储器，也就是 FLASH 启动 |
| 1 | 0 | 系统存储器 | 系统存储器启动，用于串口下载 |
| 1 | 1 | SRAM 启动 | SRAM 启动，用于在 SRAM 中调试代码 |

表 2.1 BOOT0 和 BOOT1 对应的启动模式说明

从表格的说明看来，如果程序从 flash 正常启动或者采用 SWD 方式进行程序下载和仿真时，需要将 BOOT0 拉低，BOOT1 不需要关心，如果采用串口方式下载程序，则需要将 BOOT0 拉高，BOOT1 拉低。

### 2.1.5 SWD 接口

SWD 接口原理图如图 2.5 所示：

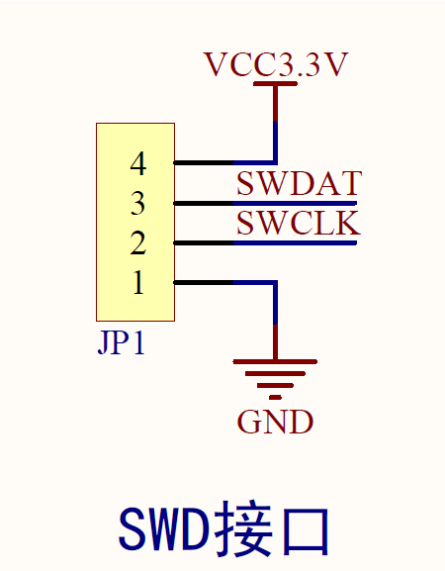



图 2.5 SWD 接口原理图

图中，SWDAT 和 SWCLK 分别接在主芯片的 PA13 和 PA14 引脚上。VCC3.3V 为 3.3V 电源输出和输入引脚，如果最小系统板已经上电，此引脚可以不接。

该接口用于 SWD 方式下载和仿真程序，只要仿真器（如 ST-LINK、JTAG）支持 SWD 模式，都可以使用该接口。如果想将 SWDAT 和 SWCLK 引脚作为普通 IO 使用，则需要在程序里将 SWD 功能禁止，此时无法使用 SWD 功能，如果想再次使用，则需要将 BTO 拉高再上电。

### 2.1.6 TFT_LCD FPC 接口

TFT LCD FPC 接口原理图如图 2.6 所示：

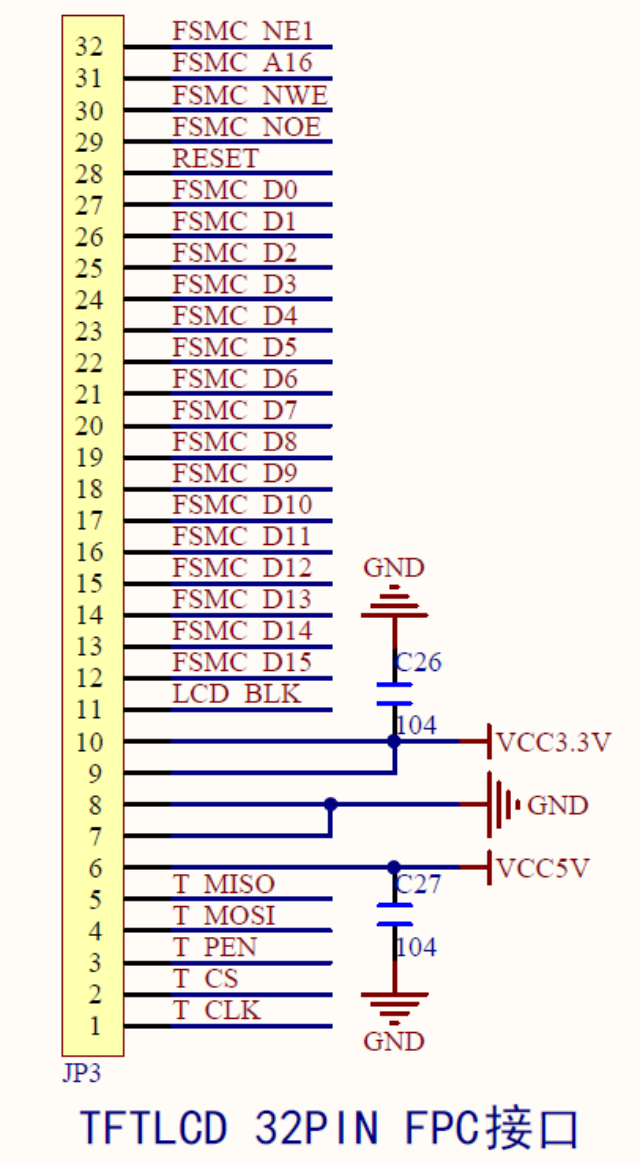



图 2.6 TFT_LCD 32PIN FPC 接口原理图

图中 JP3 是一个 32 PIN 的 FPC 座子，LCD 模块通过 FPC 软排线和其连接（LCD 模块如果没有 FPC 座子，则需通过转接板连接）。C26 和 C27 为电源旁路滤波电容。

LCD 模块采用 FSMC 总线驱动，所以 JP3 中的部分引脚是直接挂在 FSMC 总线上的，其中：

FSMC_NE1 为片选引脚连接在主芯片 PD7 引脚上；

FSMC_A16 为数据/命令选择引脚，连接在主芯片 PD11 引脚上；

FSMC_NEW 为写控制引脚，连接在主芯片的 PD5 引脚上；

FSMC_NOE 为读控制引脚，连接在主芯片的 PD4 引脚上；

RESET 为复位引脚，连接在主芯片复位引脚上；

FSMC_D0~D15 为 16 位并口数据传输引脚，其中 D0~D1 分别连接在主芯片的

PD14~PD15 引脚上，D2~D3 分别连接在 PD0~PD1 引脚上，D4~D12 分别连接在 PE7~PE15 引脚上，D13~D15 分别连接在 PD8~PD10 引脚上；

LCD_BLK 为背光控制引脚，连接在主芯片的 PB15 引脚上；

T_MISO 为触摸屏数据输出引脚，连接在主芯片 PB2 引脚上；

T_MOSI 为触摸屏数据输入引脚，连接在主芯片 PC4 引脚上；

T_PEN 为触摸屏中断检测引脚，连接在主芯片 PB1 引脚上；

T_CS 为触摸屏片选控制引脚，连接在主芯片 PC13 引脚上；

T_CLK 为触摸屏时钟引脚，连接在主芯片 PB0 引脚上。

### 2.1.7 外部扩展 IO 接口

外部扩展 IO 口原理图如图 2.7 所示：

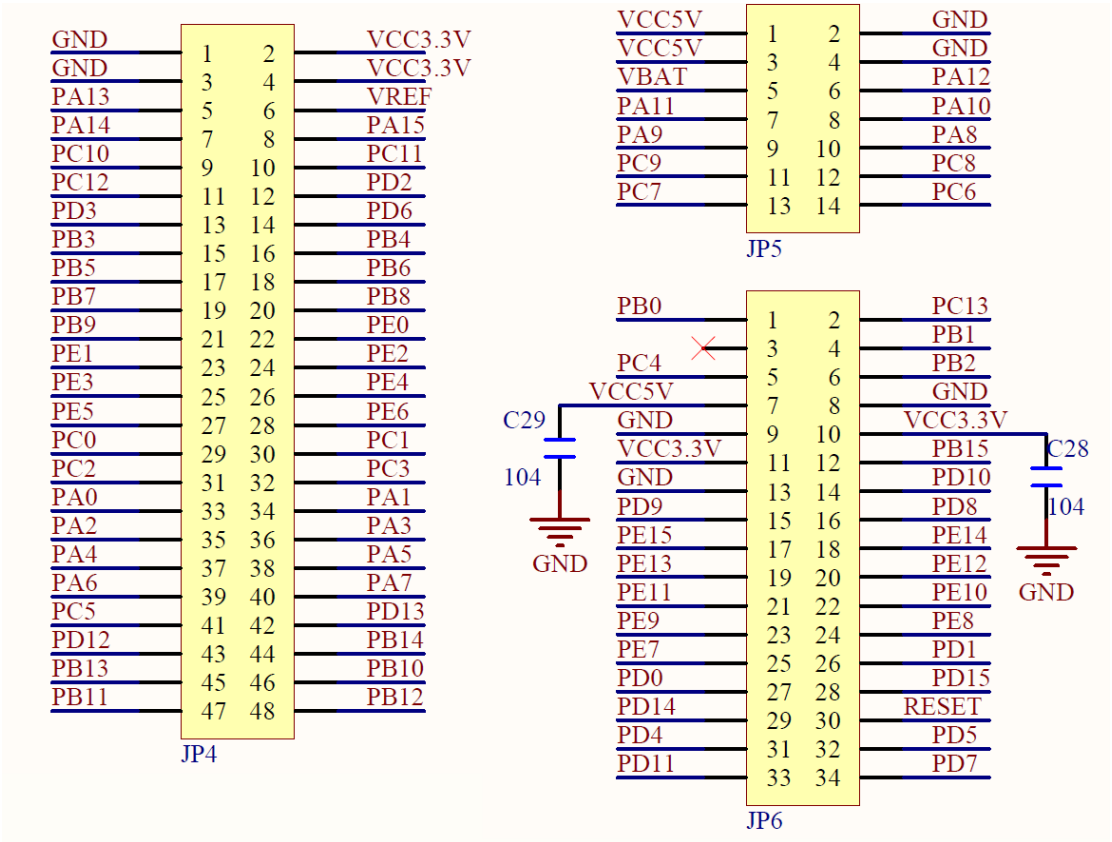



图 2.7 外部扩展 IO 接口原理图

图中扩展 IO 接口由 3 部分组成：JP4、JP5、JP6。扩展 IO 口中包含两种类型 IO：电源输出/输入 IO 和功能 IO（直接和主芯片的功能 GPIO 连接）。其中：

JP4 的 1~4 脚为 3.3V 电源输出/输入 IO，其他的都是功能 IO；

JP5 的 1~4 脚为 5V 电源输入/输出 IO，其他的都是功能 IO；

JP6 为 LCD 模块排针直插接口，当不使用 LCD 时，才能作为扩展 IO 口使用；

### 2.1.8 LED 灯控制

LED 灯控制原理图如图 2.8 所示：

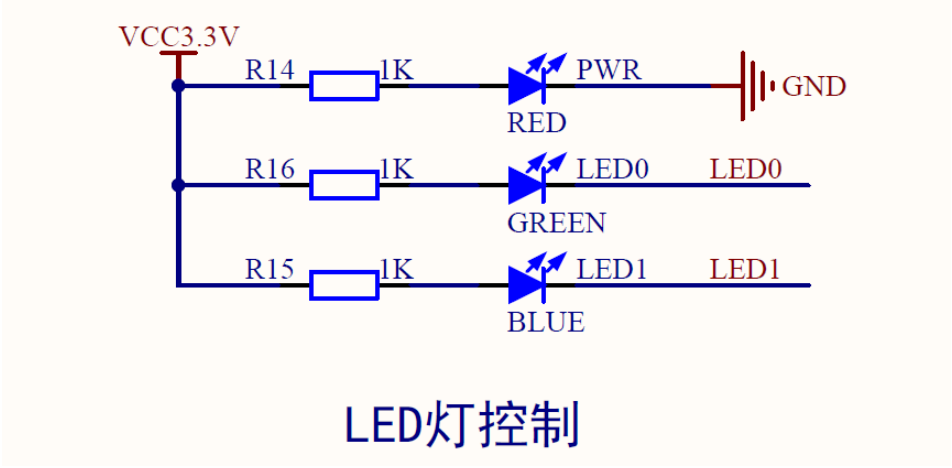



图 2.8 LCD 灯控制原理图

最小系统板共有 3 颗 LED 灯，其中：

PWR 是电源指示灯，为红色；

LED0 和 LED1 是测试用的 LED 灯，都为蓝色，它们分别接在主芯片的 PA1 和 PC5 引脚上。当 LED0 和 LED1 端口被接入低电平时，LED0 和 LED1 就会被点亮。

R14、R15、R16 是限流电阻，防止电流过大，烧毁 LED 灯。

### 2.1.9 按键控制

按键控制原理图如图 2.9 所示：

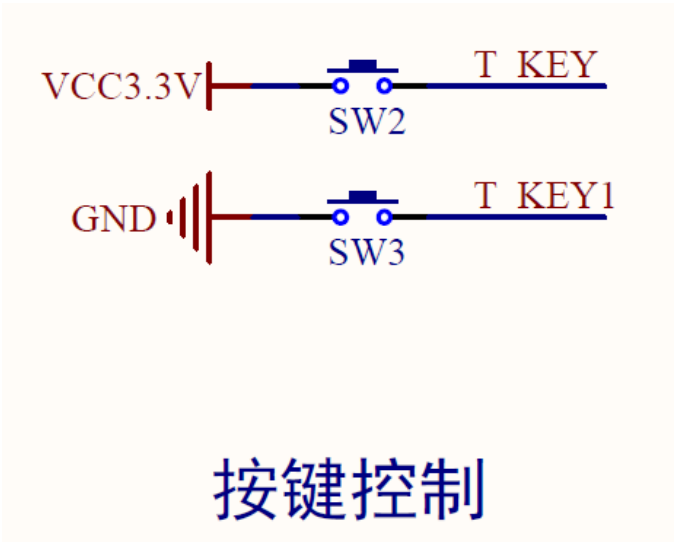



图 2.9 按键控制原理图

最小系统板共有两个输入按键：KEY0（T_KEY1）和 KEY_UP（T_KEY）。其中：

KEY0 按键连接在主芯片的 PE4 引脚上，该按键为低电平触发，所以程序中初始化

时需要配置成上拉输入。

KEY_UP 按键连接在主芯片 PA0 引脚（唤醒引脚）上，该按键为高电平触发，所以程序中初始化需要设置成下拉输入。不使用唤醒功能时，该按键可以作为普通按键。

### 2.1.10 RTC 外部电池控制

RTC 外部电池控制原理图如图 2.10 所示：

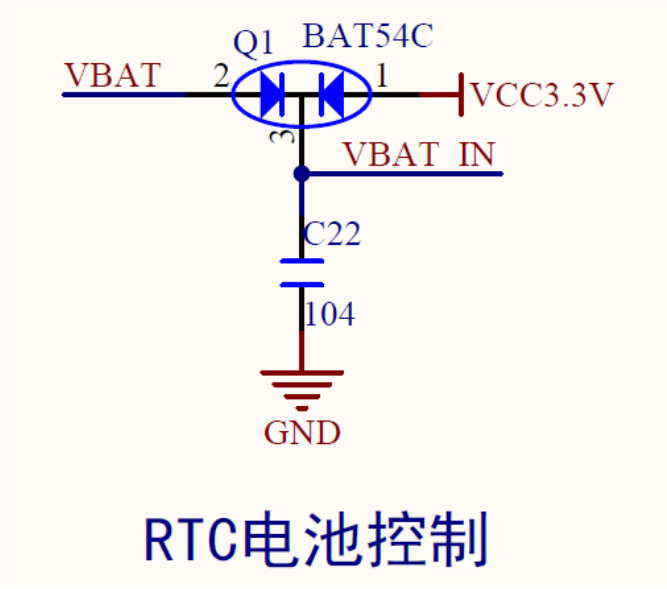



图 2.10 RTC 外部电池控制原理图

图中 Q1（BAT54C）为一个双向二极管，隔离了 VBAT 和 VCC3.3V 输入端；

VBAT 为外部电池输入端，例如接入纽扣电池；

VCC3.3 为板载 3.3V 电源输入端；

VBAT_IN 连接在主芯片的 VBAT 引脚上，给 RTC 供电；

C22 为电源旁路滤波电容，维持 RTC 输入电源稳定。

RTC 模块供电有三种情况：

A、 直接使用板载 3.3V 电源供电

这是默认情况，在没有外部电池供电接入的时候，就用板载电源供电（前提是最小系统板要上电）。

B、 使用外部电池供电

当最小系统板已经断电，但是 RTC 模块还需要继续工作，保证计时准确时，就可以采用这种方式，接入外部电池给 RTC 模块供电。

C、 同时使用板载电源和外部电池供电

这种情况就是上述两种情况的重合，供电电源以高电压为准。

### 2.1.11 复位控制

复位控制原理图如图 2.11 所示：

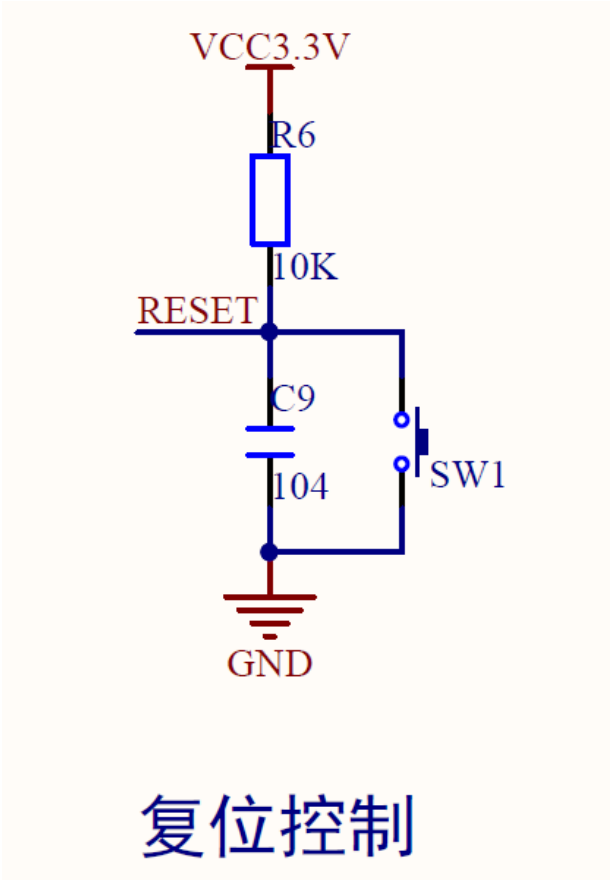



图 2.11 复位控制原理图

图中 RESET 连接在主芯片的 RST 引脚（复位引脚）上；

SW1 为复位按键，方便用户随时可以对主芯片进行复位；

R6 为上拉电阻，C9 为旁路电容；

STM32 主芯片为低电平复位，所以电路也设计成了低电平复位，其中 R6 和 C9 构成了上电复位电路。原理分析如下：

当最小系统板刚上电时，C9 会立即进行充电，此时 C9 两端相当于短路，RESET 脚接到 GND，主芯片复位引脚被拉低，进入复位状态，随着 C9 充电完毕，C9 两端相当于开路，RESET 脚被 R6 上拉到高电平，上电复位完成，所以只要 C9 容值选择恰当，保证 RESET 引脚被拉低的时间大于主芯片需要的复位时间，就可以实现上电复位。在运行过程中，一旦 SW1 按键按下，RESET 接到 GND，主芯片进入复位状态，SW1 松开，RESET 被拉高，复位完成，从而实现按键复位。

### 2.1.12 SD 卡控制

SD 卡控制原理图如图 2.12 所示：

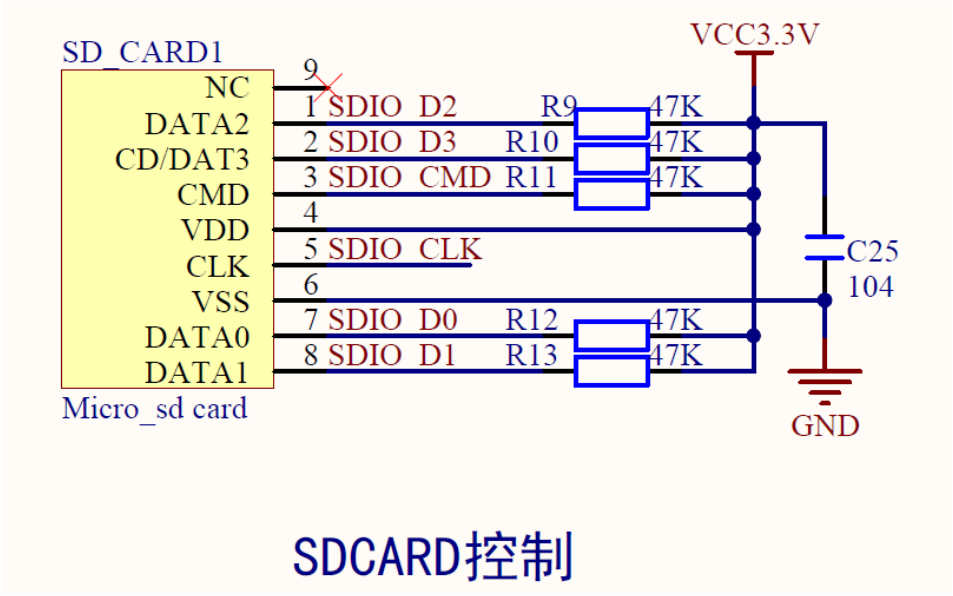



图 2.12 SD 卡控制原理图

图中 SD 卡接口采用 4 位 SDIO 方式驱动，理论最大速度可达 24MB/S。其中：

SDIO_CMD 为命令传输引脚，连接在主芯片的 PD2 引脚上；

SDIO_CLK 为时钟控制引脚，连接在主芯片的 PC12 引脚上；

SDIO_D0~D3 为 4 位数据传输引脚，分别连接在主芯片 PC8/PC9/PC10/PC11 引脚上；

当不使用 SD 卡时，以上所连接的引脚都可以当做普通 IO 使用。

R9~R13 都为 47K 上拉电阻，C25 为电源旁路滤波电容；

### 2.1.13 EEPROM 控制

EEPROM 控制原理图如图 2.13 所示：

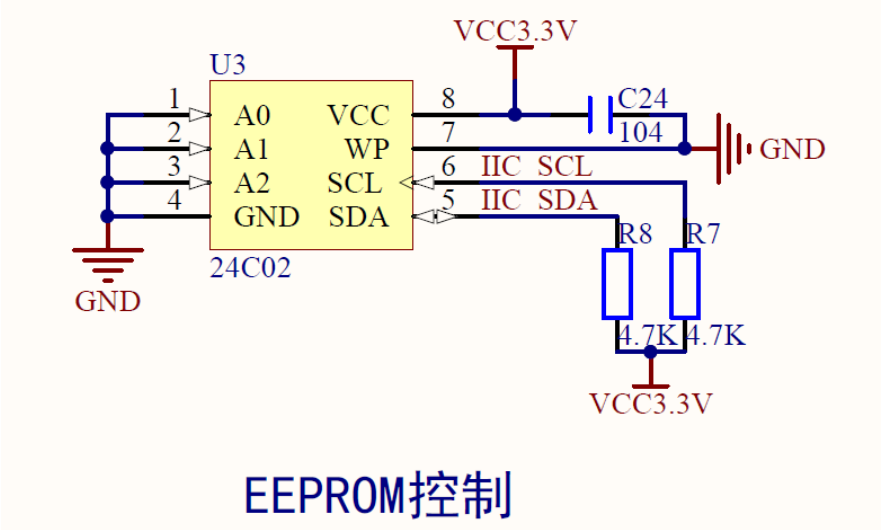



图 2.13 EEPROM 控制原理图

图中 EEPROM 选用 24C02，其容量为 2K bits(256 Bytes)，采用 IIC 总线传输数据。此原理图可以兼容同系列更大容量的 EEPROM。其中：

A0~A2 都接地，也就是说将 24C02 的地址设置为 0 了；

IIC_SCL 为 IIC 总线时钟引脚，连接在主芯片的 PB8 引脚上；

IIC_SDA 为 IIC 总线数据引脚，连接在主芯片的 PB9 引脚上；

R7、R8 为 4.7K 上拉电阻，C24 为电源旁路滤波电容；

由于 IIC_SCL 和 IIC_SDA 引脚接了 4.7K 上拉电阻，所以即使没有使用 24C02，也不建议把他们当做普通引脚使用。

### 2.1.14 SPI FLASH 控制

SPI FLASH 控制原理图如图 2.14 所示：

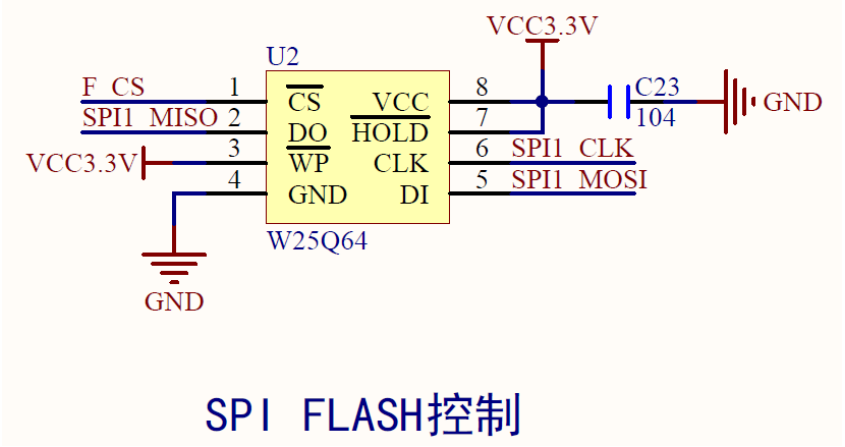



图 2.14 SPI FLASH 控制原理图

图中 SPI FLASH 选用 W25Q64，其容量为 64M bits（8M Bytes），采用 SPI 总线传输数据。此原理图可以兼容同系列更大容量的 SPI FLASH。其中：

F_CS 为片选引脚，连接在主芯片的 PB14 引脚上；

SPI1_MISO 为数据读取引脚，连接在主芯片的 PB4 上；

SPI1_MOSI 为数据写入引脚，连接在主芯片的 PB5 上；

SPI1_CLK 为时钟引脚，连接在主芯片的 PB3 上；

C23 为电源旁路滤波电容。

当 F_CS 引脚拉高（片选禁止），则 PB3/PB4/PB5 可做普通 IO 使用，不建议把 PB14 当做普通 IO 使用。

由于 PB3/PB4 分别复用了 JTAG 的 JTDO 和 JTRST 功能，所以一旦使用 JTAG 功能，SPI 总线就无法使用，SPI FLASH 也使用不了，所以推荐大家尽量使用 SWD 功能。

### 2.2 最小系统板硬件使用注意事项

1） 使用 USB 接口供电时，电源电压保证输入为 5V 左右，过低或过高的电压都会发生异常，推荐使用独立电源供电，比如 5V 的充电头。

2） 板上的保险丝最大保护电流是 750mA，所以如果负载电流超过 750mA，会导致保险丝断开，进入保护状态。所以尽量不要使用大负载的设备。

3） 开发板上提供了 3.3V 或 5V 的电源输入/输出 IO，使用时千万不要将 3.3V 和 GND、5V 和 GND 短接。另外也不要将 3.3V 引脚接入 5V 电压。

4） 使用最小系统板上某个扩展 IO 口时，先要查看原理图或者 IO 资源分配表，弄清楚所用的 IO 口有没有连接外设，如果有连接，则需要确认外设是否需要使用，这样将干扰都排除了，才能使用该 IO 口。

5） 不要用潮湿的手去碰正在运行的最小系统板，以防发生短路。

6） 使用 LCD 模块时，如果无显示（白屏或者黑屏），使用 FPC 软排线连接的模块要检查 FPC 软排线是否插好（包括有没有插反，有没有插到位等等），使用排针直插的模块要检查排针是否插好（包括有没有插反，有没有插到位等等）。如果还是不行，则打开串口查看 LCD ID 是否正确，程序是否支持该款 LCD 等等。

# 3. 软件资源详解

软件资源详解的目的就是让用户熟练掌握新建、编译、下载和仿真调试 MDK5 项目工程，从而学会一个完整的 STM32F407VxT6 最小系统板的开发流程。另外我们也提供了部分示例程序供大家熟悉主芯片的功能模块操作。

### 3.1 新建 MDK5 项目工程

资料包里已经放置一个新建的项目工程模板，路径为“**程序示例\Demo_0_STM32F407VGT6_工程模板**”，大家可以拿来对比参考。

**1） 进行准备工作**

在新建项目工程之前，建议大家新建一个文件夹专门存放当前的项目工程目录和文件，这样方便管理。在这里我们新建一个“**STM32F407VxT6_工程模板**”文件夹，在该文件夹下面再分别新建如下 7 个文件夹：**CORE、HARDWARE、OBJ、PROJECT、STM32F407xx_FWLIB、SYSTEM、USER**。当然这些文件夹的数目和名称都可以任意修改。新建好的文件夹如图 3.1 所示：

