基于 stm32F407 嵌入式底层驱动开发实践

Release 0.0.1

Mar 07, 2020

关于本项目

本目录文件为 STM32F407 开发板相关教程文档

所有资料发布于百度云与 github，

- 百度云地址：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

  目录 W104-stm32f407

- github: https://github.com/wujique/STM32F407_tech_doc.git

  在 github 上托管了教程文档。其他资料请从百度云下载。

- gitee 镜像：https://gitee.com/hokgaai/STM32F407_tech_doc

- 文档同步发布在：https://stm32f407-tech-doc.readthedocs.io/en/latest/

CHAPTER 1

百度云目录说明

本工程包含代码、资料、文档，目录说明如下。

CHAPTER 2

文档说明

`build` 是 md 转换为 html 的目录。

`source` 是 md 文档目录。其中 `spec` 是产品说明书，`base` 是基础教程，`Advanced` 是提高教程。

本教程文档使用 **markdown** 格式，使用 **Typora** 编写。

所有文档使用 **Sphinx** 管理。

md 文件经过 sphinx 处理后，生成 html 文件。文件入口在 `doc\build\html` 中的 `index.html`，使用浏览器打开即可浏览所有文档。

sphinx 可以将文档处理为 pdf，如有需要，可自行转换。

Typora 也可以将 md 文档转换为 pdf。

> 可参考：
>
> 《使用 ReadtheDocs 托管文档》
>
> https://www.xncoding.com/2017/01/22/fullstack/readthedoc.html

CHAPTER 3

屋脊雀 STM32F407 开发板产品手册

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/12o0Vh4Tv4z_O8qh49JwLjg

2018.11.28

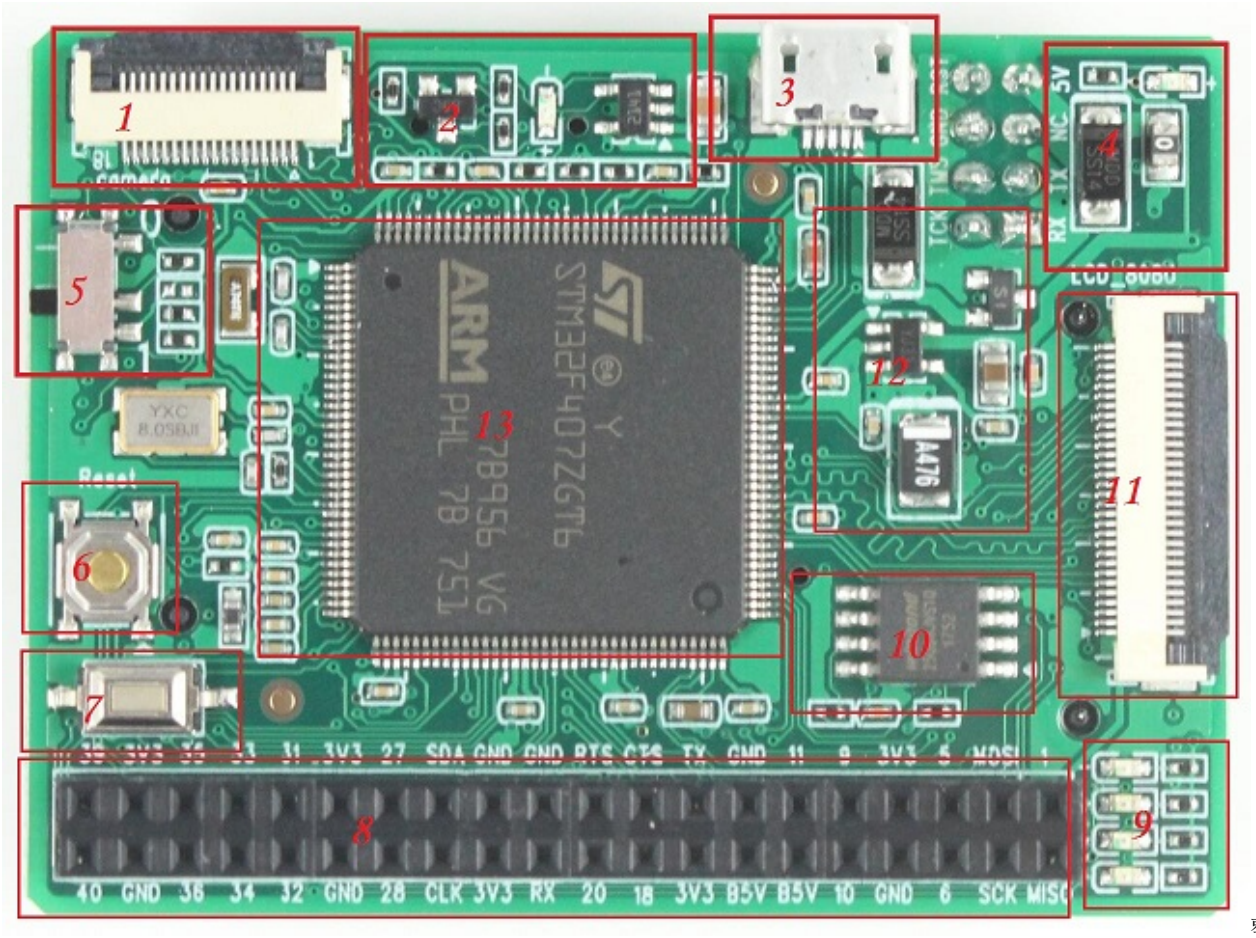
3.1 硬件

3.1.1 整体产品图

使用核心板与底板设计方式，整体长宽只有一张卡片大小。底板按照功能进行模块设计，接口全部按照功能分组引出，用户可按照接口自行开发其他芯片核心板。结构设计精巧，多处进行隐藏性设计。


CAN
HR911105A
TouchPad
STELLAR
NO.17016061009512
星美生活

3.1.2 核心板顶面

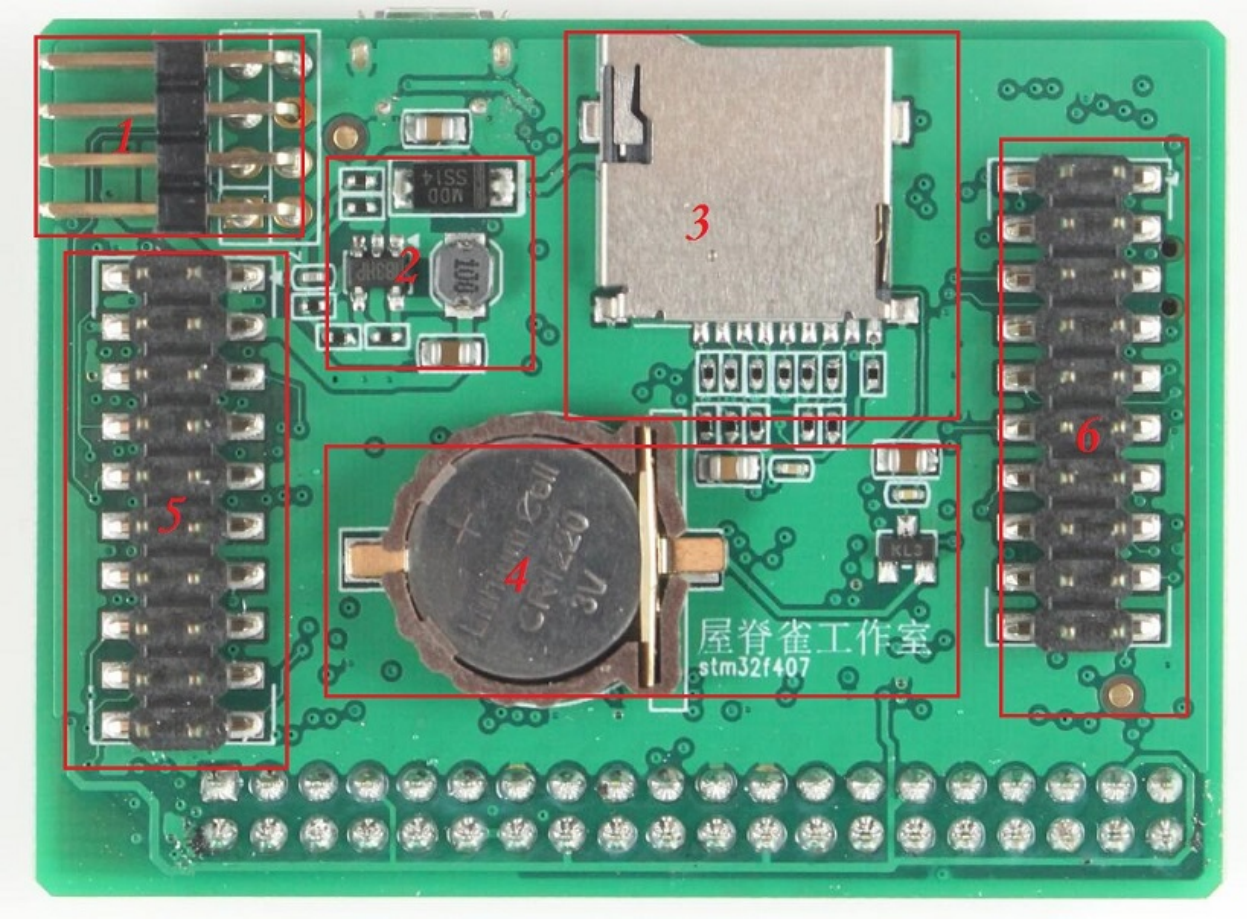


整体

1. 摄像头接口。
2. USB 电源控制电路，实现 HOST 和 DEVICE 两种工作状态切换。
3. usb 接口，做 HOST 时通过 OTG 转接线转接。
4. 调试口电源和电源指示灯。
5. BOOT0 选择。拨到下方 1 状态，进入 ISP 下载模式，也即是系统区启动。拨到上方 0，则从用户 FLASH 控件启动。BOOT1 通过跳线电阻控制，焊接 R105，则连接到高电平；焊接 R103 则连接到低电平。默认焊接 R103，不焊 R105。
6. 复位按键
7. 用户按键
8. 外扩接口
9. LED 灯
10. SPI FLASH

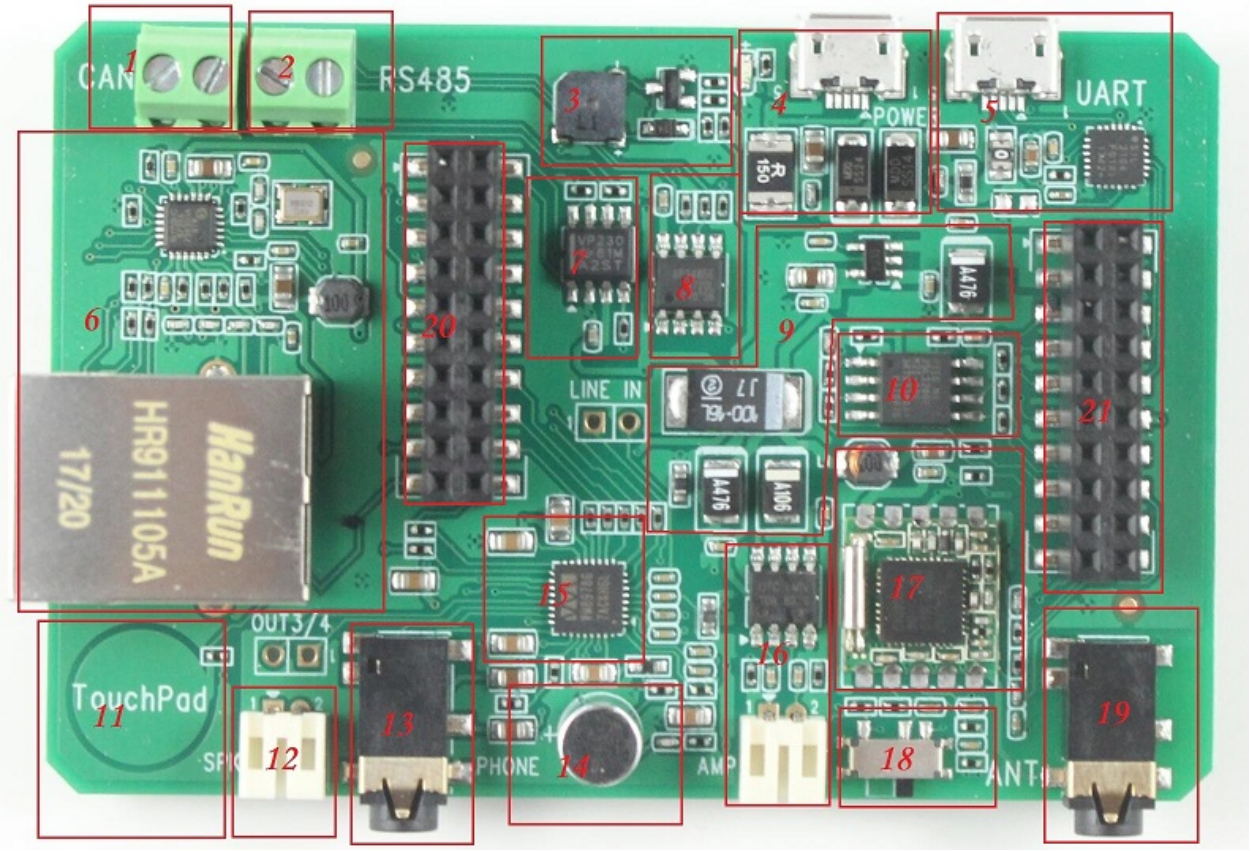
11. LCD 接口，包含触摸屏信号。
12. 电源电路
13. 主芯片

3.1.3 核心板底面



1. 调试口接口，包含 SW 调试信号，串口，电源。
2. 5V 升压电路。
3. TF 卡座。
4. 纽扣电池
5. 与底板接口
6. 与底板接口

3.1.4 底板



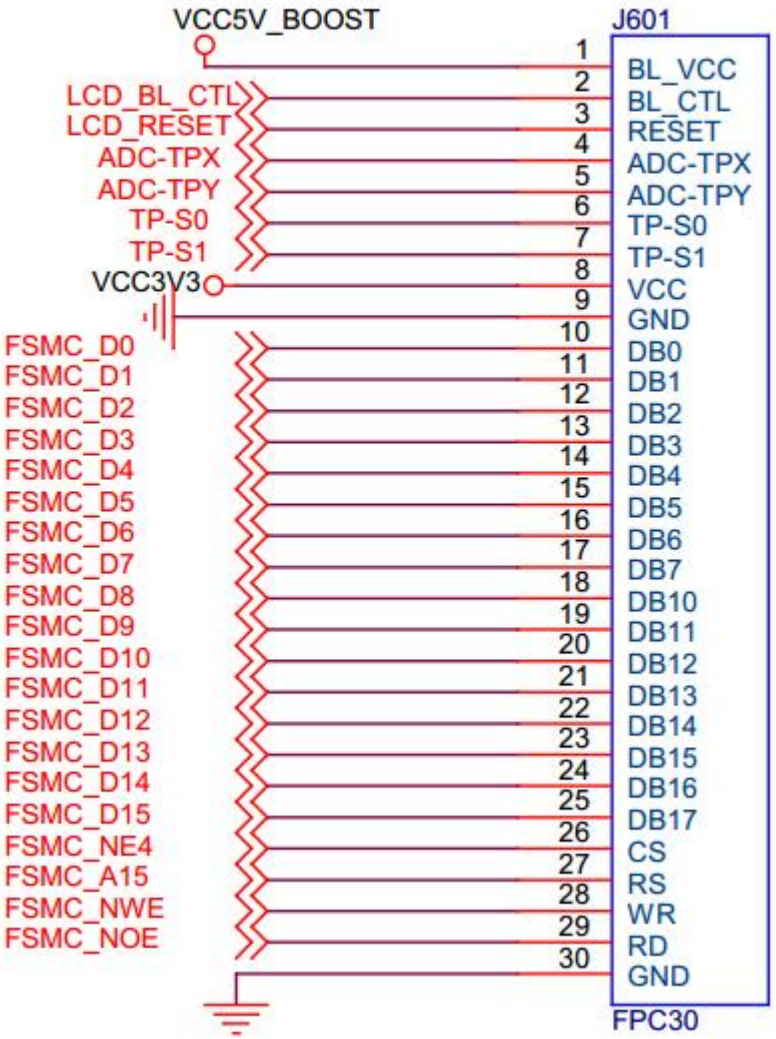
1. CAN 座子，左边是 CANL，右边是 CANH。
2. RS485 座子，左边是 A，右边是 B。
3. 蜂鸣器
4. 电源输入，使用 MICOR USB 接口，保险为 1.5A，可直接用手机充电器供电。
5. USB 转串口，同时可做电源输入，保险为 1A。
6. 网口与网络电路。
7. can 芯片
8. RS485 芯片。
9. 电源电路
10. SPI FLASH
11. 触摸按键
12. 语音芯片喇叭接口
13. 语音芯片耳机接口

14. 语音芯片 MIC
15. 语音芯片 WM8978
16. 功放和喇叭接口。
17. 收音机芯片 TEA5767
18. 功放音源选择，选择 DAC 语音或收音机声音做为输入。
19. 收音机天线座子
20. 与核心板接口
21. 与核心板接口

3.1.5 关键器件规格参数

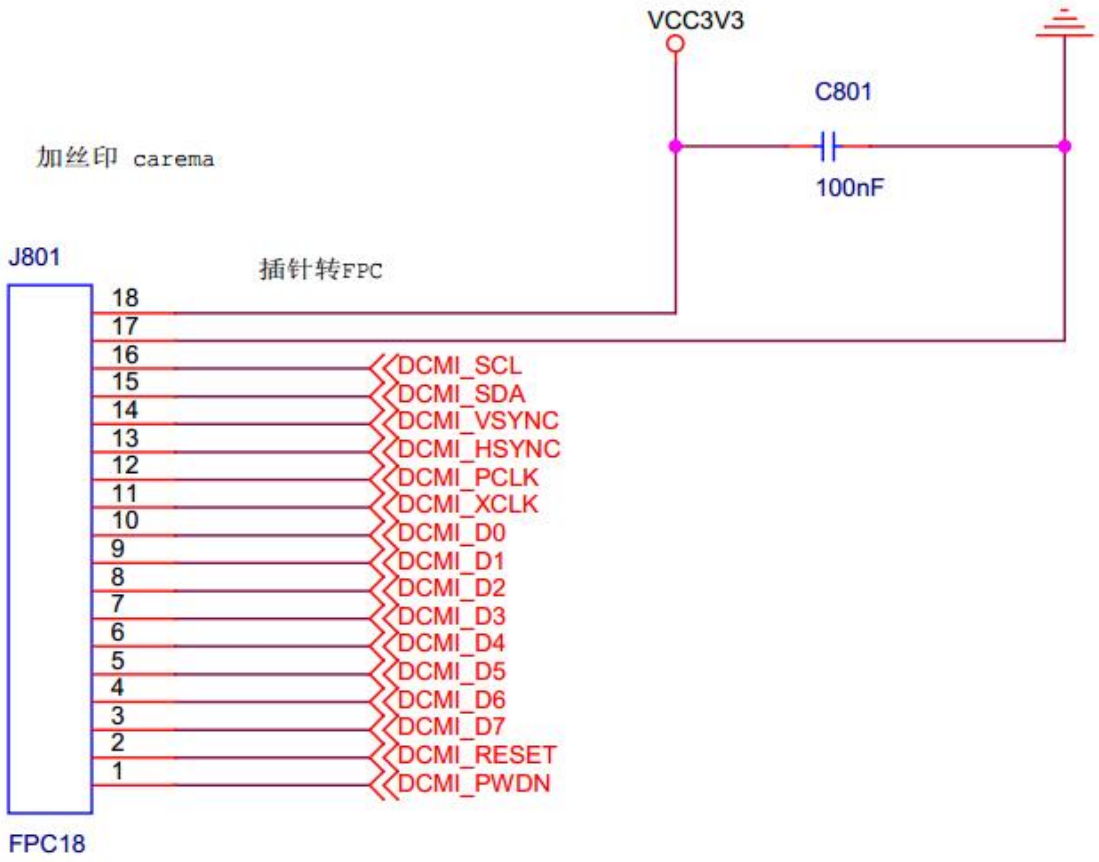
- 核心板
- 底板

3.1.6 接口说明



- 液晶接口 LCD 接口

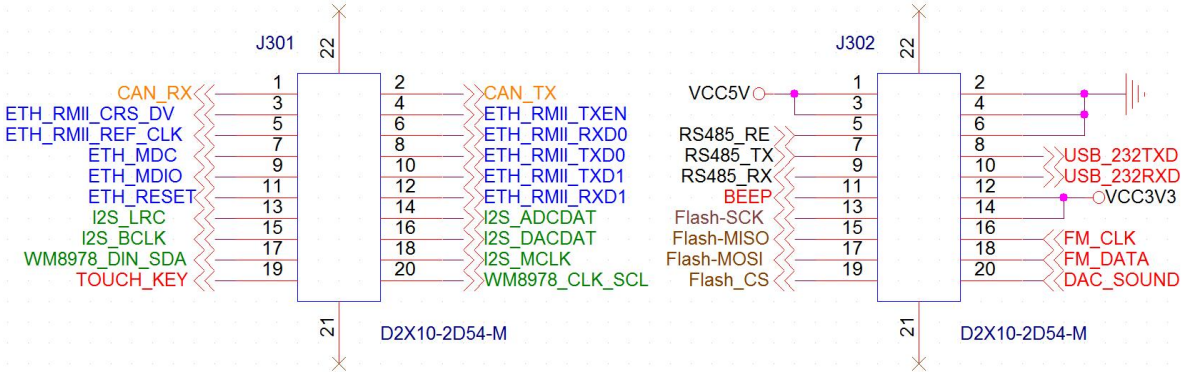
1. LCD 接口为 8080 接口，16 位总线。
2. 有两根电源线，电压分别是 5V 和 3.3V，5V 由升压芯片提供。
3. 包含 4 根触摸屏控制信号，如果使用 XPT2046，则用作模拟 SPI，如果使用 ADC 转换检测触摸屏，则作为 ADC 和 IO.



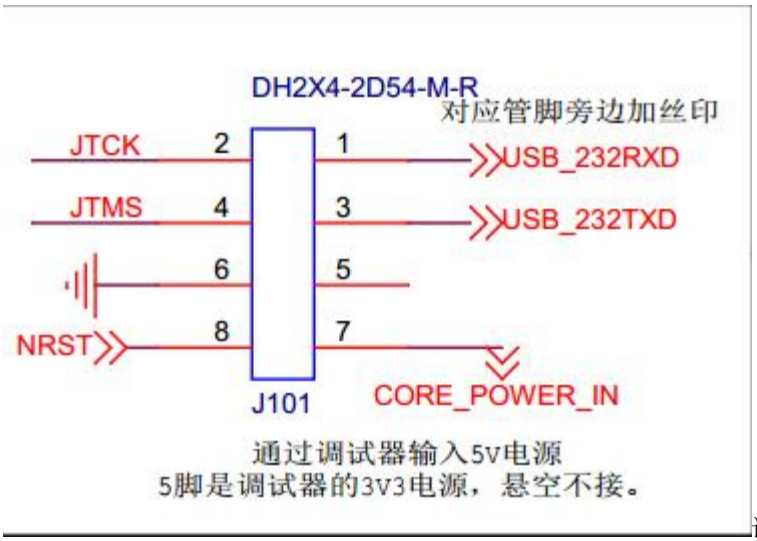
- 摄像头接口 摄像头接口

  标准通用的 DCMI 接口摄像头。

- 核心板接口 下图是核心板的信号接口，实物上排针是放在底面，需要做镜像处理，细节请查看位号图和

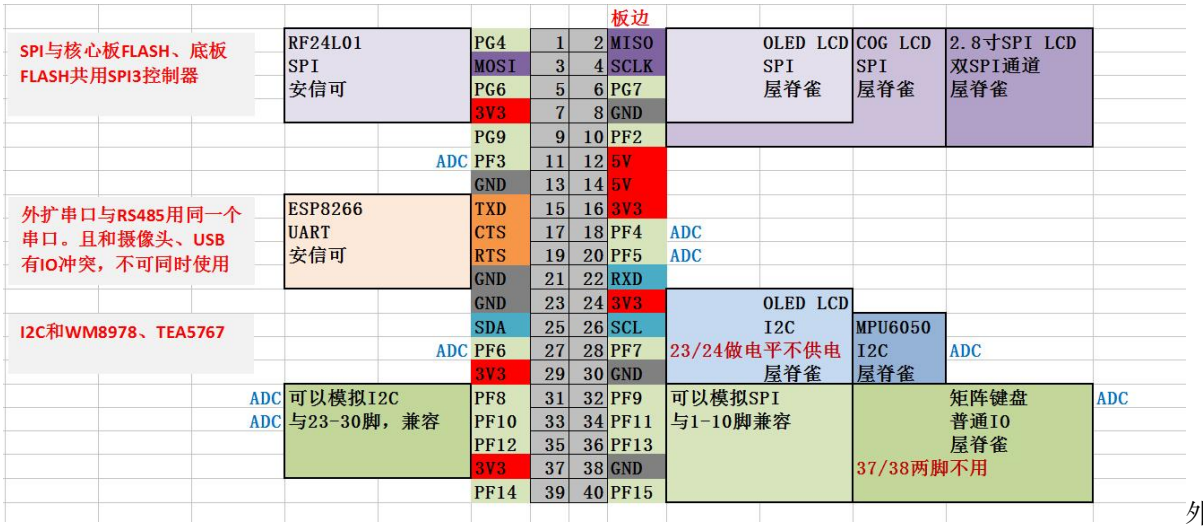


信号说明图 核心板底板接口



- 调试接口 调试接口

1. 包含 SW 调试信号，串口，5V 电源。
2. 兼容屋脊雀设计的 CMISIS DAP、DAPLink。
3. 调试串口同时通过底板经过 CP2104 转换后经 USB 输出。



- 外扩接口 外扩接口 **精心设计**

1. 主要包含 4 个功能区域：SPI、串口、I2C、普通 IO。
2. SPI 区域可以直接接安信可 RF24L01、OLED LCD(spi)、COG LCD、2.8 寸 SPI TFTLCD。图上的方向就是模块方向，例如 RF24L01，就是 PCB 朝核心板内插。
3. 外扩串口可以直接插安信可的 ESP8266 模块。
4. I2C 可接 OLED LCD(i2c)，管脚跟 SPI 兼容，也就是说，你买一个我们的 OLED LCD 模块，这个模块通过跳线电阻选择是 SPI 接口还是 I2C 接口，然后就可以接到这个外扩接口的 I2C 或者是 SPI 接口上。
5. 外扩普通 IO，布局和 SPI/I2C 兼容，意味着可以使用这些 IO 模拟 SPI 或者是 I2C。我们的模块就可

以接到这个模拟的接口上。做普通 IO 口使用，可以接我们的 4*4 矩阵按键。

外扩接口具体使用方法请查看《基本使用手册》

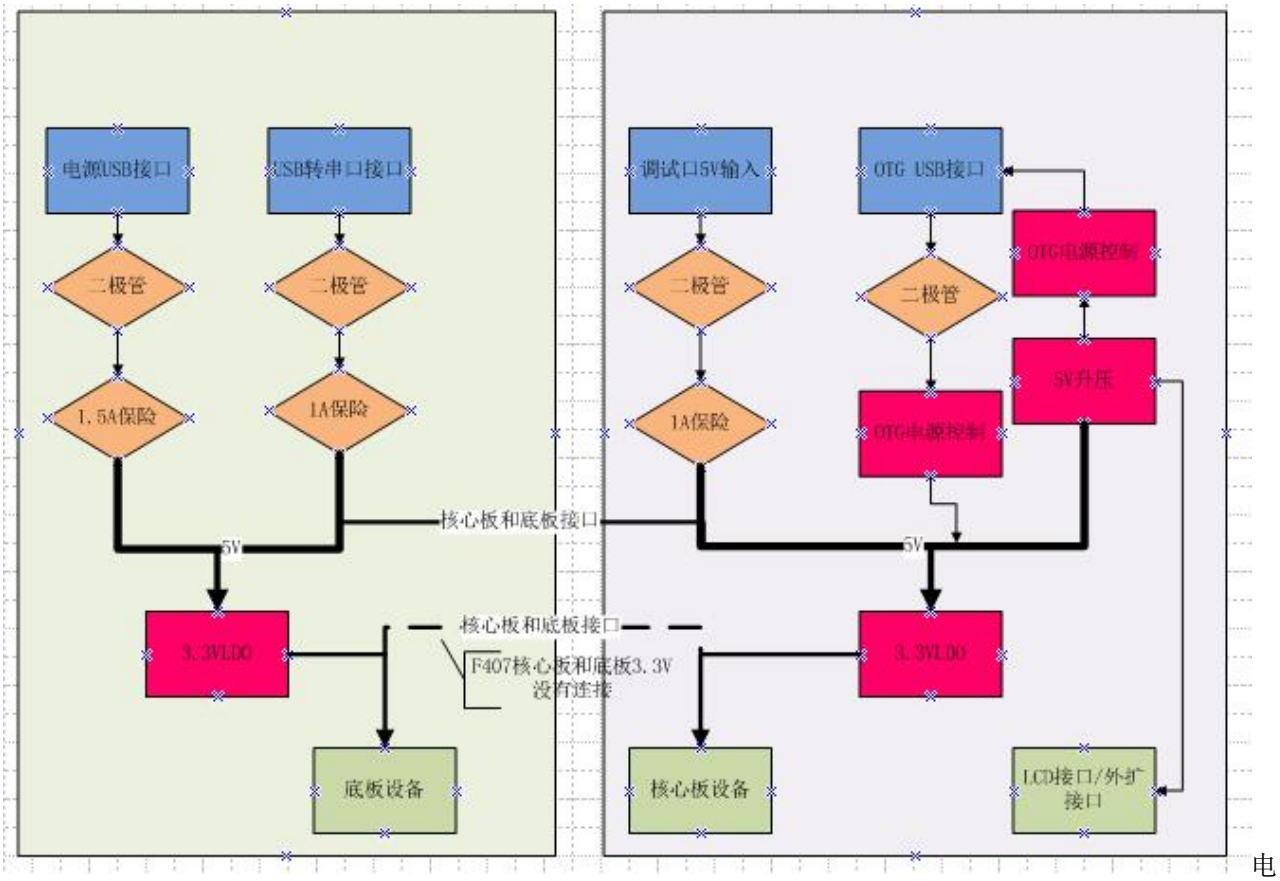
- 串口

在外扩接口上有一个外扩串口，但是跟 USB、摄像头共用了几个 IO。如果你希望使用一个串口，并且使用摄像头，例如：使用一个摄像头并且用串口接 ESP8266 模块。可以这样：

1. 不使用调试口上的调试串口。
2. 用杜邦线将 ESP8266 模块串口接到调试串口。
3. 其他控制 IO 用外扩普通 IO。
4. 禁止掉程序的所有调试信息。

-

3.1.7 电源总图



电源树上图是核心板和底板电源树。左边是底板，右边是核心板。底板将 5V 和 3V3 通过接口向核心板提供连接。我们的 F407 核心板只是连接 5V，3V3 没有接。核心板使用自己 LDO 将 5V 转为 3V3。

1. 底板有两路电源输入，都是 USB MICRO 接口。其中一路是独立电源输入，串 1.5A 保险，可以用手机充电器供电。另外一路是 USB 转串口，在用串口时，也可以对系统供电，串的保险是 1A。

2. 底板有 3V3 LDO，理论提供 500ma 电流，实际上是 700ma 关断。
3. 核心板同样有两路电源输入，一路是 OTG USB，另外一路是调试口输入。调试接口这路加有二极管和 1A 保险。OTG USB 口输入电源只有二极管没有保险丝。
4. 调试电压输入是通过 CMSIS DAP 或者 DAPLink 接入的 5V，这个 5V 通常也是由电脑提供，通常只能提供 500ma 电流。
5. 为了实现 OTG，核心板除输入电源转 3.3V 外，还有两个电源模块：一个是将输入电源升压到 5V；另外就是 OTG 电源输入输出切换电路。
6. 5V 升压理论能提供 500ma 电流，目前除了提供给 USB 外，还连接到外扩接口和 LCD 接口。屋脊雀的 LCD 模组带 LDO，因此通常使用的是升压后的 5V 供电。
7. 四路电源输入全部是 5V 输入，可以随便接。也可以同时接多个，多个能提供更多电流，不会造成倒灌。

3.1.8 常用模块功耗

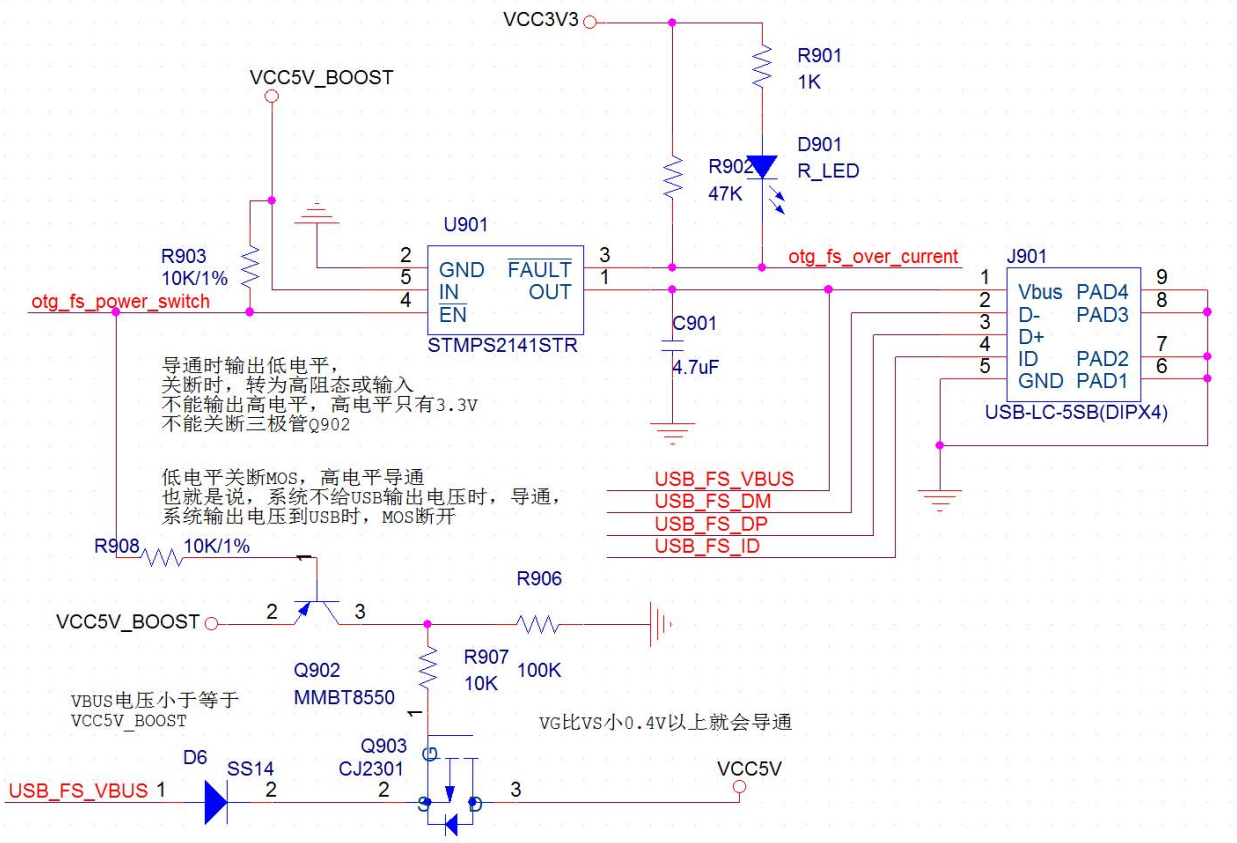
一些列表为一些常用模块功耗，如果使用中发现 LCD 屏幕背光抖动，或者其他异常，可能是电源不足，压降太大，请将底板的 1.5A 电源接上。

- 以下电流都是以工作电压测量，如有必要，请自行转换为 5V 输入端电流。

3.1.9 注意事项

- USB 口电源切换问题

为了实现 OTG 用一个 micro 接口，我们设计了一个电源切换电路。

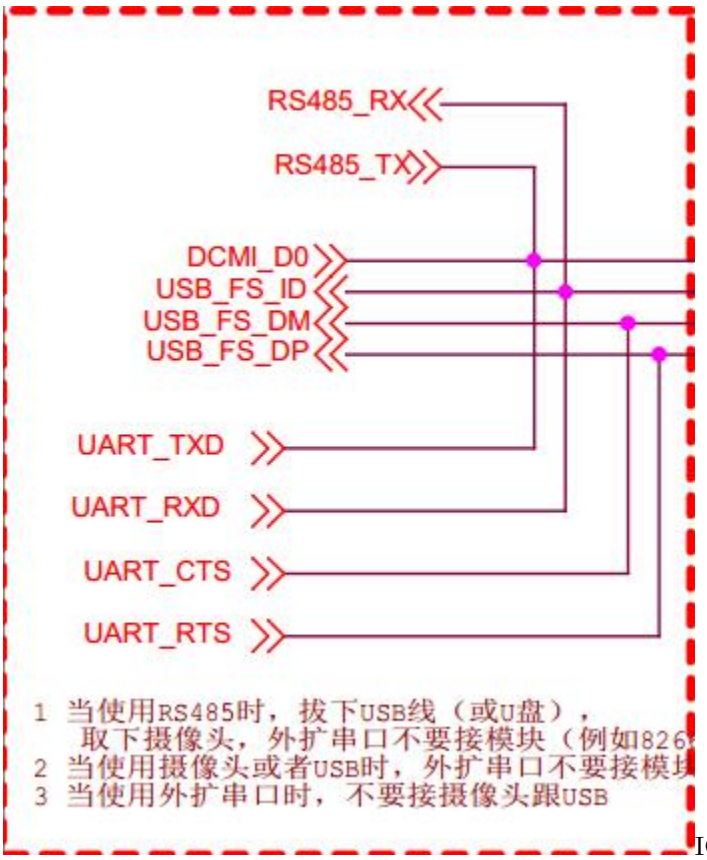


HOST 供电

接 U 盘，STM32 作为 HOST，otg_fs_power_switch 输出低电平，升压得到的 5V 通过 U901 输出，供电给 U 盘，同时 Q903 MOS 管关断，防止 USB_FS_VBUS 反向倒灌到系统 5V 电源。接电脑，STM32 作为 DEVICE，otg_fs_power_switch 转输入或者高阻态，U901 关断，Q903 导通，USB_FS_VBUS 供电给核心板使用。电路没加保险丝，也没有二极管保护。当 LCD，摄像头等都接上时，电流可能大于 500ma，直接接电脑可能会过流，建议通过带电源的 HUB 接到电脑。

- RS485 IO 共用问题

在进行硬件设计时，尽可能的少造成 IO 复用。为了提供一个干净的串口用于调试、命令行交互。只好将 485 和外扩接口共用，并且和 USB、摄像头 IO 口有复用。复用细节在原理图有标注。



IO 冲突

- FM 性能

1. 板载的 TEA5767 毕竟是一个小模组而已，性能无法和收音机相比。
2. 电脑电源会带来干扰，降低收音机灵敏度，用充电宝供电并且断开与电脑所有连接，效果会提升不少。
3. 配套的天线只能做功能测试，如果效果不好，可以在天线尾端接一段导线，并且将导线挂到高处。导线并不是越长越好，太长反而会引入其他干扰。按照 FM 的波长，天线总长 65 厘米左右，实测接一段 60 厘米的导线效果不错。
4. 空旷处（窗户边）肯定比室内效果要好。
5. 网络、摄像头、USB、SD 卡、TFT LCD 屏等，在运行时，都会发射干扰，降低收音机灵敏度。如要解决这个问题，需要增加屏蔽措施，考虑毕竟只是一块开发板，决定不做如此复杂，而且经过测试，在收音机信号良好的情况下，干扰影响不大。
6. 通过 WM8978 播放收音比 TDA2822 效果要好（工作室没能力调音，TDA2822 单声道，WM8978 立体声）。

- 核心板插拔本开发板布局比较紧密，很多器件都靠板边。在插拔式如不注意，可能会造成损坏。建议插拔时：1 先取下核心板周围的所有附件，例如摄像头、TFT lcd、USB 线、外扩 IO 口上的所有器件。2 从底部往上顶，慢慢取出核心板。3 核心板左边的 BOOT 拨动开关、上边沿的 miscro USB 接口，禁止受力。4 拔核心板时，两个排针要慢慢轮流拔出，不要一下子拔出一边，否则排针将**变歪**。

3.2 end

CHAPTER 4

嵌入式开发入门

够用的硬件

能用的代码 *

实用的教程 * 屋脊雀工作室编撰 -201801227

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

大家好，从今天开始，屋脊雀和大家一起学习《基于 STM32F407 的嵌入式软件开发实践》。在正式学习之前，我先跟大家解释一下问题：1 我们学什么？2 我们怎么学?

4.1 我们学什么

教程名字叫《基于 **STM32F407** 的**嵌入式**软件开发**实践**》，这个标题是很有意思的。

1. 我们学的是软件开发，而且，**是嵌入式软件开发**。对应嵌入式的是台式机软件开发，也叫 PC 软件开发。在行业中，嵌入式通常分两块：**LINUX 环境**和**单片机环境**。(实际上嵌入式软件开发还有很多分类)。单片机环境就是我们常说的裸奔或运行 RTOS（实时操作系统)。我们教程学习的是单片机环境的软件开发，而不是 LINUX 的 ARM 开发。
2. 基于 F407 学习嵌入式，需要硬件，也就是大家经常买的开发板。十多年前，通常从 51 单片机开始学习，然后学习 ARM7，再学习在 ARM9 上的 LINUX 开发。在今天，51 单片机虽然没有消亡，仍然在大量的家电类产品上使用。但是经过意法半导体多年经营，" 学习单片机"，已经被 "学习 STM32" 替代。为什么选择 F407 这一款 STM32 芯片呢？**首先**，407 资源丰富，是单片机中的强者。基于 407 学习之后，使用低端的芯片基本没有问题了，例如 F103 系列。**第二点**，价格便宜，一片二三十块钱。不像 F7 或 H7 系列，一片七八十块钱，价格非常昂贵。F7&H7 与 407 相比，多出来的功能，其实并不常用，初学者完全可以暂时不学习。(F7 和 H7 在实际项目上应用不多，太贵，太贵)
3. 开发实践这个是我们课程的重点，我们的软件开发，完全从实际项目流程进行，教程包含的都是多年的工作经验，知识点都是干货。这也是我们的根本目的，不仅仅要学习芯片功能，更多的要学习软件开发的技巧，驱动架构的接口设计，软件如何分层等。

4.2 我们怎么学

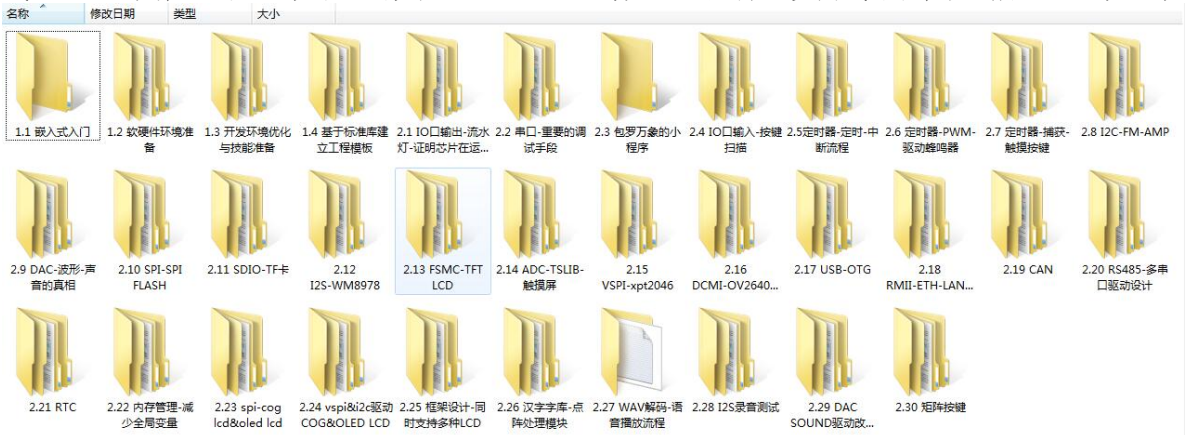
好的学习方法，事半功倍。目前网络很多教程，都比较基础，内容也只是对资料的理解，缺少实践干货。学习完那些课程后，只能算入门。缺少更深层次的理解。我们的教程，会传授很多实际经验给大家。毕竟我们都有 10 多年的开发实践，做出来的东西也更接近工作内容。

我们的教程有以下特点：

1. 基于屋脊雀的 F407 开发板这个开发板，是我们工作室精心设计，设计这个硬件是为了软件开发，是为了引出软件开发中的知识点。例如：我们底板跟核心板都有一个 SPI FLASH，为什么？为了引出多个设备共用 SPI 时如何处理，是为了引出 SPI 接口驱动如何写这个知识点。不像其他教程一样，堆砌硬件,每个硬件做教程,出几百页教程几百个视频,需要大量时间学习,无法迅速抓住软件开发的本质。

整体

2. 从 0 开始，模拟项目开发流程每一个教程，都是一步一步前进，后续的教程，基于前面的教程。例如：为了验证硬件是能用的，首先需要点一个灯，这是最基础的。只有灯亮了，才能去调试其他功能。第二个调试的外设，肯定是串口，为什么？因为串口是我们输出调试信息的重要手段。调试 LCD 功能，肯定是先调试显示一些点，显示英文字母。只有等 SPI FLASH 或者 SD 卡文件系统调试好后，才会做显示汉字图片。



教程目录

3. 重实践我们更注重实际项目开发需要的知识，更注重实际项目开发的流程，对于实际开发中不需要深入了解的，暂时不做过多解释。例如 USB 协议，实际开发中，通常是由原厂 FAE 做技术支持，我们只需要使用，所以，我们并不会对协议栈做太深入的探讨。还有，我们也不会对 STM32F407 一些太底层的细节做探讨。所以，我们是**基于库学习**，不会**直接操作寄存器**。在学习 STM32 的过程中，穿插介绍软件开发过程的其他知识，例如版本管理，编程规范等。

4.3 学习基础

学习单片机需要基础吗？我觉得需要，无论你是在学校还是自学，在学习嵌入式开发之前，最好有以下知识储备：

1. 基本的电路知识电阻电容你要知道，电压电流你要清楚。
2. 基本的 C 语言知识认真学习过学校的课程就可以了，后面我们会推荐一些自学提高的资料。

还有同学会问：要不要先学习 51？不需要，学过更好，不学也没关系。要不要汇编？不需要，会，当然很好。

> 当遇到一些深层次的问题，有汇编帮助，更容易处理问题。汇编思维会帮助你写出更健壮的程序，在写 C 代码的时候，你会时刻想着，这样写，会不会溢出啊，会不会有问题啊。（本人毕业后写过 3 年 6052 汇编）。

学习需要做什么装备？需要一台电脑，一套开发板，还有就是**定下心**。

前面我们大概介绍了教程，下面我们介绍一些入门的基本概念

4.4 单片机

单片机是什么？单片机的完整名称是**单芯片微机**。在百度百科定义是这样：

> 单片机又称单片微控制器，它不是完成某一个逻辑功能的芯片，而是把一个计算机系统集成到一个芯片上。相当于一个微型的计算机

也就是说，单片机就是个微型电脑。在 STM32 这么广泛使用之前，比较流行的单片机，是 51 单片机. 51 单片机，并不是一个型号，而是一种单片机的统称，这些单片机使用 8031 指令集。具体的有 AT 的 89S52、台湾 STC 的 51 单片机等。除此外，比较流行的单片机还有 AVR 等。