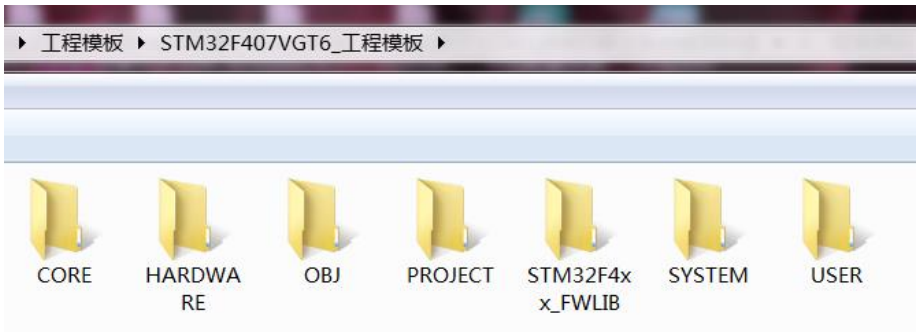



图 3.1 项目工程新建文件夹

**2） 新建 Keli 工程**

打开 keil 软件（电脑上要提前装好），点击 **Project->New uVision Project** 按钮，如图 3.2 所示：

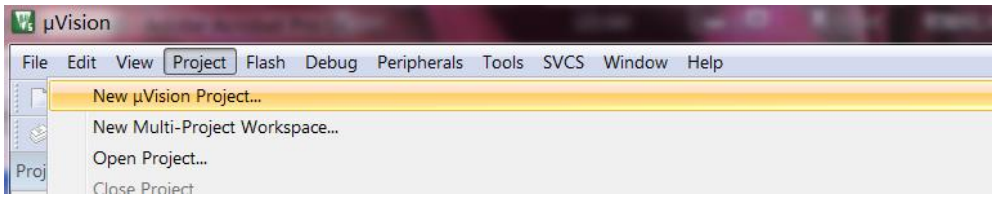



图 3.2 新建 Keil 工程

点击之后，定位到第一步准备工作中所创建的 **PROJECT** 目录下，给项目工程命名为 **TEMPLATE**，然后点击保存。如图 3.3 所示：

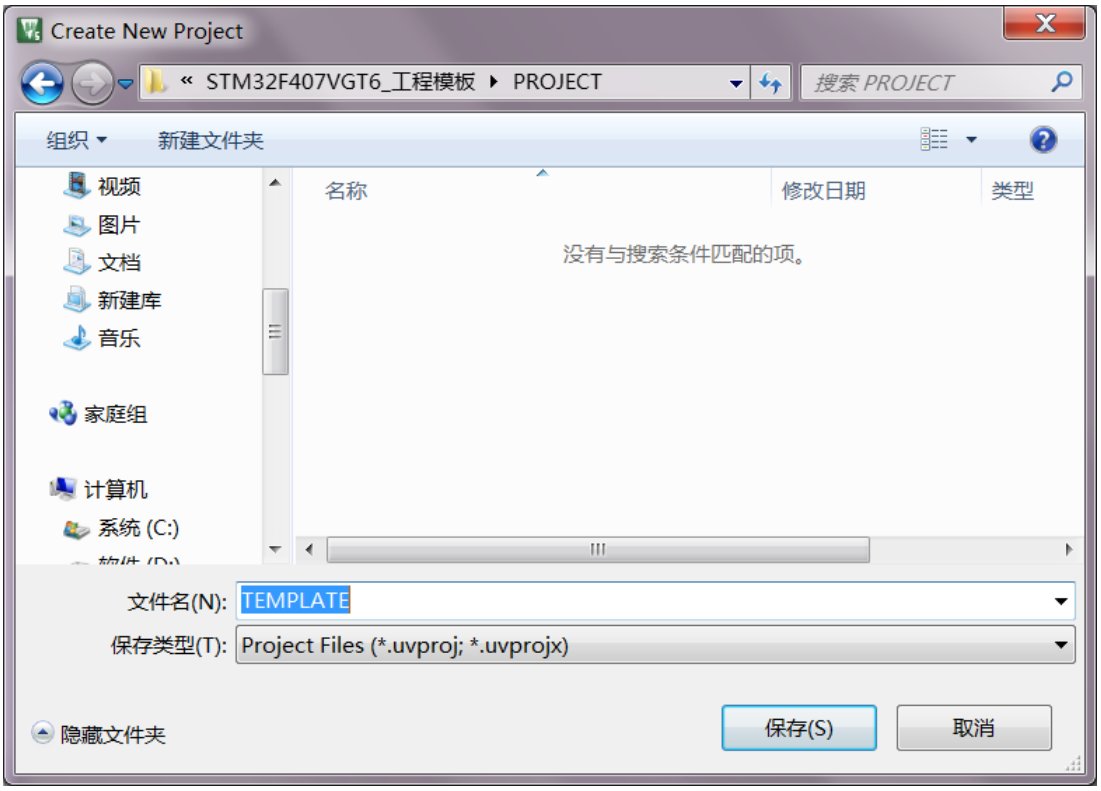



图 3.3 定义工程目录和名称

### 3） 选择芯片型号

接下来会弹出一个选择芯片型号的界面，就是选择新建工程要运行的单片机型号。如果使用 STM32F407VGT6 芯片我们选择 **STMicroelectronics->STM32F4 Series-> STM32F407->STM32F407VG-> STM32F407VGTx**；如果使用 STM32F407VET6 芯片，则选择 **STMicroelectronics->STM32F4 Series->STM32F407->STM32F407VE->STM32F407VETx**。如图 3.4 所示：

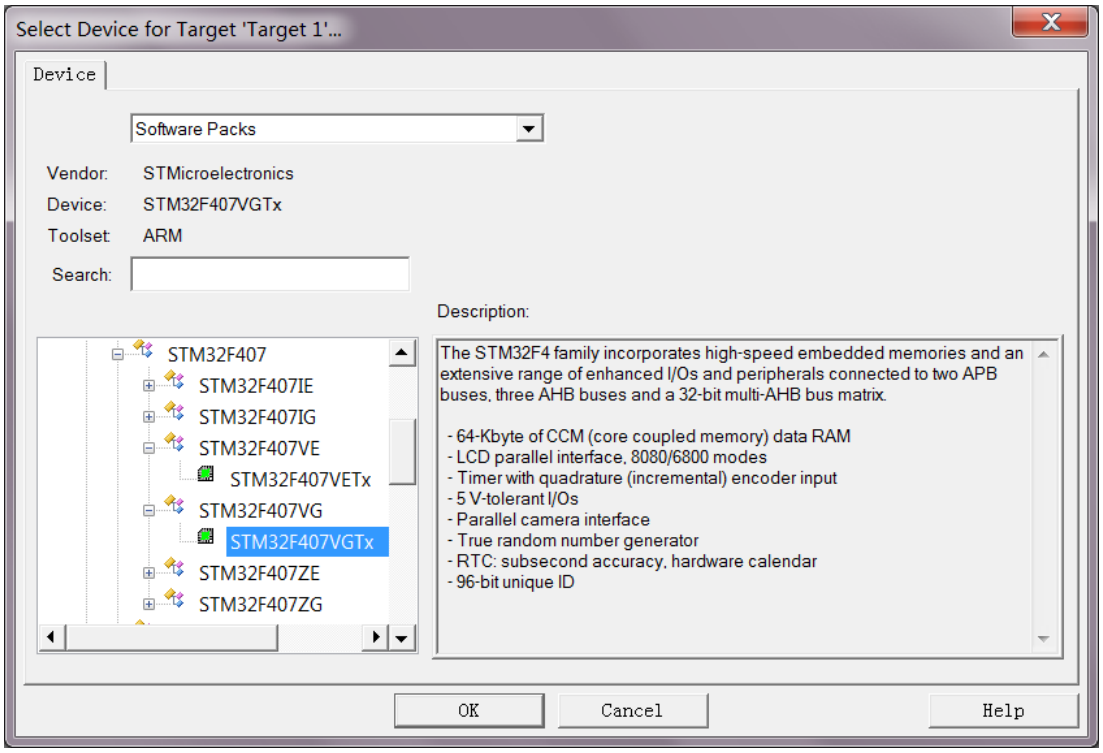



图 3.4 选择芯片型号

点击 OK，选择芯片信号完成后，会弹出一个构建开发环境的选择界面，如图 3.5 所示，这里直接点击 **Cancel**。

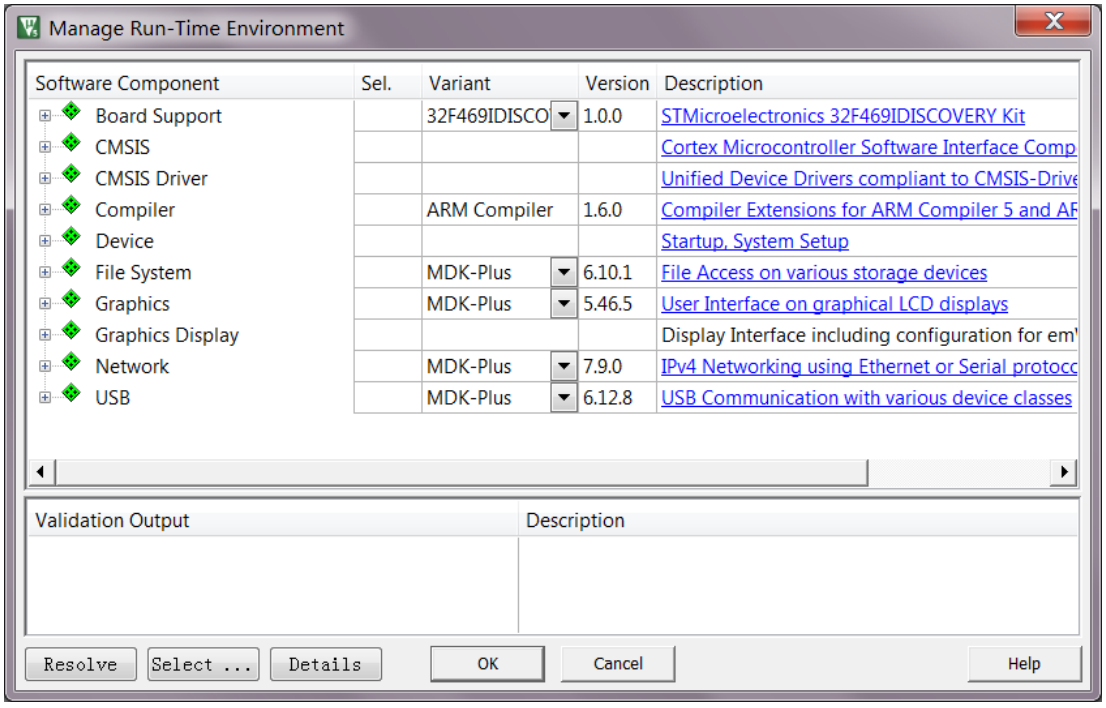



图 3.5 构建开发环境的选择界面

**4） 查看工程文件**

接下来可以看到新建的工程界面如图 3.6 所示：

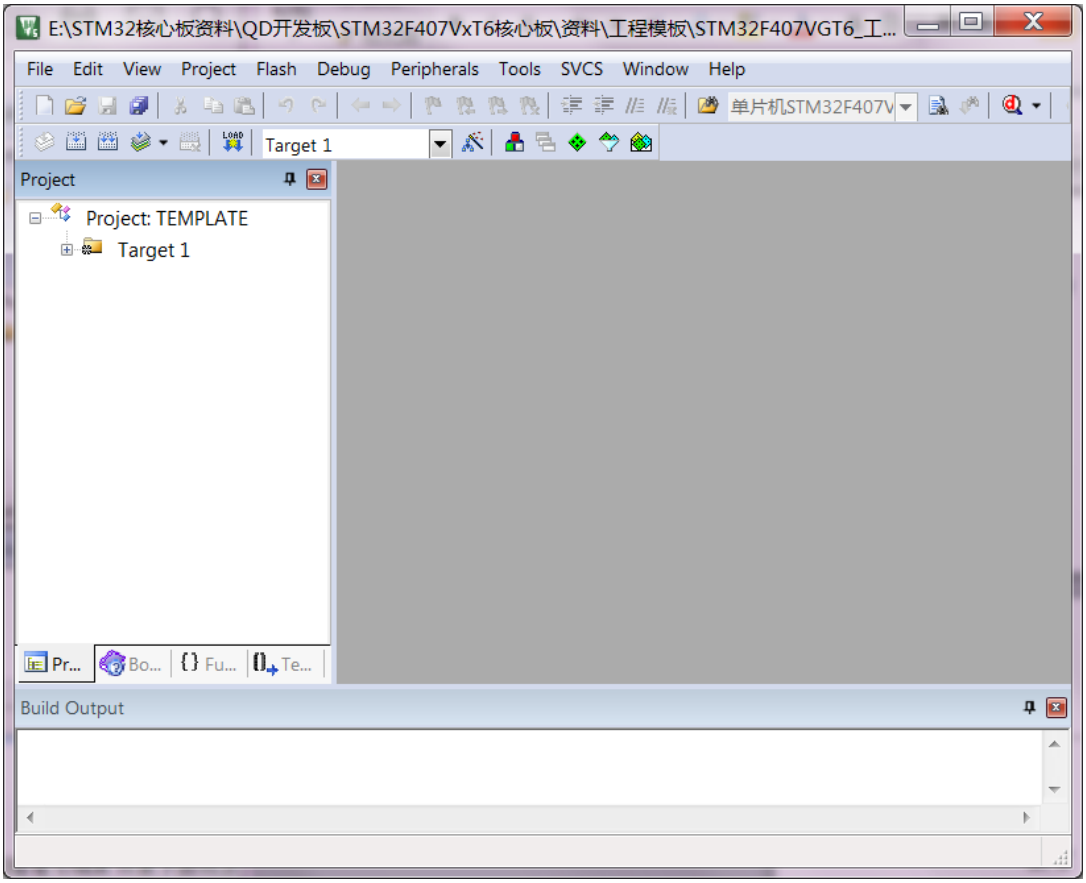



图 3.6 工程初步建立界面

查看 PROJECT 目录下内容，如图 3.7 所示：

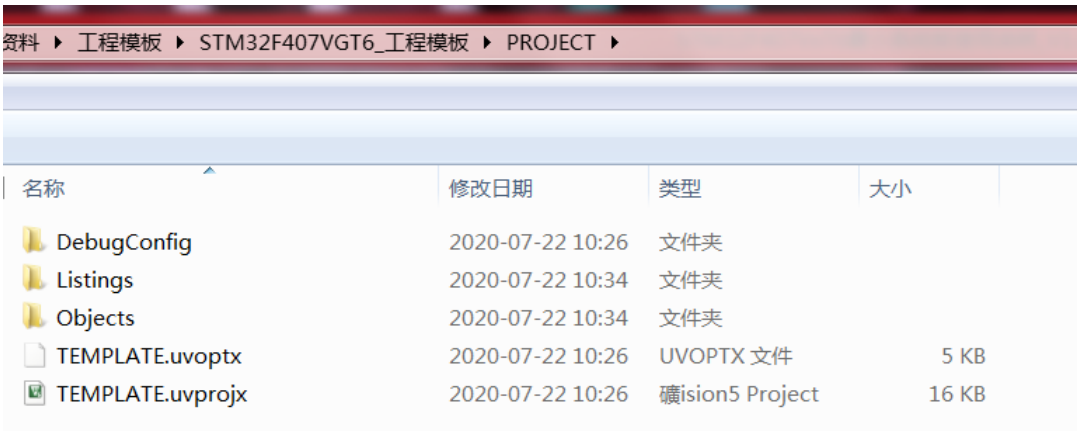



图 3.7 项目工程 PROJECT 目录内容

图中 **TEMPLATE.uvprojx** 为项目工程文件，直接双击就可以打开整个工程，此文件禁止删除。DebugConfig、Listings、Objects 这三个目录为 Keil 软件自动生成的，其中 DebugConfig 目存放仿真器的配置文件，Listings 和 Objects 这两个目录存放编译过程生成的中间文件和编译成功后生成的 HEX 文件。我们已经自己定义目录来存放编译生成的文件，所以这两个目录用不到，可以删除。

**5） 添加官方固件库文件到工程目录**

打开资料包里的**官方固件库\STM32F4xx_官方标准库**目录，解压固件库后找到 **STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\STM32F4xx_StdPeriph_Driver** 目录，将该目录下的 **inc** 和 **src** 文件夹拷贝到第一步准备工作中新建的 **STM32F407xx_FWLIB** 目录下，如图 3.8 所示：


图 3.8 STM32F407xx_FWLIB 目录存放内容

找到 **STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm** 目录，将该目录下 **startup_stm32f40_41xxx.s** 文件拷贝

到第一步准备工作中新建的 CORE 目录下，找到 STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Include 目录，将该目录下的 core_cm4.h、core_cm4_simd.h、core_cmFunc.h、core_cmInstr.h 这四个头文件也拷贝到 CORE 目录下，如图 3.9 所示：


图 3.9 CORE 目录存放内容

找到 STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Libraries\CMSIS\Device\ST\STM32F4xx\Include 目录，将该目录下 stm32f4xx.h 和 system_stm32f4xx.h 头文件拷贝到第一步准备工作中新建的 USER 目录下。找到 STM32F4xx_DSP_StdPeriph_Lib_V1.4.0\Project\STM32F4xx_StdPeriph_Templates 目录，将该目录下的 main.c、stm32f4xx_conf.h、stm32f4xx_it.c、stm32f4xx_it.h、system_stm32f4xx.c 文件也拷贝到 USER 目录下，如图 3.10 所示：

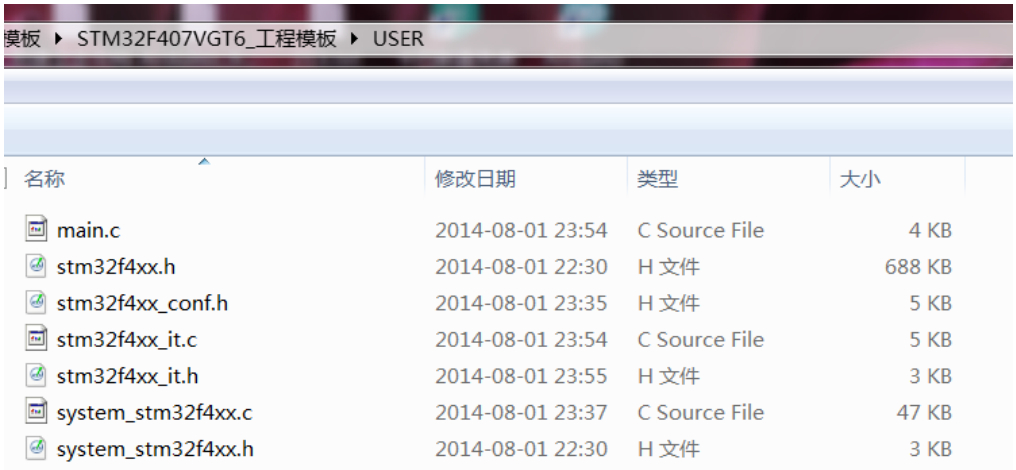



图 3.10 USER 目录存放内容

### 6） 导入官方固件库文件到项目工程

官方库文件添加完毕后，接下里就是把他们导入到工程项目里，打开 Keil 软件，鼠标右键单击 Target1，选择 Manage Project Items，如图 3.11 所示。

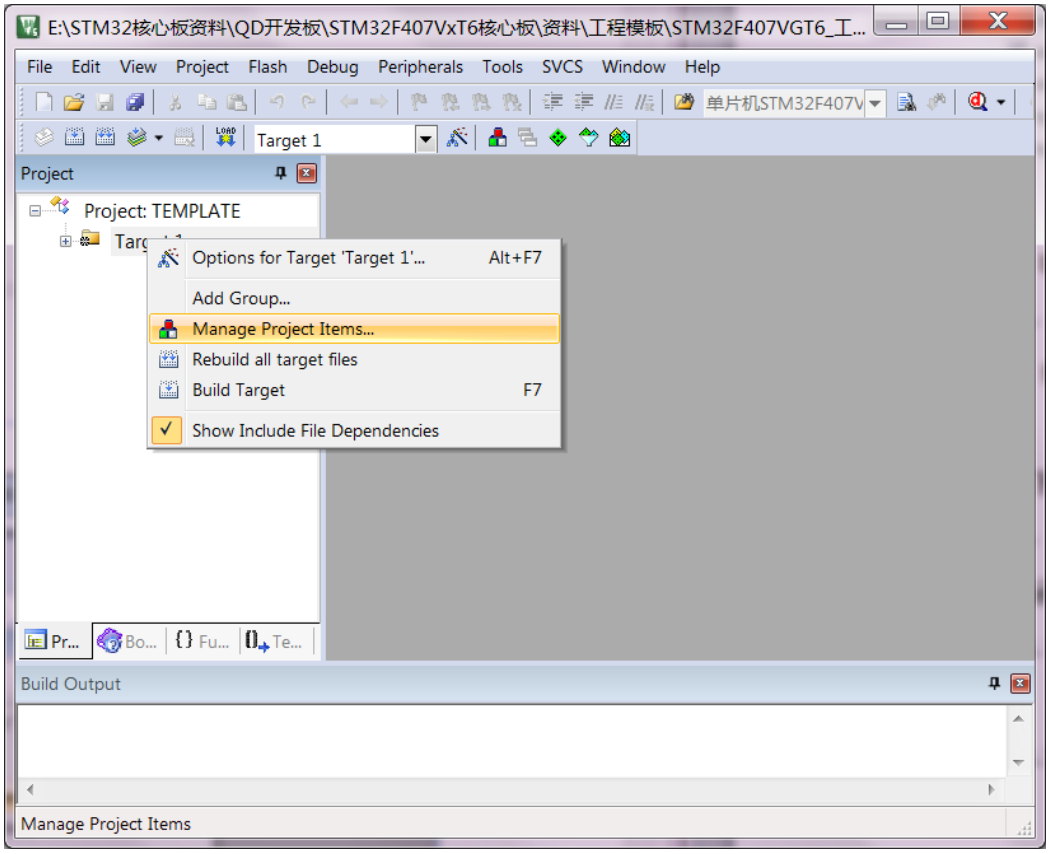



图 3.11 选择 Manage Project Items

在 **Project Targets** 下将 Target1 改为 **TEMPLATE**，在 **Groups** 下将 Source Group1 删掉，新建 **CORE**、**FWLIB**、**USER** 三个 Groups，如图 3.12 所指示。点击 OK 保存，界面如图 3.13 所示。

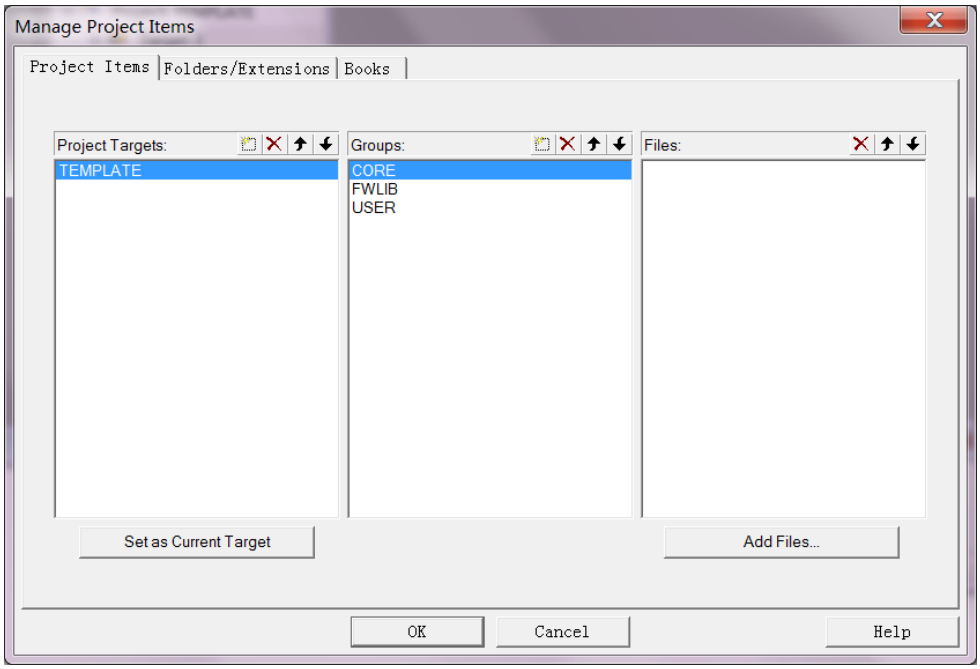



图 3.12 新建 Groups

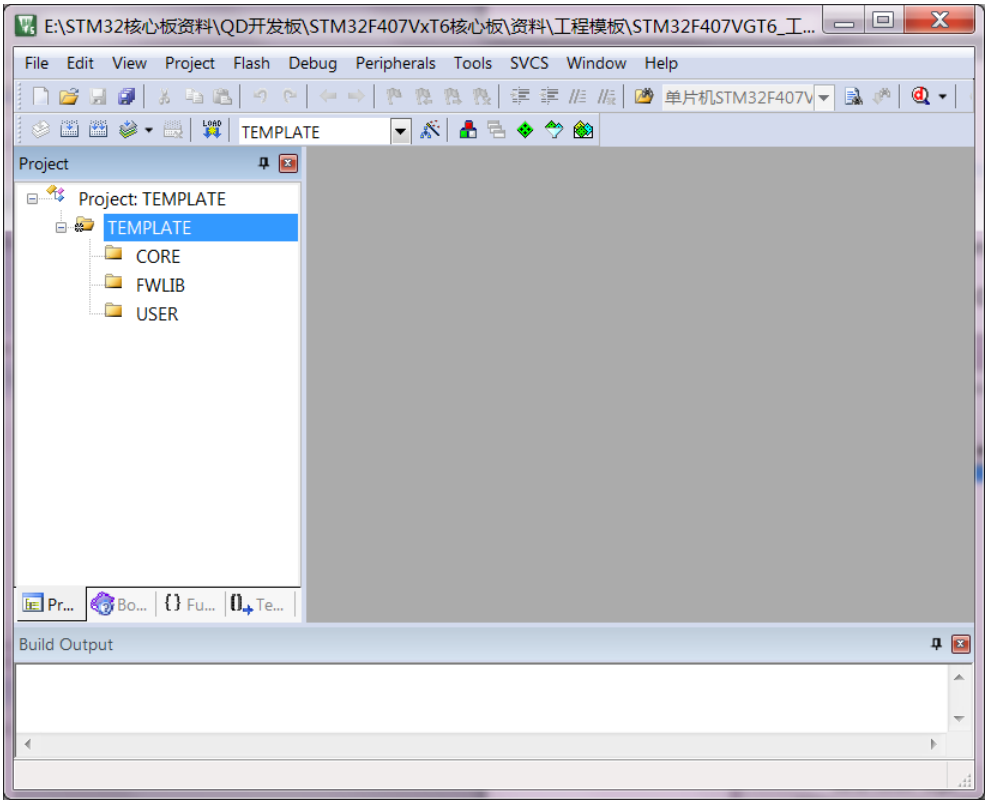



图 3.13 工程 Groups 界面

再按照上述方法，鼠标右键单击 TEMPLATE,选择 Manage Project Items 进入 Manage Project Items 界面。先给 CORE 添加文件，选中 CORE，鼠标点击右下角 Add Files 按钮，找到 CORE 目录，此时需要将文件类型设为 All files (*.*)，目录下的文件才会显示出来，如图 3.14 所示，待文件显示后，选中 startup_stm32f40_41xxx.s 文件，再点击 add 按钮，最后点击 Close 按钮。可看到文件已添加，如图 3.15 所示：