当年我们学完 51 后，就会进一步学习 “ARM”。那 ARM 又是什么？ARM 其实不是芯片，也不是单片机。ARM 是一个内核设计公司，最早，他们设计了 ARM7 内核，ARM9 内核。习惯上我们把用这两种内核的芯片我们都称之为 ARM 芯片。ARM 公司并不生产芯片，他只是卖内核给 IC 公司，芯片是 IC 公司设计生产的。当年流行的芯片，ARM7 有 NXP 的 LPC2132 等，ARM9 有三星的 2410 等。

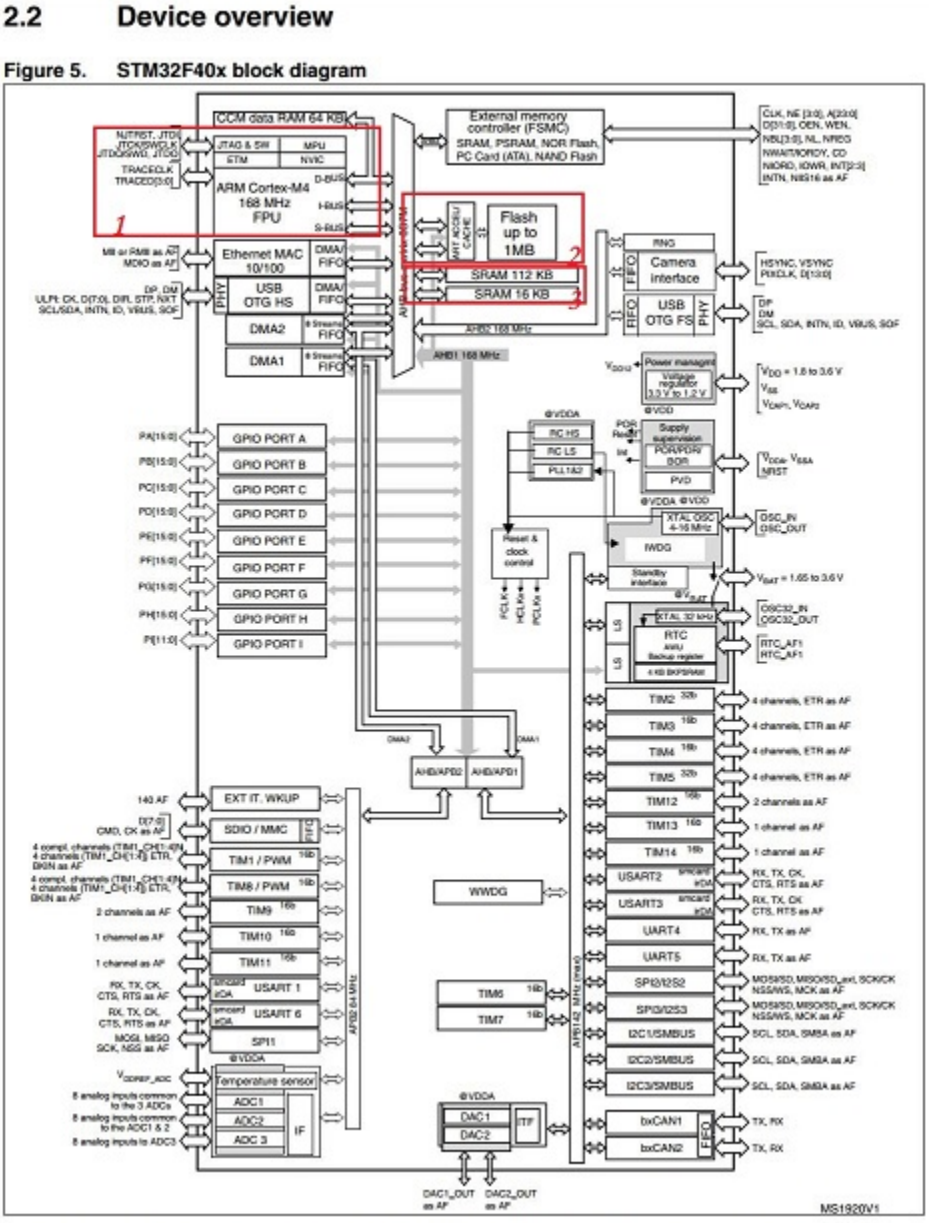
一般来说，ARM7 内核的芯片，不跑 LINUX；ARM9 可以跑 LINUX。ARM7 芯片类似现在的 STM32，或者说是 CONTEX 内核的芯片。

STM32，我们都知道是 ST 生产的芯片，这些芯片用的内核，也是 ARM 公司设计的，也就是 contex 内核。因为 STM32 推广的太成功了，所以现在没有说学习 contex 芯片，都是说学习 STM32。

还有一个，其实我们把 ARM9 内核的芯片，排除在单片机之外。。。。。所以我们俗称的单片机，都是裸奔或跑 RTOS 的，不跑 LINUX 的。

4.5 单片机的组成

前面说到，单片机就是一个小电脑。其实更形象的说，单片机类似一个电脑主机，不包含显示器和键盘等外设。**在每个芯片的规格书上，都会有芯片的组成框图**。其中 STM32F407 的，在文档《STM32F407_ 数据手册.pdf》第 17 页：



1. The timers connected to APB2 are clocked from TIMxCLK up to 168 MHz, while the timers connected to APB1 are clocked

stm32 block diagram

- CPU: 在 F407 芯片的框图上，左上角红框 1 框住的，就是芯片的内核，也就是 ARM 设计的 contex 内核。这一块内核，就相当于 CPU。
- 硬盘红框 2，是 407 芯片内部的 NOR FLASH，相当于硬盘。单片机 FLASH 与硬盘不同的是，FLASH 可以运行程序，硬盘是不能运行程序的，电脑程序都是从硬盘加载到内存中运行。跑 LINUX 的“单片

机”跟电脑一样，程序也是在内存中运行，因此，通常我们不把这些芯片叫做单片机了。

- 内存红框 3，是单片机内置的 SRAM，相当于内存。通常只用 SRAM 存储程序运行过程需要的变量。特殊情况下也可以加载程序到 RAM 运行。

此外：

1. 从这个框图，我们可以看出一个芯片，有多少功能，有多少外设，串口有几个，SPI 有几个等信息。
2. 有一些单片机其实是没有 RAM 和 flash 的，需要外接，例如 LPC2220。
3. 大部分跑 LINUX 的芯片，是没有内部 RAM 和 FLASH 的。例如 MT7688。

4.6 程序开发流程

待补充！

4.7 嵌入式项目开发流程

待补充！

4.8 end

CHAPTER 5

软硬件环境准备

够用的硬件

* 能用的代码

实用的教程 **

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

在项目开发中，当你拿到硬件提供的板子，第一步做什么呢？如果理所当然的认为可以直接上电调试，你就错了，有时，会直接烧板的。因为硬件工程师，不一定可信，而且很多公司的硬件工程师，不会对板子做基本验证，焊好就直接丢给软件用。

5.1 硬件检测

收到板子板后：

1. 先检查是否有明显焊接错误，了解板子的基本构成，查看各 IC 是否焊接正确。
2. 用万用表检测 **GND** 跟 **VCC**、**V33** 之间是否短路。

   我们第一版底板 VCC5V 跟 VCC3V3 之间短路。分析解决：未贴片的 PCB 两者没有短路，PCB 应该没有问题。查看原理图，发现 CAN 模块预留了两个 0 欧电阻，用来选择电压。只能焊接其中一个，样板两个都焊上了。焊下期中一个电阻即可解决问题。

3. 用 USB 线接上底板，上电测试。USB 线插到带 USB 转串口芯片 CP2140 的 micro USB 座子。底板红色电源灯亮，没闻到糊味，也没听到响声，更没看到青烟，开局良好。电脑提示安装驱动失败，USB 转串口用的是 CP2104 芯片，需要安装驱动才能正常使用。到官网下载驱动：https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers

   Download for Windows 7/8/8.1/10 (v6.7.5)

Platform	Software	Release Notes
Windows 7/8/8.1/10	Download VCP (5.3 MB) (Default)	Download VCP Revision History
Windows 7/8/8.1/10	Download VCP with Serial Enumeration (5.3 MB) Learn More »	Download VCP Revision History

   CP210X 驱动下载下载第一个默认的驱动 CP210x_Windows_Drivers.zip 安装过程，根据系统选择 X86 或 X64，然后一直点击下一步即可。

   CP210X 安装安装成功后重新插拔 USB 线，对开发板重新上电后，在**我的电脑**–> **设备管理**里面可以看

到 USB 转的串口：Silicon Labs CP210x USB to UART Bridge(COM6)(串口号由电脑自动分配)

CP210X 虚拟串口

4. 检测核心板用万用表检测 **GND** 跟 **VCC**、**V33** 之间是否短路

   我们第一版核心板 VCC3V3 与地线短路。分析解决：用万用表测试空 PCB，不短路。先分析原理图，看哪里可能会有问题，然后再对比焊接的 PCB 实物，看哪里焊错料。查原理图发现，画原理图的时候，SMT32F407 的 143PIN，PDR-ON，接了两个 0R 电阻，发正式版的时忘记修改了。PDR-ON 用法参考《STM32F407_ 数据手册.pdf》, 5.1.6 Power supply scheme 章节：

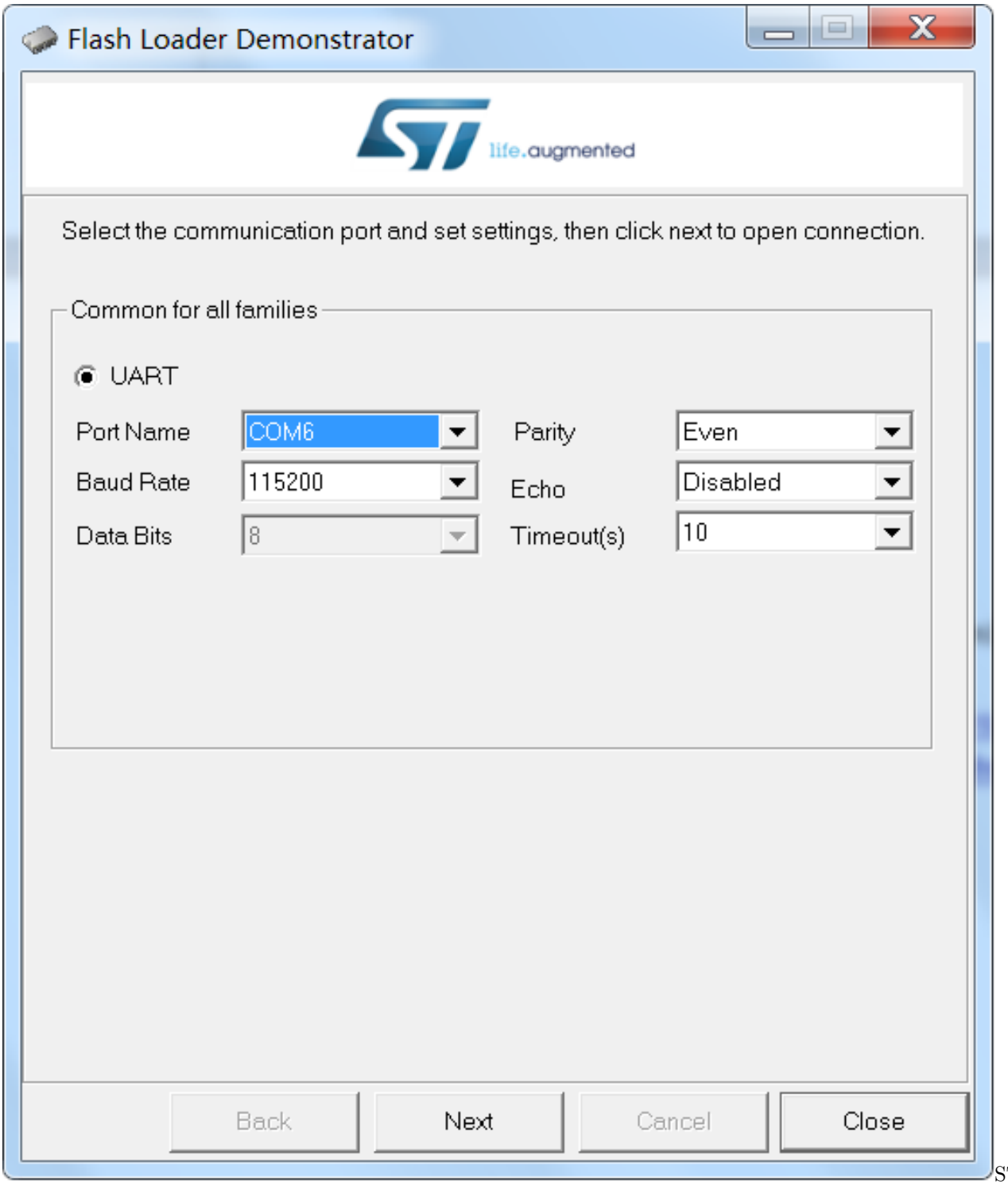
   1. Each power supply pair must be decoupled with filtering ceramic capacitors as shown above. These capacitors must be placed as close as possible to, or below, the appropriate pins on the underside of the PCB to ensure the good functionality of the device.
   2. To connect BYPASS_REG and PDR_ON pins, refer to *Section 2.2.17: Real-time clock (RTC), backup SRAM and backup registers*.
   3. The two 2.2 µF ceramic capacitors should not be connected when the voltage regulator is OFF.
   4. The 4.7 µF ceramic capacitor must be connected to one of the V_{DD} pin.
   5. V_{DDA}=V_{DD} and V_{SSA}=V_{SS}.

   PDR-ON 管脚说明阅读 2.2.15 16 等章节可以了解这个管脚的作用。我们将 PDR-ON 接到 VCC3V3，去掉 R1006 电阻。

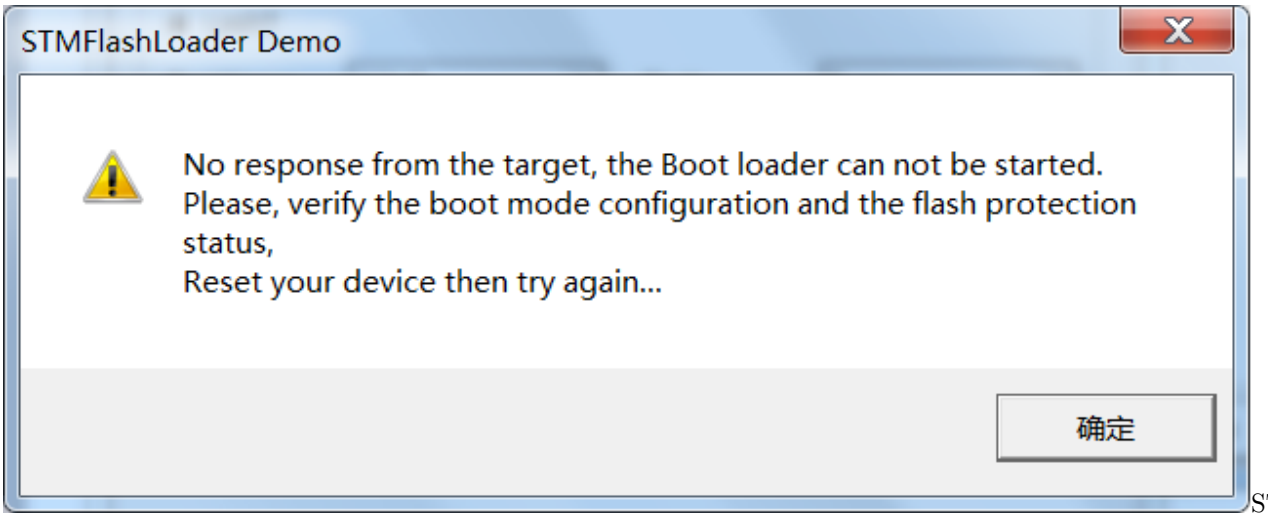
1. 核心板插上底板，万用表检测 VCC5V 跟 VCC3V3、GND 之间，没有短路。
2. 上电（USB 线连 CP2104 接口），底板跟核心板电源灯都正常点亮。没闻到糊味，也没听到响声，更没看到青烟，开局良好。

5.2 芯片测试

硬件电路经过检测后，可以上电测试了。如何验证芯片电路正确呢？可以使用 STM32 官方下载工具，FLASHER-STM32，《en.flasher-stm32.zip》，将一段程序下载到芯片内，如果下载成功，则说明芯片的最小系统是可以使用的。软件下载路径如下：http://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-programmers/flasher-stm32.html 解压后安装。安装后运行，界面如下，串口选择开发板上的 USB 转串口。**超时时间 *Timeout* 不要选太短，否则会擦除超时而失败，特别是 *STM32* 大容量型号，我们用的 *ZG* 就是 *1M flash* 的。**



ST FLASH 运行如果现在点击 next，下载软件卡一段时间后提示下图错误:

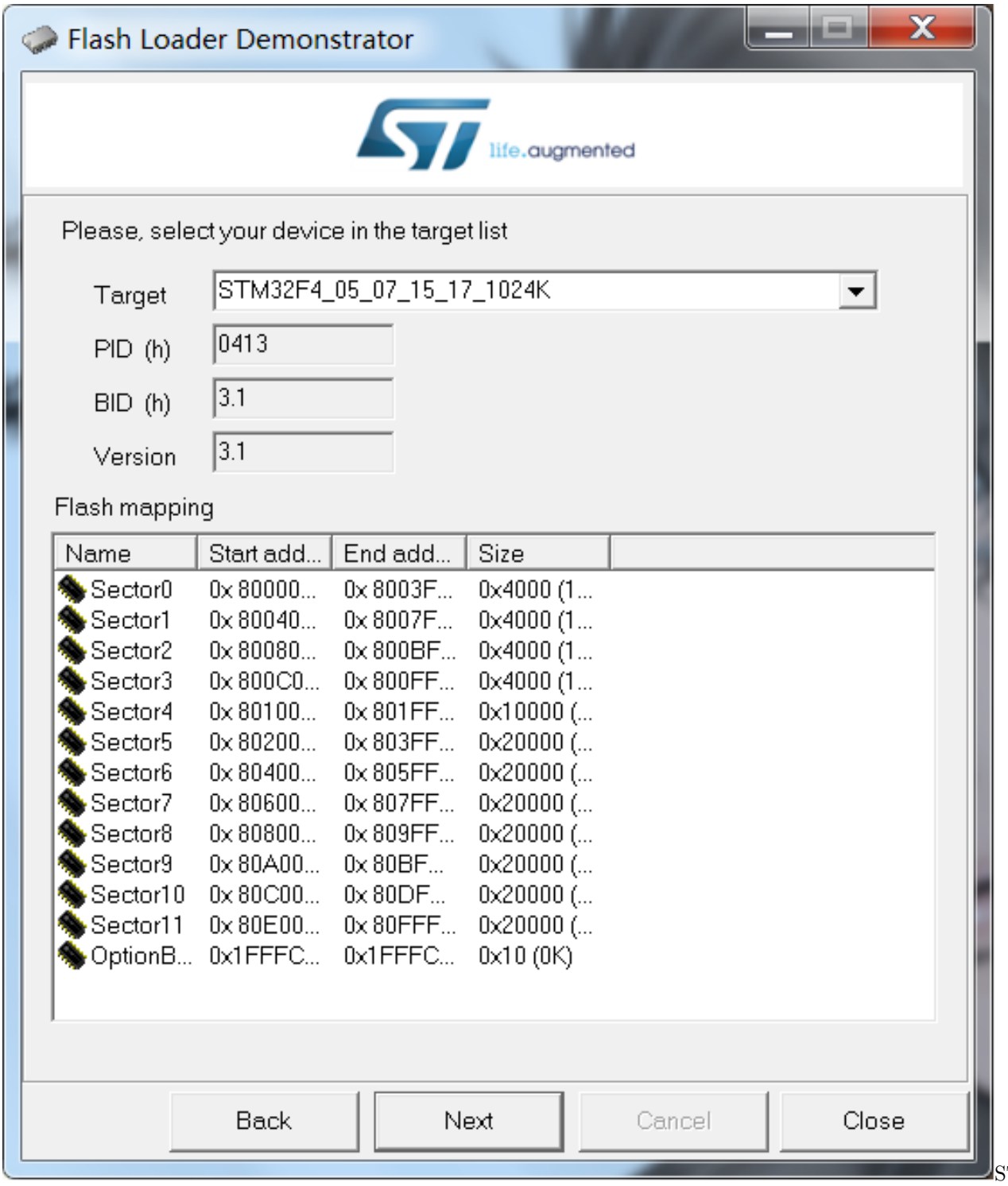


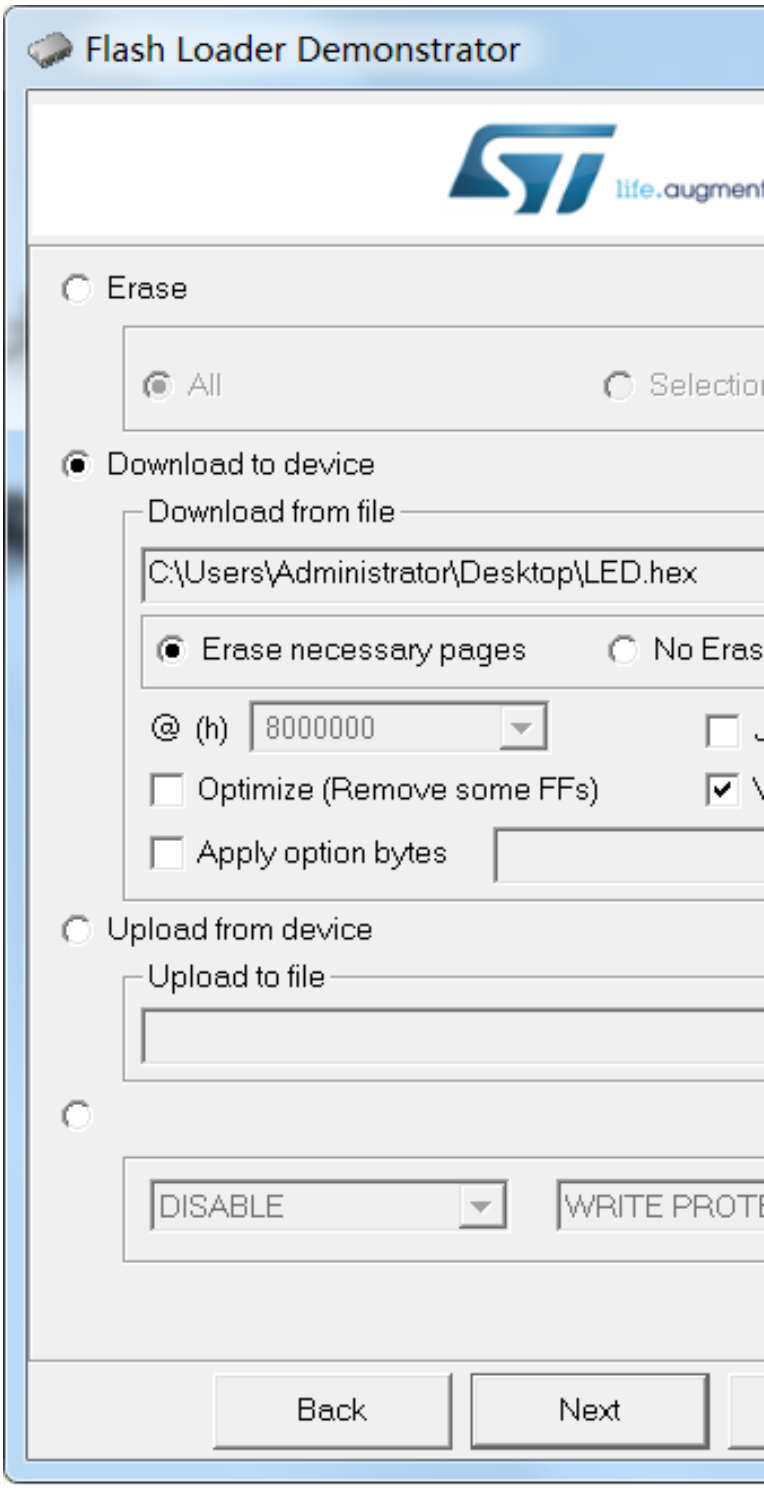
ST FLASh 下载失败这是因为我们还没有让 CPU 进入下载模式。阅读文档《STM32™ 微控制器系统存储器自举模式》，STM32 有 3 种启动模式。

表 2. 自举引脚配置

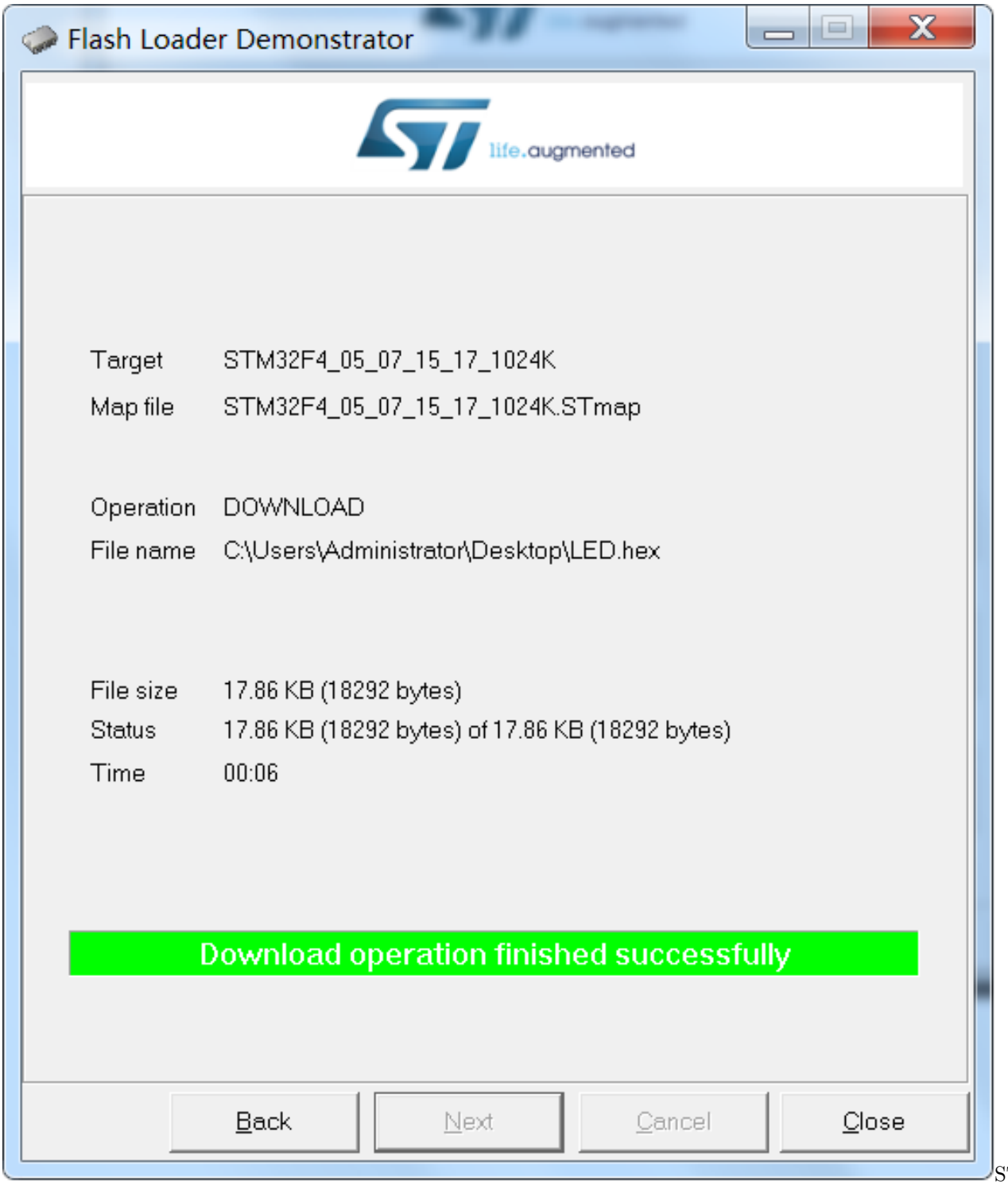
自举模式选择引脚		自举模式	别名使用
BOOT1	BOOT0		
X	0	用户 Flash	选择用户 Flash 作为自举空间
0	1	系统存储器	选择系统存储器作为自举空间
1	1	嵌入式 SRAM	选择嵌入式 SRAM 作为自举空间

BOOT 配置通常我们是将程序下载到 FLASH 中。下载程序时，让芯片从系统存储器启动，运行芯片自带的 BOOT。下载完成后，让芯片从 FLASH 启动，运行下载的程序。因此我们需要将 BOOT0 管脚接到高电平，BOOT1 接低电平，芯片进入系统存储器启动。在屋脊雀开发板上，只需要将 BOOT0 拨动开关拨动到 1，BOOT1 管脚电路上已经接到 0。启动模式设置好后，按核心板上的复位键复位芯片。再点击 next，next，就可以检测到芯片型号了，如下图。


Flash Loader Demonstrator
life.augmented
Please, select your device in the target list
Target
STM32F4_05_07_15_17_1024K
PID (h)
0413
BID (h)
3.1
Version
3.1
Flash mapping
Name
Start add...
End add...
Size
Sector0 0x 80000... 0x 8003F... 0x4000 (1...
Sector1 0x 80040... 0x 8007F... 0x4000 (1...
Sector2 0x 80080... 0x 800BF... 0x4000 (1...
Sector3 0x 800C0... 0x 800FF... 0x4000 (1...
Sector4 0x 80100... 0x 801FF... 0x10000 (...
Sector5 0x 80200... 0x 803FF... 0x20000 (...
Sector6 0x 80400... 0x 805FF... 0x20000 (...
Sector7 0x 80600... 0x 807FF... 0x20000 (...
Sector8 0x 80800... 0x 809FF... 0x20000 (...
Sector9 0x 80A00... 0x 80BF... 0x20000 (...
Sector10 0x 80C00... 0x 80DF... 0x20000 (...
Sector11 0x 80E00... 0x 80FFF... 0x20000 (...
OptionB... 0x1FFFC... 0x1FFFC... 0x10 (0K)
Back
Next
Cancel
Close
ST



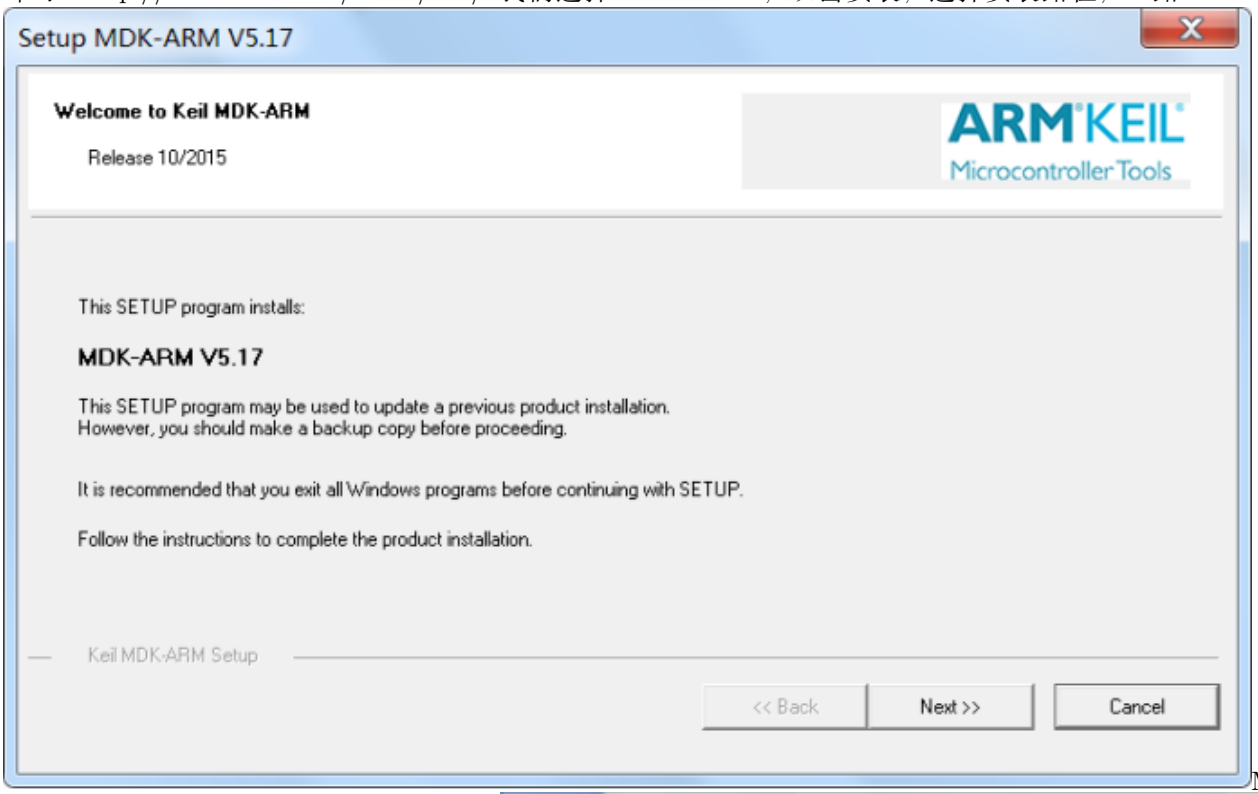
FLASH 芯片信息随便选择一个.hex 文件，测试是否能烧录。FLASH 选操作烧录成功，说明硬件核心部分基本正常，可以开始调试软件了。


Flash Loader Demonstrator
life.augmented
Target STM32F4_05_07_15_17_1024K
Map file STM32F4_05_07_15_17_1024K.STmap
Operation DOWNLOAD
File name C:\Users\Administrator\Desktop\LED.hex
File size 17.86 KB (18292 bytes)
Status 17.86 KB (18292 bytes) of 17.86 KB (18292 bytes)
Time 00:06
Download operation finished successfully
Back
Next
Cancel
Close

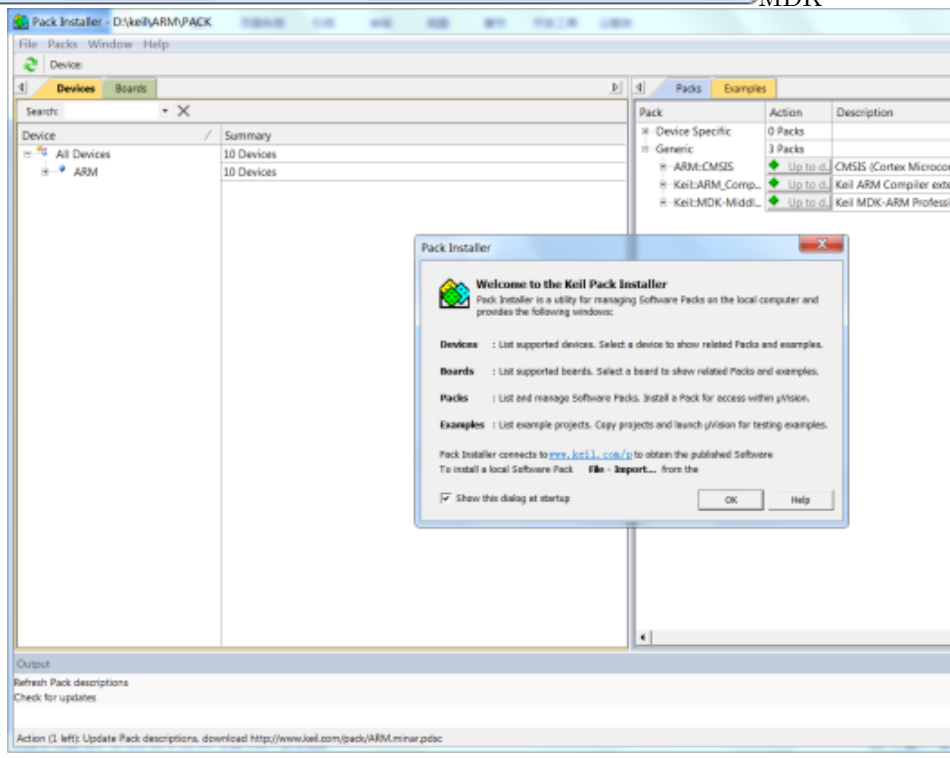
ST FLASH 成功

5.3 编译工具安装

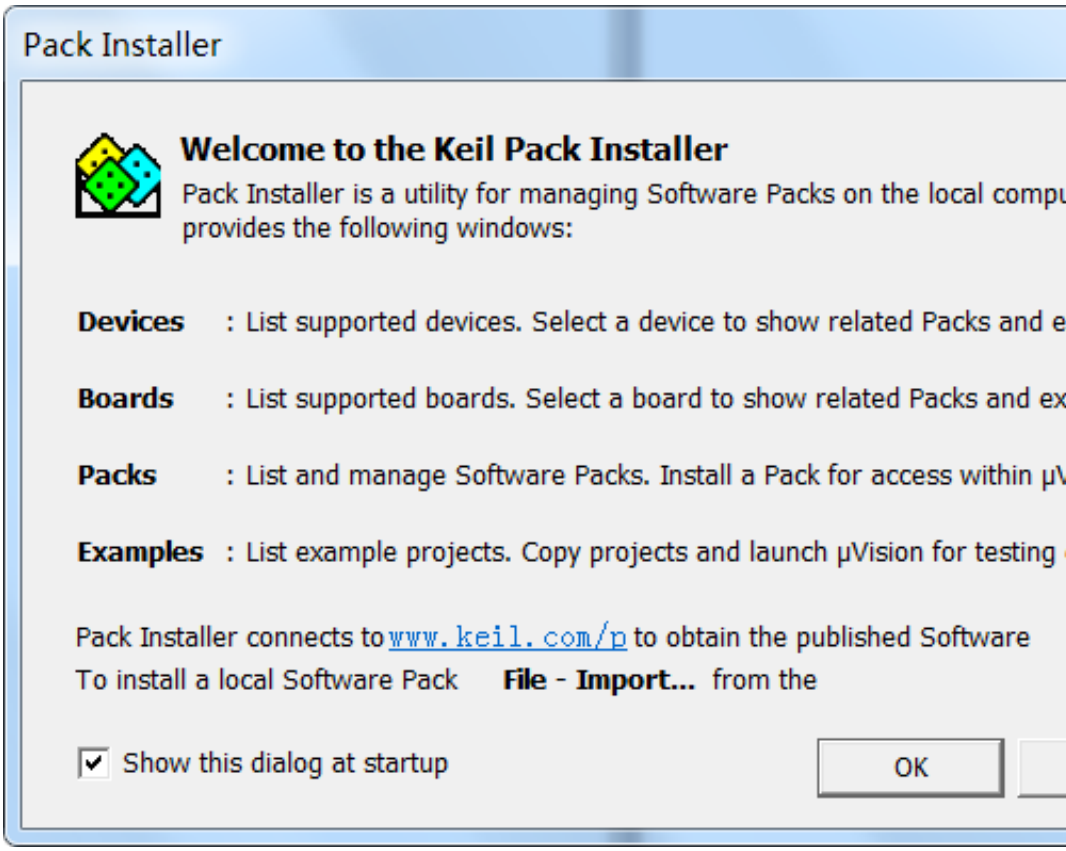
开发 STM32 我们使用 keil MDK 集成开发环境，MDK 在持续更新，我们选用一个较新的稳定版本即可。http://www2.keil.com/mdk5/524/ 我们选择 mdk524.exe，双击安装，选择安装路径，一路 Next



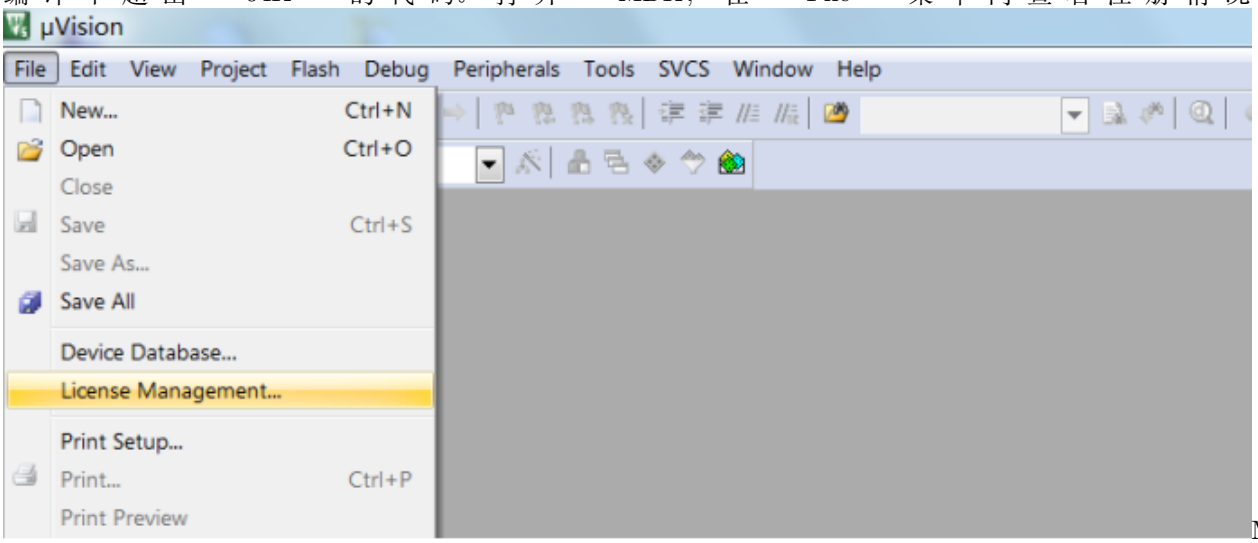
MDK



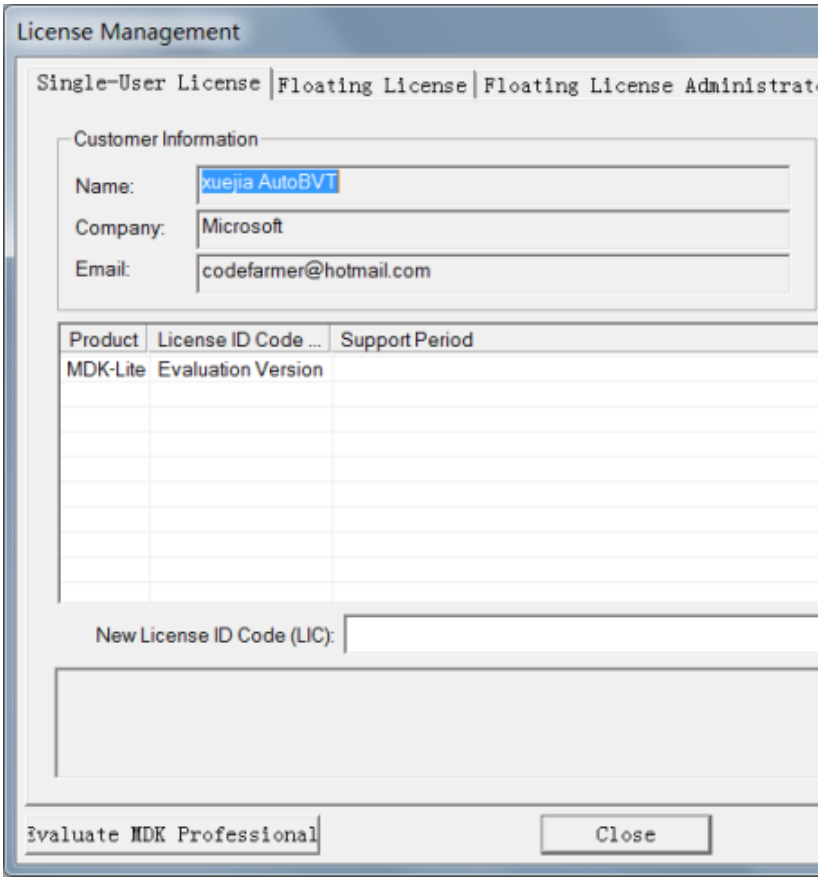
安装软件安装结束后，弹出一个库配置界面



库配置提示框说明了库配置信息配置信息此处可暂时不配置。MDK 安装结束后，需要注册，否则只能编译不超出 64K 的代码。打开 MDK，在 File 菜单内查看注册情况