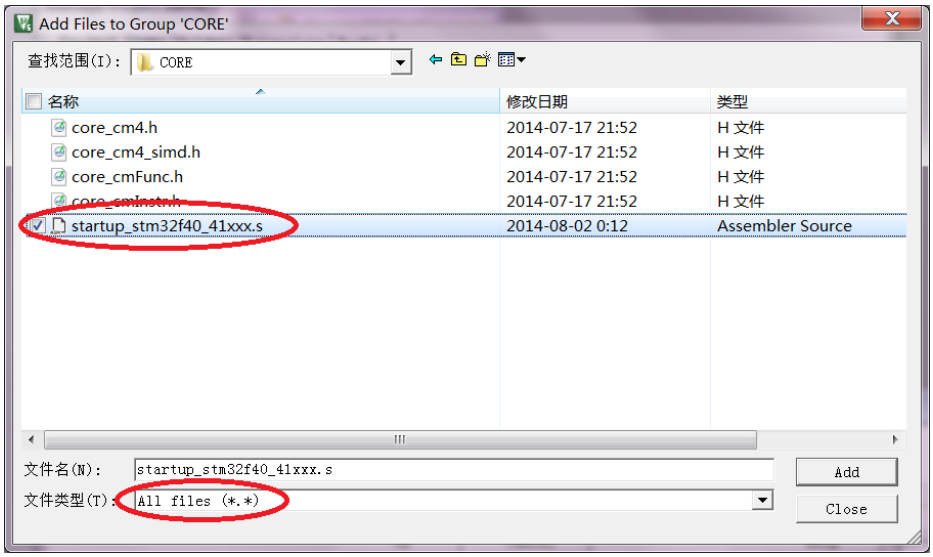



图 3.14 添加 startup_stm32f40_41xxx.s

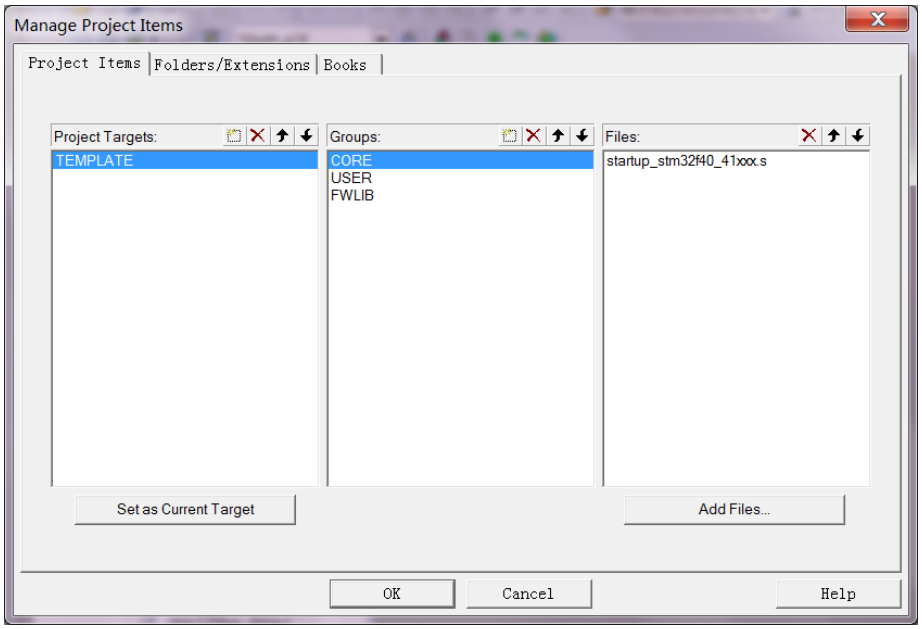



图 3.15 添加文件到 CORE Group

FWLIB 和 USER 添加文件的方法和上述基本一样，只是这两个 Groups 只需要添加.c 文件，所以文件类型不需要设为 All files (*.*)。值的注意的是：

FWLIB 添加文件时需要找到 **STM32F4xx_FWLIB\src** 目录，除了 **stm32f4xx_fmc.c** 文件外，其他外设文件都要添加，如图 3.16 所示。需要说明的是，我们做模板时为例展示，添加了所有的外设文件，在实际项目中，并不需要添加所有的外设文件，只需添加所用的外设文件。

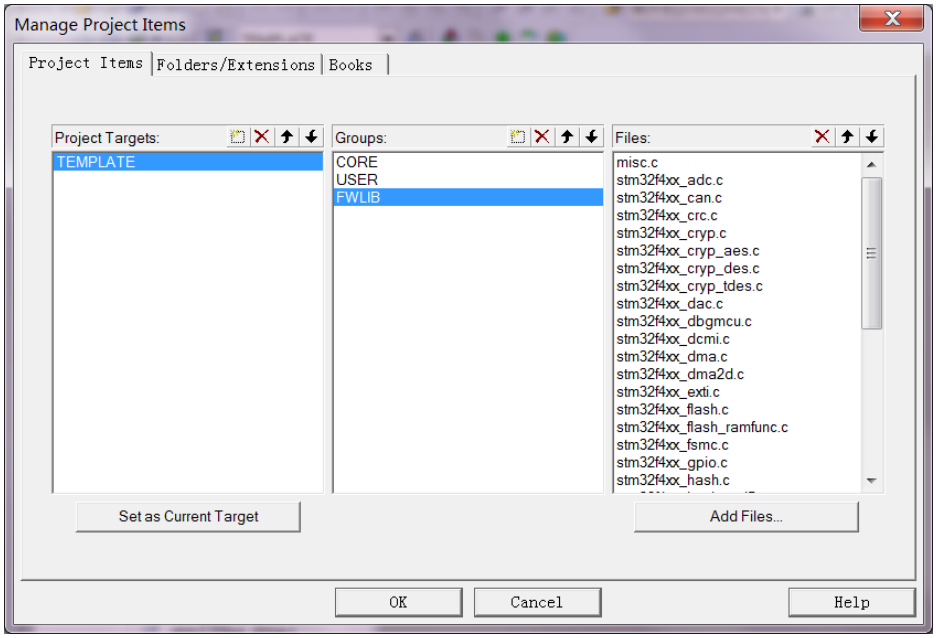



图 3.16 添加文件到 FWLIB Group

USER 添加文件时需要找到 **USER** 目录，添加 **main.c**、**stm32f4xx_it.c**、**system_stm32f4xx.c** 文件。如图 3.17 所示。

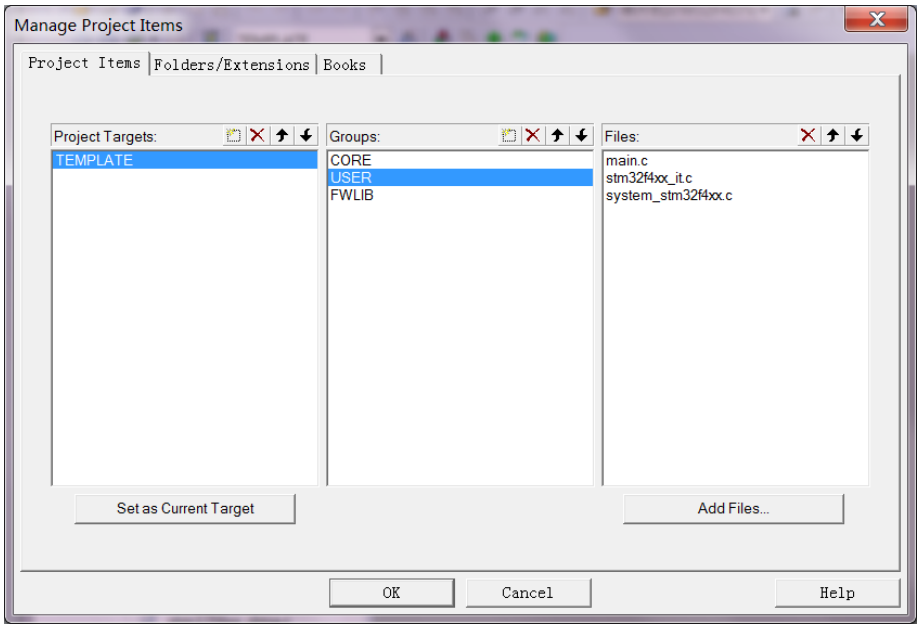



图 3.17 添加文件到 USER Group

文件添加成功后，点击 OK 保存，回到工程界面，可以看到添加的文件已显示出来，如图 3.18 所示：

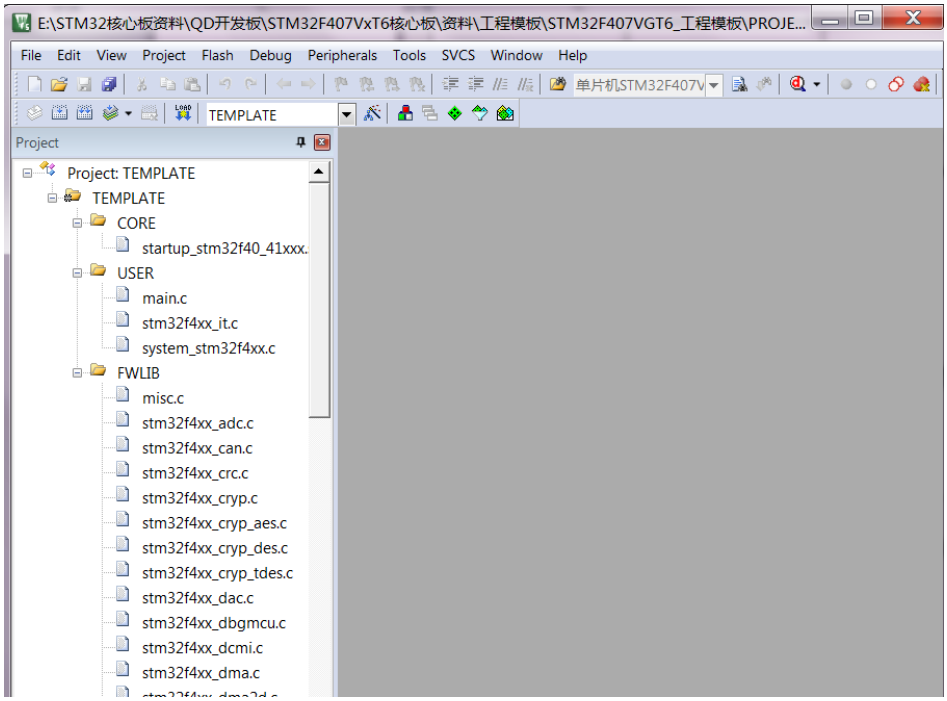



图 3.18 工程文件显示

### 7） 设置工程头文件路径

接下来需要设置头文件路径，不然编译会出错。打开 keil 软件，按照图 3.19、图 3.20 和图 3.21 所示步骤进行设置。

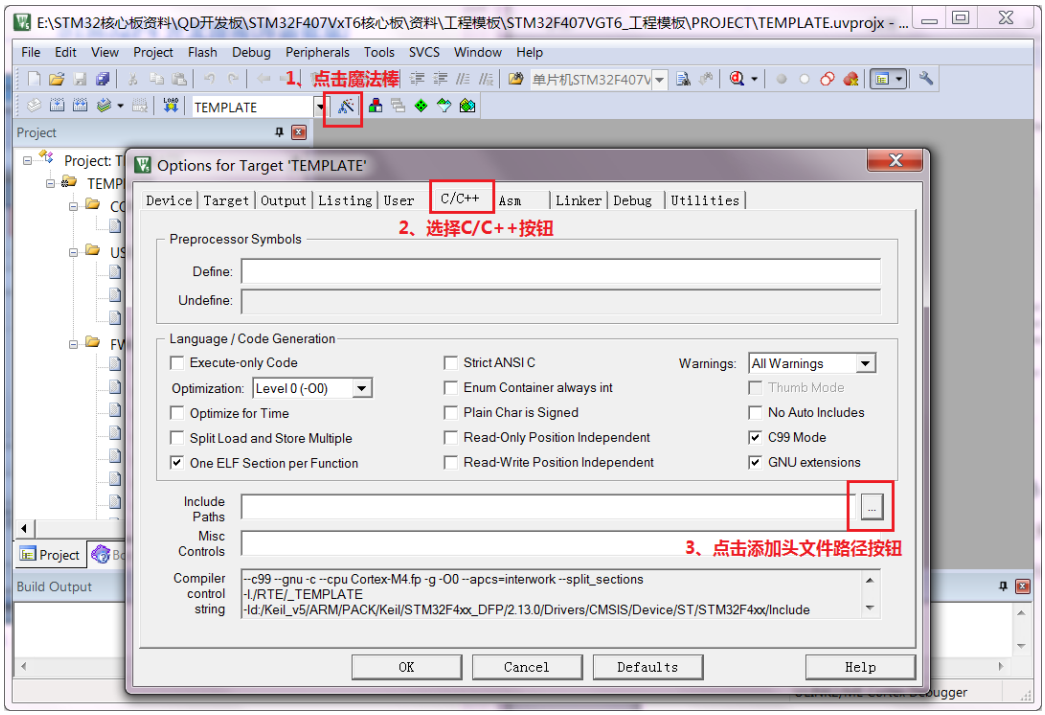



图 3.19 打开头文件路径添加界面

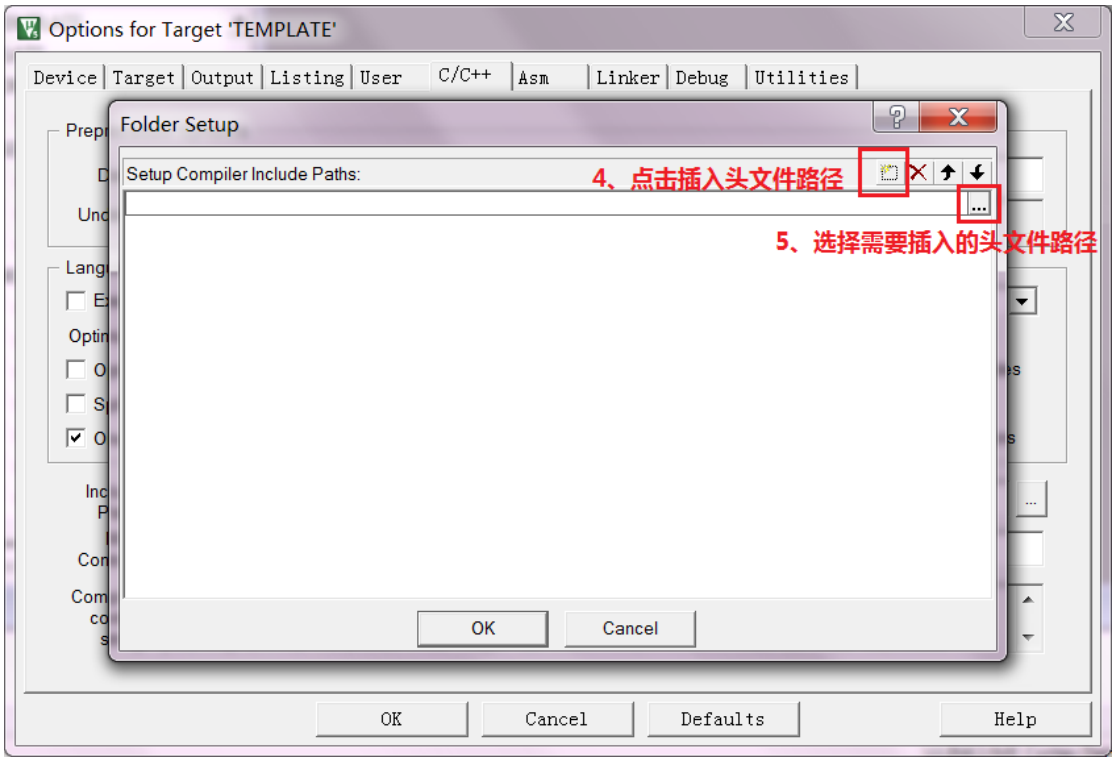



图 3.20 添加头文件路径

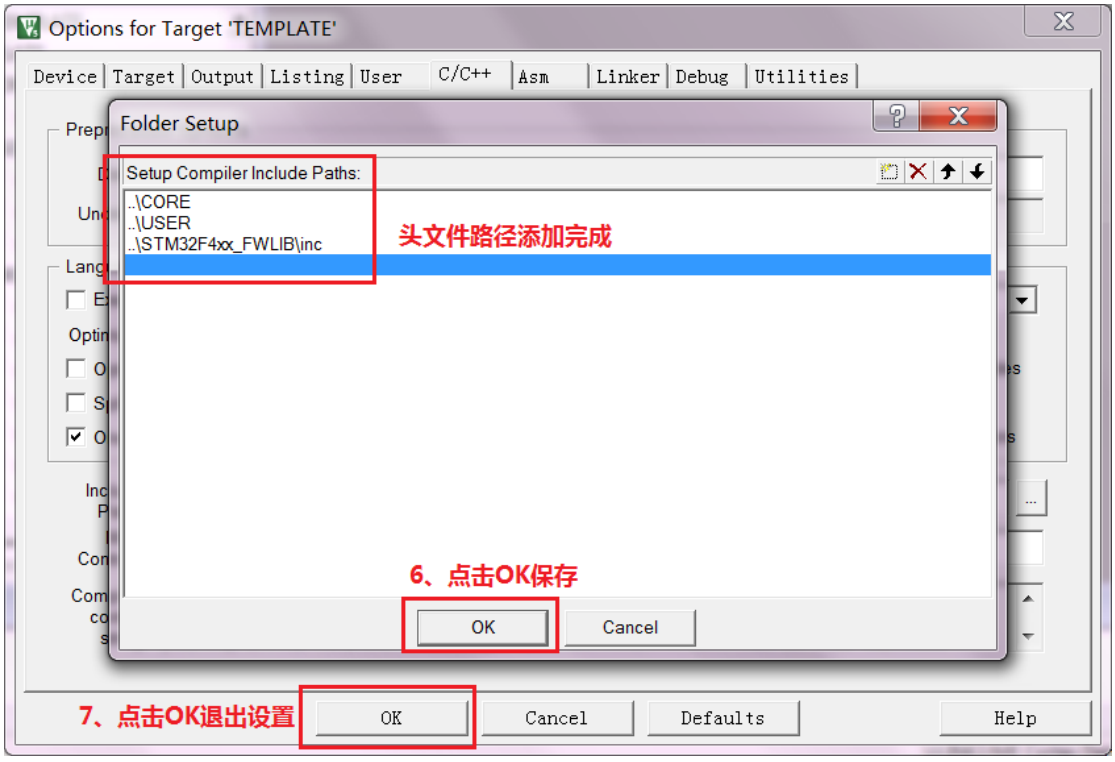



图 3.21 头文件设置完成

需要注意的是头文件路径一定要设置到头文件所在目录，不能设置到上级目录。

**8） 添加全局宏定义标识符**

头文件路径添加完毕后，还需要添加全局宏定义标识符，不然编译会报错。步骤如下：

点击魔法棒 -> 选择 C/C++ 按钮 -> 在 Define 栏输入 STM32F40_41xxx,USE_STDPERIPH_DRIVER->点击 OK 保存。如图 3.22 所示步骤。

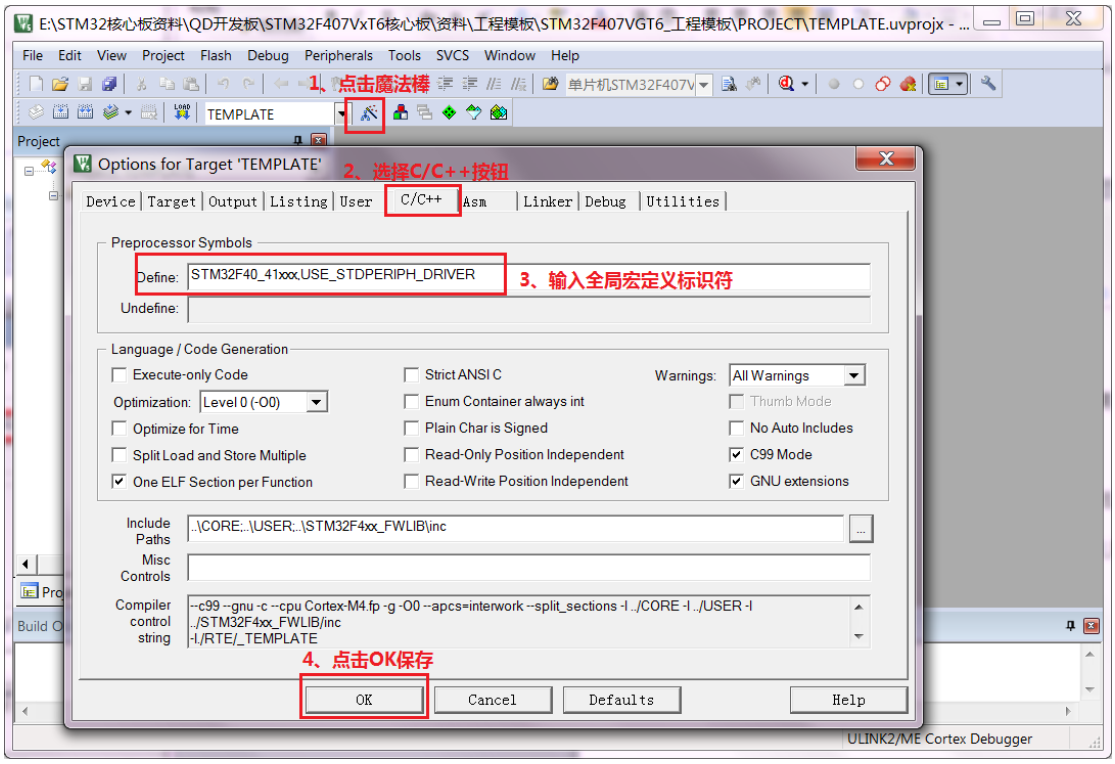



图 3.22 添加全局宏定义标识符

需要注意的是上述添加了两个全局宏定义标识符，他们之间用逗号隔开的。

**9） 设置编译文件存放目录**

编译文件包含编译中间文件和编译后生成的文件。设置步骤如下：

点击魔法棒->选择 Output->点击 Select Folder Objects 按钮选择 OBJ 目录->将 Debug Information、Create HEX File、Browse Information 这三个选项都勾上->点击 OK 保存退出。如图 3.23 所示：

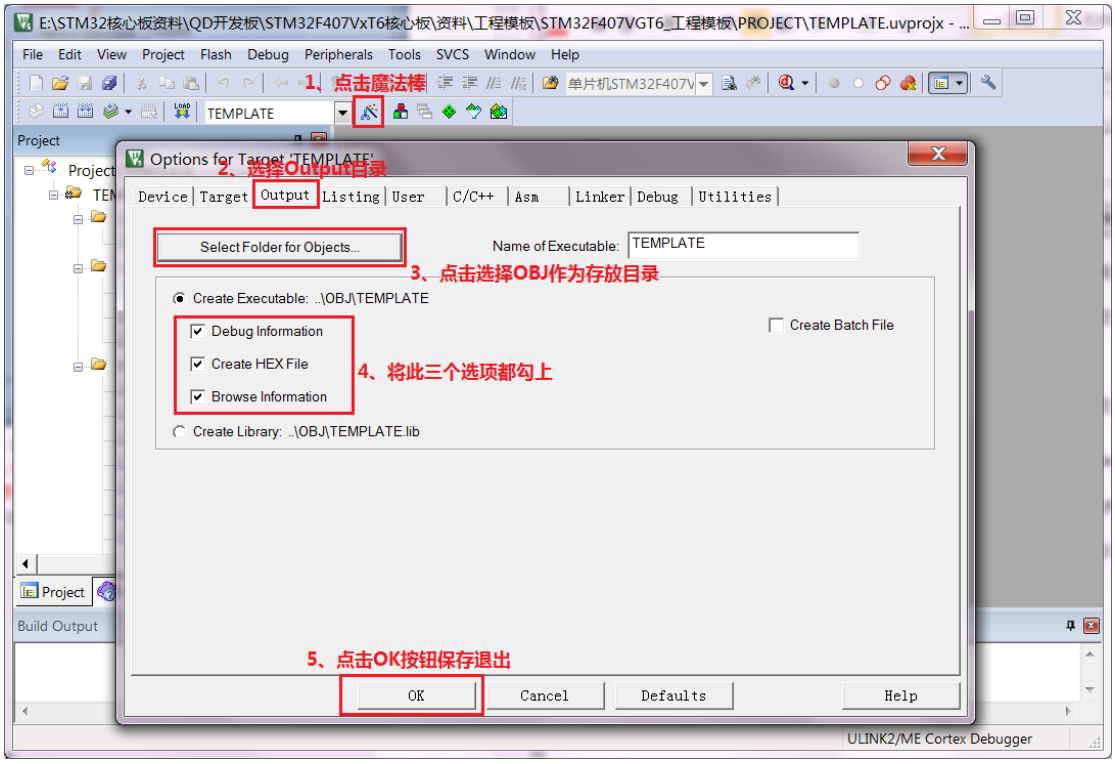



图 3.23 设置编译文件存放目录

10）修改文件

由于官方库文件的一些配置和我们的最小系统板不匹配，所以需要修改。这里需要修改如下 4 个文件：**main.c、stm32f4xx_it.c、System_stm32f4xx.c、stm32f4xx.h。**

替换 main.c 文件内容是因为我们想更新自己编写的测试内容，大家可以到“**程序示例\Demo_0_STM32F407VGT6_工程模板**”里去复制 main.c 内容，替换内容如下：

```
#include "delay.h"
#include "sys.h"

int main(void)
{
    delay_init(168);        //初始化延时函数
    //初始化 PA1 引脚
    GPIO_InitTypeDef    GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);//使能
GPIOF 时钟
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;//LED0 对应 IO 口
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//普通输出模式
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//100MHz
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
  GPIO_Init(GPIOA, &GPIO_InitStructure);//初始化 GPIO
```

```
GPIO_SetBits(GPIOA,GPIO_Pin_1);//PA1 设置高，灯灭
  //循环测试
    while(1)
    {
        GPIO_SetBits(GPIOA,GPIO_Pin_1);//PA1 设置高，灯灭
        delay_ms(500); //间隔 500ms
        GPIO_ResetBits(GPIOA,GPIO_Pin_1);//PA1 设置低，灯亮
        delay_ms(500); //间隔 500ms
    }
}
```

修改 stm32f4xx_it.c 文件是因为 main.h 头文件不需要使用，系统时钟相关的中断处理函数我们也不需要使用，修改如下：

打开 stm32f4xx_it.c 文件，将 32 行对 main.h 头文件引入的内容删除，如图 3.24 所示，将 144 行 SysTick_Handler 函数内容删除，如图 3.25 所示：

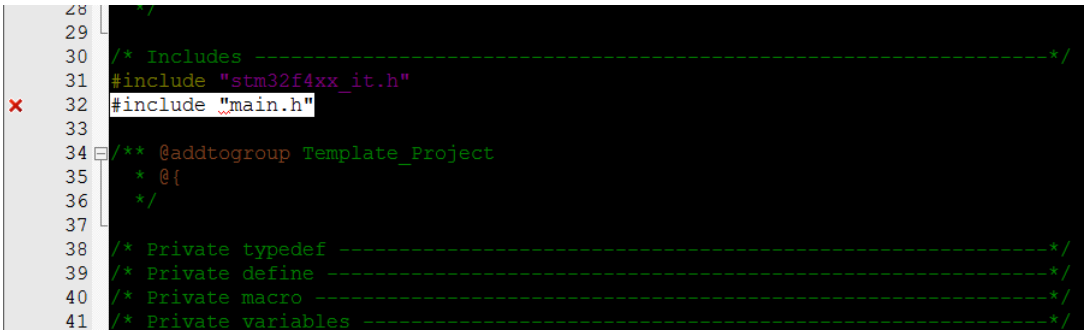



图 3.24 删除 stm32f4xx_it.c 文件里 main.h 文件引入

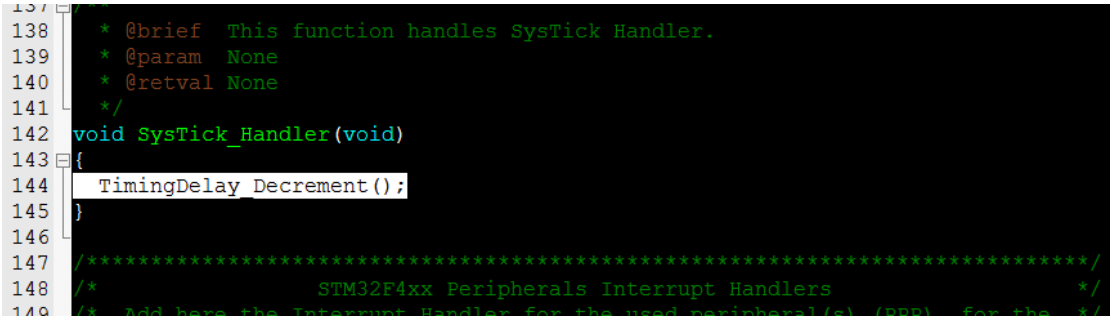



图 3.25 删除 stm32f4xx_it.c 文件里 SysTick_Handler 函数内容

修改 System_stm32f4xx.c 和 stm32f4xx.h 文件是因为官方固件库的 PLL 第一级分频系数为 25，而我们的最小系统板只需配置为 8，这样主频才能达到 168MHz。