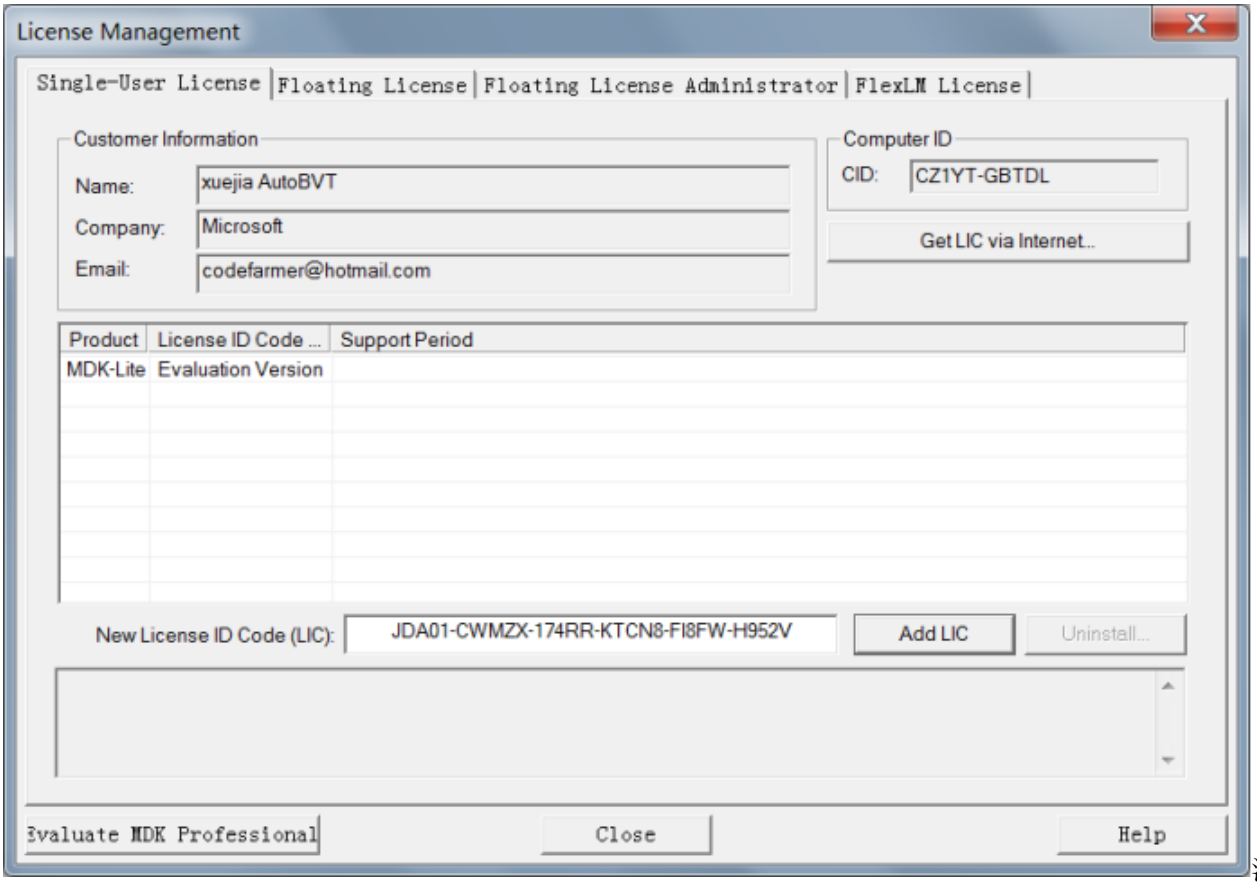


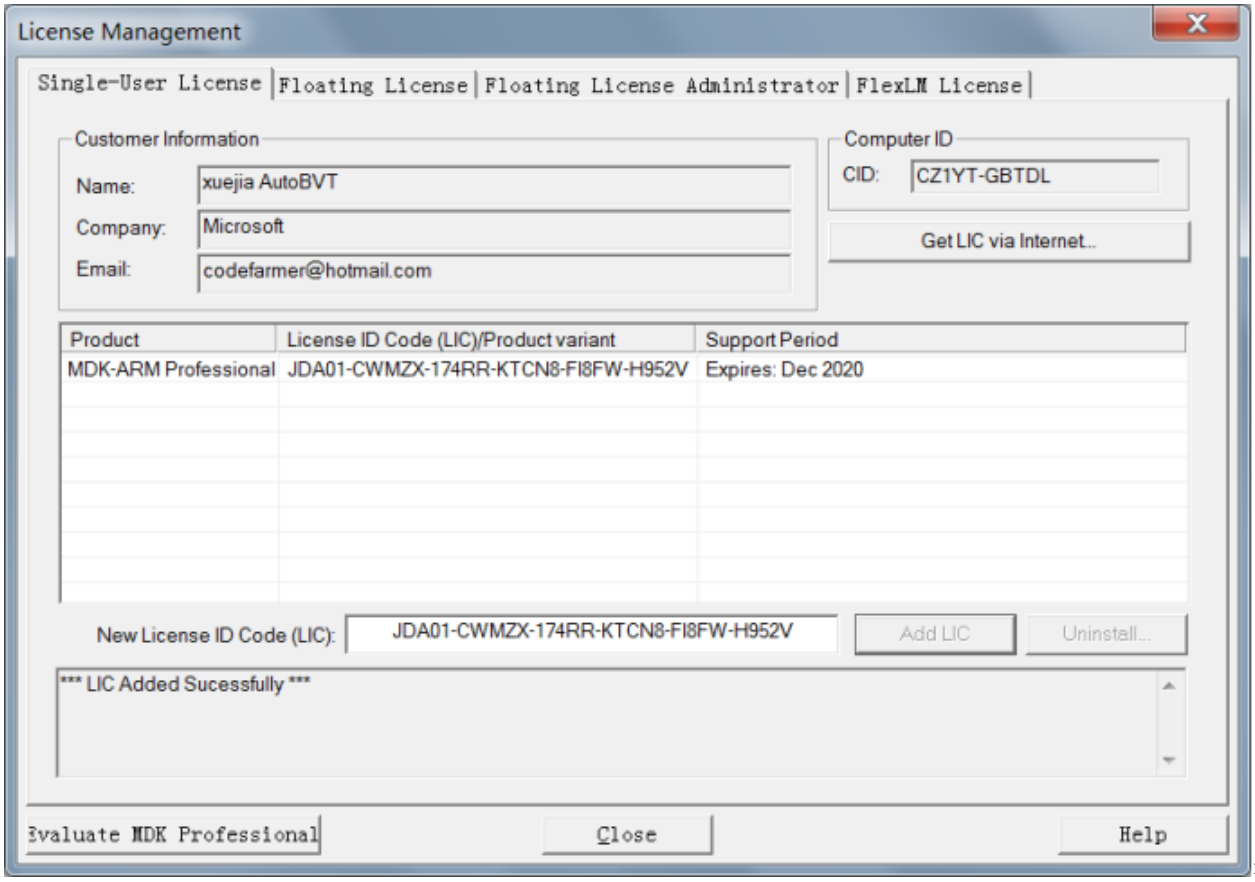
MDK



FILE 菜单目前属于未注册状态，右上角有一个 CID 号未注册 **可以通过 CID 获取到一个注册码，如何获取请自行百度**。

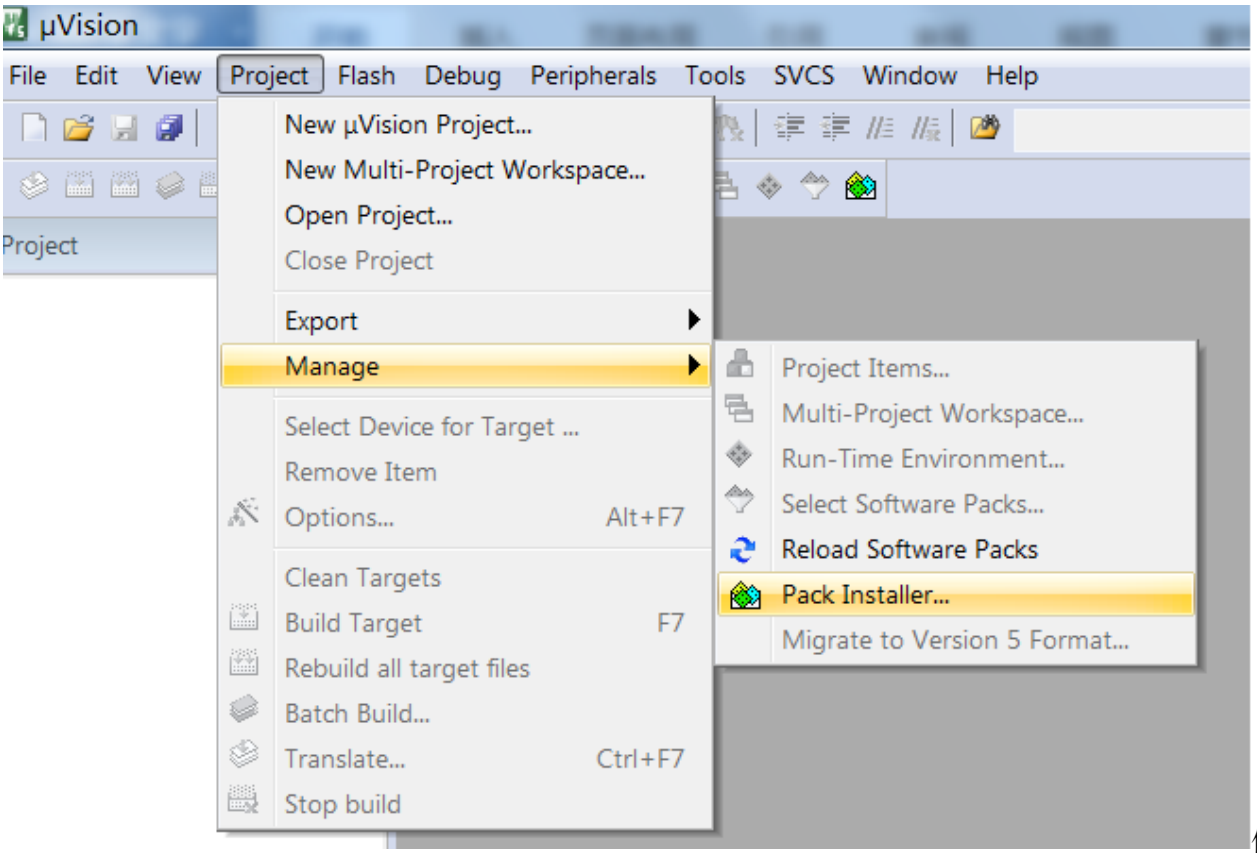
将注册码拷贝到 MDK 对话框下部 New License ID Code 框内，点击 Add LIC，


License Management
Single-User License
Floating License
Floating License Administrator
FlexLM License
Customer Information
Name:
xuejia AutoBVT
Company:
Microsoft
Email:
codefarmer@hotmail.com
Computer ID
CID:
CZ1YT-GBTDL
Get LIC via Internet...
Product
License ID Code ...
Support Period
MDK-Lite
Evaluation Version
New License ID Code (LIC):
JDA01-CWMZX-174RR-KTCN8-FI8FW-H952V
Add LIC
Uninstall...
Evaluate MDK Professional
Close
Help
注

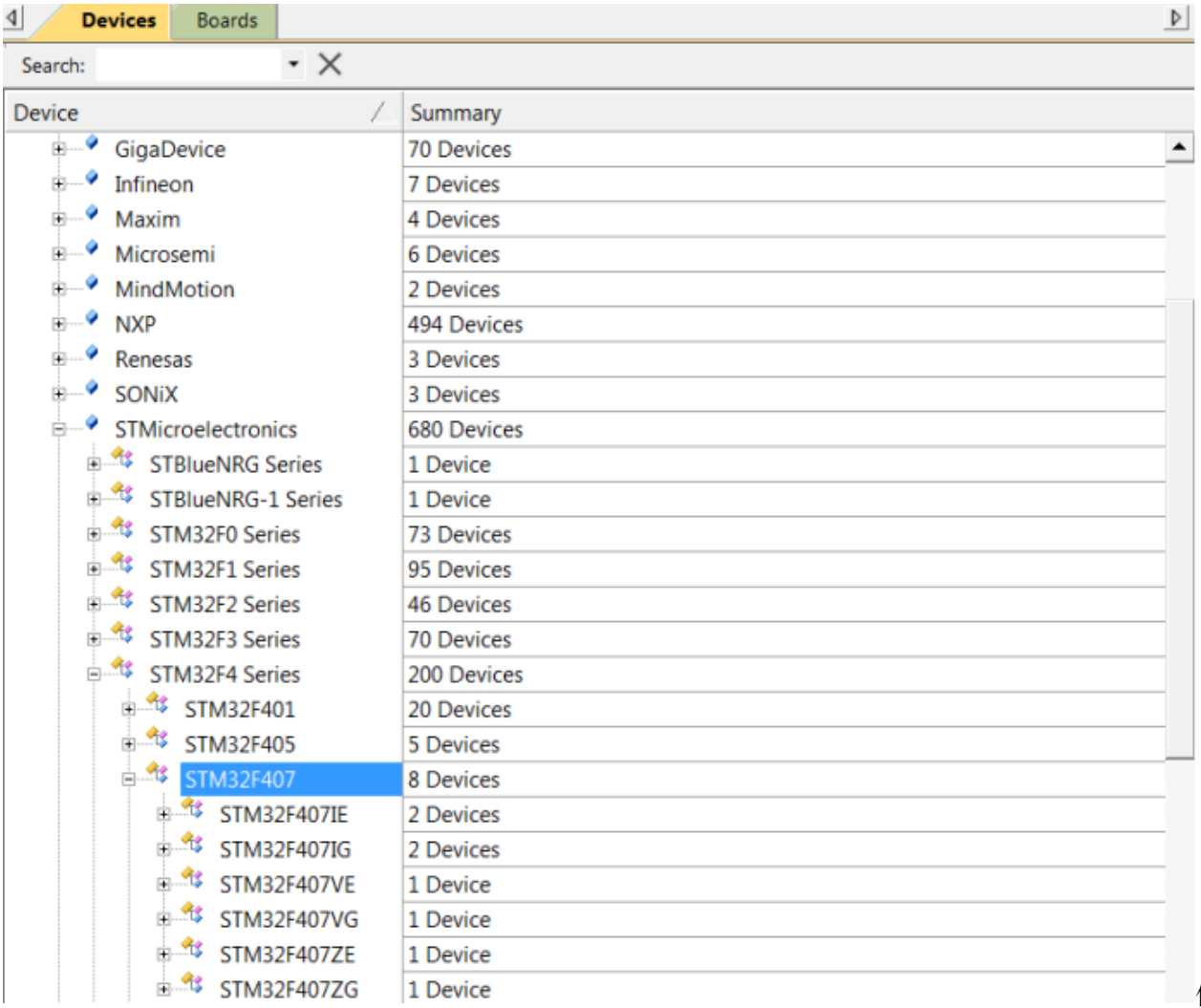


册注册成功注册成功

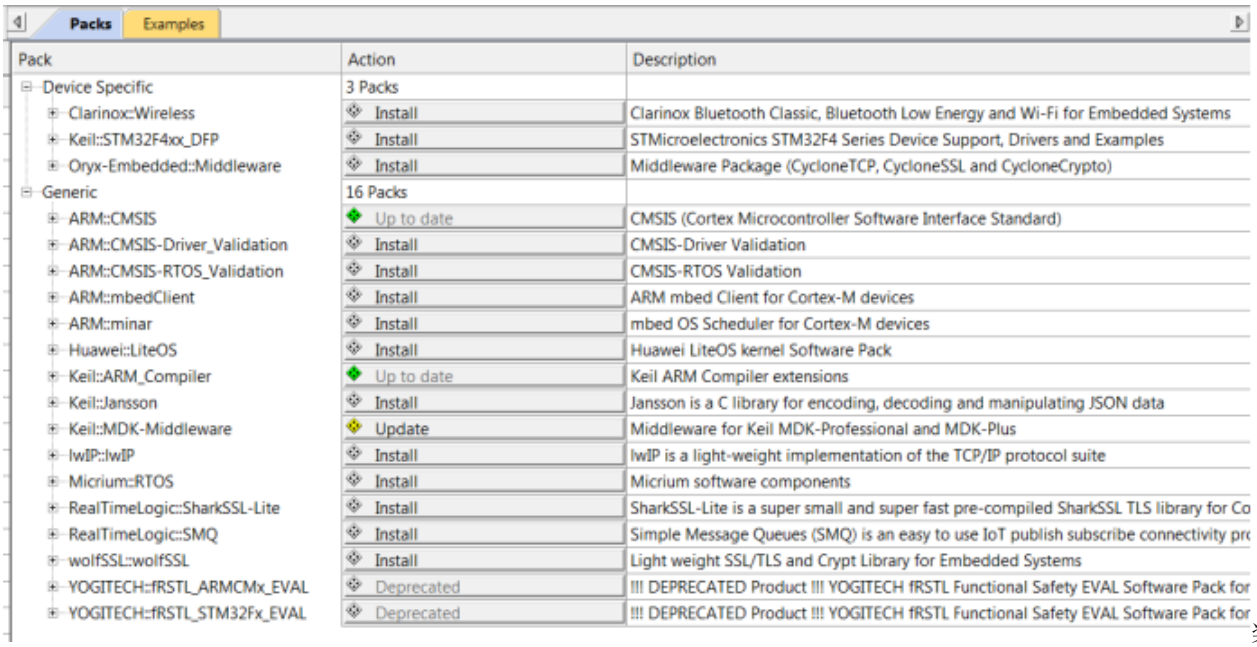
比较早的 KEIL 版本，只要安装好 IDE，就可以正常工作了。但是后来越来越多芯片，KEIL 就分成两部分了，一部分是 MDK IDE 环境，另一部分是不同芯片的依赖包。依赖包 Pack 下载



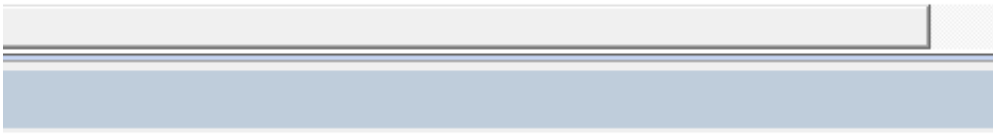
依赖包安装左边对话框内找到芯片 **(第一次安装 MDK 时，看不到芯片，只有 ARM 一个选项，请双击对应的内核，下载芯片支持列表)**



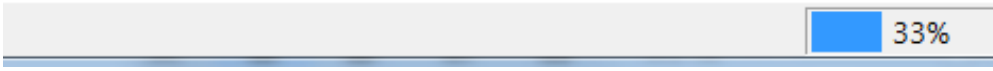
依赖包芯片右边则是库说明与安装，我们暂时只选择前面 3 个芯片支持包，其他的扩展包暂时不使用。

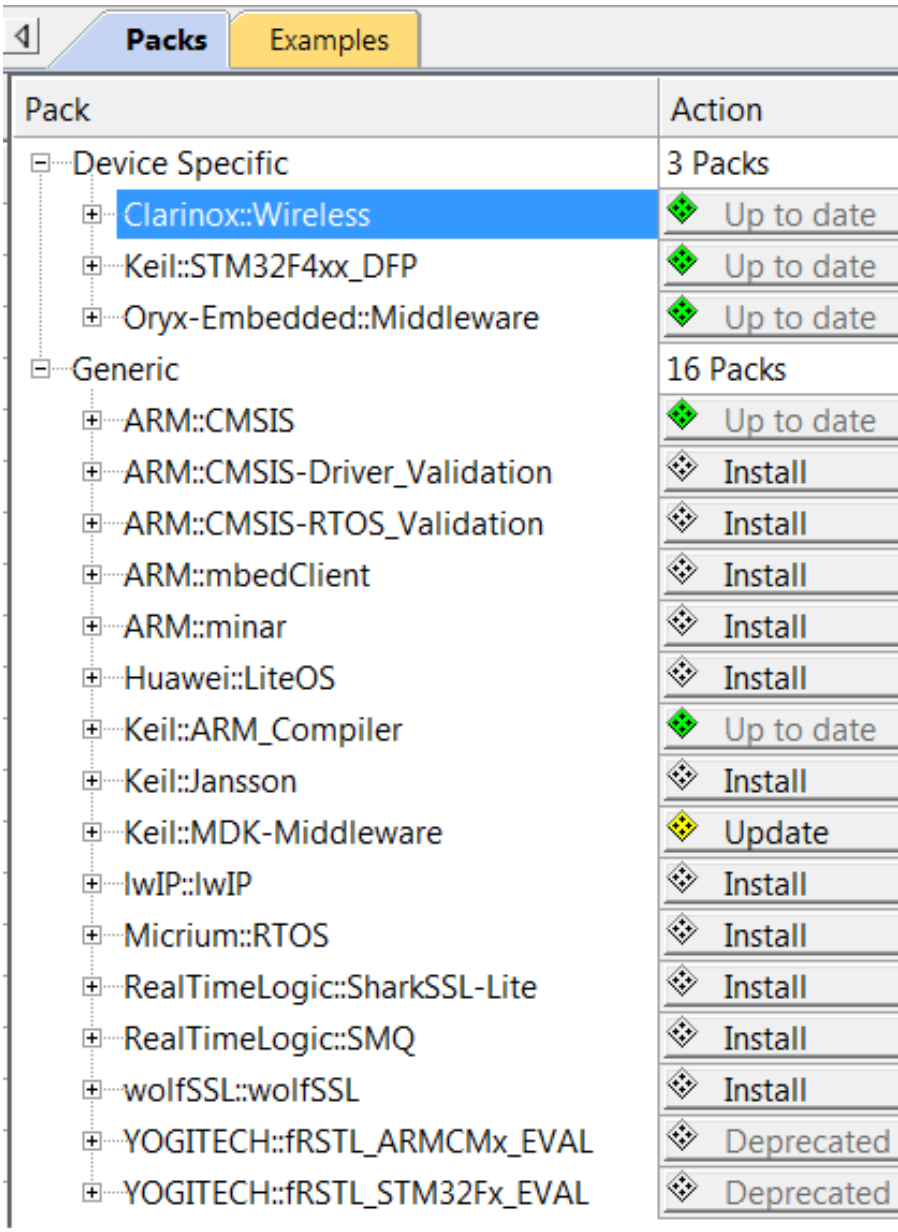


装



3 个包点击安装后左下角有安装进度条





度安装结束后,安装的 Pack 显示绿色棱形图标。装结束

下载 pack 很慢，可以直接到 http://www.keil.com/dd2/pack/#/eula-container，用下载工具下载。在资料包内我们提供了 407 需要的包。下载完成后，直接双击下载包就可以安装了。

5.4 CMSIS DAP

前面我们测试芯片，是通过串口下载程序验证芯片是否正常工作。这个方法有点麻烦，而使用调试器就相对简单。调试器有很多种，有 JLINK、STLINK 等。以前很多人都是使用盗版的 JLINK，除了有版权问题，JLINK 还经常会丢固件，固件丢失后 JLINK 就变砖头了。现在多了一个选择，那就是 CMSIS DAP (DAPLink)。CMSIS DAP 是 ARM 公司开源 mbed 项目的一个附属品，用于调试 **Cortex** 内核的芯片。这

个小工具除了能像 JLINK 一样下载调试程序外，还自带 USB 转串口功能。**我们根据开源的 CMSIS DAP，优化设计了一版本硬件，推出一款新的 DAP，物美价廉**一个小巧的 DAP，可以替代 JLINK+USB 转串口 + 电源线，接线非常简洁，电脑桌也更少线缆了。关于 DAP，请查阅《CMSIS DAP 产品手册》。资料下载：https://pan.baidu.com/s/1bHUVe6X6tymktUHk_z91cA

5.5 结束

到此，开发 STM32 需要的软硬件基本准备好。

5.6 end

CHAPTER 6

开发环境优化与技能准备

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

上一章节我们准备好了软硬件开发环境，为了提高开发效率，我们还需要做一些必要的工具介绍与环境优化。例如：编码工具的选择。

6.1 编码工具的选择

MDK，是一个 IDE，除了编译程序，当然包含了编码环境。但是 MDK 的编码环境相对其他编码环境来说，功能比较弱。码农常用的编码环境有：sourceinsight、ATOM、Sublime Text、Eclipse。综合考虑我们选择 sourceinsight 作为 win 环境下的代码编辑工具。

> Source Insight 是一个面向项目开发的程序编辑器和代码浏览器，它拥有内置的对 C/C++, C# 和 Java 等程序的分析。能分析源代码并在工作的同时动态维护它自己的符号数据库，并自动显示有用的上下文信息。

话说这个工具十年前发布了 3.5 版本后就一直停滞更新，搞得我们这些码农就使用了 10 年 3.5 版本。就在大家对更新不再抱任何希望的时候，竟然更新了，就在 2017。官网：https://www.sourceinsight.com/ **如果有条件的，请支持正版**。安装说明见资料包内文档《si4.0 安装与 doxygen 宏配置.docx》大家感受下其界面 SI 界面

6.1.1 SI 常用操作

见资料包内的文档《si 使用入门与常见操作.docx》

6.2 C 语言提高

当年我在学校学 C 语言的时候，指针都不教，所以如果你是刚毕业或者是在校生，请你将 C 语言加深学习。

1. 首先建议人手备一本《C 程序设计语言》C 程序设计语言这本书要放在工作电脑边，实时查阅。
2. 建议读《C 语言深度解剖 (普通下载).pdf》，如果你懂了这本书，面试中遇到的 C 语言肯定没问题。可以买一本实体书支持作者
3. 其他资料包内的其他文件也很有用，可以一读，如果现在不理解，等教程全部学习完之后再回头看也可以。

6.3 版本管理

版本管理，也叫版本控制。通俗易懂的说法，就是版本备份。当你完成一个功能的时候，一定要进行备份。当你想做一个修改的时候，也请备份。为什么要备份呢?

1. 防止代码丢失。
2. 修改代码过程中，经常会出现一些其他问题，这时就可以将修改前后代码进行对比，根据差异点快速定位问题。

那么如何做备份呢? 最简单的办法就是拷贝一份，然后用日期或者修改关键字命名。这个只能算备份，称不上版本管理。版本管理通常需要版本管理软件。常用的版本管理软件有 git 和 SVN，这两个软件的最主要区别就是 git 是分布式，SVN 是集中式。git 是 linux 版本控制的工具，我们推荐大家用 git，因为自己单机使

用，git 最方便，而 SVN 需要一个服务器（可以自己电脑搭一个）。git 的使用并不难，最简单的基础就几条指令。

具体使用方法见资料中的《git 基本使用.pdf》

6.4 对比工具

前面版本管理中提到对比，git 本身就有对比功能。不过我还是推荐大家使用 beyoncompare。

具体使用方法见资料中的《beyondcompare 基本使用.pdf》

6.5 end

CHAPTER 7

基于标准库建立工程模板

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

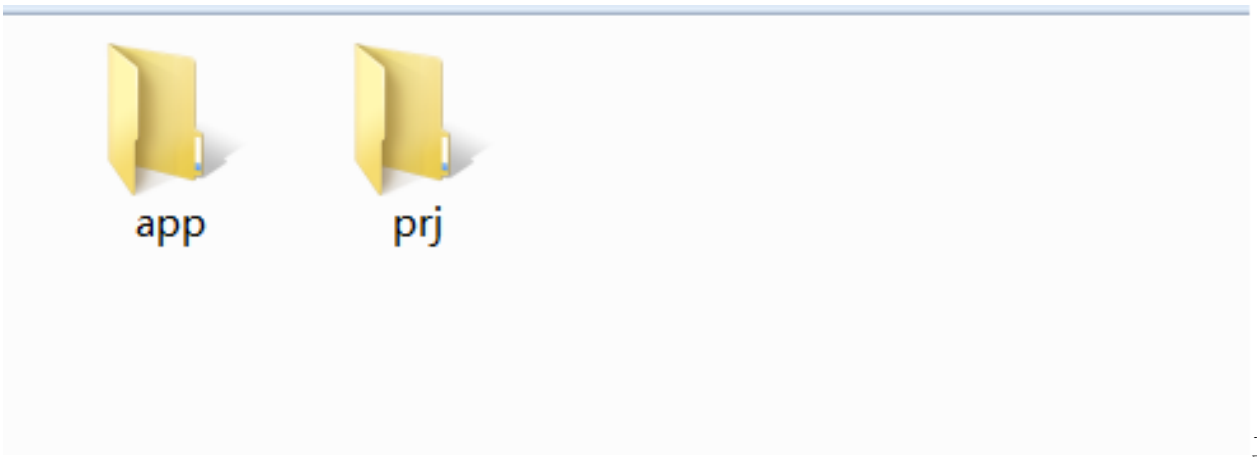
资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

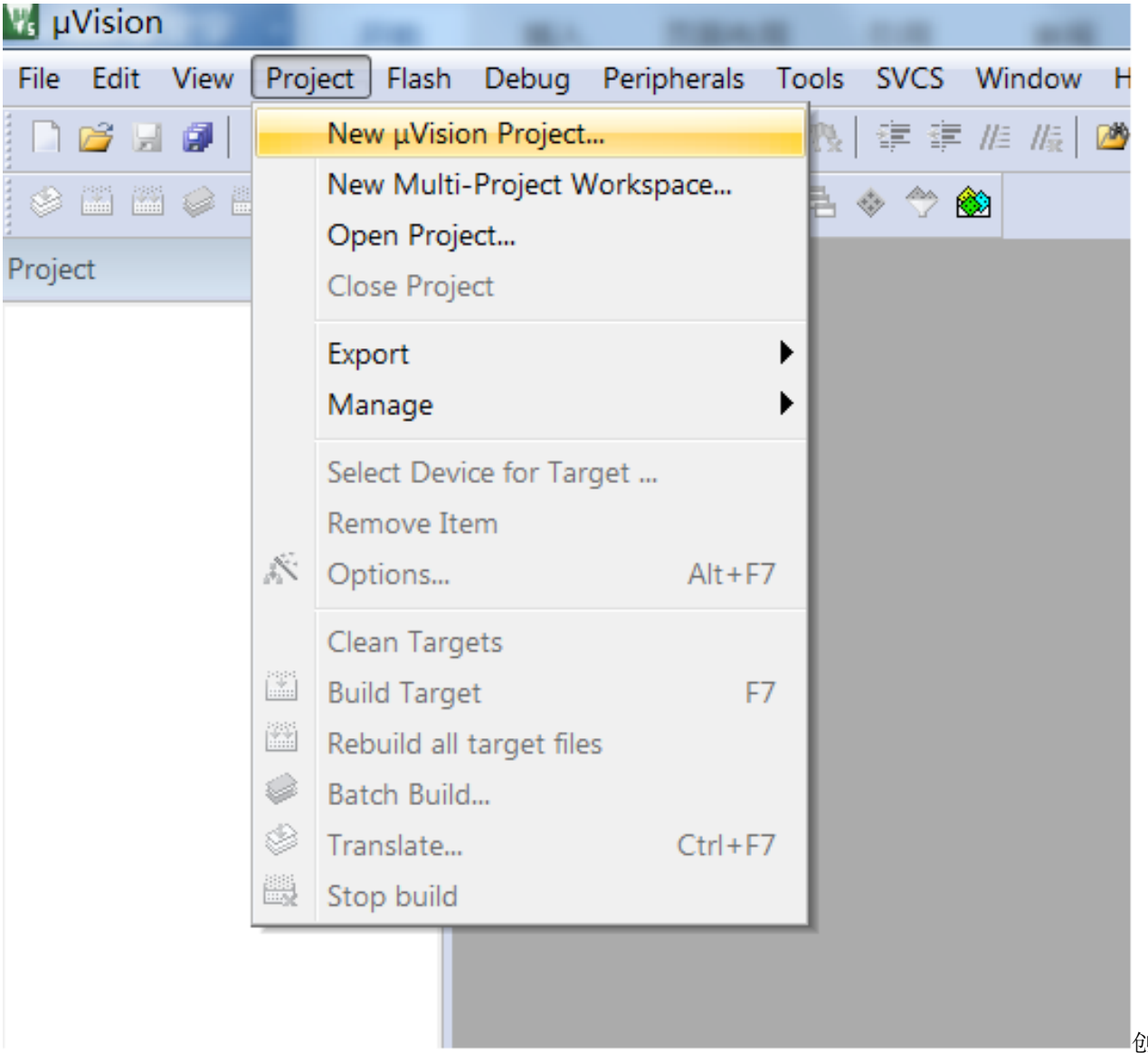
前面几个小节，我们了解了基本的软件开发流程，配置好了开发环境，今天，我们正式开始软件开发。

7.1 建立工程

首先在电脑建立目录，保存工程文件，app 文件夹用来存放用户代码，prj 文件夹用来保存 MDK 工程文件。

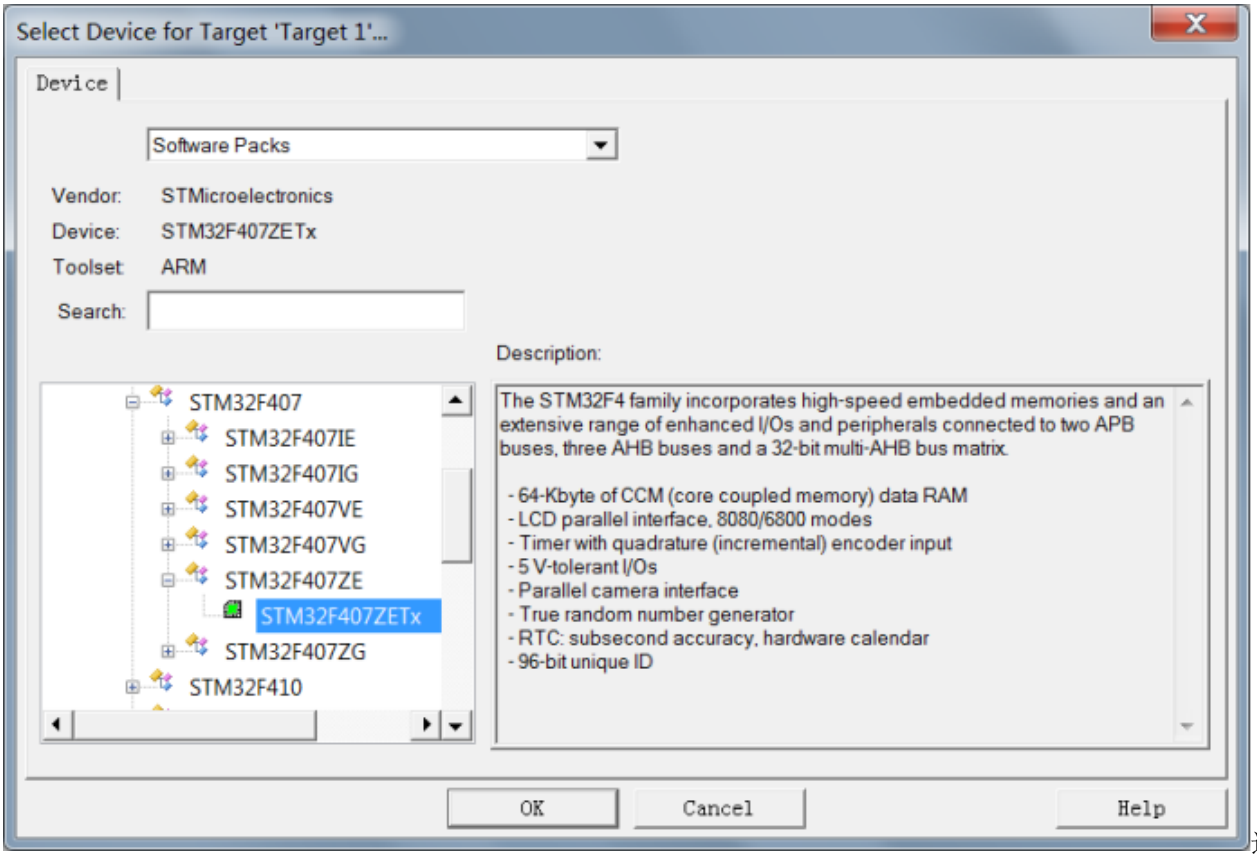


工程目录打开 MDK 软件，创建新工程

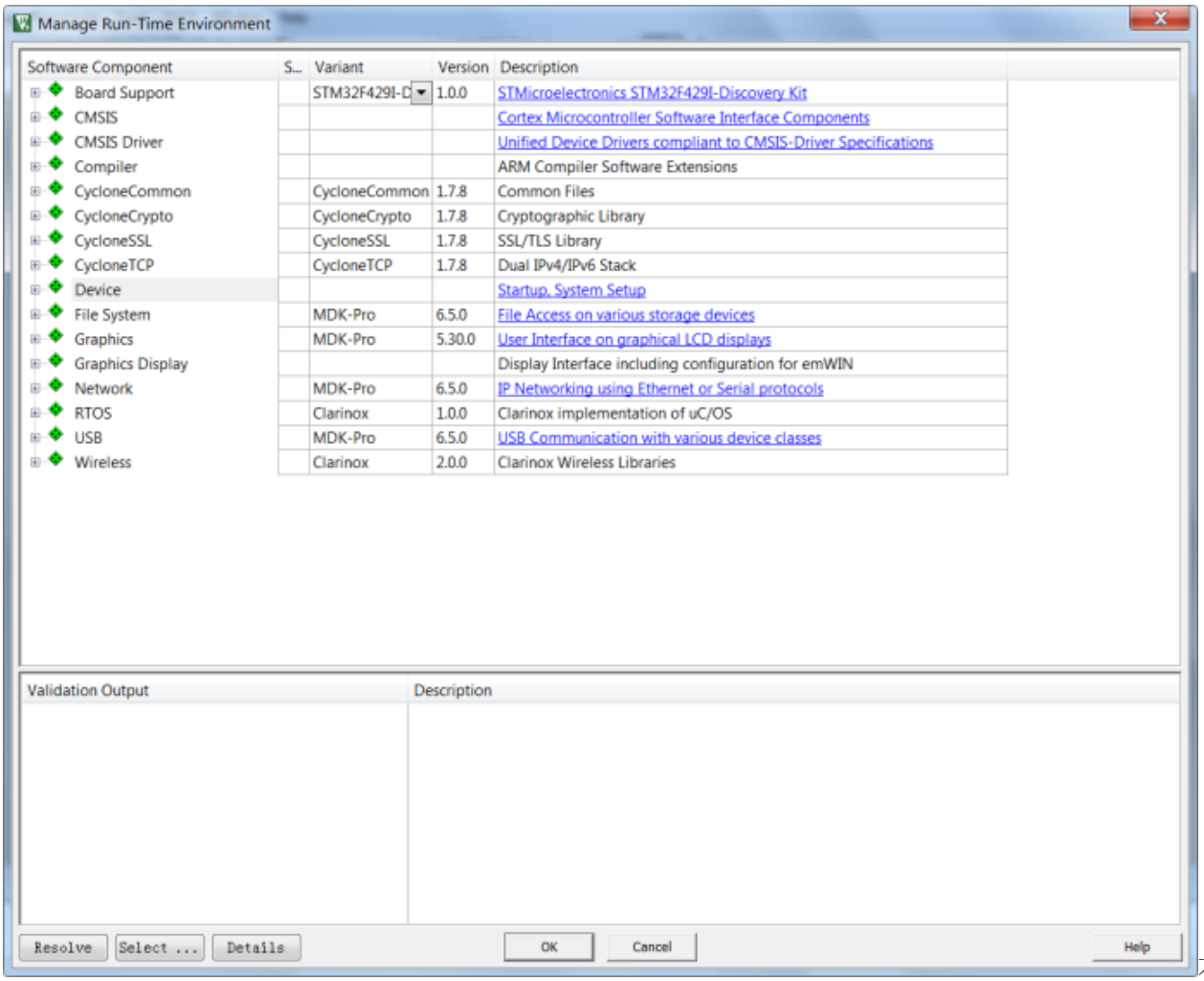


创建工程

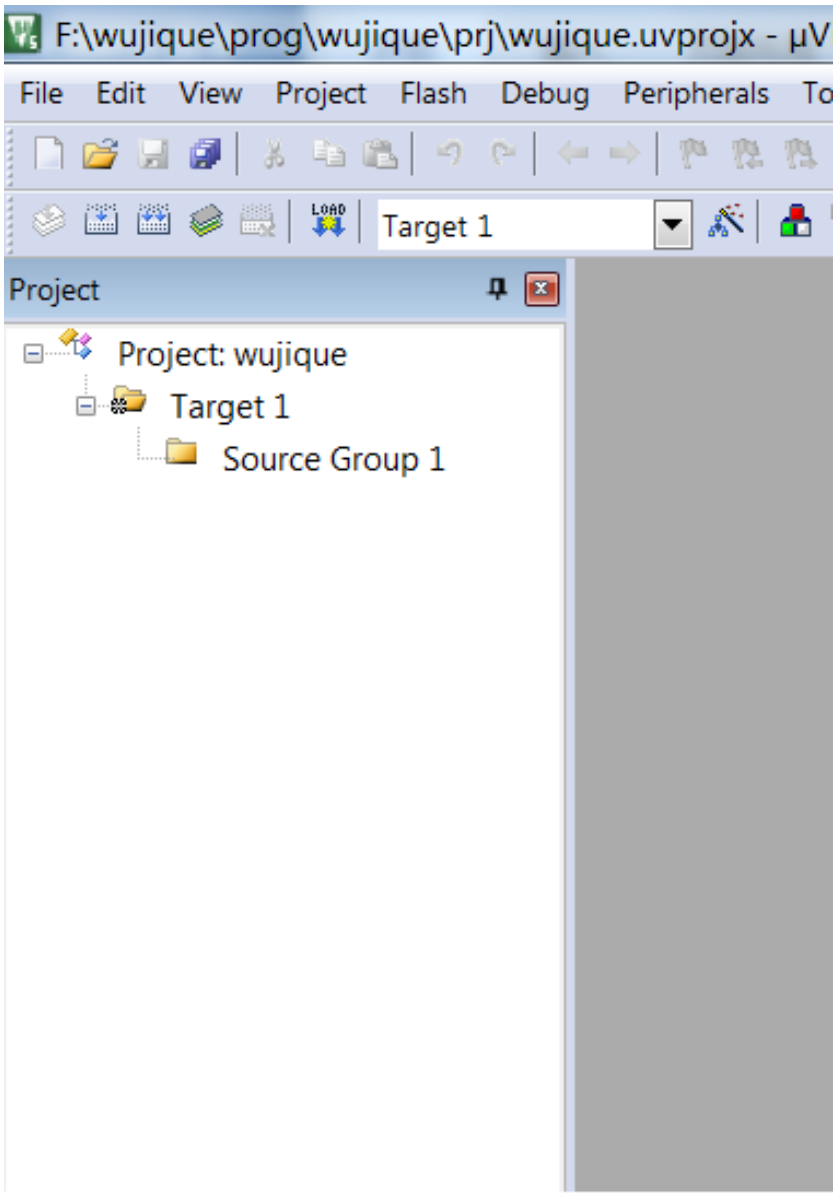
根据实际情况选择选择芯片，我们使用 407 系列，Z 系列，是 ZGT。



选择芯片点击 OK 后弹出一个组件选择界面，我们暂时不选择组件

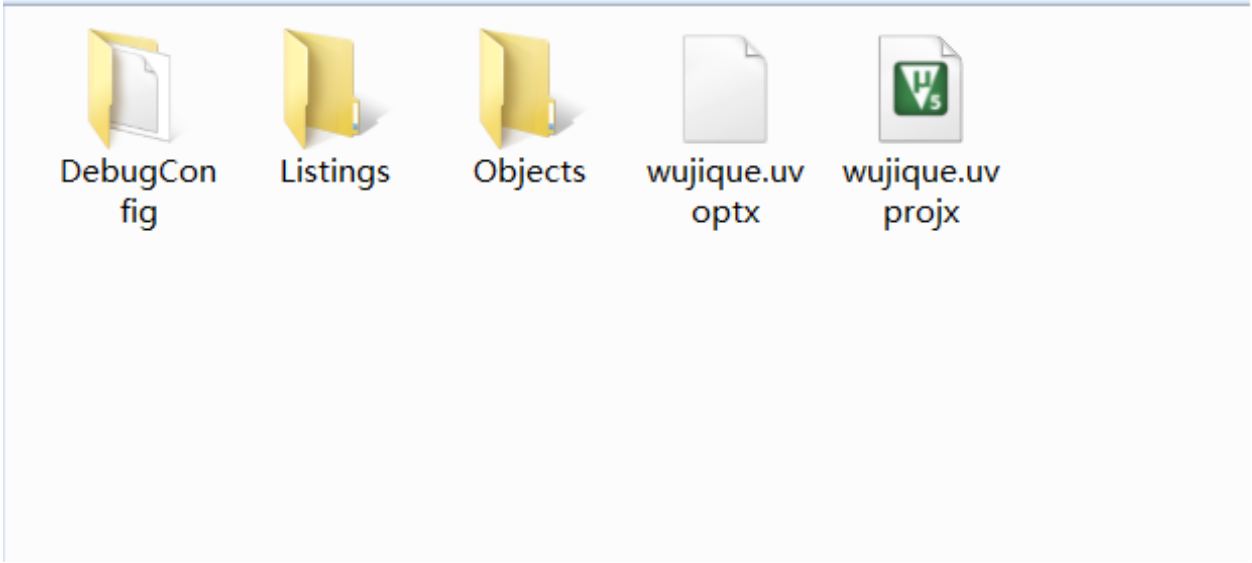

Manage Run-Time Environment
Software Component
S...
Variant
Version
Description
Board Support
STM32F429I-D
1.0.0
STMicroelectronics STM32F429I-Discovery Kit
CMSIS
Cortex Microcontroller Software Interface Components
CMSIS Driver
Unified Device Drivers compliant to CMSIS-Driver Specifications
Compiler
ARM Compiler Software Extensions
CycloneCommon
CycloneCommon
1.7.8
Common Files
CycloneCrypto
CycloneCrypto
1.7.8
Cryptographic Library
CycloneSSL
CycloneSSL
1.7.8
SSL/TLS Library
CycloneTCP
CycloneTCP
1.7.8
Dual IPv4/IPv6 Stack
Device
Startup, System Setup
File System
MDK-Pro
6.5.0
File Access on various storage devices
Graphics
MDK-Pro
5.30.0
User Interface on graphical LCD displays
Graphics Display
Display Interface including configuration for emWIN
Network
MDK-Pro
6.5.0
IP Networking using Ethernet or Serial protocols
RTOS
Clarinox
1.0.0
Clarinox implementation of uC/OS
USB
MDK-Pro
6.5.0
USB Communication with various device classes
Wireless
Clarinox
2.0.0
Clarinox Wireless Libraries
Validation Output
Description
Resolve
Select ...
Details
OK
Cancel
Help

不

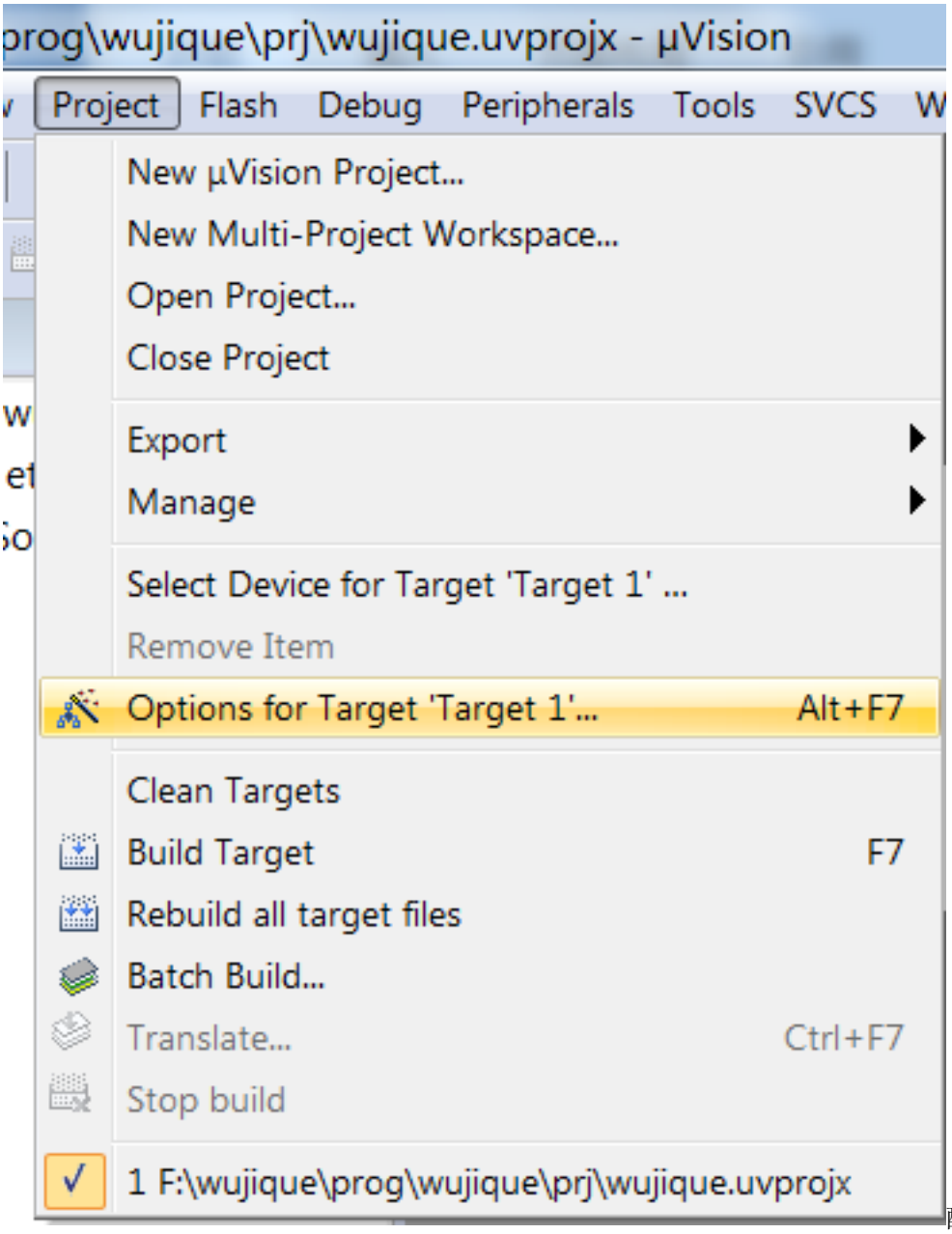


选择组件点击 OK，工程如图，左边就是工程文件目录

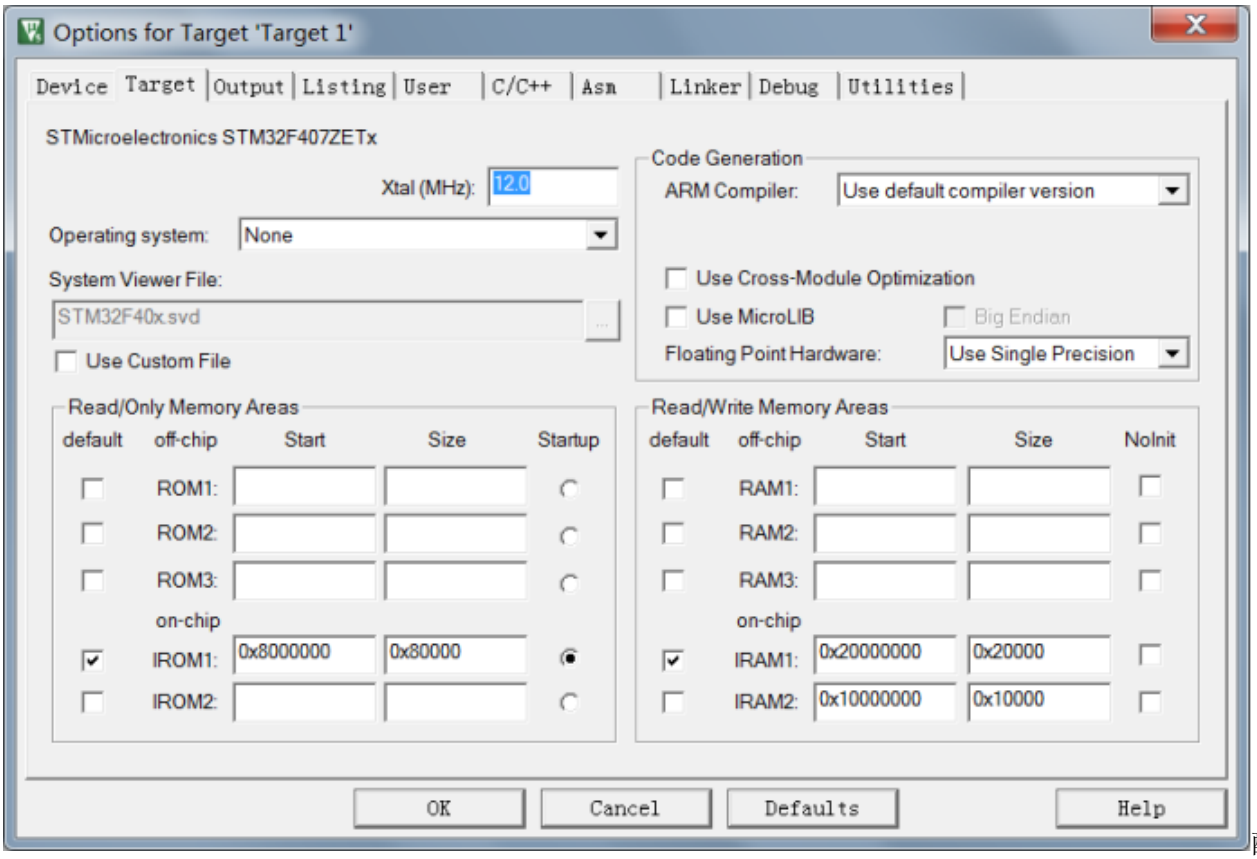
程打开 prj 文件夹，内容如下，.uvprojx 后缀的文件就是工程文件，以后可以直接双击这个文件打开工程。