打开 System_stm32f4xx.c 文件，将 **PLL_M** 值修改为 **8**，如图 3.26 所示。

打开 stm32f4xx.h 文件，将 **HSE_VALUE** 值修改为 **8000000**，如图 3.27 所示。

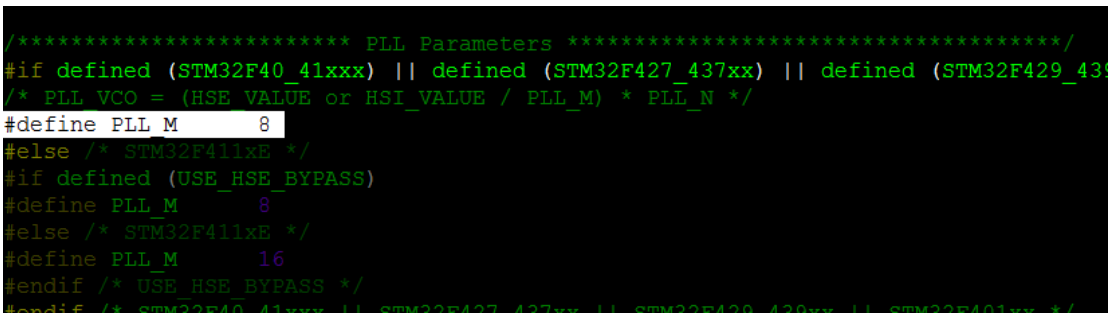



图 3.26 修改 System_stm32f4xx.c 文件里 PLL_M 值

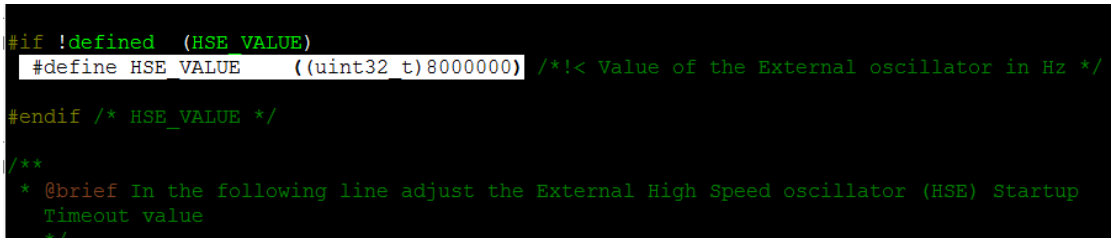



图 3.27 修改 stm32f4xx.h 文件 HSE_VALUE 值

11） **添加外部非官方文件**

经过上述步骤，一个标准的官方库项目工程已经成功建立，但是在实际项目如果只有官方库是不够的，我们还需要添加自己编写的代码文件来完善项目工程。这里我们只添加 SYSTEM 目录文件，大家可以到“**程序示例\Demo_0_STM32F407VGT6_工程模板**”里去复制 SYSTEM 目录内容放在新建项目工程的 SYSTEM 目录下。添加工程文件方法和设置头文件路径方法和上述方法一致，这里就不再说明了。另外在准备工作中还新建了一个 HARDWARE 目录，此目录专门用于放置外设文件，比如 LCD、SPI FLASH 等等。

## 3.2 编译项目工程

项目工程新建完毕，接下里就是编译项目工程，点击编译按钮就可以进行编译了，编译结果出现 0 Error(s),0 Warning(s),就说明编译成功。如图 3.28 所示：

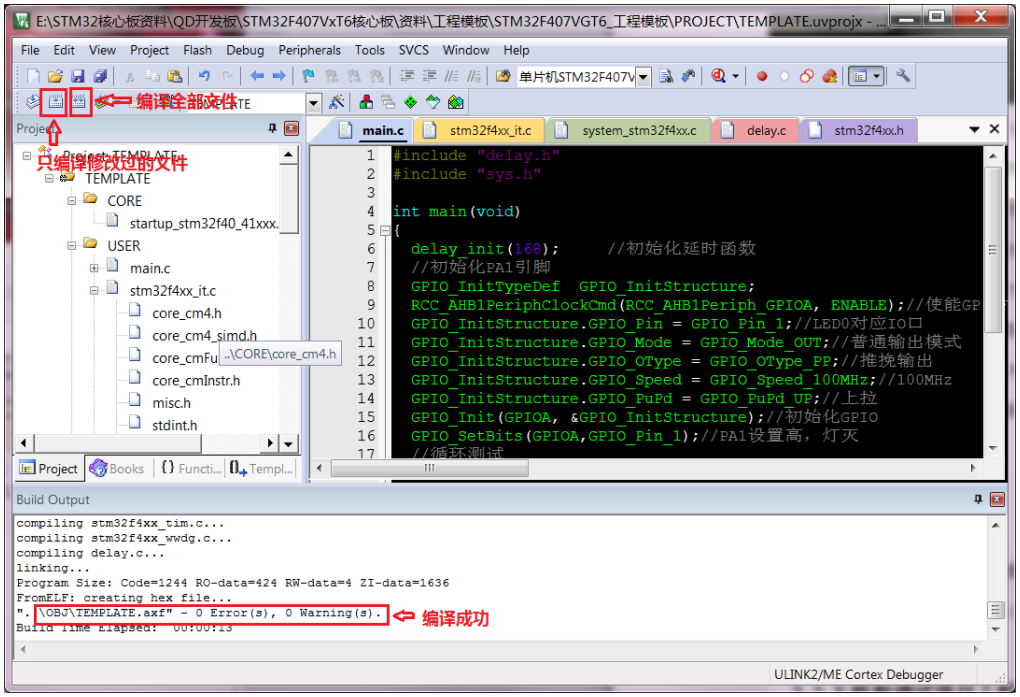



图 3.28 程序编译界面

图中编译全部文件是指编译所有的项目工程文件，当工程第一次编译或者改动文件较多时采用这种编译方式，但是编译过程比较久。

只编译修改过的文件是指是指修改过的文件会被重新编译，没有修改过的文件不会被编译，当工程文件改动较少时，可以采用这种编译方式，其优势是编译过程较短，在调试程序时建议采用这样编译方式。

## 3.3 下载和调试项目工程

上述我们介绍了怎么新建项目工程、怎么编译项目工程，接下来我们将介绍怎么下载和调试项目工程。

### 3.3.1 串口下载

串口下载是最简单的一种下载方式。最小系统板使用串口 1 下载，但是板上没有 USB 转串口模块（USB TO TTL），所以需要用户自行配备。串口下载具体说明，请查看资料包中**下载与调试方法**目录下的“**串口下载说明.pdf**”文档，如图 3.29 所示：

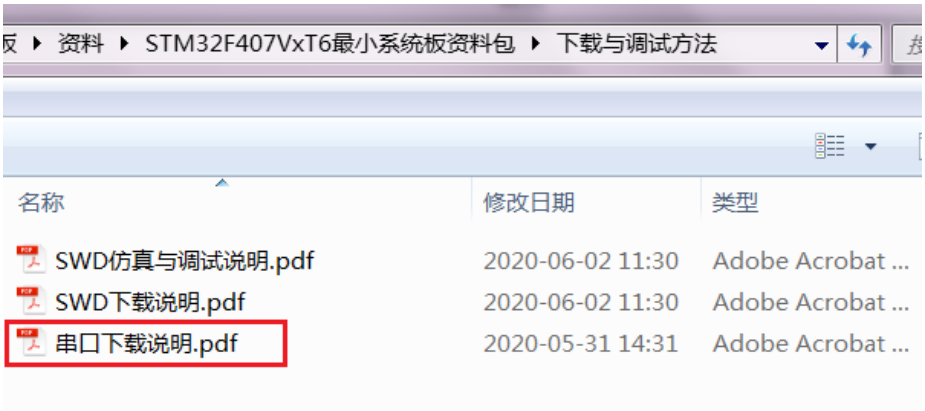



图 3.29 串口下载说明文档

### 3.3.2 SWD 下载

仿真器下载也是一种程序下载方式。仿真器有很多种，例如 JTAG 和 ST-LINK 等等，它们下载的端口有 JTAG 和 SW 两种模式，因为 JTAG 模式需要占用许多 IO 口，接线比较麻烦，而 SW 模式占用 IO 口较少，且最小系统板专门留有 SWD 下载接口，所以建议大家使用 SW 模式。这里也只介绍 SWD 下载方式，具体说明请查看资料包中**下载与调试方法**目录下的“**SWD 下载说明.pdf**”文档，如图 3.30 所示：

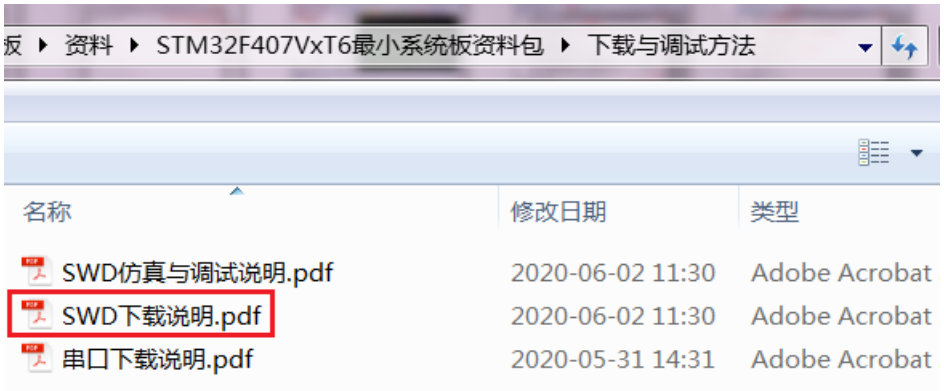



图 3.30 SWD 下载说明文档

### 3.3.3 SWD 仿真与调试

串口只能下载代码，不能调试代码。仿真器可以调试代码，它可以做到实时跟踪代码运行状态，快速找到 bug，这对于大工程代码来说，非常实用，可以节省很多开发时间。这里只介绍 SWD 仿真与调试，具体说明见资料包中**下载与调试方法**目录下的“**SWD 仿真与调试说明.pdf**”文档，如图 3.31 所示：

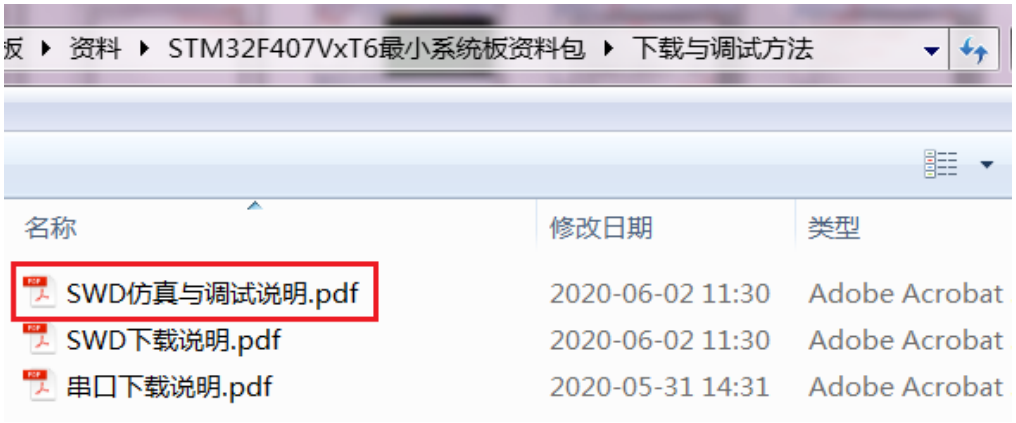



图 3.31 SWD 仿真与调试说明文档

## 3.4 示例程序详解

通过以上介绍，大家对 STM32F407VxT6 最小系统板的硬件开发平台和软件开发平台都有了一定的了解。接下来将通过对 20 个示例讲解，让大家更加熟悉最小系统板的开发。

### 3.4.1 LED 灯示例

**1） 示例目的**

让大家掌握 STM32F407 的 IO 口输出操作。

**2） 硬件说明**

本示例用到 LED0 和 LED1 两颗 LED 灯，其中 LED0 接在 PA1 上，LED1 接在 PC5 上，最小系统板上已经连接好，不需要再接线，硬件连接如图 3.32 所示：

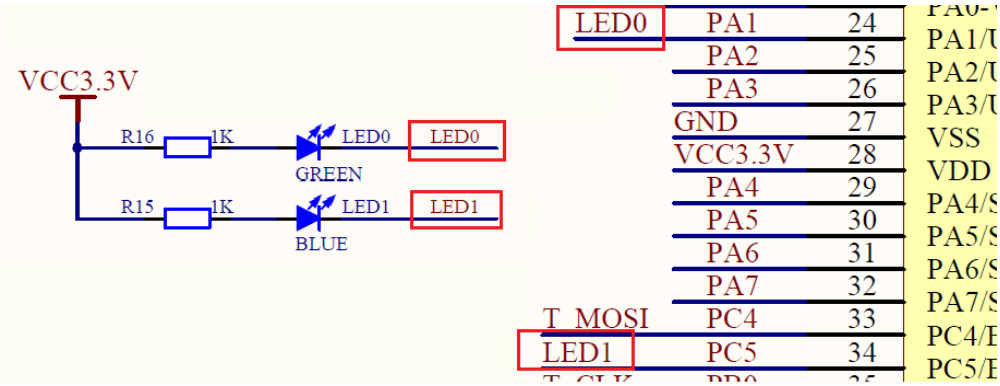



图 3.32 LED 灯与 STM32F407 连接原理图

**3） 软件设计**

从硬件连接原理图上看只要软件控制 PA1 引脚输出低电平，LED0 发光二极管导通，就会亮，反之输出高电平，LED0 发光二极管截止，就会灭。LED1 控制也

是一样的原理。基于此原理，我们开始软件设计，步骤如下：

A、按照上述新建 MDK5 项目工程的方法新建一个名称为 LED 的项目工程（如果是直接从模板拷贝过来的，需要将项目工程文件名称由 Template.uvproj 改为 LED.uvproj）。

B、在项目工程的 HARDWARE 目录下新建 LED 目录，在 LED 目录下新建 led.c 和 led.h 文件（可以在电脑上直接新建，也可以通过 keil 软件新建后保存到 LED 目录下，如图 3.33 步骤所示，新建 led.h 方法一样）。

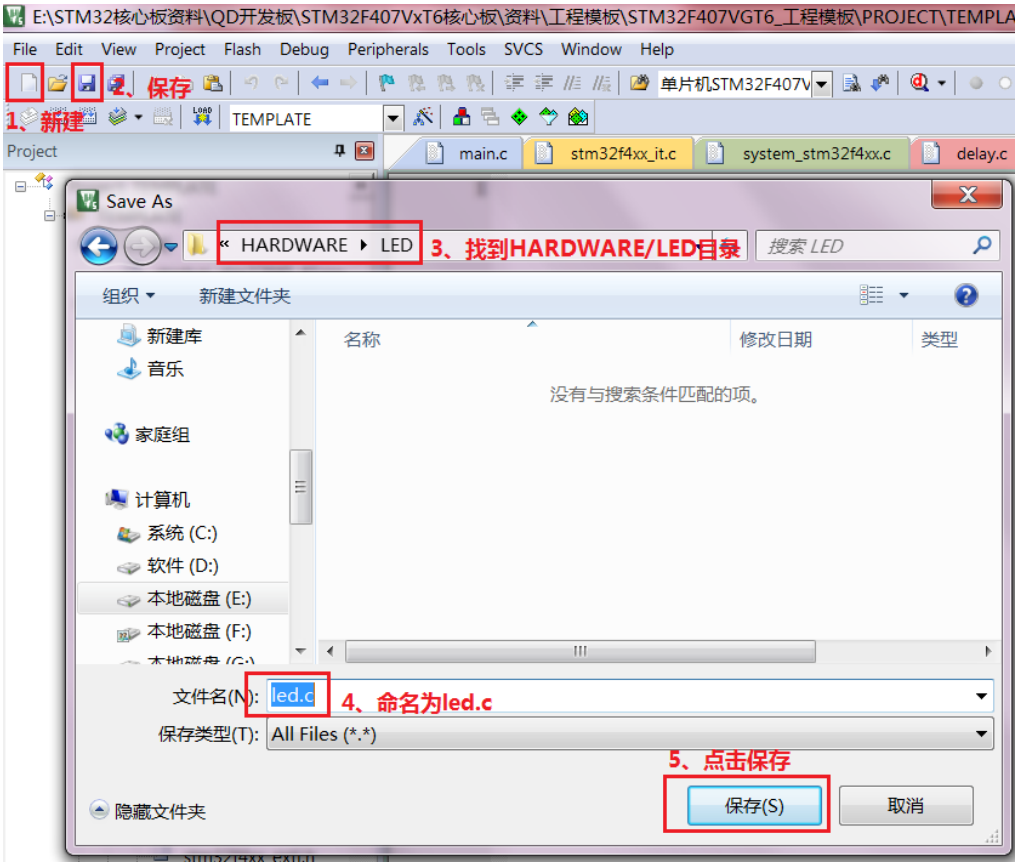



图 3.33 keil 软件新建并保存文件步骤

C、 在 led.c 文件里添加如下内容并保存（大家可以直接去示例程序里拷贝）

```
#include "led.h"
/**************************************************************************
************
* @name          :void LED_Init(void)
* @date          :2020-05-08
* @function      :Initialize LED GPIO
* @parameters :None
* @retvalue      :None
**************************************************************************/
```

```
void LED_Init(void)
{
GPIO_InitTypeDef GPIO_InitStructure;
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPIOC,
ENABLE);//使能 GPIOA、GPIOC 时钟
//PA1,PC5 初始化设置
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;                  //LED0 对应 IO 口
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;          //普通输出模式
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;         //推挽输出
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100MHz
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;            //上拉
 GPIO_Init(GPIOA, &GPIO_InitStructure);                    //初始化 GPIOA
   GPIO_SetBits(GPIOA,GPIO_Pin_1);                              //PA1 设置高，
LED0
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;                 //LED1 对应 IO 口
GPIO_Init(GPIOC, &GPIO_InitStructure);                    //初始化 GPIOC
GPIO_SetBits(GPIOC,GPIO_Pin_5);                              //PC5 设置高，LED1
灯灭
}
```

这里面只有一个 LED_Init(void)函数，此函数就是对 LED 灯所连接的 GPIO 口进行初始化。首先使能 GPIO 时钟，接下来设置对应的 IO 口、输入输出模式、IO 电平状态、IO 速度、上下拉状态，最后调用 GPIO_Init 函数将前面设置的参数都设置到底层寄存器里去。接下里调用 GPIO_SetBits 函数将 IO 上拉，让 LED 灯默认为熄灭状态。

在 led.h 文件里添加如下内容并保存（大家可以直接去示例程序里拷贝）

```
#ifndef __LED_H
#define __LED_H
#include "sys.h"
//LED 端口操作方式切换
#define LED_IO_DEF 0     //0-位操作，1-库函数操作，2-寄存器操作

//LED 端口定义
#if LED_IO_DEF==0   //使用位操作控制 LED 灯端口
#define LED0 PAout(1)   // LED0
#define LED1 PCout(5)   // LED1
#define LED0_SET LED0=1   //拉高 PA1
#define LED0_CLR LED0=0   //拉低 PA1
#define LED1_SET LED1=1   //拉高 PC5
#define LED1_CLR LED1=0   //拉低 PC5
#elif LED_IO_DEF==1     //使用库函数控制 LED 灯端口
```

```
#define LED0 GPIO_Pin_1       //PA1
#define LED1 GPIO_Pin_5       //PC5
#define LED0_SET GPIO_SetBits(GPIOA,LED0)     //拉高 PA1
#define LED0_CLR GPIO_ResetBits(GPIOA,LED0) //拉低 PA1
#define LED1_SET GPIO_SetBits(GPIOC,LED1)     //拉高 PC5
#define LED1_CLR GPIO_ResetBits(GPIOC,LED1) //拉低 PC5
#else                          //使用寄存器控制 LED 灯端口
#define LED0 GPIO_Pin_1       //PA1
#define LED1 GPIO_Pin_5       //PC5
#define LED0_SET GPIOA->BSRRL=LED0    //拉高 PA1
#define LED0_CLR GPIOA->BSRRH=LED0    //拉低 PA1
#define LED1_SET GPIOC->BSRRL=LED1    //拉高 PC5
#define LED1_CLR GPIOC->BSRRH=LED1    //拉低 PC5
#endif
void LED_Init(void);   //初始化 GPIO
#endif
```

里面定义了三种 GPIO 操作方式：位操作、库函数操作和寄存器操作。使用宏定义 LED_IO_DEF 来进行区分。其中位操作使用起来最简便，它的定义放在 SYSTEM 目录下的 sys.h 文件里面。库函数操作使用起来比较直观易懂。寄存器操作使用起来有点复杂，需要查看寄存器名称。大家可以根据需求自由选择。

D、 文件内容写入完成后，在项目工程里新建 HARDWARE group，将 led.c 导入进去，再添加头文件路径，最后将 FWLIB Group 里不需要的外设库文件删除。以上操作方法和新建项目工程里的一致，这里不再阐述。

E、 在 main.c 文件里添加如下内容：

```
#include "delay.h"
#include "sys.h"
#include "led.h"

int main(void)
{
    delay_init(168);        //初始化延时函数
    LED_Init();             //初始化 LED 灯 GPIO
    while(1)
    {
        LED0_CLR;           //LED0 亮
        LED1_SET;           //LED1 灭
        delay_ms(500); //间隔 500ms
        LED0_SET;           //LED0 灭
        LED1_CLR;           //LED1 亮
```

```
            delay_ms(500); //间隔 500ms
        }
}
```

可以看到 main.c 文件里先对延时函数进行初始化（包含时钟和主频设置），再对 LED 灯所连的 GPIO 进行初始化。接下来就执行一个死循环，里面 LED0 和 LED1 交替闪烁，时间间隔 500ms。

F、所有文件都添加完成后，就可以进行编译了。

**4） 示例效果**

编译成功后，烧录到 STM32F407VxT6 最小系统板里，成功运行后，可以看到 LED0 和 LED1 不停的交替闪烁，时间间隔 500ms。

### 3.4.2 按键示例

**1） 示例目的**

让大家掌握 STM32F407 的 IO 口输入操作。

**2） 硬件说明**

本示例需要用到 LED 灯：LED0 和 LED1，按键：KEY0，KEY_UP，其中 LED 灯硬件在 LED 示例已说明，这里只介绍按键硬件连接。如图 3.34 所示，KEY_UP(T_KEY)连接在 PA0 引脚上，KEY0（T_KEY1）连接在 PE4 上。

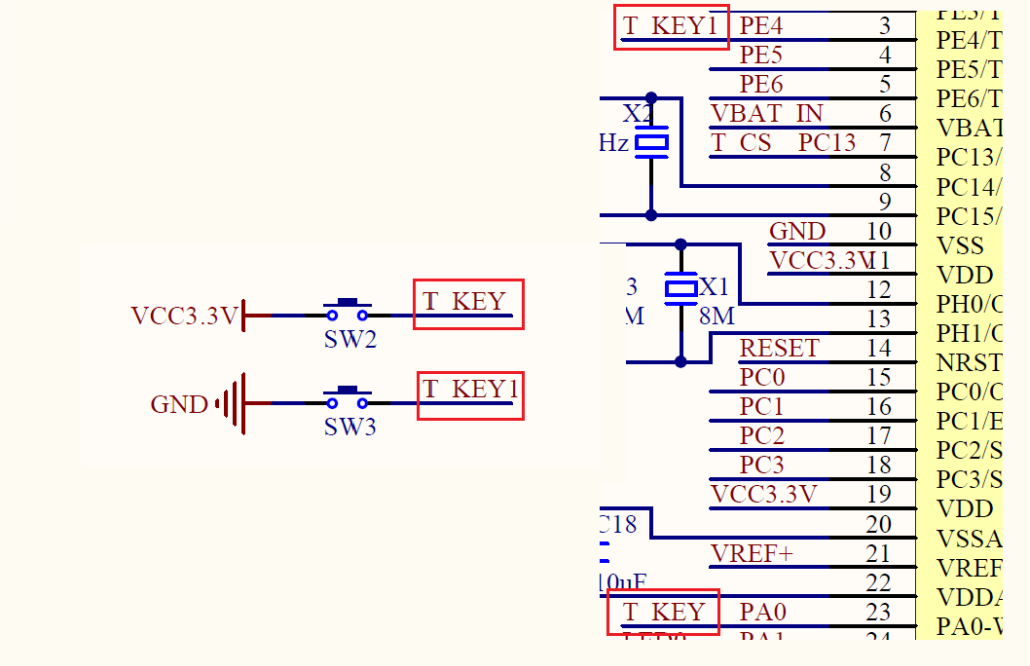



图 3.34 按键与 STM32F407 连接原理图

### 3） 软件设计

从硬件连接原理图上看，只要 PE4 引脚设为输入并实时读取 PE4 上的电平状态，一旦 SW3（KEY0）按下，PE4 被接到低电平，软件上只要判断 PE4 引脚电平为低电平，就认为 SW3（KEY0）按下了。SW2（KEY_UP）按键工作原理是一样的，只是需要判断 PA0 为低电平时才认为 SW2（KEY_UP）按下了。根据此原理，软件设计步骤如下：

A、按照上述新建 MDK5 项目工程的方法新建一个名称为 KEY 的项目工程（如果是直接从模板或者 LED 示例拷贝过来的，需要将项目工程文件名称改为 KEY.uvproj）。

B、这里只说明 KEY 相关的文件怎么添加，其他外设文件添加请参考相关示例。在项目目录的 HARDWARE 目录下新建 KEY 目录，在 KEY 目录下新建 key.c 和 key.h 文件（新建方法见上述示例说明）。

C、在 key.c 文件里添加如下内容并保存（大家可以直接去示例程序里拷贝）

```
#include "key.h"
#include "delay.h"

/**************************************************************************
************
 * @name            :void KEY_Init(void)
 * @date            :2020-05-08
 * @function        :Initialize KEY GPIO
 * @parameters :None
 * @retvalue        :None
**************************************************************************
************/
void KEY_Init(void)
{
    GPIO_InitTypeDef    GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPI
OE, ENABLE);//使能 GPIOA,GPIOE 时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;               //KEY0  对应引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;            //普通输入模式
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //100M
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP ;           //上拉
    GPIO_Init(GPIOE, &GPIO_InitStructure);                  //初始化 PE4
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;               //KEY_UP 对应 PA0
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN ;         //下拉
    GPIO_Init(GPIOA, &GPIO_InitStructure);                  //初始化 PA0
}
```

```
/**************************************************************************
************
 * @name            :u8 KEY_Scan(u8 mode)
 * @date            :2020-05-08
 * @function        :Scan whether key is pressed or not(Key response
priority:KEY0>KEY_UP)
 * @parameters :mode:0-Continuous pressing is not supported
                                1-Support continuous press
 * @retvalue       :0-no key is pressed
                        1-KEY0 is pressed
                                                2-KEY_UP is pressed
**************************************************************************
*************/
u8 KEY_Scan(u8 mode)
{
    static u8 key_up=1;//按键按松开标志
    if(mode)key_up=1;   //支持连按
    if(key_up&&(KEY0_VALUE==0||KEY_UP_VALUE==1))
    {
        delay_ms(10);//去抖动
        key_up=0;
        if(KEY0_VALUE==0)
        {
            return KEY0_PRES;
        }
        else if(KEY_UP_VALUE==1)
        {
            return KEY_UP_PRES;
        }
    }
    else if(KEY0_VALUE==1&&KEY_UP_VALUE==0)
    {
        key_up=1;
    }
    return 0;// 无按键按下
}
```

这里面包含有 2 个函数：KEY_Init 和 KEY_Scan，其中 KEY_Init 对按键所连的 GPIO 进行初始化，可以看到 GPIO 口都被初始化为输入模式，且 KEY0 所接 GPIO 被设置成上拉，KEY_UP 所接 GPIO 口被设置为下拉，因为他们的触发电平不一样。KEY_Scan 为按键扫描函数，一般放在死循环里面采用轮询的方式进行扫描。此函数带有去抖动和支持连按功能。

在 key.h 文件里添加如下内容并保存（大家可以直接去示例程序里拷贝）

```
#ifndef __KEY_H
#define __KEY_H
#include "sys.h"

//KEY 端口读取方式切换
#define KEY_IO_DEF 0     //0-位操作，1-库函数操作，2-寄存器操作

#if KEY_IO_DEF==0       //位操作方式读取按键端口值
#define KEY0_VALUE     PEin(4)    //PE4
#define KEY_UP_VALUE PAin(0)    //PA0
#elif KEY_IO_DEF==1    //库函数方式读取按键端口值
#define KEY0_VALUE            GPIO_ReadInputDataBit(GPIOE,GPIO_Pin_4) //PE4
#define KEY_UP_VALUE         GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0)
      //PA0
#else    //寄存器方式读取按键端口值
#define KEY0_VALUE       ((GPIOE->IDR&GPIO_Pin_4)?1:0)
#define KEY_UP_VALUE    ((GPIOA->IDR&GPIO_Pin_0)?1:0)
#endif
#define KEY0_PRES          1  //KEY0 按下
#define KEY_UP_PRES     2    //KEY_UP 按下
void KEY_Init(void);//IO 初始化
u8 KEY_Scan(u8);                //按键扫描函数
#endif
```

里面定义了三种 GPIO 读取方式：位操作、库函数操作和寄存器操作。使用宏定义 KEY_IO_DEF 来进行区分。其中位操作使用起来最简便，它的定义放在 SYSTEM 目录下的 sys.h 文件里面。库函数操作使用起来比较直观易懂。寄存器操作使用起来有点复杂，需要查看寄存器名称。大家可以根据需求自由选择。

D、文件内容写入完成后，在项目工程里新建 HARDWARE group，将 key.c 导入进去，再添加头文件路径，最后将 FWLIB Group 里不需要的外设库文件删除。以上操作方法和新建项目工程里的一致，这里不再阐述。