工程文件夹建立好的工程，还需要进行一些个性化配置。通过 **Project** 下的 **Option** 菜



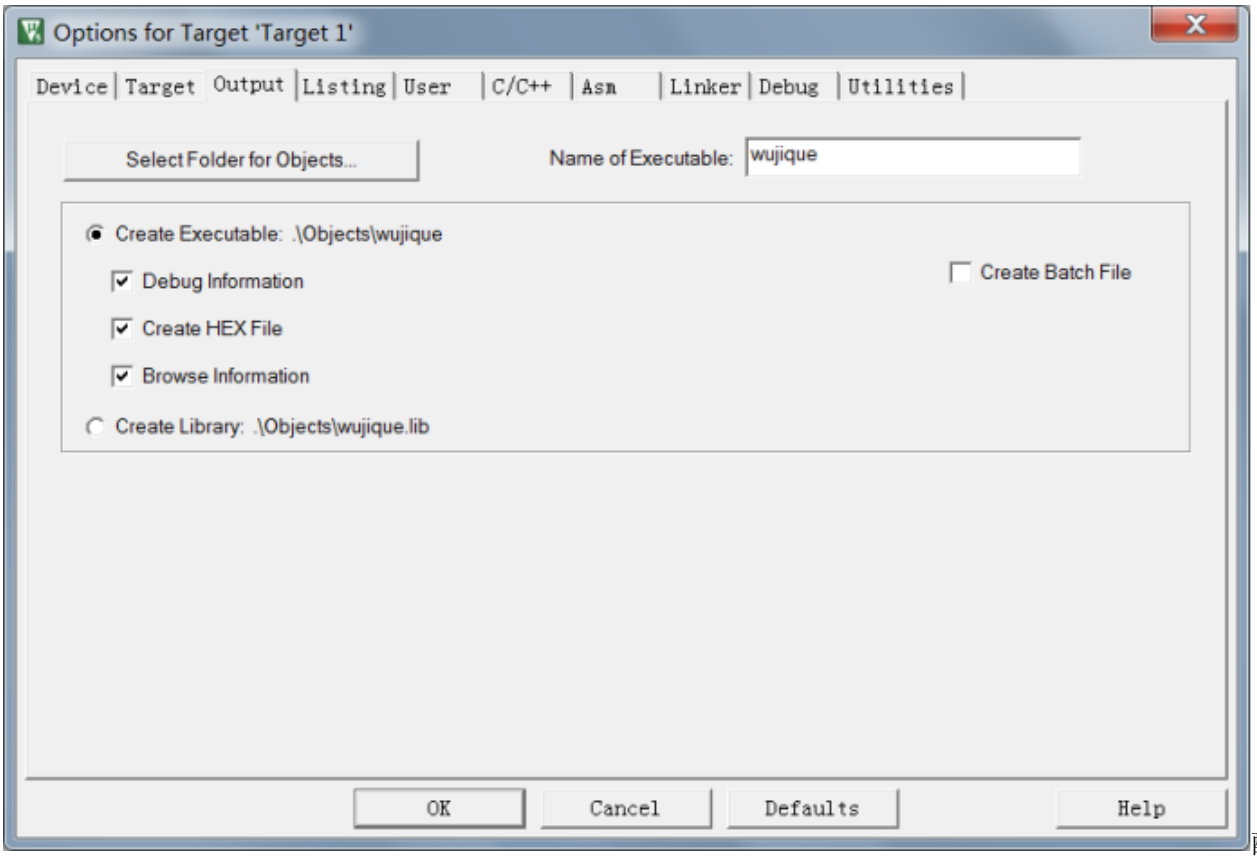
单配置。配置工程弹出配置界面，点击 **Target 标签**：晶振原来是 12，改为 8，也就是 8MHZ 外部晶振。底下的片上 FLASH（IROM1）跟 RAM(IRAM1) 暂时使用默认的，后续需要我们再修改。左边的 **Use MicroLIB** 在某些情况下也会用到，当前暂时不勾。



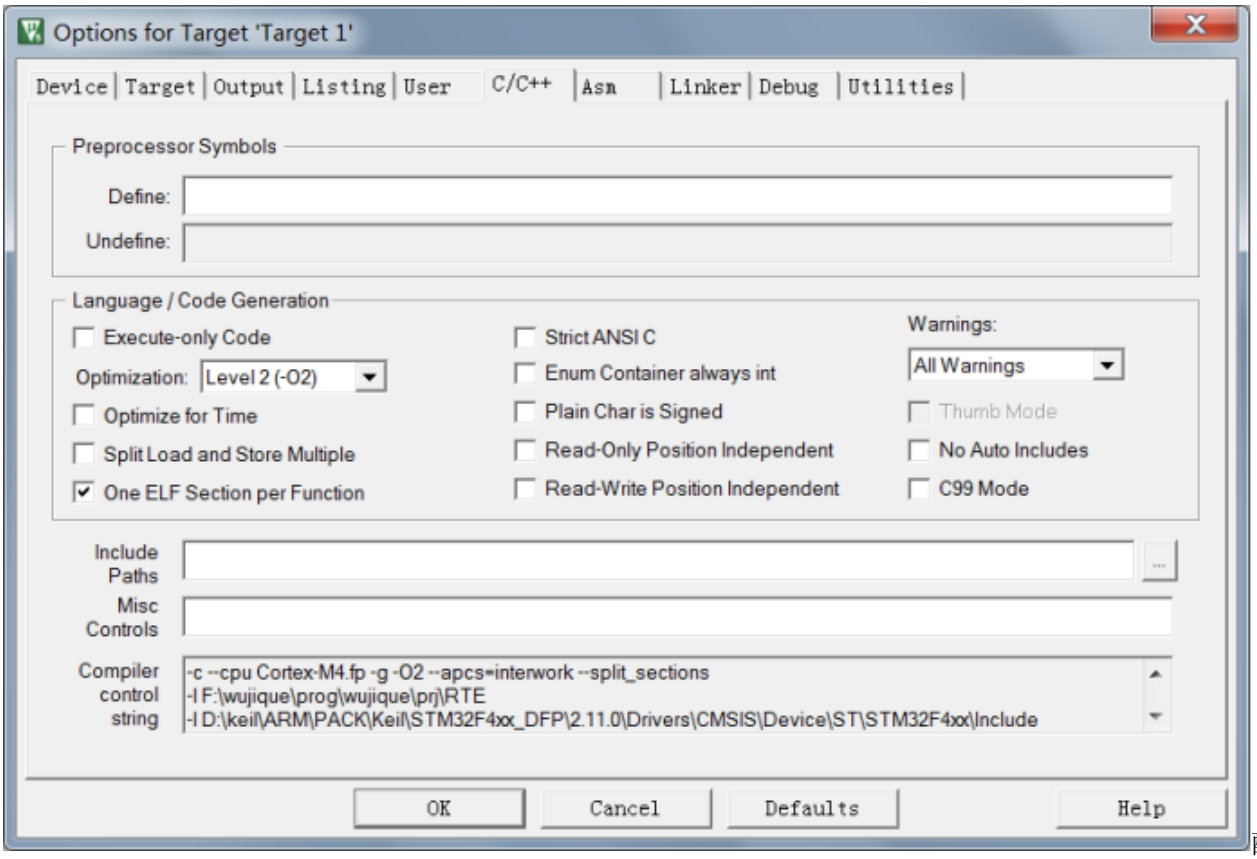
配置芯片

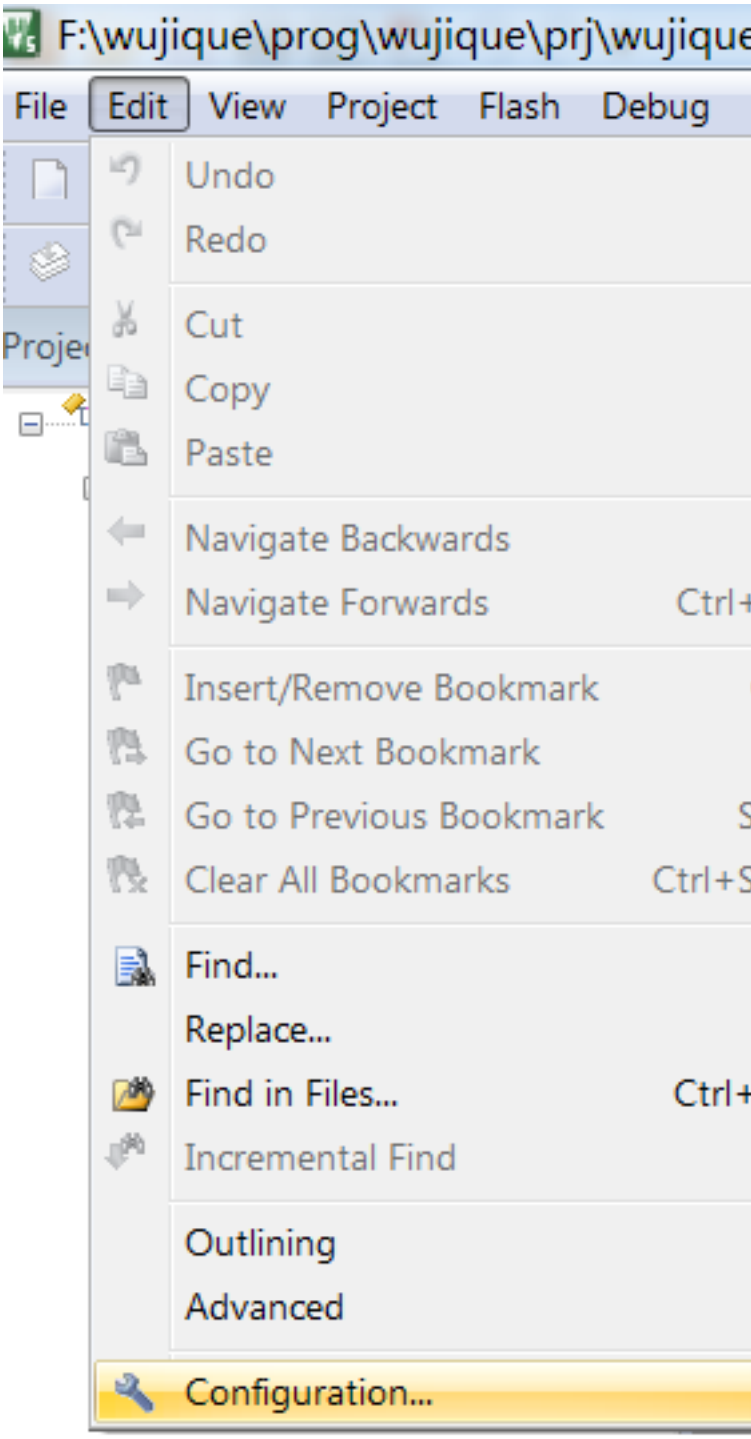
LIB，库文件。在源码中，我们经常看到 #include <stdio.h> 等语句，这就是包含库文件的意思。库文件是一些通用的，标准的功能函数，例如一些常见的数学方法。MicroLIB 是一个精简的库，当空间不够时可以选择使用。但是相对标准的完整库，少很多功能。在做一些大模块移植，例如二维码解码库时，可能会编译不过。更多细节后续慢慢了解。

点击 **Output 标签**，勾上 Create HEX File，也就是编译后生成 HEX 文件



配置输出格式点击 **C/C++ 标签**配置好代码优化等级 Level1(-O1)。**优化等级一定要在工程一开始就设置，如果在开发中途修改优化等级，所有代码必须重新测试，修改优化等级会造成很多莫名奇妙的问题**。底部 Include Paths 是头文件路径，后续添加文件夹时需要在这里添加对应目录路径，否则编译会出现错误。


Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Preprocessor Symbols
Define:
Undefine:
Language / Code Generation
Execute-only Code
Optimization: Level 2 (-O2)
Optimize for Time
Split Load and Store Multiple
One ELF Section per Function
Strict ANSI C
Enum Container always int
Plain Char is Signed
Read-Only Position Independent
Read-Write Position Independent
Warnings:
All Warnings
Thumb Mode
No Auto Includes
C99 Mode
Include Paths
Misc Controls
Compiler control string
-c --cpu Cortex-M4.fp -g -O2 --apcs=interwork --split_sections
-I F:\wujique\prog\wujique\prj\RTE
-I D:\keil\ARM\PACK\Keil\STM32F4xx_DFP\2.11.0\Drivers\CMSIS\Device\ST\STM32F4xx\Include
OK
Cancel
Defaults
Help
配



置头文件配置完 Option 后，设置 **Edit 下的 Configuration**

置 CONFIG 主要是把右边中间 File&Project Handling 下的 Automatic reload of externally modified files 勾上，意思就是当文件被其他编辑器修改的时候，自动加载最新的。为啥要勾上这个呢？因为基本上所有的 IDE 编辑功能都不好用，编写代码我们将使用另外一个软件。

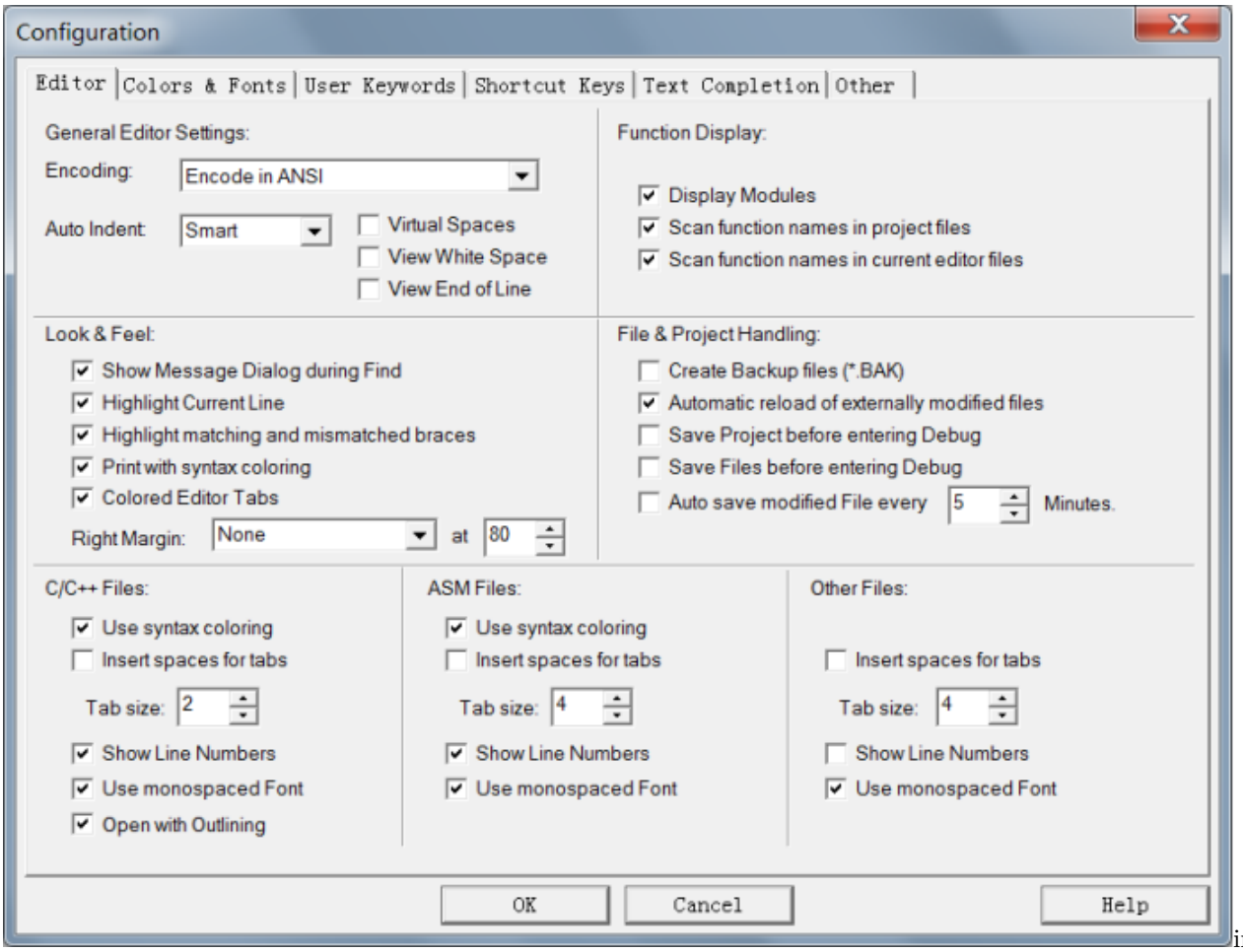


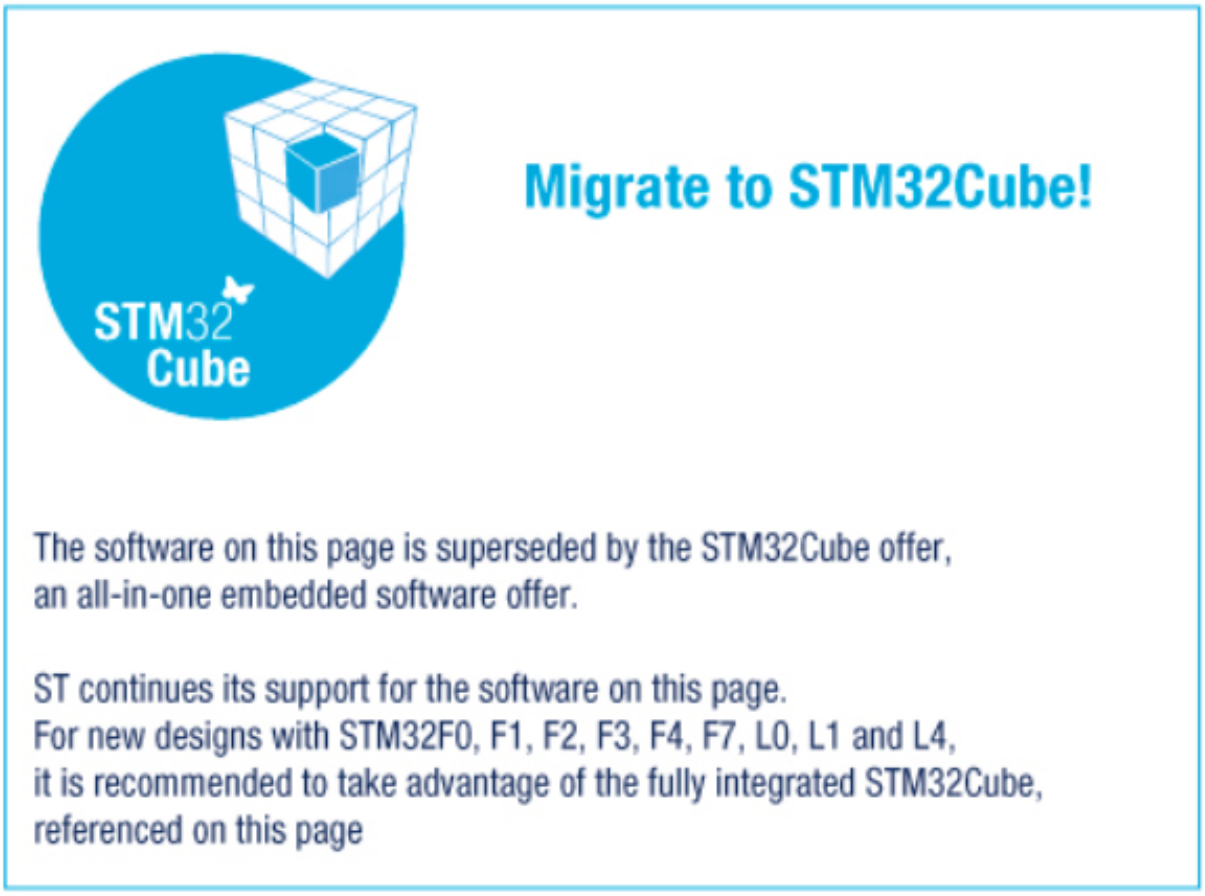
image_1bucn6ck

149.4kB 到此为止，编译工具 MDK 基本配置完成。

7.2 STM32 标准库

为了大力推广 STM32，ST 公司编写了 STM32 芯片的标准库。以前，我们使用一个芯片，都是自己控制芯片的寄存器。非常繁琐，ST 推出标准库后，解放了码农，不再需要关心芯片底层寄存器，更加专注于上层驱动和应用开发。**不推荐大家再使用寄存器的方式写程序，如果要学习操作寄存器，直接看 ST 的库吧。**