E、在 main.c 文件里添加如下内容：

```
#include "delay.h"
#include "sys.h"
#include "led.h"
#include "key.h"
```

```
int main(void)
{
    u8 key_value,key0_flag=0,key_up_flag=0;
    delay_init(168);        //初始化延时函数
    KEY_Init();             //初始化按键 GPIO
    LED_Init();             //初始化 LED 灯 GPIO
    while(1)
    {
        key_value=KEY_Scan(0);   //扫描按键
        if(key_value==KEY0_PRES)   //KEY0 被按下
        {
            key0_flag=!key0_flag;
            if(key0_flag)
            {
                LED0_CLR;         //LED0 亮
            }
            else
            {
                LED0_SET;         //LED0 灭
            }
        }
        else if(key_value==KEY_UP_PRES) //KEY_UP 被按下
        {
            key_up_flag=!key_up_flag;
            if(key_up_flag)
            {
                LED1_CLR;         //LED0 亮
            }
            else
            {
                LED1_SET;         //LED0 灭
            }
        }
        else   //无按键按下
        {
            delay_ms(10);
        }
    }
}
```

可以看到 main.c 文件里先对延时函数进行初始化（包含时钟和主频设置），再对 LED 灯和 KEY 按键所连的 GPIO 进行初始化。接下来就执行一个死循环，里面每隔 10ms 扫描一次按键状态，KEY0 控制 LED0，默认 LED0 是灭的，按下 KEY0，LED0

就亮，再按下 KEY0，LED0 就灭，一直这样循环处理；KEY_UP 控制 LED1，现象和 LED0 一致。

E、 所有文件都添加完成后，就可以进行编译了。

**4）示例效果**

编译成功后，烧录到 STM32F407VxT6 最小系统板里，成功运行后，按 KEY0 和 KEY_UP 按键，可以控制 LED0 和 LED1 亮灭。

### 3.4.3 串口示例

**1） 示例目的**

让大家掌握 STM32F407 的串口发送和接收数据操作。

**2） 硬件说明**

本示例用到串口 1，如图 3.35 所示，RX 接在 PA10 上，TX 接在 PA9 上。

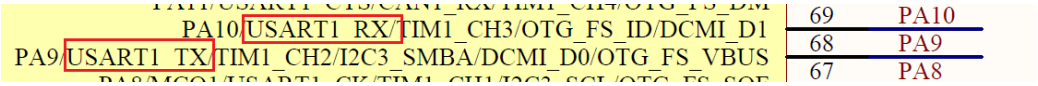



图 3.35 串口连接原理图

使用串口和 PC 机连接时，需要外接 USB 转串口模块，且 PA10（RX）需要接在 USB 转串口模块的 TX 上，PA9（TX）需要接在 USB 转串口模块的 RX 上。

**3） 软件设计**

本示例将实现串口和上位机通信，上位机发送过来字符，最小系统板接收后将全部返回给上位机。

建立项目工程、添加文件和头文件路径以及修改文件部分这里就不在阐述，和 LED 示例方法一致。着重讲解一下程序代码。

直接打开串口示例的 usart.c 文件，里面包含三个部分的内容：设置 printf 函数从串口 1 输出、串口 1 初始化、串口 1 中断复位函数设计。

设置 printf 函数从串口 1 输出，代码如下：

```
#if !USE_MICROLIB
#pragma import(__use_no_semihosting) //导入__use_no_semihosting 符号，确保没有从 C 库使用半主机的函数
//标准库需要的支持函数
struct __FILE
{
    int handle;
};
FILE    stdout;
```

```
//定义_sys_exit()以避免使用半主机模式
void _sys_exit(int x)
{
    x = x;
}
#endif
//重定义 fputc 函数
int fputc(int ch, FILE *f)
{
    while (USART_GetFlagStatus(USART1,USART_FLAG_TC)==RESET);//循环发送,直到发送完毕
    USART_SendData(USART1,(u8)ch);
    return ch;
}
```

方法有两种：一种是使用 MicroLIB；一种是不使用 MicroLIB。不管用哪种方法都要重定义 fputc 函数，将函数的内容从标准的输入输出改为串口输入输出。代码中 USE_MICROLIB 宏定义就是控制着两种方法切换的。当 USE_MICROLIB 为 0 时，不使用 MicroLIB，就执行上述代码的内容；当 USE_MICROLIB 为 1 时，使用 MicroLIB，需要在 keil 软件里面设置，打开 keil 软件，按照图 3.36 所示步骤设置。

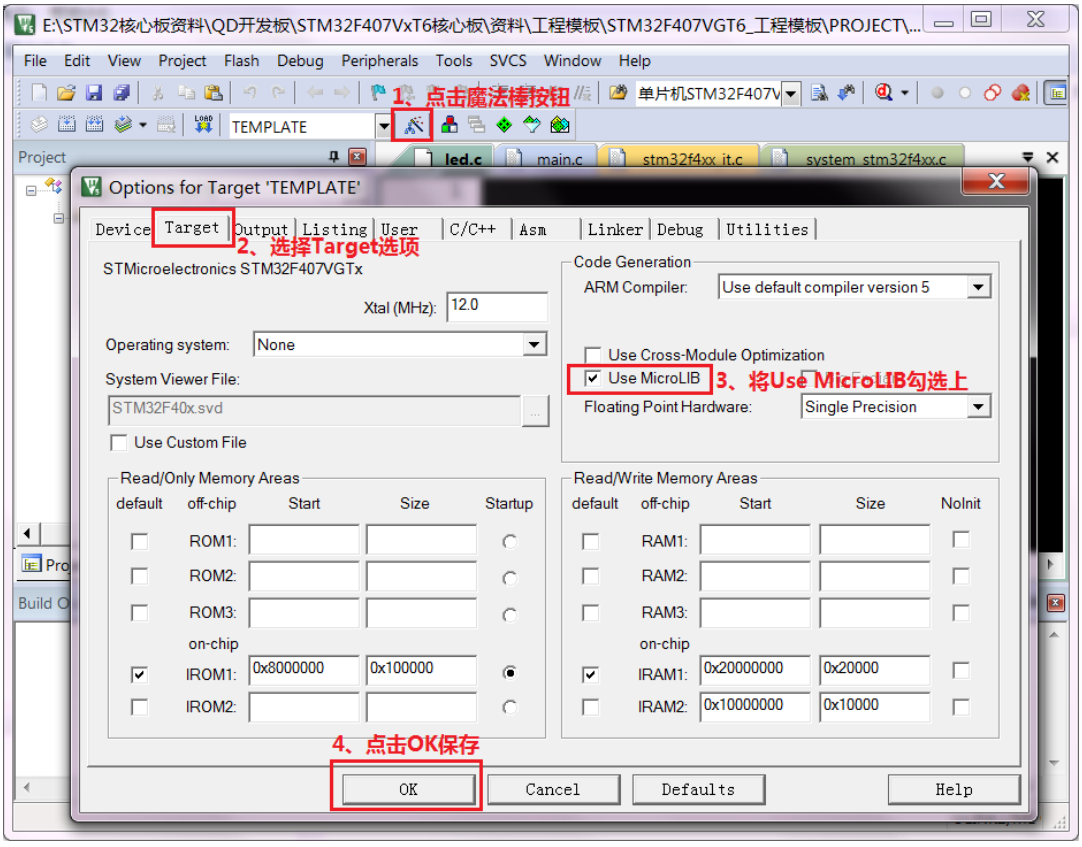



图 3.36 设置使用 MicroLIB

USART1_Init 函数用于串口 1 初始化，这里大致讲解一下初始化过程，具体代码大家可以去示例中查看。初始化过程如下：

A、 使能 GPIOA 时钟和串口 1 时钟；

B、 将 PA9 和 PA10 复用映射；

C、 初始化 PA9 和 PA10 引脚，将它们初始化为复用功能、速度 50MHz、推挽复用输出、上拉。

D、 初始化串口 1，设置串口波特率、发送或接收字长、停止位、奇偶校验位、硬件数据流控制、收发模式等。

E、 使能串口 1、清除标志位、开启串口中断。

F、 配置中断参数，包括中断通道、优先级等等。

USART1_IRQHandler 为串口 1 中断服务函数，用来接受数据。这里大致讲解一下它的工作流程：

```
void USART1_IRQHandler(void)
{
    u8 Res;
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)  //接收中断(接收到的数据必须是 0x0d 0x0a 结尾)
    {
        Res =USART_ReceiveData(USART1);    //读取接收到的数据
        if((USART_RX_STA&0x8000)==0)//接收未完成
        {
            if(USART_RX_STA&0x4000)//接收到了 0x0d
            {
                if(Res!=0x0a)
                {
                    USART_RX_STA=0;//接收错误,重新开始
                }
                else
                {
                    USART_RX_STA|=0x8000;    //接收完成了
                }
            }
            else //还没收到 0X0D
            {
                if(Res==0x0d)
                {
                    USART_RX_STA|=0x4000;
                }
```

```
                else
                {
                    USART_RX_BUF[USART_RX_STA&0X3FFF]=Res ;
                    USART_RX_STA++;
                    if(USART_RX_STA>(USART_REC_LEN-1))
                    {
                        USART_RX_STA=0;//接收数据错误,重新开始接收

                    }
                }
            }
        }
    }
```

首先判断上位机是否发送数据，一旦发送数据，STM32F407 立即进行接收，每次接收一个字节，当接收的数据以 0x0d 0x0a 结尾，则表示数据接收完毕，否则继续接收数据，如果数据只以 0x0d 或者 0x0a 结尾，则表示接收数据出错，需要重新接收。

main 函数的内容如下：

```
#include "delay.h"
#include "sys.h"
#include "usart.h"
#include "led.h"
#include "key.h"

int main(void)
{
    u16 rlen=0,times=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分
组 2
    delay_init(168);        //初始化延时函数
    USART1_Init(115200); //初始化串口 1
    KEY_Init();              //初始化按键 GPIO
    LED_Init();              //初始化 LED 灯 GPIO
    while(1)
    {
        if(USART_RX_STA&0x8000)
        {
            rlen=USART_RX_STA&0x3fff;//得到此次接收到的数据长度
            printf("\r\n 您输入的内容为:");
            printf("%.*s\r\n\r\n",rlen,USART_RX_BUF);//发送输入的内容
            USART_RX_STA=0;
        }else
```

```
        {
            times++;
            if(times%5000==0)
            {
                printf("\r\nSTM32F407VxT6 最小系统开发板串口示例\r\n");
            }
            if(times%200==0)printf("请输入内容,点击发送\r\n");
            if(times%30==0)LED0=!LED0;//闪烁 LED,提示系统正在运行.
            delay_ms(10);
        }
        if(KEY_Scan(0)==KEY0_VALUE) //按下 KEY0 按键
        {
            delay_ms(200);   //放在连续输出
            printf("\r\nKEY0 按键被按下\r\n");
        }
    }
}
```

可以看到代码里实时在接收上位机发送的数据，接收完成后，立即将数据发送给上位机。LED0 指示程序运行状态，按下 KEY0 键，也会给上位机发提示信息。

**4） 示例效果**

程序运行后可以看到，打开串口工具上位机，可以看到效果和 main 函数运行的一致。如图 3.37 所示。

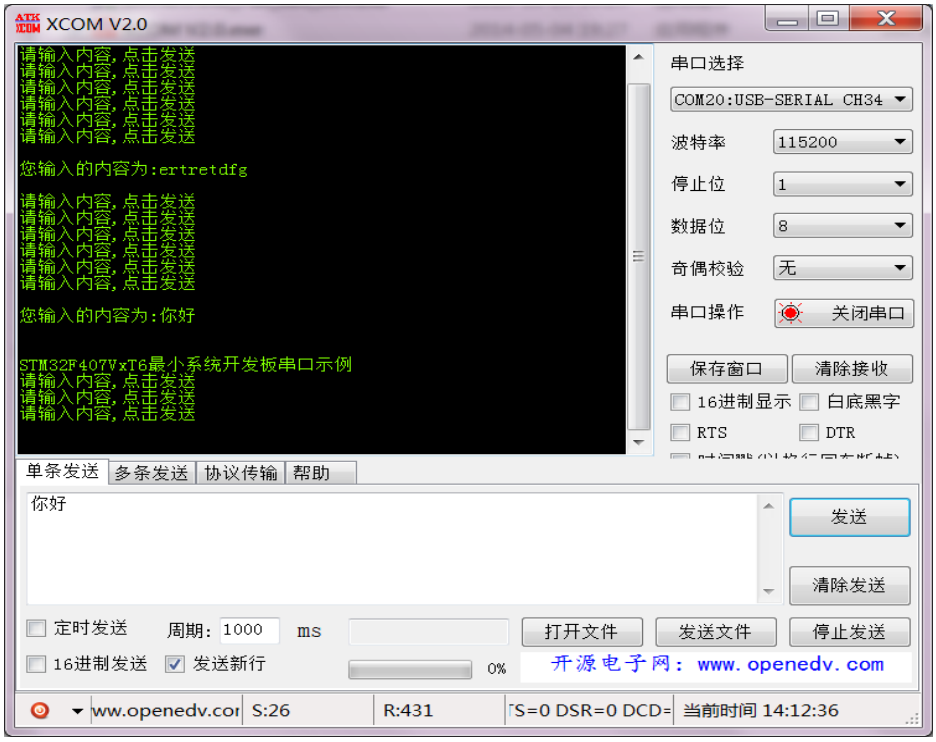



图 3.37 串口示例

### 3.4.4 外部中断示例

#### 1） 示例目的

让大家熟练掌握 IO 中断操作。

#### 2） 硬件说明

本示例用的硬件和按键示例一样，这里不再说明。

#### 3） 软件设计

打开示例程序 main.c 文件，内容如下：

```
#include "delay.h"
#include "sys.h"
#include "led.h"
#include "key.h"
#include "exti.h"

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);        //初始化延时函数
    KEY_Init();             //初始化按键 GPIO
    EXTIX_Init();           //初始化外部中断输入
    LED_Init();             //初始化 LED 灯 GPIO
    while(1)
    {
        delay_ms(1000);     //1s 延时
    }
}
```

可以看到本示例和上述按键示例是有区别的，上述按键示例采用轮询的方式检测按键事件，本示例采用中断方式检测按键示例，只要按键按下就会触发中断，进入中断服务程序。

EXTIX_Init 为外部中断初始化函数，大家可以打开示例程序看代码，它主要的作用就是对中断引脚、中断触发方式、中断优先级进行设置。

下面以 KEY0 按键为例介绍一下中断服务程序，内容如下：

```
void EXTI4_IRQHandler(void)
{
    delay_ms(10); //消抖
    if(KEY0_VALUE==0)
    {
```

```
            LED0=!LED0;
        }
        EXTI_ClearITPendingBit(EXTI_Line4);//清除 LINE4 上的中断标志位
}
```

可以看到一旦 KEY0 按键按下，会立即进行消抖处理，然后控制 LED0 亮灭，最后将当期的中断标志位清除，为下一次中断触发做准备。

#### 4） 示例效果

程序成功运行后，可以看到示例效果和按键示例一致。

### 3.4.5 独立看门狗示例

#### 1） 示例目的

让大家熟练掌握独立看门狗操作。

#### 2） 硬件说明

独立看门狗存在 STM32F407 内部，没有外部电路，它是一个计数器，一旦在设定的时间内计数器清 0，则会产生一个复位信号，让 STM32F407 复位重启。示例中用到的其他硬件在之前的示例中已经说明。

#### 3） 软件设计

打开示例程序中 main.c 代码，内容如下：

```
#include "delay.h"
#include "sys.h"
#include "led.h"
#include "key.h"
#include "exti.h"
#include "iwdg.h"

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);        //初始化延时函数
    KEY_Init();             //初始化按键 GPIO
    EXTIX_Init();           //初始化外部中断输入
    LED_Init();             //初始化 LED 灯 GPIO
    delay_ms(200);          //延时 200ms
    IWDG_Init(5,500);       //分频因子为 128,重载值为 500,溢出时间为 2s
    LED1=0;                 //点亮 LED1，使其常亮
    while(1)
```

```
    {
        delay_ms(1000);       //1s 延时
    }
}
```

这里其他的内容就不介绍了，重点介绍下 IWDG_Init 函数，内容如下：

```
void IWDG_Init(u8 prer,u16 rlr)
{
    IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable); //使能对 IWDG->PR
IWDG->RLR 的写
    IWDG_SetPrescaler(prer); //设置 IWDG 分频系数,分频因子=4*2^prer.但最
大值只能是 256!
    IWDG_SetReload(rlr);     //设置 IWDG 装载值
    IWDG_ReloadCounter(); //reload
    IWDG_Enable();           //使能看门狗
}
```

可以看到此函数首先对看门狗进行写使能，然后设置分频系数和装载值并重载，最后使能看门狗。其中喂狗时间间隔由分频系数和装载值决定，计算公式为：  tout=((4*2^prer)*rlr)/32(ms)，其中 prer 为分频系数，rlr 为装载值。

接下里我们来看按下 KEY_UP 按键怎么实现喂狗，中断服务函数内容如下：

```
void EXTI0_IRQHandler(void)
{
    delay_ms(10); //消抖
    if(KEY_UP_VALUE==1)
    {
        IWDG_Feed();
    }
    EXTI_ClearITPendingBit(EXTI_Line0); //清除 LINE0 上的中断标志位
}
```

可以看到，按下 KEY_UP 按键后，就会调用 IWDG_Feed 函数喂狗，而 IWDG_Feed 函数则调用 IWDG_ReloadCounter 函数进行值重载。

**4） 示例效果**

程序运行成功后，会看到 LED1 常亮，如果一直不按 KEY_UP 按键，LED1 会每隔大约 2 秒熄灭一次；如果不停的按 KEY_UP 按键（间隔小于 2 秒），LED1 则不会熄灭。

### 3.4.6 定时器示例

#### 1） 示例目的

让大家熟练掌握 STM32F407 通用定时器操作。

#### 2） 硬件说明

本示例要用到 LED0、LED1 以及通用定时器 3（TIM3）。LED 灯前面已经介绍，定时器在 STM32F407 内部，只需要软件设置。

#### 3） 软件设计

打开示例程序中的 mian.c 文件，内容如下：

```
#include "delay.h"
#include "sys.h"
#include "led.h"
#include "timer.h"

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);        //初始化延时函数
    LED_Init();              //初始化 LED 灯 GPIO
    //tout=((9999+1)*(8399+1))/84(us)=1s
    TIM3_Int_Init(9999,8399);   //定时器 3 初始化，时钟 84M，分频系数 8400，所以 84M/8400=10Khz 的计数频率，计数 10000 次为 1s
    while(1)
    {
        LED0=!LED0;          //LED0 翻转
        delay_ms(200);        //200ms 延时
    }
}
```

这里重点看下 TIM3_Int_Init 函数，内容如下：

```
void TIM3_Int_Init(u16 arr,u16 psc)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE);   ///使能 TIM3 时钟
  TIM_TimeBaseInitStructure.TIM_Period = arr;  //自动重装载值
    TIM_TimeBaseInitStructure.TIM_Prescaler=psc;   //定时器分频
```

```
    TIM_TimeBaseInitStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
    TIM_TimeBaseInitStructure.TIM_ClockDivision=TIM_CKD_DIV1;
    TIM_TimeBaseInit(TIM3,&TIM_TimeBaseInitStructure);//初始化 TIM3
    TIM_ITConfig(TIM3,TIM_IT_Update,ENABLE); //允许定时器 3 更新中断
    TIM_Cmd(TIM3,ENABLE); //使能定时器 3
    NVIC_InitStructure.NVIC_IRQChannel=TIM3_IRQn; //定时器 3 中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0x01; //抢占优先级 1
    NVIC_InitStructure.NVIC_IRQChannelSubPriority=0x03; //子优先级 3
    NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

从代码来看，主要是设置定时器的计时间隔和中断以及开启定时器。

定时器中断服务函数内容如下：

```
void TIM3_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM3,TIM_IT_Update)==SET) //溢出中断
    {
        LED1=!LED1;//LED1 翻转
    }
    TIM_ClearITPendingBit(TIM3,TIM_IT_Update);  //清除中断标志位
}
```

从代码来看，产生中断时，控制 LED1 亮灭。

**4） 示例效果**

程序正常运行后，可以看到 LED0 每隔 400ms 亮灭一次，LED1 每隔 2s 亮灭一次。

### 3.4.7 PWM 输出示例

**1） 示例目的**

让大家掌握使用 STM32F407 的通用定时器产生 PWM 输出。

**2） 硬件说明**

本示例要用到 LED0 和 LED1 以及定时器 TIM5(将 TIM5 的 CH2 输出到 PA1)。

通过定时 TIM5 的 CH2 通道输出 PWM 到 LED0（接在 PA1 上)。

**3） 软件设计**

打开示例程序的 main.c 文件，内容如下：

```
#include "delay.h"
#include "sys.h"
#include "led.h"
#include "pwm.h"

int main(void)
{
    u8 dir=1;
    u16 PWM_Value=0;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);        //初始化延时函数
    LED_Init();              //初始化 LED 灯 GPIO
    //tout=((499+1)*(83+1))/84(us)=500us,PWM 频率为 2KHz
    TIM5_PWM_Init(499,83);    //定时器 5 PWM 初始化，时钟 84M，分频系数 84，所以 84M/84=1Mhz 的计数频率，计数 500 次为 500us
    LED1=0;                  //点亮 LED1
    while(1)
    {
        delay_ms(10);
        if(dir)
        {
            PWM_Value++;    //dir==1 led0pwmval 递增
        }
        else
        {
            PWM_Value--;//dir==0 led0pwmval 递减
        }
        if(PWM_Value>300)
        {
            dir=0;          //led0pwmval 到达 300 后，方向为递减
        }
        if(PWM_Value==0)
        {
            dir=1;           //led0pwmval 递减到 0 后，方向改为递增
        }
        TIM_SetCompare2(TIM5,PWM_Value); //修改比较值，修改占空比
    }
}
```

可以看到整个过程就是驱动 LED0 从暗到亮，再从亮到暗变化。

TIM5_PWM_Init 函数用于初始化 PWM 设置，这里只说明 PWM 设置部分，内容如下：

```
    //初始化 TIM5 Channel2 PWM 模式
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; //选择定时器模式:TIM 脉冲宽度调制模式 2
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出使能
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low; //输出极性:TIM 输出比较极性低
    TIM_OC2Init(TIM5, &TIM_OCInitStructure);  //根据 T 指定的参数初始化外设 TIM5
    TIM_OC2PreloadConfig(TIM5, TIM_OCPreload_Enable);  //使能 TIM5 在 CCR1 上的预装载寄存器
```

从代码看 PWM 设置需要在通用定时器初始完成的基础上再选择 PWM 模式，设置输出使能和极性等等。

**4） 示例效果**

代码成功运行后，可以看到 LED1 常亮，LED0 从暗到亮，再从亮到暗循环变化。

### 3.4.8 TFT_LCD 显示示例

**1） 示例目的**

让大家掌握使用 STM32F407 的 FSMC 总线驱动 LCD 显示。

**2） 硬件说明**

本示例要用到 LED0、USART1 和 LCD。其中 LED0 和 USART1 前面已经介绍过了，这里只介绍 LCD 硬件连接。

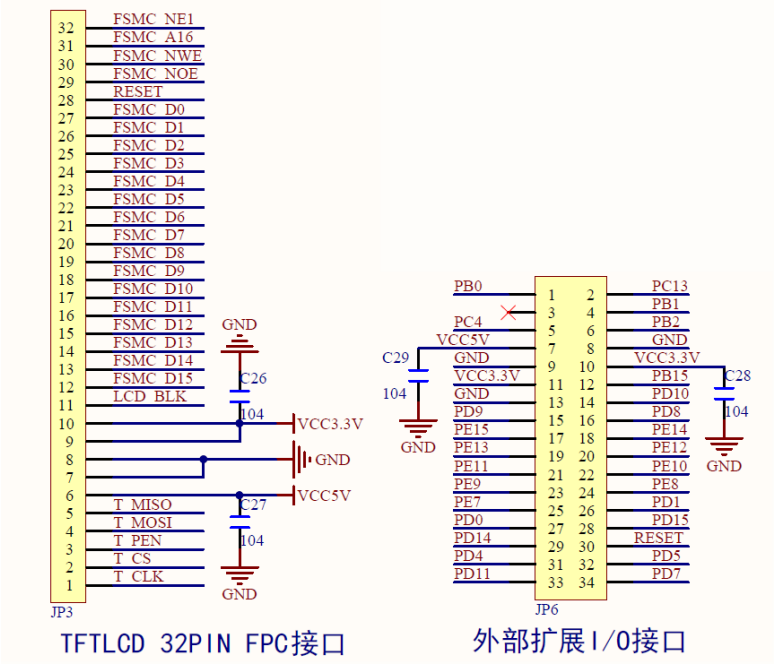



图 3.38 LCD 和 STM32F407 连接原理图

原理图中展示了最小系统板上连接 LCD 模块的两个接口：32pin FPC 接口和 34pin 排针接口。这里要说明的是 LCD 模块各引脚连接：

片选引脚接 FSMC_NE1（PD7）；

数据/命令选择引脚接 FSMC_A16（PD11）；

写控制引脚接 FSMC_NEW（PD5）；

读控制引脚接 FSMC_NOE（PD4）；

16bit 数据引脚接 FSMC_D0~D15；

背光接 PB15

**3） 软件设计**

打开示例程序中的 main.c 文件，内容如下：

```
#include "delay.h"
#include "sys.h"
#include "lcd.h"
#include "gui.h"
#include "test.h"
#include "usart.h"
#include "led.h"

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断优先级分组 2
    delay_init(168);       //初始化延时函数
    USART1_Init(115200); //串口 1 初始化
    LED_Init();             //初始化 LED
    LCD_Init();          //液晶屏初始化
    LED0=0;           //点亮 LED0
  //循环测试
    while(1)
    {
        main_test();         //测试主界面
        Test_Read();       //读 ID 和颜色值测试
        Test_Color();              //简单刷屏填充测试
        Test_FillRec();       //GUI 矩形绘图测试
        Test_Circle();        //GUI 画圆测试
        Test_Triangle();      //GUI 三角形绘图测试
        English_Font_test();//英文字体示例测试
        Chinese_Font_test();//中文字体示例测试
        Pic_test();                //图片显示示例测试
        Test_Dynamic_Num(); //动态数字显示测试
```

```
            Rotate_Test();     //旋转显示测试
            POINT_Test();            // Loading  测试
            Combo_Test();              //  下拉列表测试
            ProgresBar_Test();   //  进度条测试
            BarReport_Test();        //  条形报表测试
            Window_Test();               //  窗口测试
            Button_Test();           //  按钮测试
        }
}
```

从代码来看此示例进行了十几项 LCD 显示测试。

这里介绍一下 LCD_Init 函数，由于代码量比较大，只大致讲解一下内容。

```
void LCD_Init(void)
{
    LCD_GPIOInit();//LCD GPIO 初始化
    lcddev.id=LCD_Read_ID();
    printf("LCD ID:0x%X\r\n",lcddev.id);
    LCD_Set_BWTR();  //重新配置写时序寄存器的时序使 WR 时序为最快
    delay_ms(100);
    //LCD_RESET();      //如果不使用开发板复位引脚，则调用此复位函数
    switch(lcddev.id)
    {
      ……     //各种 LCD IC 初始化
    }
    LCD_direction(USE_HORIZONTAL);//设置 LCD 显示方向
    LCD_LED=1;//点亮背光
    LCD_Clear(WHITE);//清全屏白色
}
```

从代码来看，LCD 初始化流程如下：

A、 初始化 GPIO,包括 FSMC 总线初始化；

B、 读取 LCD 内部 IC 的 ID，然后通过串口打印出来；

C、 重新设置 FSMC 写时序；

D、 进行 LCD 复位；

E、 根据读取到的 ID 进行 LCD 初始化；

F、 设置 LCD 显示方向，包括 LCD 坐标写入命令和 GRAM 写入和读取命令设置；

G、 点亮背光并将 LCD 清为白屏；

因为代码量大，关于 LCD 的写入和读取函数、窗口设置函数，这里就不介绍了，大家可以到示例代码里查看。

4） 示例效果

示例成功运行，可以看到 LCD 显示图形和文字的界面，如图 3.39 所示：


图 3.39 LCD 显示界面

### 3.4.9 RTC 实时时钟示例

1） 示例目的

让大家掌握 RTC 实时时钟使用方法。

2） 硬件说明

本示例要用到 LED0、LED1、LCD、USART1 以及 RTC 实时时钟模块，其他 RTC 模块在 STM32F407 内部，其他的硬件之前都有说明。

3） 软件设计

打开示例程序里的 main.c 文件内容如下：

```
extern u8 rtc_show_flag;
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);                    //初始化延时函数
    USART1_Init(115200);                //串口 1 初始化
    LED_Init();                         //初始化 LED
    LCD_Init();                         //液晶屏初始化
    RTC_Set_Init();                     //RTC 初始化
    RTC_Set_WakeUp_IRQ(RTC_WakeUpClock_CK_SPRE_16bits,0);    //配置 WAKE UP 中断,1 秒钟中断一次
    RTC_Set_AlarmA_IRQ();               //设置闹钟中断
    RTC_Set_AlarmA(5,9,31,0);           //设置闹钟时间 周四 19:10:00
    RTC_test();
    RTC_Show_AlarmA_time();             //显示设置的闹钟时间
  //循环测试
    while(1)
    {
        if(rtc_show_flag)
        {
            RTC_Show_time();
            rtc_show_flag=0;
        }
        delay_ms(1);
    }
}
```

从代码里看，开始初始化了 RTC 模块，然后设置唤醒中断，接下里设置闹钟，最后将实时时间显示在 LCD 上。

在 rtc.c 文件里包含 RTC 设置的全部函数,这里简单说明一下各个函数的功能。

RTC_Set_Time 用于设置 RTC 实时时间的时、分、秒。

RTC_Set_Date 用于设置 RTC 实时时间的年、月、日。

RTC_Set_Init 用于初始化 RTC 模块并设置初始化时间,上电后只会初始化一次。

RTC_Set_AlarmA 用于闹钟时间，包括星期、时、分、秒等。

RTC_Set_AlarmA_IRQ 用于设置闹钟中断。

RTC_Set_WakeUp_IRQ 用于设置实时时钟唤醒中断，包括唤醒间隔。

RTC_Alarm_IRQHandler 为闹钟中断服务函数,这里设置闹钟到了就点亮 LED1。

RTC_WKUP_IRQHandler 为时钟中断服务函数，这里设置中断触发标志位并且切

换 LED0 状态。

**4） 示例效果**

本示例运行成功后，LED0 间隔 2 秒闪烁一次，LCD 上显示实时时钟和设置的闹钟时间，当闹钟时间到了， LED1 会点亮，同时 LCD 显示相应的提示信息。如图 3.40 所示：

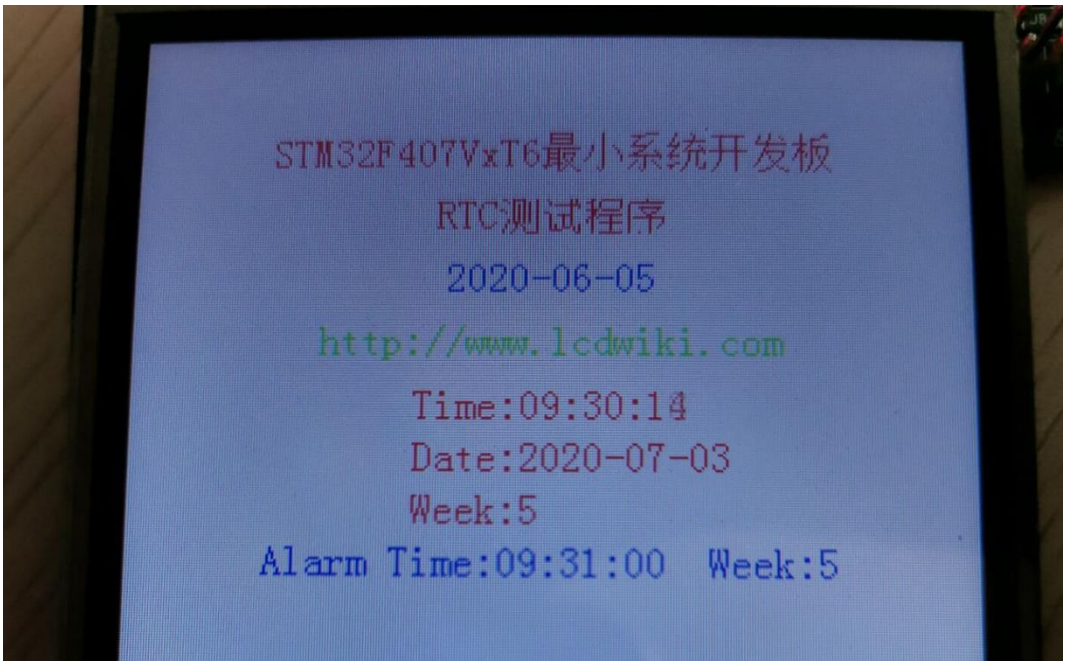



图 3.40 RTC 实时时钟显示界面

### 3.4.10 待机唤醒示例

**1） 示例目的**

让大家掌握 STM32F407 进入待机模式和待机唤醒的方法。

**2） 硬件说明**

本示例要用到 LED0、KEY_UP、LCD，这几个硬件前面都介绍过了。待机和唤醒是 STM32F407 内部的功能，外部通过 KEY_UP 来触发。

**3） 软件设计**

打开示例程序中的 main.c 文件，找到如下内容：

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);                //初始化延时函数
    USART1_Init(115200);            //串口 1 初始化
    LED_Init();                     //初始化 LED
```

```
    LCD_Init();                          //液晶屏初始化
    WKUP_Init();                        //wake up 初始化
    wakeup_test();                      //wake up test 显示
  //循环测试
    while(1)
    {
        LED0=!LED0;
        delay_ms(250);//延时 250ms
    }
}
```

其中，WKUP_Init 为待机唤醒初始化函数，wakeup_test 为 LCD 显示函数，将示例标题内容显示到 LCD 上。

待机唤醒相关的函数都定义在 wkup.c 文件里，我们大致讲解一下，大家可以到示例程序查看相关的代码：

Sys_Enter_Standby 为进入系统待机模式的函数。在进入待机模式之前，需要先复位所有 IO，使能 Power 时钟，使能后备区域访问，关闭 RTC 中断，清除唤醒标志位，设置唤醒使能，然后才进入待机模式。

Check_WKUP 函数用于处理 KEY_UP 按键功能的，在正常开机时，长按 KEY_UP 按键超过 3 秒，就会进入待机模式；在待机模式下，长按 KEY_UP 按键超过 3 秒，就会正常开机。

EXTI0_IRQHandler 就是 KEY_UP 所接的待机唤醒引脚的中断服务函数，只要 KEY_UP 长按超过 3 秒，就会触发中断进入待机模式。

WKUP_Init 为待机唤醒初始化函数，就是对 KEY_UP 所接的待机唤醒引脚进行初始化，需要注意的是，执行初始化后会进入待机状态。

**4） 示例效果**

本示例运行成功后，LCD 会闪一下，然后进入待机状态，此时长按 KEY_UP 按键 3 秒，LCD 显示，LED0 闪烁，进入唤醒状态，再长按 KEY_UP 按键 3 秒，LCD 显示关闭，LED0 熄灭，进入待机状态。需要注意唤醒状态下才能下载程序。

### 3.4.11 ADC 示例

**1） 示例目的**

让大家掌握使用 STM32F407 的 ADC 通道来采集外部电压值。

**2） 硬件说明**

本示例用到 LED0、串口 1、LCD 和 ADC1 第 5 通道（接在 PA5 上）。前面三个硬

件说明已经介绍。ADC 模块在 STM32F407 内部。本示例就是要通过 ADC1 的通道 51 来读取外部电压值，这里我们直接读取 3.3V 电压，需要使用杜邦线将 PA5 接到 3.3V 引脚，同时参考电压引脚 VREF 需要用跳帽接到 3.3V 上。如图 3.41 所示：

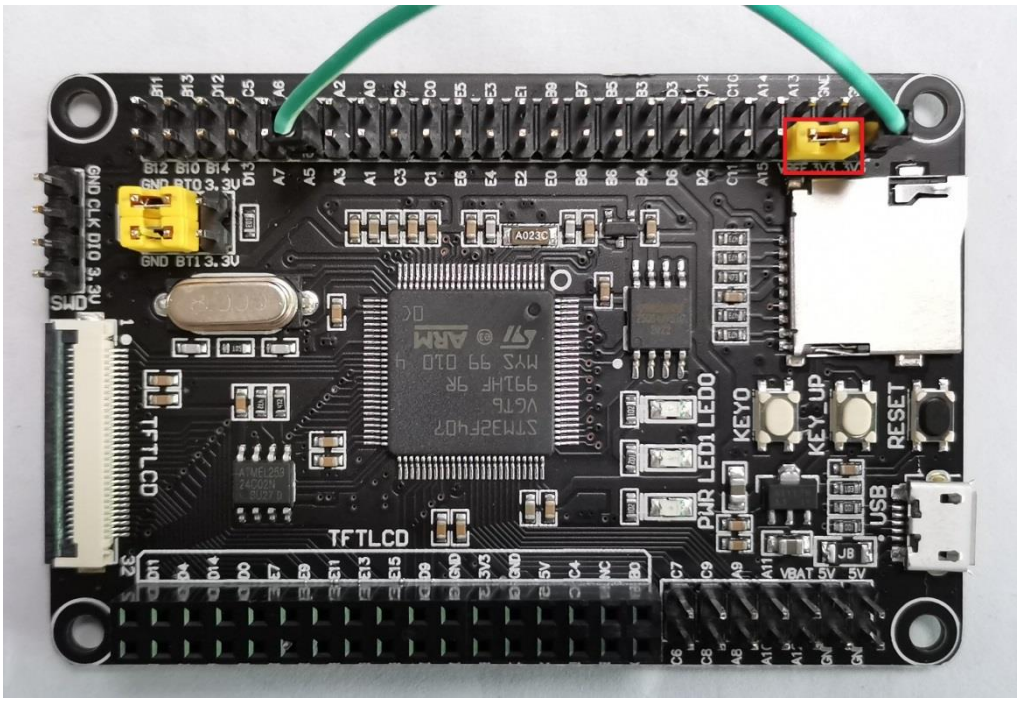



图 3.41 ADC 示例接线图

### 3） 软件设计

打开示例程序 main.c 功能，找到如下内容：

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);                //初始化延时函数
    USART1_Init(115200);            //串口 1 初始化
    LED_Init();                     //初始化 LED
    LCD_Init();                     //液晶屏初始化
    ADC1_Init();                    //ADC 通道初始化
    ADC_test();                     //显示 ADC 信息
  //循环测试
    while(1)
    {
        Show_ADC_Value();           //显示获取到的 ADC 值
        LED0=!LED0;
        delay_ms(250);//延时 250ms
    }
}
```

其中 ADC1_Init 为 ADC 初始化函数，ADC_test 为示例标题显示函数，Show_ADC_Value 为将 ADC 值显示到 LCD 上的函数。

ADC 相关的函数都定义在 adc.c 里面，大家可以到示例中查看，这里简单介绍一下：

ADC1_Init 为 ADC 初始化函数，内容包括初始化 PA5（注意 PA5 要初始化成模拟输入且不带上下拉），设置 ADC 模式，转换频率等等。

Get_ADC_Value 为获取 ADC 转换值的函数，在获取转换值之前需要先设置采样周期，然后启动转换功能，待转换结束后，返回转换值。

Get_ADC_Average_Value 为获取转换平均值的函数，就是多转换几次，然后求取平均值，这样可以提高转换值的准确度。

**4） 示例效果**

本示例运行成功后，可以看到 LED0 闪烁，LCD 上显示 ADC 值和实际电压值，两个值相应发生变化，当将 PA5 引脚接到 3.3V 电压引脚上时，可以看到采集到的 ADC 值基本接近最大值。如图 3.42 所示：


图 3.42 ADC 示例测试界面

### 3.4.12 DAC 示例

**1） 示例目的**

让大家掌握使用 STM32F407 的 DAC 来输出模拟电压。

**2） 硬件说明**

本示例要用到 LED0、KEY0、KEY_UP、串口 1、ADC1 第 5 通道（接在 PA5 上）、

DAC 通道 1（接在 PA4 上）。我们要使用 DAC 通道 1 输出模拟电压，然后使用 ADC 通道 5 对该输出电压进行读取。使用 KEY_UP 对 DAC 进行加，使用 KEY_UP 对 DAC 进行减。参考电压引脚 VREF 需要用跳帽接到 3.3V 上，如图 3.43 所示：

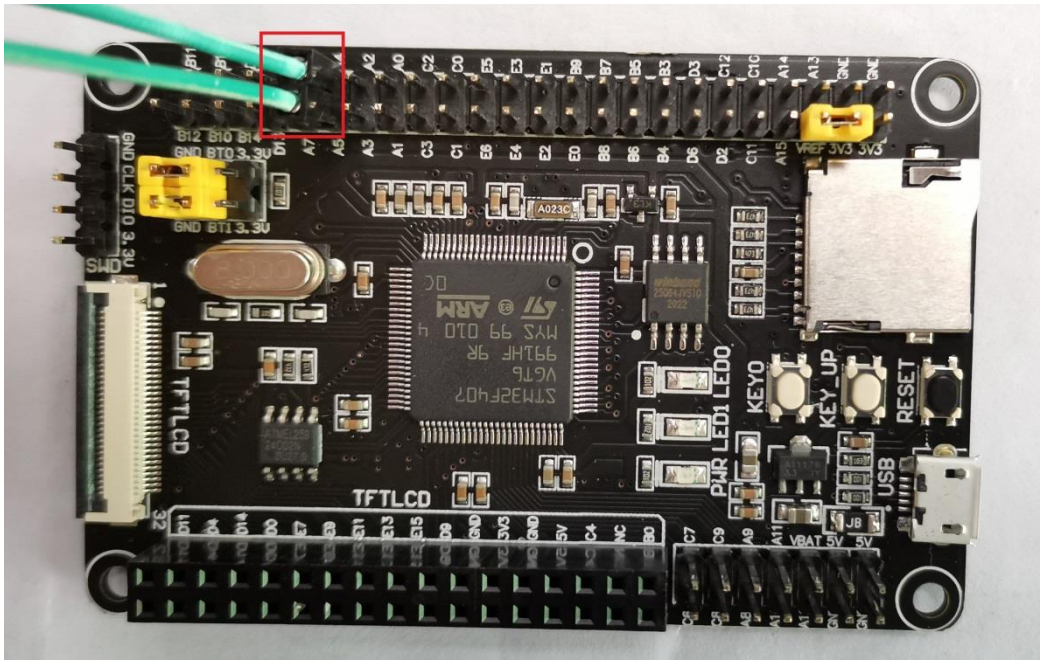



图 3.43 DAC 示例接线图

### 3） 软件设计

打开示例程序的 main.c 文件，内容如下：

```
int main(void)
{
    u8 i;
    u16 dac_value=0;
    u8 key_value;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断优先级分组 2
    delay_init(168);                    //初始化延时函数
    USART1_Init(115200);                //串口 1 初始化
    LED_Init();                         //初始化 LED
    LCD_Init();                         //液晶屏初始化
    ADC1_Init();                        //ADC 通道初始化
    DAC1_Init();                        //DAC 通道初始化
    KEY_Init();                         //按键初始化
    DAC_test();                         //显示 DAC 信息
    while(1)
    {
        key_value=KEY_Scan(0);
        if(key_value==KEY_UP_PRES)
        {
```

```
            if(dac_value<4000)
            dac_value+=100;
        }
        else if(key_value==KEY0_PRES)
        {

            if(dac_value>=100)
            dac_value-=100;
        }
        DAC_SetChannel1Data(DAC_Align_12b_R, dac_value);//设置 DAC 值;
        if((key_value==KEY_UP_PRES)||(key_value==KEY0_PRES))
        {
            Show_DAC_Value();              //显示获取到的 DAC 值
            Show_ADC_Value();              //显示获取到的 ADC 值
        }
        if(i%20==0)
            LED0=!LED0;
        i++;
        delay_ms(10);//延时 100ms
    }
}
```

其中 DAC1_Init 为 DAC 初始化函数,DAC_test 为将 DAC 值显示在 LCD 上的函数，DAC_SetChannel1Data 为设置 DAC 值函数，其内容就是将 ADC 值设置到 DAC 寄存器里面。Show_DAC_Value 为将获取到的 DAC 值显示到 LCD 上函数，Show_ADC_Value 为将获取到的 ADC 值显示到 LCD 上函数。

DAC 相关的代码都放在 dac.c，这里大致讲解一下，大家可以到示例代码里去查看：

DAC1_Init 为 DAC 初始化函数，包括引脚初始化和通道初始化。

**4） 示例效果**

本示例运行成功后,可以看到 LED0 闪烁,LCD 上显示 ADC 值和 DAC 电压值，将 PA4 和 PA5 连接起来，主 IC 内部经过 DAC 转换后由 DAC 通道 1（PA4）输出模拟电压，然后通过 ADC1 的通道 5（PA5）采集输出的模拟电压，并显示在 LCD 上，按下 KEY0 键，DAC 值减 100，按下 KEY_UP 键，DAC 值加 100。如图 3.43 所示：


图 3.43 DAC 示例显示界面

### 3.4.13 DMA 示例

#### 1） 示例目的

让大家掌握 STM32F407 的 DMA 操作方式，熟练使用 DMA 传输数据。

#### 2） 硬件说明

本示例要用到 LED0、KEY0、串口 1、LCD 和 DMA（DMA2 数据流 7 通道 4），其他硬件前面都已经介绍了，这里只介绍 DMA。DMA，全称为：Direct Memory Access，即直接存储器访问。DMA 传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过硬件为 RAM 与 I/O 设备开辟一条直接传送数据的通路，能使 CPU 的效率大为提高。本示例要使用串口 1 通过 DMA 发送数据。串口 1 的 TX 位于 DMA2 数据流 7 的通道 4。

#### 3） 软件设计

打开示例代码的 main.c 文件，内容如下：

```
extern u8 sbuf[SEND_SIZE];
int main(void)
{
    u8 i;
    u8 key_value;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断优先级分组 2
```

```
    delay_init(168);                    //初始化延时函数
    USART1_Init(115200);                //串口 1 初始化
    LED_Init();                         //初始化 LED
    LCD_Init();                           //液晶屏初始化
    KEY_Init();                         //按键初始化
    DMA_Config(DMA2_Stream7,DMA_Channel_4,(u32)&USART1->DR,(u32)sb
uf,SEND_SIZE);//DMA2,STEAM7,CH4,外设为串口 1,存储器为 sbuf,长度
为:SEND_SIZE.
    DMA_test();                           //DMA test
    fill_sbuf();                        //填充发送缓冲区
  //循环测试
    while(1)
    {
        key_value=KEY_Scan(0);
        if(key_value==KEY0_PRES)
        {
            Show_DMA_Trans();
        }
        if(i%20==0)
        {
            LED0=!LED0;
            i=0;
        }
        i++;
        delay_ms(10);//延时 100ms
    }
}
```

从代码来看，DMA_Config 为 DMA 配置函数，DMA_test 为 DMA 示例显示函数，Show_DMA_Trans 为 DMA 传输进度显示函数。

DMA 相关的函数定义在 dma.c 文件里，这里大致说明一下，具体代码大家可以去示例程序里查看。

DMA_Config 为 DMA 设置函数，需要设置过程为：

A、 DMA 时钟使能；

B、 判断 DMA 是否可配置；

C、 待 DMA 可配置后， 设置 DMA 通道；

D、 设置 DMA 外设地址；

E、 设置 DMA 存储器地址；

F、 设置 DMA 传输模式和数据传输量；

G、 设置 DMA 外设和存储器模式；

H、 设置外设和存储器数据长度；

I、 设置 DMA 数据模式、传输优先级等；

J、 设置 DMA FIFO 模式和突发传输模式；

K、 调用 DMA_Init 函数对上述值进行设置（写入寄存器）；

DMA_Enable 为开启一次 DMA 传输函数，过程设置如下：

A、 调用 USART_DMACmd 函数使能串口 1 的 DMA 发送模式；

B、 调用 DMA_Cmd 先关闭当前 DMA 传输；

C、 调用 DMA_GetCmdStatus 函数获取 DMA 状态，确保 DMA 可以设置；

D、 调用 DMA_SetCurrDataCounter 函数设置 DMA 数据传输量；

E、 调用 DMA_Cmd 开启 DMA 传输。

**4） 示例效果**

本示例运行成功后，可以看到 LED0 闪烁，按下 KEY0，就开始 DMA 传送，同时在 LCD 上面显示传送进度,打开串口调试助手,可以收到 DMA 发送的内容。如图 3.44 所示。


图 3.44 DMA 示例显示界面

### 3.4.14 IIC 示例

#### 1） 示例目的

让大家掌握使用普通 IO 口模拟 IIC 是时序，对 24C02 EEPROM 进行读写，并将结果显示在 LCD 上。

#### 2） 硬件说明

本示例需要用到 LED0、KEY0、KEY_UP、串口 1、LCD 和 24C02 EEPROM。前面的硬件已经介绍，这里只介绍 24C02 EEPROM。24C02 EEPROM 采用 IIC 接口通信，其中 IIC 时钟引脚接在主芯片的 PB8 上，IIC 数据引脚接在主芯片的 PB9 上。如图 3.45 所示

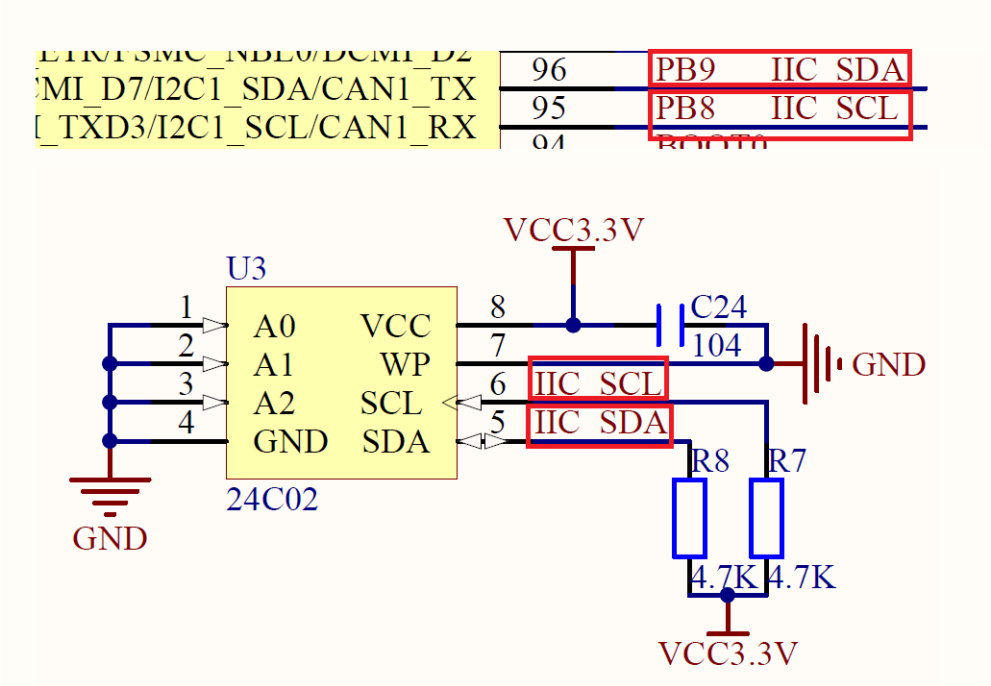



图 3.45 24C02 与 STM32F407 连接原理图

#### 3） 软甲设计

打开示例程序的 main.c 文件，内容如下：

```
int main(void)
{
    u8 i;
    u8 key_value;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);                //初始化延时函数
    USART1_Init(115200);            //串口 1 初始化
    LED_Init();                     //初始化 LED
```

```
    LCD_Init();                         //液晶屏初始化
    KEY_Init();                        //按键初始化
    AT24C02_Init();                    //初始化 IIC
    IIC_24C02_test();                  //IIC test
    //循环测试
    while(1)
    {
        key_value=KEY_Scan(0);
        if(key_value==KEY_UP_PRES)   //KEY_UP 按下
        {
            Show_24C02_Write();          //写入 24C02
        }
        else if(key_value==KEY0_PRES) //KEY0 按下
        {
            Show_24C02_Read();              //读取 24C02
        }
        if(i%20==0)
        {
            LED0=!LED0;
            i=0;
        }
        i++;
        delay_ms(10);//延时
    }
}
```

代码中 AT24C02_Init 为 24C02 初始化函数，IIC_24C02_test 为 IIC 测定内容显示函数（显示到 LCD 上），Show_24C02_Write 为 24C02 内容写入函数，Show_24C02_Read 为 24C02 内如读取函数。

IIC 相关的函数定义在 iic.c 文件里，本示例采用 IO 口模拟 IIC 时序，并没有采用硬件 IIC。这里大致说明一下，大家可以到示例程序里查看：

IIC_Init 为 IIC 引脚初始化函数，初始化 PB8 和 PB9 引脚，值得注意的是，IIC 引脚初始化完成后，需要拉高，等待 IIC 开始。

IIC_Start 为 IIC 开始函数，当时钟引脚为高电平时，数据引脚由高到低，IIC 开始。

IIC_Stop 为 IIC 停止函数，当时钟引脚为高电平时，数据引脚由低到高，IIC 停止。

IIC_Wait_Ack 为等待从机应答函数，先将时钟信号和数据信号拉高，然后读取数据信号电平值，在规定的时间内，如果读取到低电平值，则表示等待应答成

功，否则失败。

IIC_Ack 为主机产生应答信号函数，将数据引脚拉低，发送出去，就表示发送了应答信号。

IIC_NAck 为主机不发送应答信号函数，将数据引脚拉高，发送出去，就表示不发送应答信号。

IIC_Send_Byte 为 IIC 发送 1 个字节数据函数。

IIC_Read_Byte 为 IIC 读取 1 个字节数据函数，可以选择是否需要发送应答信号。

24C02 相关的函数定义在 24c02.c 文件里，这里大致说明一下，大家可以到示例程序里查看：

AT24C02_Init 为初始化函数，实际是调用 IIC_Init 对 IIC 进行初始化。

AT24C02_ReadOneByte 为从 24C02 的指定地址读取一个字节数据的函数。

AT24C02_WriteOneByte 为写入一个字节数据到 24C02 指定地址的函数。

AT24C02_WriteLenByte 为写入多个字节数据到 24C02 指定地址的函数。

AT24C02_ReadLenByte 为从 24C02 的指定地址连续读取 4 个字节数据的函数。

AT24C02_Check 检查 24C02 硬件是否正常的函数。

AT24C02_Read 用于连续从 24C02 指定地址读取指定长度的数据。

AT24C02_Write 用于连续往 24C02 指定地址写入指定长度的数据。

**4） 示例效果**

本示例运行成功后，可以看到 LED0 闪烁，按下 KEY_UP 按键，写入数据到 24C02，按下 KEY0 按键，从 24C02 读取数据，并在 LCD 上显示读取的内容。如图 3.46 所示：


图 3.46 IIC 测试示例显示界面

### 3.4.15 SPI 示例

#### 1） 示例目的

让大家掌握使用 STM32F407 硬件 SPI 对 SPI FLASH（W25Q64）进行读写，并将结果显示在 LCD 上。

#### 2） 硬件说明

本示例需要用到 LED0、KEY0、KEY_UP、串口 1、LCD 和 W25Q64 SPI FLASH。前面几个硬件已经介绍过，这里只介绍 W25Q64 SPI FLASH。W25Q64 SPI FLASH 接在主芯片的 SPI1 上，其中 SPI 时钟引脚接在 PB3、主机写数据引脚接在 PB5 上，主机读数据引脚接在 PB4 上，W25Q64 片选引脚接在 PB14 上如图 3.47 所示：

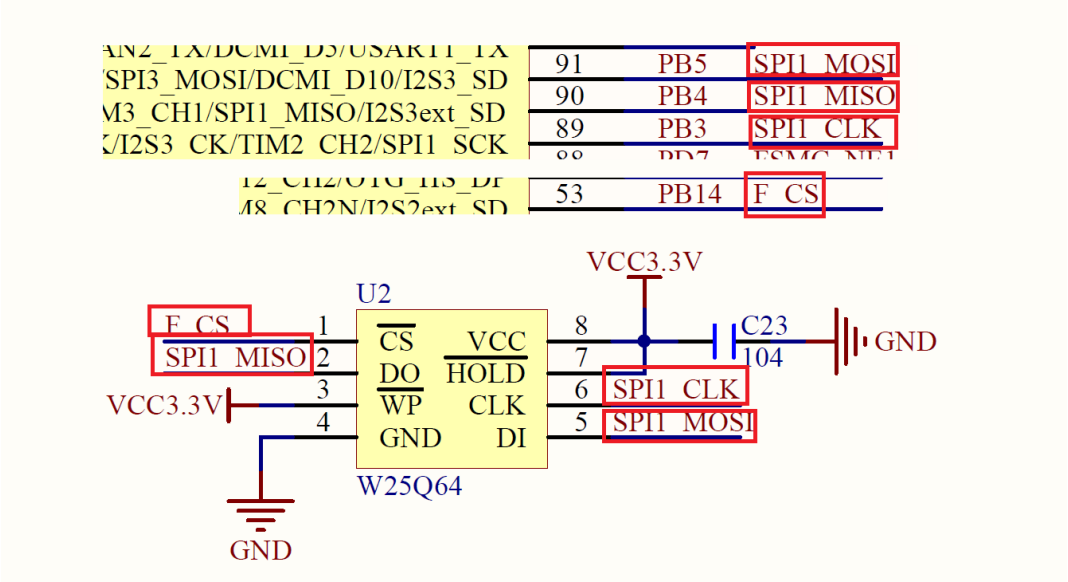



图 3.47 W25Q64 与 STM32F407 连接原理图

### 3） 软件设计

打开示例程序 main.c 文件，内容如下：

```
int main(void)
{
    u8 i;
    u8 key_value;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级
分组 2
    delay_init(168);                //初始化延时函数
    USART1_Init(115200);             //串口 1 初始化
    LED_Init();                      //初始化 LED
    LCD_Init();                        //液晶屏初始化
    KEY_Init();                      //按键初始化
    W25Q64_Init();                    //W25Q64 初始化
    SPI_W25Q64_test();                //W25Q64 测试
    //循环测试
    while(1)
    {
        key_value=KEY_Scan(0);
        if(key_value==KEY_UP_PRES)   //KEY_UP 按下
        {
            Show_W25Q64_Write();          //写入 24C02
        }
        else if(key_value==KEY0_PRES) //KEY0 按下
        {
            Show_W25Q64_Read();           //读取 24C02
        }
```

```
        if(i%20==0)
        {
            LED0=!LED0;
            i=0;
        }
        i++;
        delay_ms(10);//延时 100ms
    }
}
```

其中 W25Q64_Init 为 W25Q64 初始化函数，SPI_W25Q64_test 为 SPI 测试示例内容显示函数（显示在 LCD 上），Show_W25Q64_Write 为显示 W25Q64 写入内容的函数，Show_W25Q64_Read 为显示 W25Q64 读取内容的函数。

SPI 相关的函数都定义在 spi.c 文件里面，这里大致说明一下，具体代码大家可以到示例程序里查看。

SPI1_Init 为 SPI1 初始化函数，初始化过程如下：

A、 使能 GPIO 时钟和 SPI1 时钟；

B、 初始化 SPI1 连接的 GPIO（PB3、4、5），注意要将 GPIO 设置为复用功能；

C、 将 SPI1 停止复位；

D、 将 SPI1 设置为双线双线全双工模式；

E、 将 SPI1 设置为主 SPI 工作模式；

F、 将 SPI1 的接收或者发送数据大小设为 8 位；

G、 将 SPI1 的时钟极性设为空闲状态为高电平，第二个跳变沿数据报采集；

H、 将 SPI1 的 NSS 设为软件管理；

I、 将 SPI1 的速率设为 256、数据传输从高位开始，CRC 值设为 7；

J、 调用 SPI_Init 函数设置上述参数，调用 SPI_Cmd 函数使能 SPI1 外设。

SPI1_SetSpeed 函数用于实时设置 SPI1 的传输速率。

SPI1_ReadWriteByte 函数用于同步发送和读取数据。

W25Q64 相关的函数定义在 w25q64.c 文件里面，这里大致说明一下，具体代码大家可以到示例程序里去查看。

W25Q64_Init 为 W25Q64 初始化函数，主要初始化片选引脚，初始化 SPI1、设置 SPI1 读写速率，读取 SPI FLASH 的 ID。

W25Q64_ReadSR 为读取 W25Q64 的状态寄存器函数，寄存器的位定义大家可以查看 W25Q64 的数据手册。

W25Q64_Write_SR 为写 W25Q64 的状态寄存器函数。

W25Q64_Write_Enable 为 W25Q64 写使能函数。

W25Q64_Write_Disable 为 W25Q64 写禁止函数。

W25QXX_ReadID 为读取 W25Q64 ID 的函数。

W25Q64_Read 为在 W25Q64 指定地址读取指定长度数据的函数。

W25Q64_Write_Page 为写入一页数据（小于 256 字节）到 W25Q64 的函数。

W25Q64_Write_NoCheck 为不校验写入数据到到 W25Q64 的函数。

W25Q64_Write 为带有检验功能写入数据到到 W25Q64 的函数。

W25Q64_Erase_Chip 为擦除整个 W25Q64 的函数（运行时间较久）。

W25Q64_Erase_Sector 为擦除 W25Q64 一个扇区的函数。

W25Q64_Wait_Busy 为 W25Q64 等待空闲函数。

W25Q64_PowerDown 为 W25Q64 进入待机模式函数。

W25Q64_WAKEUP 为 W25Q64 待机唤醒函数。

**4） 示例效果**

本示例运行成功后，可以看到 LED0 闪烁，按下 KEY_UP 按键，写入数据到 W25Q64，按下 KEY0 按键，从 W25Q64 读取数据，并在 LCD 上显示读取的内容。如图 3.48 所示：


图 3.48 SPI 测试示例显示界面

### 3.4.16 触摸屏示例

#### 1） 示例目的

让大家掌握使用 STM32F407 来驱动触摸屏（电阻触摸屏和电容触摸屏），实现一个手写画板功能。

#### 2） 硬件说明

本示例要用到 LED0、KEY0、串口 1、24C02 EEPROM、LCD、电阻触摸屏（XPT2046）、电容触摸屏（GT911 和 FT5426），这里只介绍触摸屏，其他硬件已经介绍过了。电阻触摸屏采用 SPI 通信，有 5 个引脚：中断引脚接在主芯片 PB1 上，SPI 读引脚接在 PB2 上，SPI 写引脚接在 PC4 上，片选引脚接在 PC13 上，SPI 时钟引脚接在 PB0 上。电容触摸屏采用 IIC 通信，有 4 个引脚：中断引脚接在主芯片 PB1 上，IIC 数据引脚接在 PC4 上，复位引脚接在 PC13 上，IIC 时钟引脚接在 PB0 上。如图 3.49 所示：

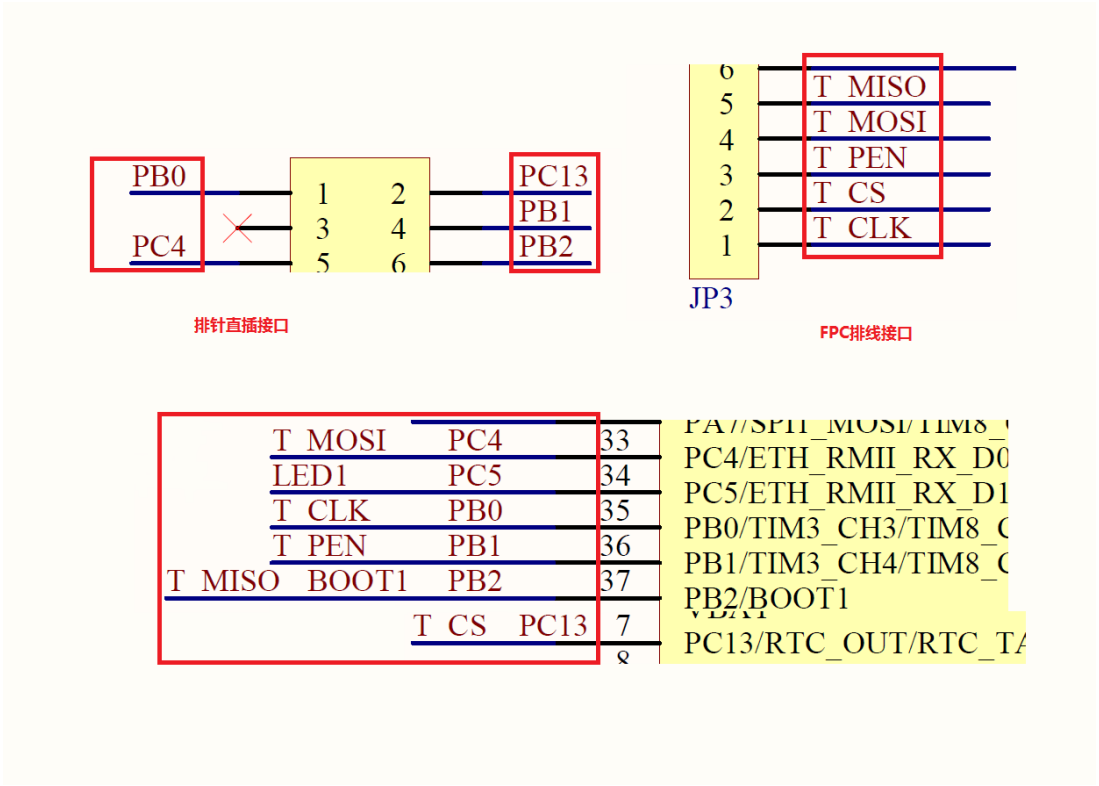



图 3.49 触摸屏和 STM32F407 连接原理图

#### 3） 软件设计

打开示例程序的 main.c 文件，内容如下：

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);                //初始化延时函数
    USART1_Init(115200);            //串口 1 初始化
    LED_Init();                     //初始化 LED
    LCD_Init();                     //液晶屏初始化
    KEY_Init();                     //按键初始化
    AT24C02_Init();                 //初始化 IIC
    TP_Init();                      //触摸屏初始化
  //循环测试
    while(1)
    {
        Touch_Test();   //触摸屏测试
    }
}
```

代码中 TP_Init 为触摸屏初始化函数，Touch_Test 为触摸屏测试函数。

TP_Init 定义在 touch.c 文件里，这里大致说明一下它的内容，大家可以到示例程序里查看具体代码。它首先根据 **TP_TYPE** 宏定义判断是初始化电阻触摸屏还是电容触摸屏，TP_TYPE 在 touch.h 中定义，说明如下：

```
#define TP_TYPE 0     //0-电阻触摸屏，1-电容触摸屏
```

当 TP_TYPE 为 0 时，就执行 RTP_Init 初始化电阻触摸，电阻触摸相关的函数都定义在 rtp.c 中，我们来带着讲解一下其中的内容：

RTP_Write_Byte 为电阻触摸数据写入函数，写入一个字节的数据到电阻触摸 IC（XPT2046）。

RTP_Read_AD 为电阻触摸数据读取函数，从电阻触摸 IC（XPT2046）读出 12 位有效的 ADC 值。

RTP_Read_XOY 为读取 X 坐标或者 Y 坐标值函数，采用多次读值去平均值的方法提供精度。

RTP_Read_XY 为读取一个点的坐标值函数（包括 X 坐标和 Y 坐标）。

RTP_Read_XY2 是在 RTP_Read_XY 基础上增加了提高准确度的方法。

RTP_Scan 为电阻触摸屏触摸事件扫描函数，实时检测是否有触摸事件发生。

RTP_Save_Adjdata 为保存校准参数函数，校准参数保存在 24C02 里面。

RTP_Get_Adjdata 为读取校准参数函数，从 24C02 里面读取校准参数。

RTP_Adjust 为电阻屏校准函数，电阻屏初次使用都是要校准的。

RTP_Init 为电阻触摸屏初始化函数，先对 GPIO 进行初始化，然后读取校准标志位，如果已经校准则初始化完成，如果没有校准，则进入校准程序，校准完成后，保存校准参数，重新读取校准参数。

校准流程如图 3.50 所示：

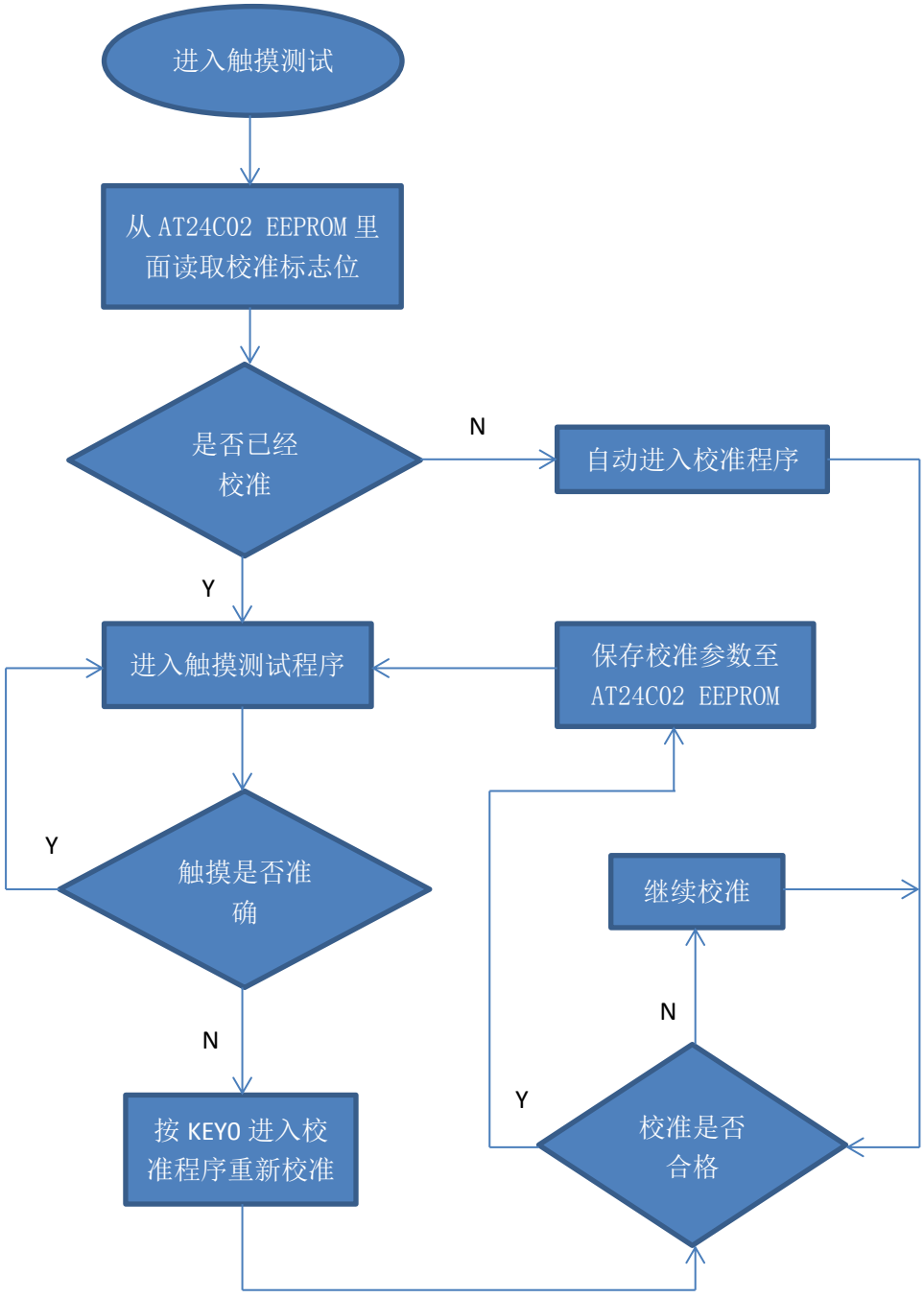



图 3.50 电阻触摸屏校准流程图

电阻触摸屏的校准方法如下：

A、运行本示例代码。

B、按 KEY0 按键进入校准界面（如果没有校准则直接进入校准界面）。

C、依次点击屏幕上的 4 个十字架（十字+小圈圈）的最中央（最好用笔尖）

D、直到屏幕提示："Touch Screen Adjust OK!"字样。

E、校准完成，自动保存校准参数。

当 TP_TYPE 为 1 时，则执行电容触摸屏初始化程序，由于电容触摸 IC 有很多种（本示例只用到 GT911 和 FT5426），不同的 IC 初始化方法和扫描方式都不相同，所以这里采用读取 ID 的方法来区分电容触摸 IC，扫描函数直接赋值给一个函数指针，方便统一调用。还定义了一个数据位来区分电阻触摸屏和电容触摸屏，触摸屏测试函数就是根据此数据位来调用电阻触摸屏和电容触摸屏测试函数的。

电容触摸屏函数定义是根据电容触摸 IC 来分类存放的，如 GT911 电容触摸存放在 GT911.c 里面，FT5426 电容触摸存在 ft5426.c 里面。电容触摸屏的函数定义里主要说明一下四个函数：GT911_Init、GT911_Scan、FT5426_Init、FT5426_Scan，具体代码大家可以到示例程序里查看。

GT911_Init 就是 GT911 电容触摸初始化函数；

GT911_Scan 就是 GT911 电容触摸扫描函数；

FT5426_Init 就是 FT5426 电容触摸初始化函数；

FT5426_Scan 就是 FT5426 电容触摸扫描函数；

另外 touch.h 里面还定义了一个选择触摸屏工作模式的宏定义：

```
#define SCAN_TYPE 0    //0-轮询模式，1-中断模式
```

可以选择轮询还是终端工作模式。

**4） 示例效果**

本示例运行成功后，可以看到 LED0 闪烁，LCD 显示触摸屏测试界面，此时可以在 LCD 上划线，电容触摸同时支持 5 点触摸，显示 5 种颜色的线条；电阻触摸屏只能支持一点触摸，点击相应区域可以改变线条的颜色，点击 RST 区域可以清屏，按 KEY0 键可以进入触摸校准程序。如图 3.51 所示：

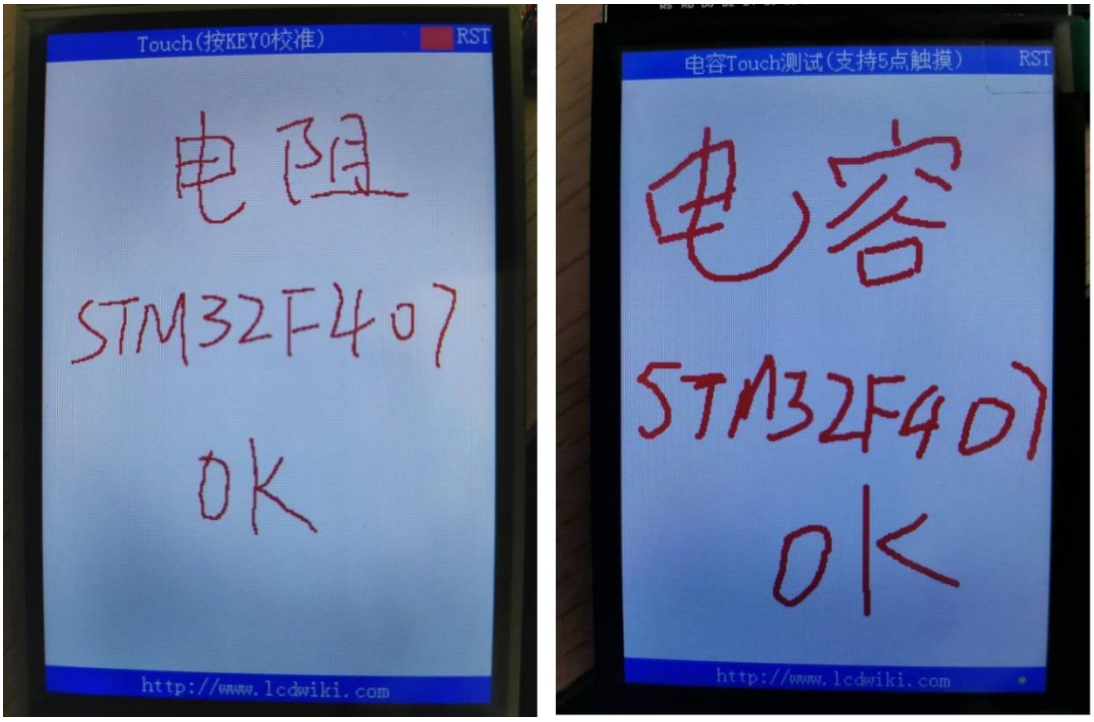



图 3.51 触摸屏测试界面

### 3.4.17 内部 flash 示例

#### 1） 示例目的

让大家掌握 STM32F407 内部 FLASH 的读写方法。

#### 2） 硬件说明

本示例需要用到 LED0、KEY0、KEY_UP、串口 1、LCD 和 STM32F407 内部 FLASH，前面的硬件都已经介绍，这里只介绍 STM32F407 内部 FLASH。STM32F407 内部 FLASH 属于 STM32F407 的内部资源，没有外部电路连接。

#### 3） 软件设计

打开示例程序的 main.c 函数，找到如下内容：

```
int main(void)
{
    u8 i;
    u8 key_value;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);                //初始化延时函数
    USART1_Init(115200);            //串口 1 初始化
    LED_Init();                     //初始化 LED
    LCD_Init();                        //液晶屏初始化
    KEY_Init();                     //按键初始化
```

```
    FLASH_test();                    //FLASH 测试
  //循环测试
    while(1)
    {
        key_value=KEY_Scan(0);
        if(key_value==KEY_UP_PRES)   //KEY_UP 按下
        {
            Show_FLASH_Write();          //写入 flash
        }
        else if(key_value==KEY0_PRES) //KEY0 按下
        {
            Show_FLASH_Read();           //读取 flash
        }
        if(i%20==0)
        {
            LED0=!LED0;
            i=0;
        }
        i++;
        delay_ms(10);//延时 100ms
    }
}
```

其中 FLASH_test 为示例内容显示函数，Show_FLASH_Write 为内部 flash 写入显示函数，Show_FLASH_Read 为内部 flash 读取显示函数。

内部 FLASH 相关的函数都定义在 flash.c 文件里面，这里大致说明一下，具体代码大家可以到示例程序里查看。

FLASH_ReadWord 为从内部 flash 指定地址读取半字（16bit）函数。

FLASH_GetFlashSector 为获取内部 flash 某个地址所在的 flash 扇区函数。

FLASH_Write 为写入指定长度的数据到内部 flash 的指定地址的函数，注意必须写到用户代码区以外的地址，写入的地址必须是 4 的整数倍。

FLASH_Read 为从内部 flash 的指定地址读取指定长度数据的函数。

**4） 示例效果**

本示例运行成功后，可以看到 LED0 闪烁，按下 KEY_UP 按键，写入数据到内部 flash，按下 KEY0 按键，从内部 flash 读取数据，并在 LCD 上显示读取的内容。如图 3.52 所示：

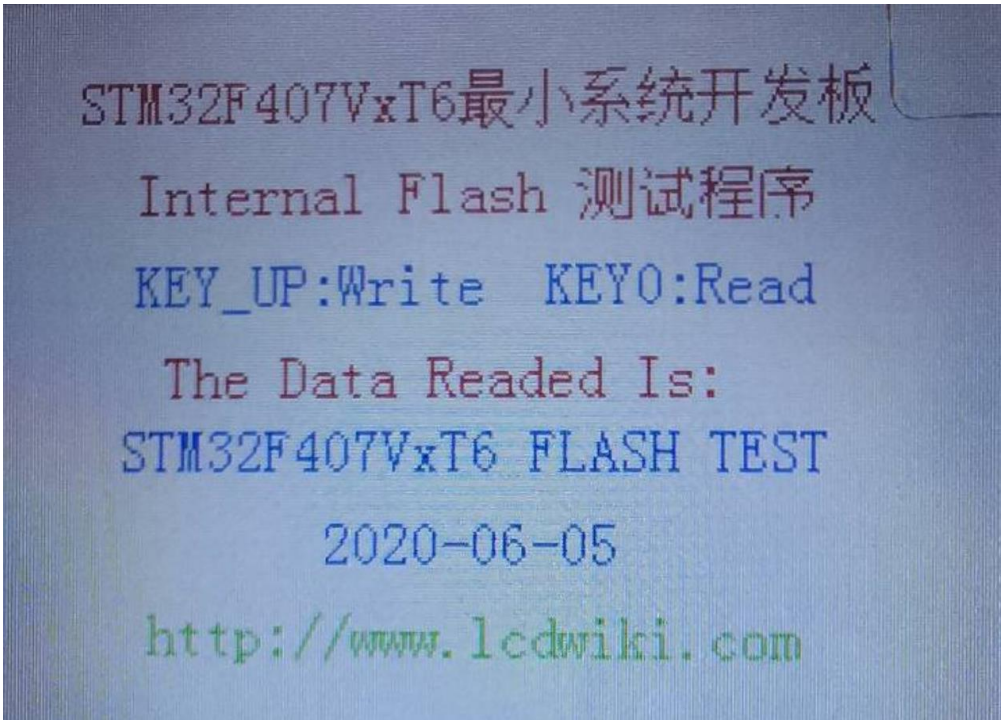



图 3.52 内部 flash 示例显示界面

### 3.4.18 SD 卡示例

#### 1） 示例目的

让大家掌握使用 STM32F407 从 SD 卡中读取图片并解码，然后在 LCD 上显示出来。

#### 2） 硬件说明

本示例要用到 LED0、KEY0、KEY_UP、串口 1、LCD 和 Micro SD 卡（插入 SD 卡槽使用），前面的硬件已经介绍，这里只介绍 SD 卡槽。SD 卡槽的 SD_D0~D3 分别接在主芯片的 PC8~11 引脚上，SD_CLK 接在主芯片的 PC12 引脚上，SD_CMD 接在主芯片的 PD2 上，如图 3.53 所示：

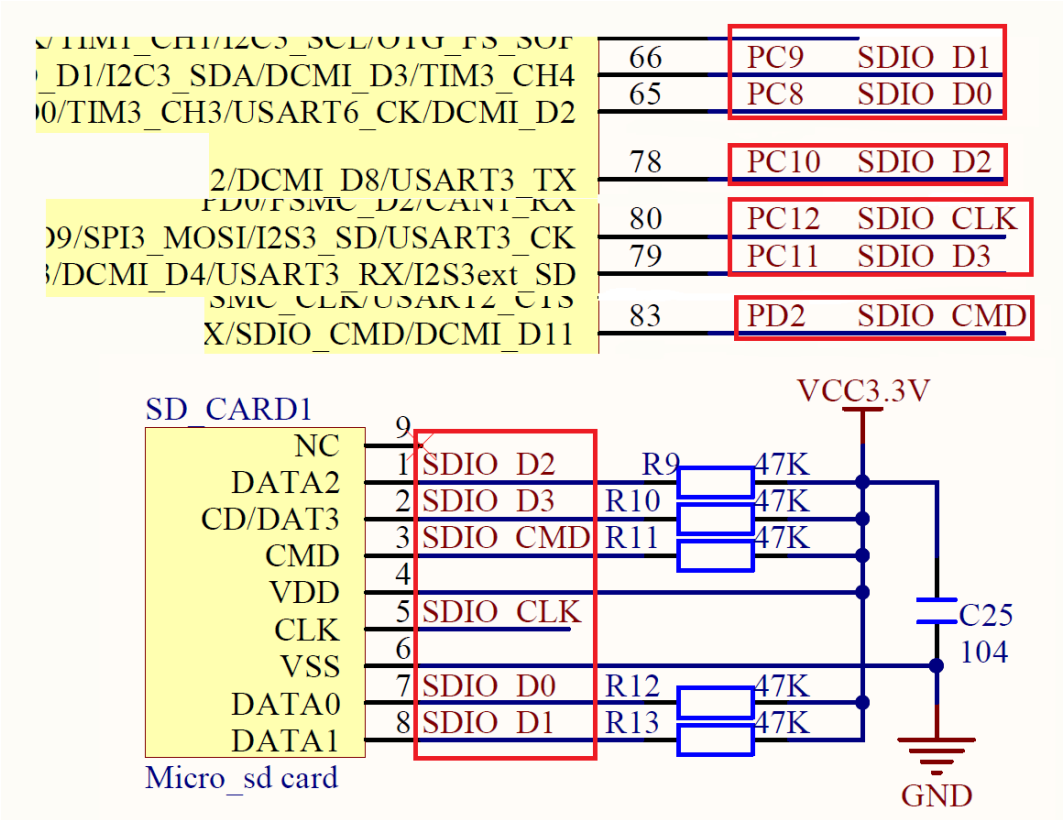



图 3.53 SD 卡槽和 STM32F407 连接原理图

### 3) 软件设计

打开示例程序的 main.c 函数，内容如下：

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断优先级分组 2
    delay_init(168);  //初始化延时函数
    USART1_Init(115200);        //初始化串口波特率为 115200
    LED_Init();                 //初始化 LED
    LCD_Init();                 //LCD 初始化
    KEY_Init();                 //按键初始化
    W25Q64_Init();              //初始化 W25Q64
    Check_SDCard();          //检测 SD 卡
    my_mem_init(SRAMIN);      //初始化内部内存池
    my_mem_init(SRAMCCM);       //初始化 CCM 内存池
    exfuns_init();            //为 fatfs 相关变量申请内存
  f_mount(fs[0],"0:",1);      //挂载 SD 卡
    f_mount(fs[1],"1:",1);    //挂载 FLASH.
    Check_Font();              //检查 W25Q64 中字库是否存在
    SDCard_test();            //显示 SD 卡测试内容
    Show_Picture();           //显示 SD 中的图片
}
```

从代码来看，首先将需要用到的硬件初始化，然后检测 SD 卡是否存在，如果不存在，则不停的显示提示，直到插入 SD 卡，当 SD 卡存在后，则初始化内外部内存池，并为 fatfs 相关变量申请内存，接下来挂载 SD 和 FLASH，然后检查 W25Q64 里面字库是否存在，如果不存在则需要按下 KEY0 键（长按 2 秒左右）更新，字库已经存在，则显示测试内容界面，然后读取 SD 卡根目录下的 PICTURE 文件夹，如果文件夹不存在就报错，如果存在就读取文件夹里的图片，如果图片不存在就提示，如果存在就显示到 LCD 上。

**下面几个方面需要注意：**

A、字库更新方法

找到测试示例文件夹下的“SD 卡测试文件”目录，将里面的“FONT”文件夹拷贝到 SD 卡的根目录下，然后将 SD 卡插入到最小系统板，运行测试程序，当程序运行到提示按 KEY0 键更新字库时，按下 KEY0 键（长按大概 2 秒左右），进入字库更新阶段。

B、图片更新方法

可以将测试示例文件夹下的“SD 卡测试文件”目录里的“PICTURE”文件夹拷贝到SD卡的根目录下，直接使用里面的图片，或者自己再SD卡更目录新建“PICTURE”文件夹，将自己准备的图片放进去（支持 BMP/JPG/JPEG/GIF 等格式的图片）。

C、如果有些 jpg/jpeg 格式不能显示的话，用 Windows 自带的画图工具打开，再保存一下，一般就可以显示了。

D、JPEG/JPG/BMP 图片，程序会根据 LCD 分辨率自动缩放，以适应 LCD，而 GIF 则必须小于屏幕分辨率才可以显示。

本示例代码比较庞大，大致介绍一下各个目录下代码的内容，大家可以到示例代码中查看相关的代码。

test.c 文件里面定义的是功能测试函数，在 main.c 中被调用，其中：

Check_SDCard 为检查 SD 卡是否插入的函数。

Check_Font 为检查 W25Q64 里字库是否存在的函数，如果不存在则需要更新。字库更新方法，上面注意事项里有介绍。

SDCard_test 为显示 SD 卡测试示例界面的函数。

pic_get_tnum 为获取图片数目的函数，如果没有图片，则需要更新。图片更新方法，上面注意事项里有介绍。

Show_Picture 为将图片显示到 LCD 上的函数。

示例程序里各个功能模块代码分目录存放，下面来介绍一下各个目录存放的代码。

FATFS 目录存放的是 FAT 文件系统代码，支持文件读写功能。

HARDWARE 目录下只介绍 SDIO 目录，它存放的是 SDIO 总线代码，专门用于 SD 卡操作。

MALLOC 目录存放的是处理内存申请的代码，用于申请内存和释放内存。

PICTURE 目录存放的是图片解码代码，包括 bmp、gif、jpg 等图片解码。

TEXT 目录存放的是字符显示代码，包括升级字库，将字符显示到 LCD 上等功能。

**4） 示例效果**

本示例运行成功后，可以看到 LED0 闪烁，然后检测 SD 卡是否存在，如果不存在，则提示 SD 卡错误，如果存在则提示 SD 卡 OK，接下里检测字库是否正常，如果不正常，提示按 KEY0 更新字库，字库更新成功或者字库本来就正常，则去 SD 卡根目录下寻找 PICTURE 文件夹，如果没找到则提示 PICTURE 文件夹错误，如果找到了，则到 PICTURE 文件夹下寻找图片，如果没找到图片则提示没有图片文件，如果找到图片文件，则开始显示图片，按 KEY0 可以快速浏览下一张，按 KEY_UP 可以快速返回上一张。如图 3.54 和图 3.55 所示。


图 3.54 SD 测试示例显示界面

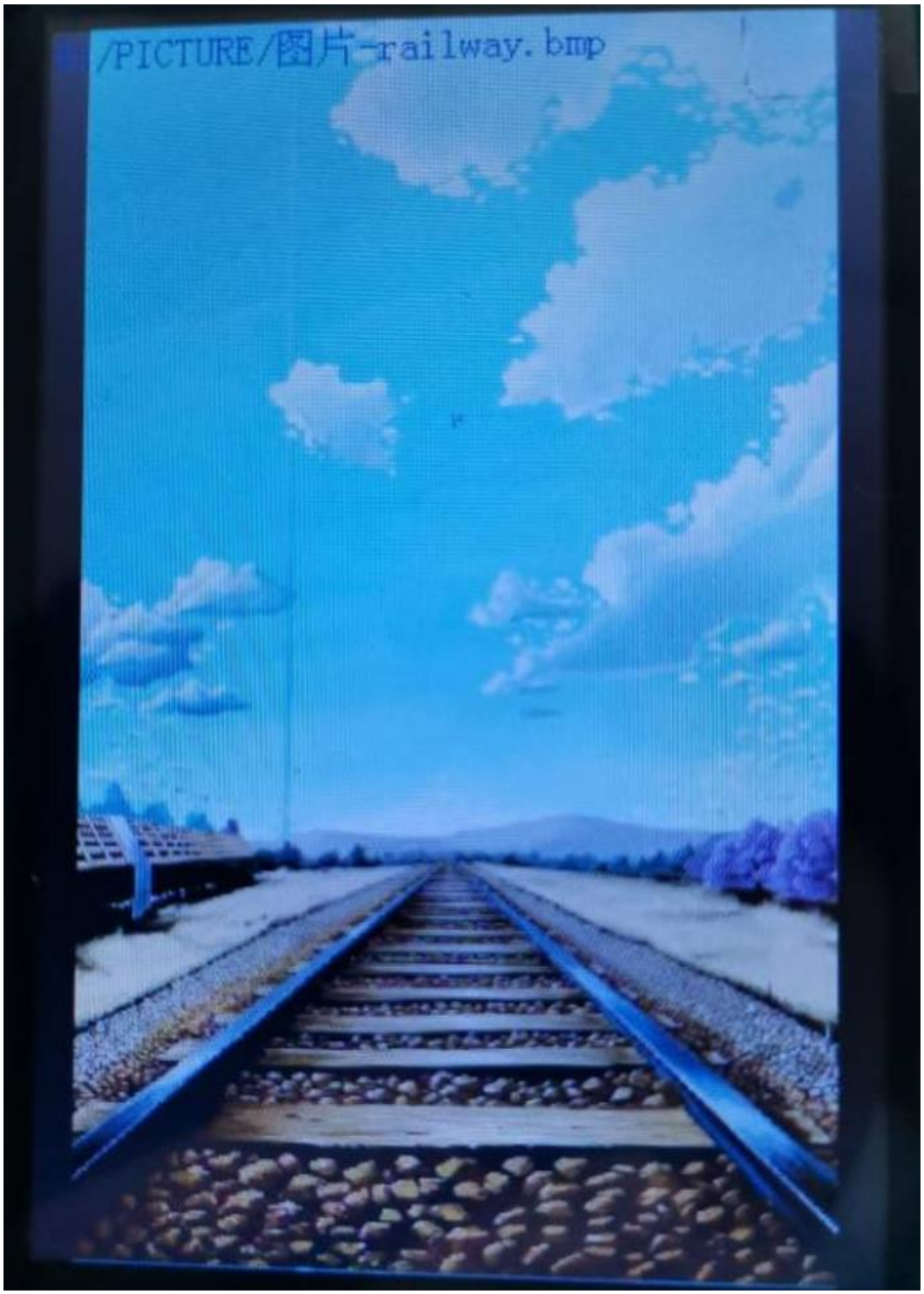



图 3.55 SD 卡测试示例图片显示效果

### 3.4.19 USB 从设备示例

#### 1） 示例目的

让大家熟悉使用 USB SLAVE 功能，实现通过 USB 通信读取板上 SPI Flash 和 SD 卡的内容同时也可以往 FLASH 和 SD 卡里写内容（和 USB 读卡器功能类似）。

2） 硬件说明

本示例需要用到 Micro SD 卡、SD 卡槽、W25Q64、LED0、LED1、KEY0、KEY_UP 、串口 1、LCD 和 Micro USB 接口，前面几个硬件都已经介绍了，这里只介绍 Micro USB 接口。Micro USB 接口既可以用于 USB 通信，也可以用于供电，其中 USB_D+接在主芯片的 PA12 引脚上，USB_D-接在主芯片的 PA11 引脚上，如图 3.56 所示：

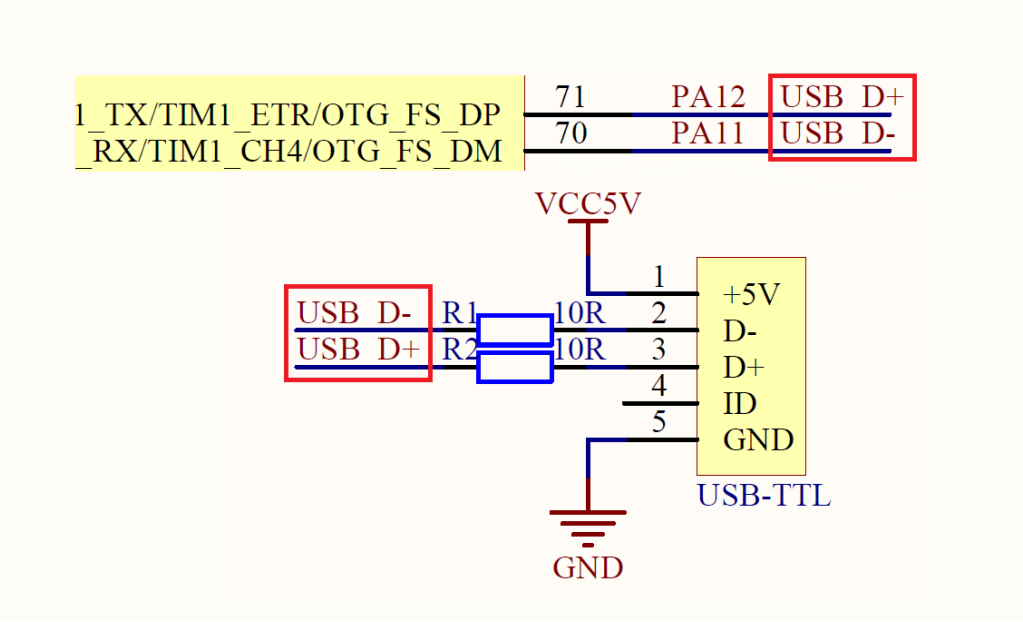



图 3.56 USB 接口和 STM32F407 连接原理图

3） 软件设计

打开示例程序的 main.c 函数，内容如下：

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);        //初始化延时函数
    USART1_Init(115200); //串口 1 初始化
    LED_Init();              //初始化 LED
    LCD_Init();                 //液晶屏初始化
    W25Q64_Init();                 //初始化 W25Q64
    USB_Slave_test();      //USB 从设备测试
    Check_SDCard();         //检测 SD 卡
    Check_W25Q64();         //检测 W25Q64
    USB_Slave_Read_Write(); //USB 从设备读写测试
}
```

USB_Slave_test 为显示测试内容的函数，USB_Slave_Read_Write 为从设备读写

函数，这些函数都定义在 test.c 文件里面，大家可以去查看。

USB 相关的代码都定义在 USB 目录下，该套代码都是从官方库移植过来的，大家可以到资料包的学习资料目录里 **USB 学习资料**文件夹下查看。这里简单介绍一下其中的几个文件和注意事项。

usb_bsp.c 提供了几个 USB 库需要用到的底层初始化函数，包括：IO 设置、中断设置、VBUS 配置以及延时函数等，需要我们自己实现。USB Device(Slave)和 USB Host 共用这个.c 文件。

usbd_desc.c 提供了 USB 设备类的描述符，直接决定了 USB 设备的类型、断点、接口、字符串、制造商等重要信息。

usbd_usr.c 提供用户应用层接口函数，即 USB 设备类的一些回调函数，当 USB 状态机处理完不同事务的时候，会调用这些回调函数。

usbd_storage_msd.c 提供一些磁盘操作函数，包括支持的磁盘个数，以及每个磁盘的初始化和读写等函数。

**注意事项：**

A、 如果需要使用 USB_OTG_FS,则需要添加 USE_USB_OTG_FS 宏定义，否则编译出错，按照图 3.57 所示步骤添加：

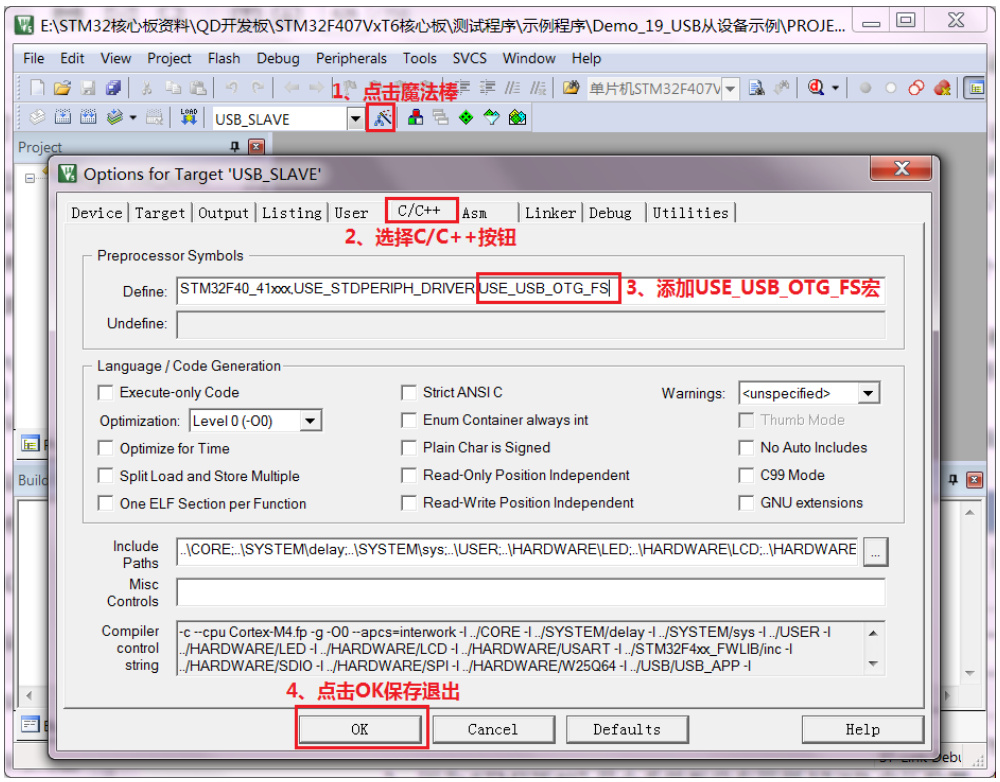



图 3.57 添加 USE_USB_OTG_FS 宏定义

B、 因为 STM32F407 最小系统板没有用到 VUSB 电压检测，所以要在 usb_conf.h 里面，将宏定义：#define VBUS_SENSING_ENABLED，屏蔽掉。

C、 通过修改 usbd_conf.h 里面的 MSC_MEDIA_PACKET 定义值大小，可以一定程度提高 USB 读写速度（越大越快），本例程我们设置 12*1024，也就是 12K

D、 官方例程不支持大于 4G 的 SD 卡，得修改 usbd_msc_scsi.c 里面的 SCSI_blk_addr 类型为 uint64_t，才可以支持大于 4G 的卡，官方默认是 uint32_t，最大只能支持 4G 卡。注意：usbd_msc_scsi.c 文件，是只读的，得先修改属性，去掉只读属性，才可以更改。

**4） 示例效果**

本示例运行成功后，LED0 闪烁，先检测 SD 卡是否插上，再检查 SPI FLASH 是否正常工作，如果检测都正常，则获取 SD 和 SPI Flash 的容量并显示在 LCD 上，如果不正常，则显示出错信息。接下来显示 USB 正在连接，如果连接正常则提示已经连接，此时 PC 机上会显示两个可移动磁盘，LCD 上会显示 SD 卡和 SPI Flash 的读写状态。如图 3.58、图 3.59、图 3.60 所示：


图 3.58 USB 从设备示例显示界面

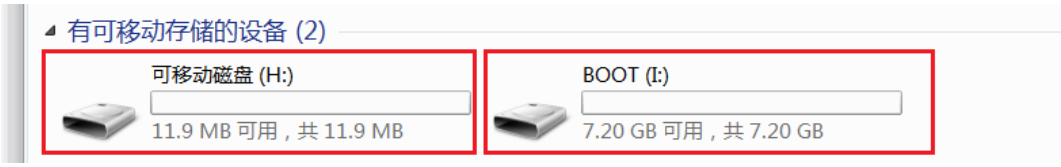



图 3.59 PC 机上显示的两个可移动磁盘

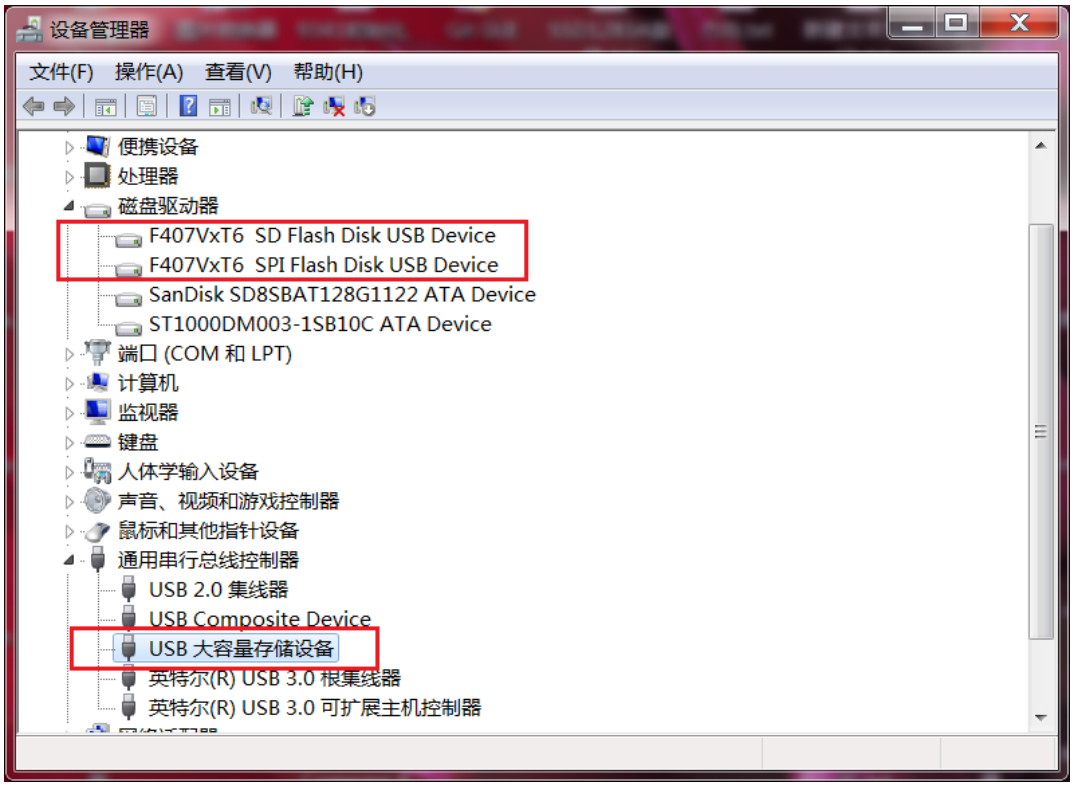



图 3.60 设备管理器界面显示的磁盘驱动器

### 3.4.20 USB 主设备示例

#### 1） 示例目的

让大家熟悉使用 USB HOST 功能，实现通过 USB 通信读取或者写入外挂 U 盘的内容。

#### 2） 硬件说明

本示例需要用到 Micro SD 卡、SD 卡槽、W25Q64、LED0、LED1、KEY0、KEY_UP 、串口 1、LCD、Micro USB 接口和 Micro USB OTG 转接线。前面那些硬件都已经介绍，这里只说明一下 Micro USB OTG 转接线怎么连接 U 盘和最小系统板。如图 3.61 所示：

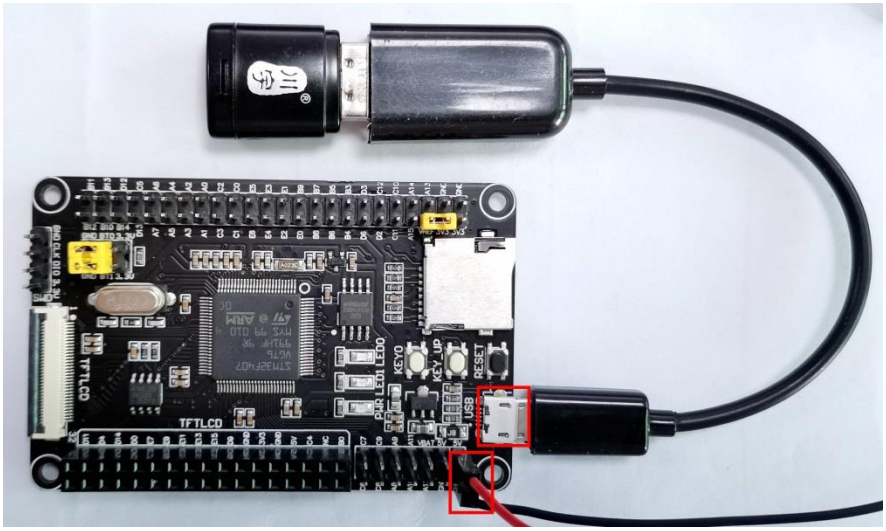


图 3.61 U 盘和 STM32F407 连接

需要注意的是连接了 Micro USB OTG 转接线后，Micro USB 接口就不能给最小系统板供电了，此时需要通过 5V 电源排针引脚供电。

**3） 软件设计**

打开示例程序 main.c 文件，内容如下：

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);        //初始化延时函数
    USART1_Init(115200); //串口 1 初始化
    LED_Init();             //初始化 LED
    KEY_Init();                 //按键初始化
    LCD_Init();               //液晶屏初始化
    W25Q64_Init();              //初始化 W25Q64
    my_mem_init(SRAMIN);//初始化内部内存池
    exfuns_init();            //为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1);    //挂载 SD 卡
    f_mount(fs[1],"1:",1);    //挂载 SD 卡
  f_mount(fs[2],"2:",1);      //挂载 U 盘
    Check_Font(); //检测 W25Q64 中字库是否存在
    USB_Host_test(); //USB Host test
    USB_Host_Process(); //USB Host 处理任务
}
```

其中 USB_Host_test 为 USB 主机示例程序内容显示函数，USB_Host_Process 为 USB 主机任务处理函数，该函数必须在主函数里面，被循环调用，而且调用频率得比较快才行（越快越好），以便及时处理各种事务。注意，USBH_Process 函数仅

在 U 盘识别阶段，需要频繁反复调用，但是当 U 盘被识别后，剩下的操作（U 盘读写），都可以由 USB 中断处理。

USB 相关的代码都定义在 USB 目录下，该套代码是从官方库移植过来的，大家可以到资料包的学习资料目录里 **USB 学习资料**文件夹下查看。这里介绍一个文件：

usbh_usr.c 提供用户应用层接口函数，相比前两章例程。

其他文件和 USB 从设备对应，只不过此示例用的是主设备功能。

**注意事项：**

如果需要使用 USB_OTG_FS,则需要添加 USE_USB_OTG_FS 宏定义，否则编译出错，添加方法在 USB 从设备里面已经介绍。

**4） 示例效果**

本示例运行成功后，先检查 SPI Flash 里面有没有字库，如果没有 LCD 上提示字库错误，需要将字库文件拷贝到 SD 卡里，按 KEY0 键（长按 3 秒）更新字库。U 盘没有连接成功之前，LED0 闪烁，此时不断查询 U 盘是否连接成功，一旦 U 盘连接成功，LED1 闪烁，LCD 显示显示 U 盘总容量和剩余容量。接下来打开串口调试工具，按 KEY0 键可以查看 U 盘根目录下的内容，按 KEY_UP 按键，会在 U 盘根目录下创建一个 USB_Host_test.txt 文件并写入“STM32F407VxT6 最小系统开发板 USB Host test!”内容。此时再按下 KEY0 按键，会发现 U 盘根目录下多了一个 USB_Host_test.txt 文件。如图 3.62、图 3.63、3.64 所示


图 3.62 USB 主设备示例显示界面

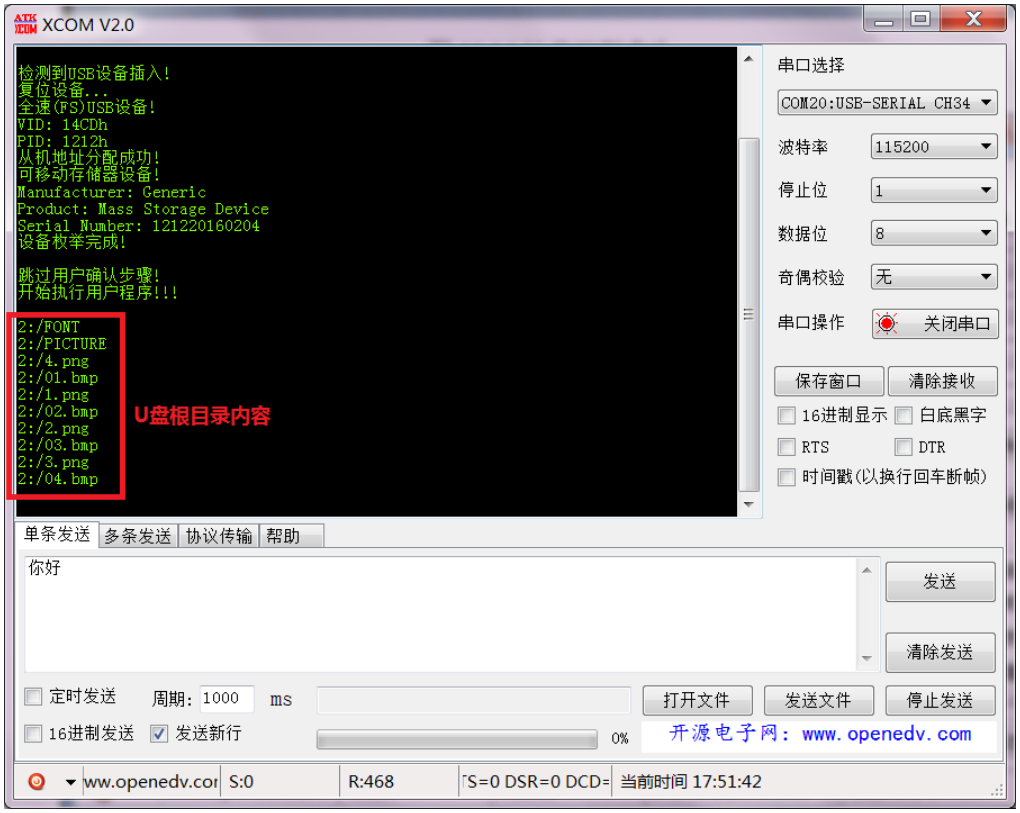



图 3.63 按 KEY0 键在串口终端显示根目录内容

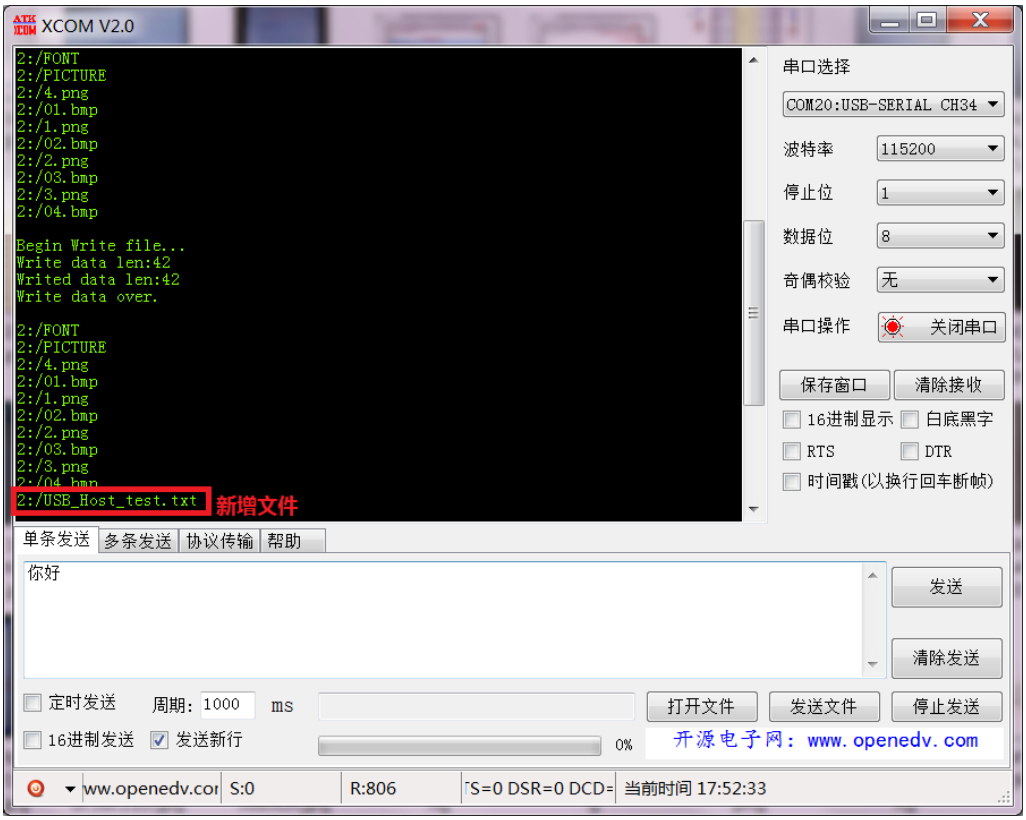



图 3.64 按 KEY_UP 按键新建 USB_Host_test.txt 文件