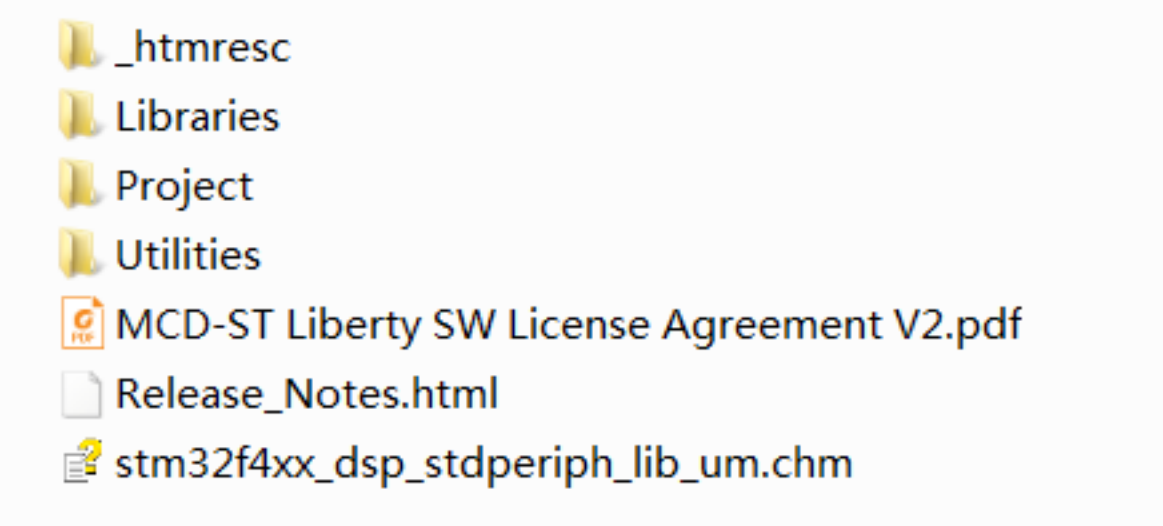
ST 以前推行标准库，现在推 HAL 库跟 LL 库，新出的型号已经不再支持标准库。



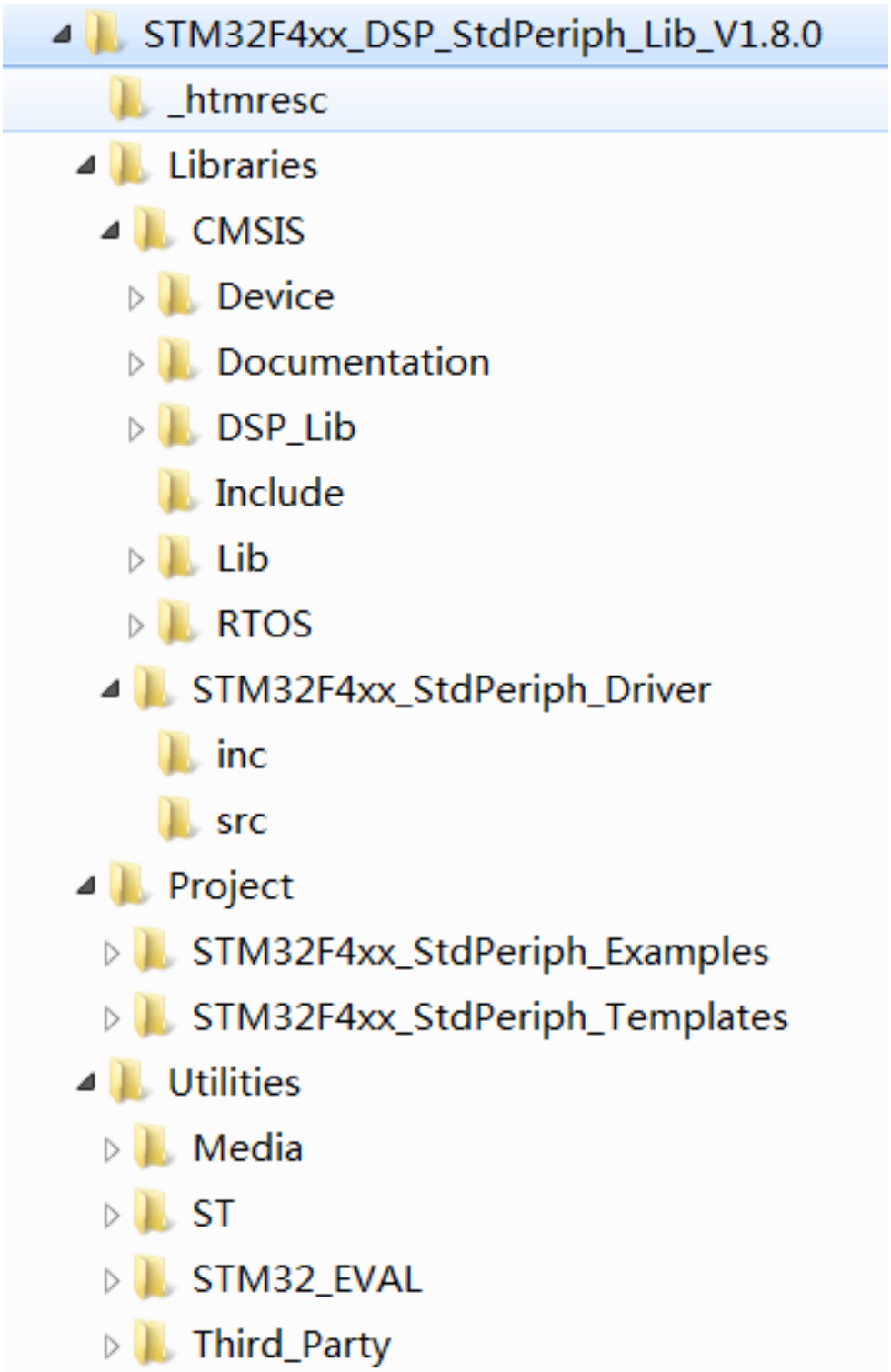
推荐 CUBE 三种库之间的关系请参考：http://blog.csdn.net/zcshoucsdn/article/details/54613202 针对 F407，我们依然使用标准库进行开发。后续 429、726 等高级型号再考虑使用 HAL 库。用标准库的一个原因是我们后面可能会转到 *GCC* 环境编译

官方库地址： http://www.st.com/content/st_com/zh/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32-standard-peripheral-libraries/stsw-stm32065.html 官网下不了，可以从 STMCU 下：http://www.stmcu.org/document/detail/index/id-213641

下载后得到一个压缩文件 en.stm32f4_dsp_stdperiph_lib.zip，解压后目录如下

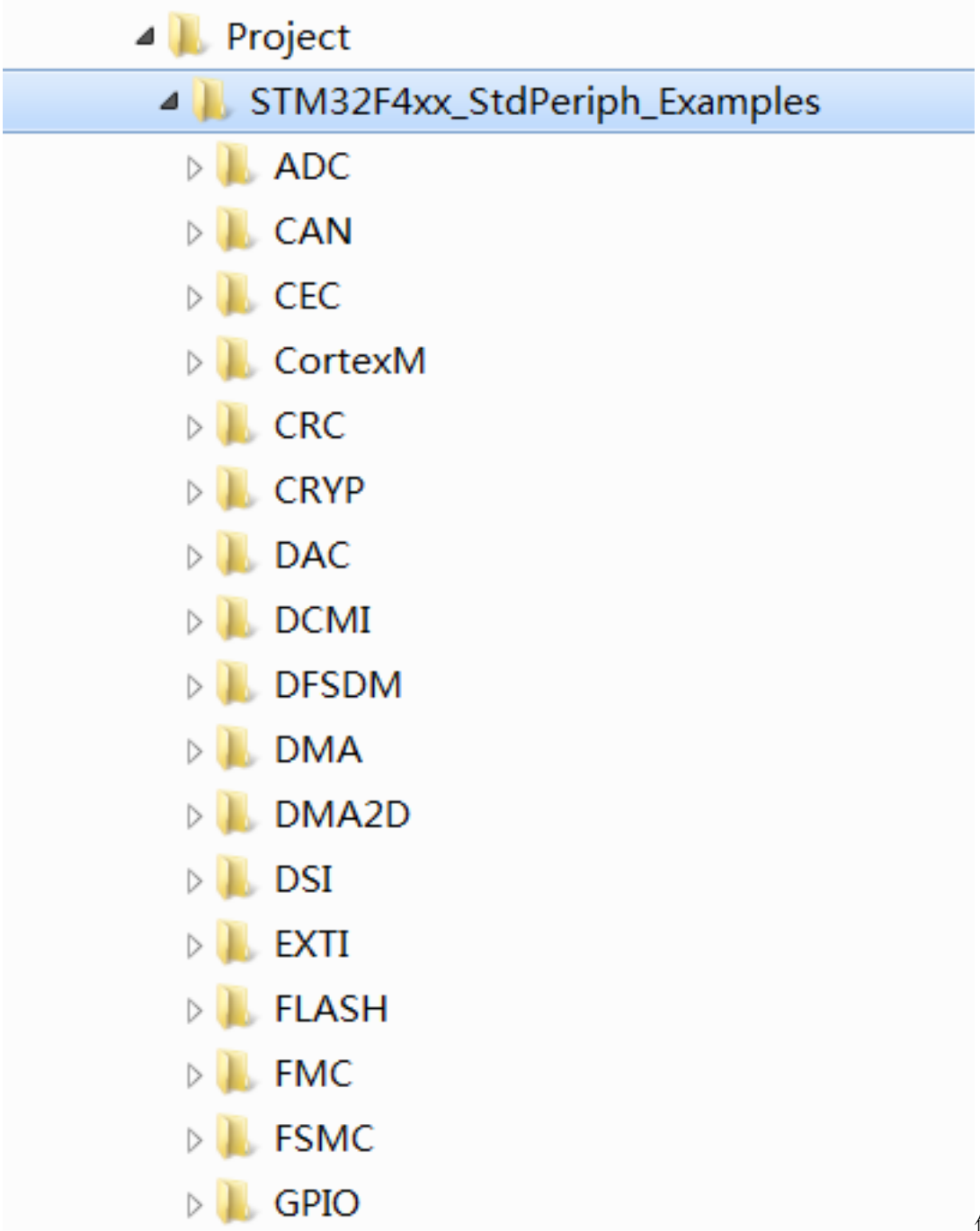

_htmresc
Libraries
Project
Utilities
MCD-ST Liberty SW License Agreement V2.pdf
Release_Notes.html
stm32f4xx_dsp_stdperiph_lib_um.chm

库解压



目录目录树如下：目录树 B

如何使用标准库，请参考官方文档。除了库代码，更重要的是里面的 Examples 例程。里面的例程可以说是非常全面。在开发过程中可以参考。特别是 USB，ETH 等外设的例程。**参考官方例程，是软件开发过程的一个**

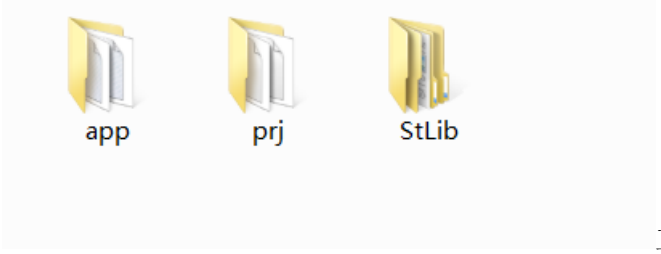


重要手段。例程

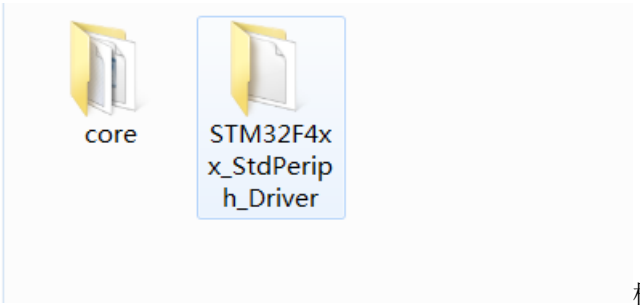
7.3 工程模板

前面我们已经建立了一个工程，但是到目前为止，这个工程还是一个空工程。下面我们将 ST 的库添加到工程中。

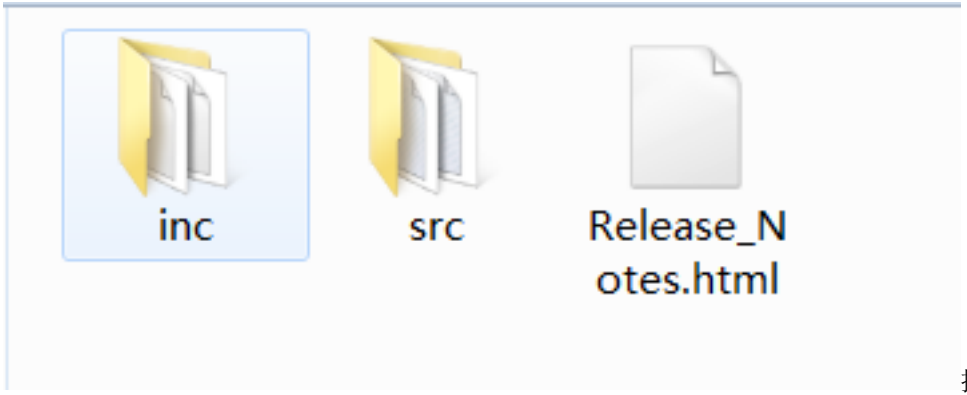
1. 建立文件夹 StLib，专门用于保存 ST 官方提供的库，目前我们使用了**标准外设库**，后面还会使用 **USB**



库，网络 ETH 库。 工程模板文件夹



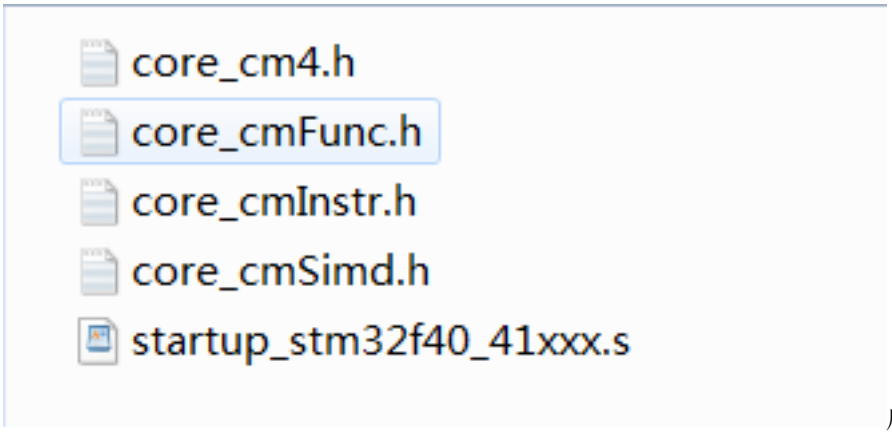
2. 在 StLib 内建立文件夹 core 标准库文件夹
3. 将 STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Libraries\STM32F4xx_StdPeriph_Driver 文件夹拷贝



到 StLib 目录下 拷贝库文件 B

4. STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm 内的启动代码 startup_stm32f40_41xxx.s 拷贝到 StLib\core
5. 从 STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Libraries\CMSIS\Include 内拷贝四个头文件到 StLib\core

   core_cm4.h core_cmFunc.h core_cmInstr.h core_cmSimd.h



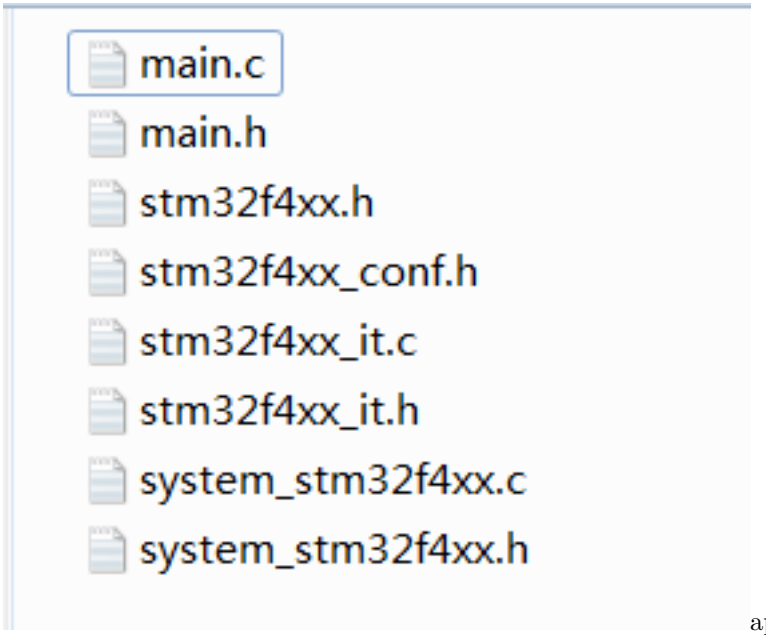
StLib\core 文件夹如下：启动代码等

1. 从 STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Libraries\CMSIS\Device\ST\STM32F4xx\Include 拷贝两个头文件到 app 文件夹

   stm32f4xx.h system_stm32f4xx.h

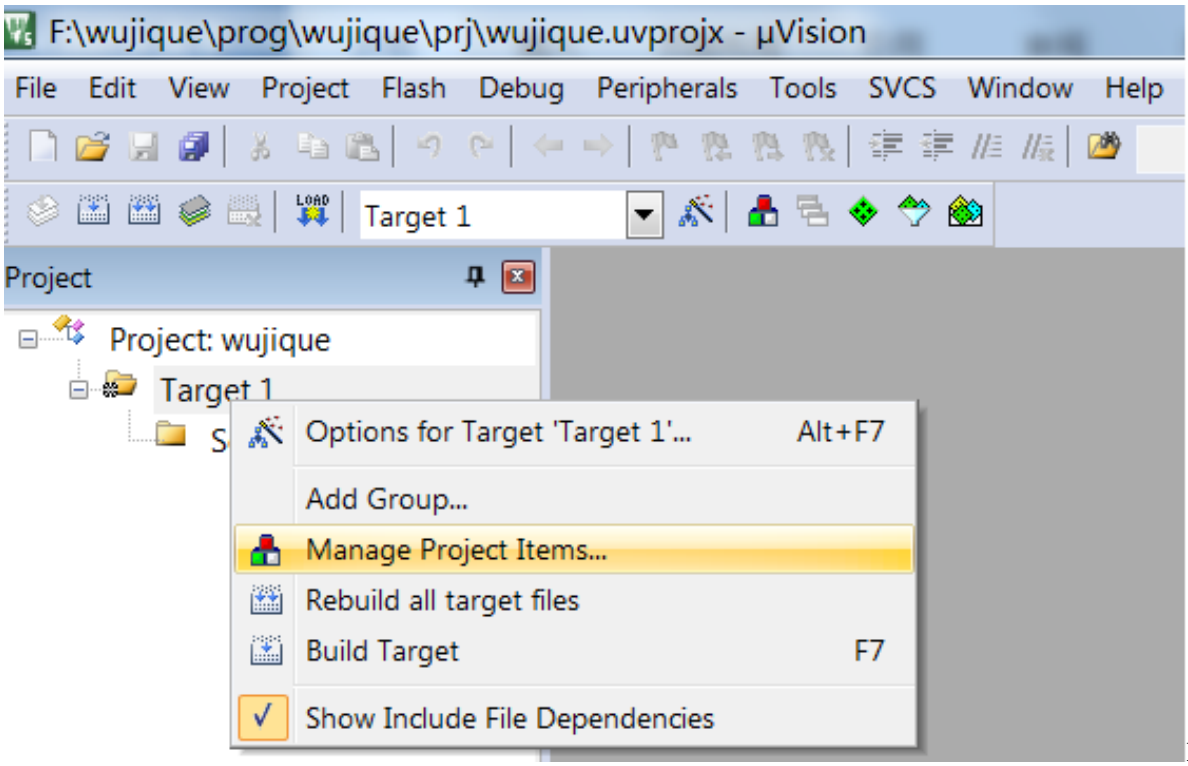
1. 从 STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Project\STM32F4xx_StdPeriph_Templates 拷贝以下文件到 app 文件夹

   main.c main.h stm32f4xx_conf.h stm32f4xx_it.c stm32f4xx_it.h system_stm32f4xx.c



拷贝完成后 app 目录如下 app 目录

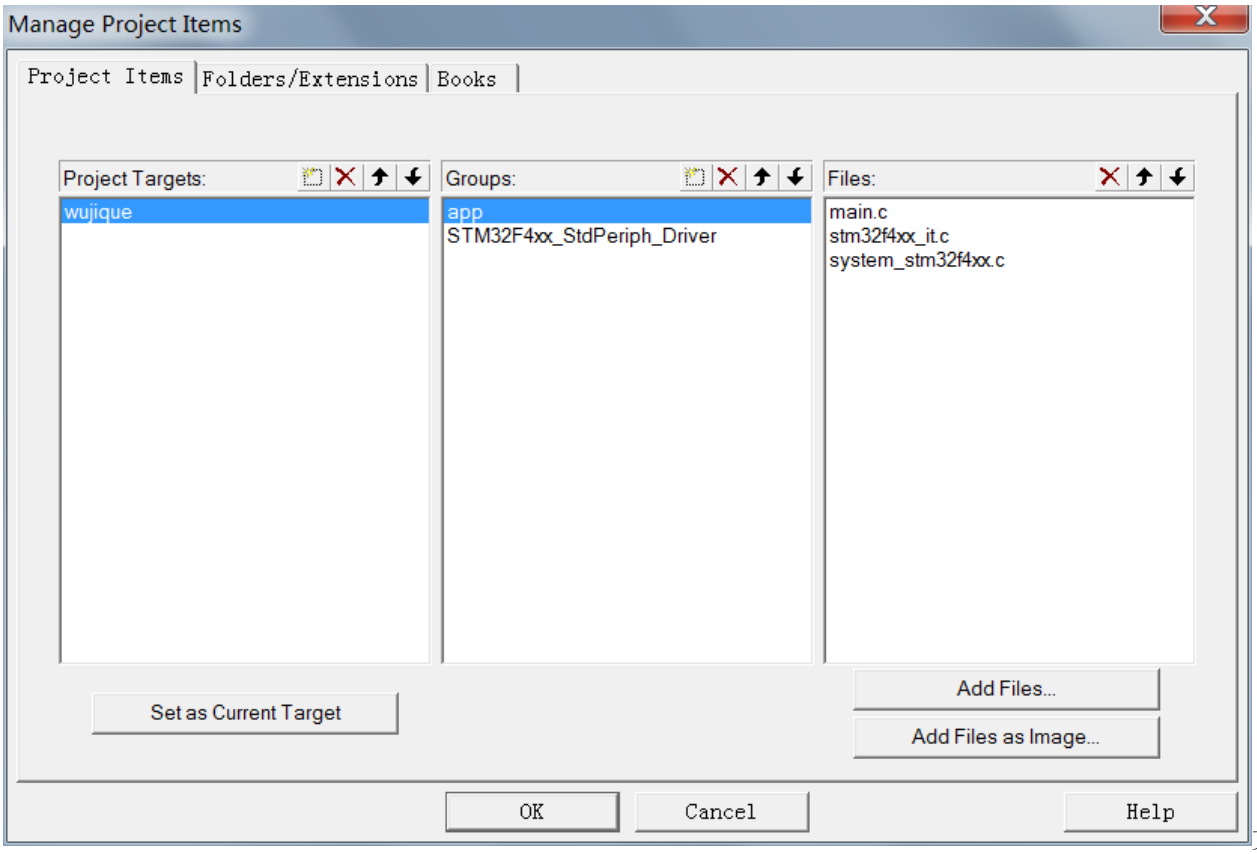
1. 将文件添加到工程右键点击 Target1 进入 Manage Project Items



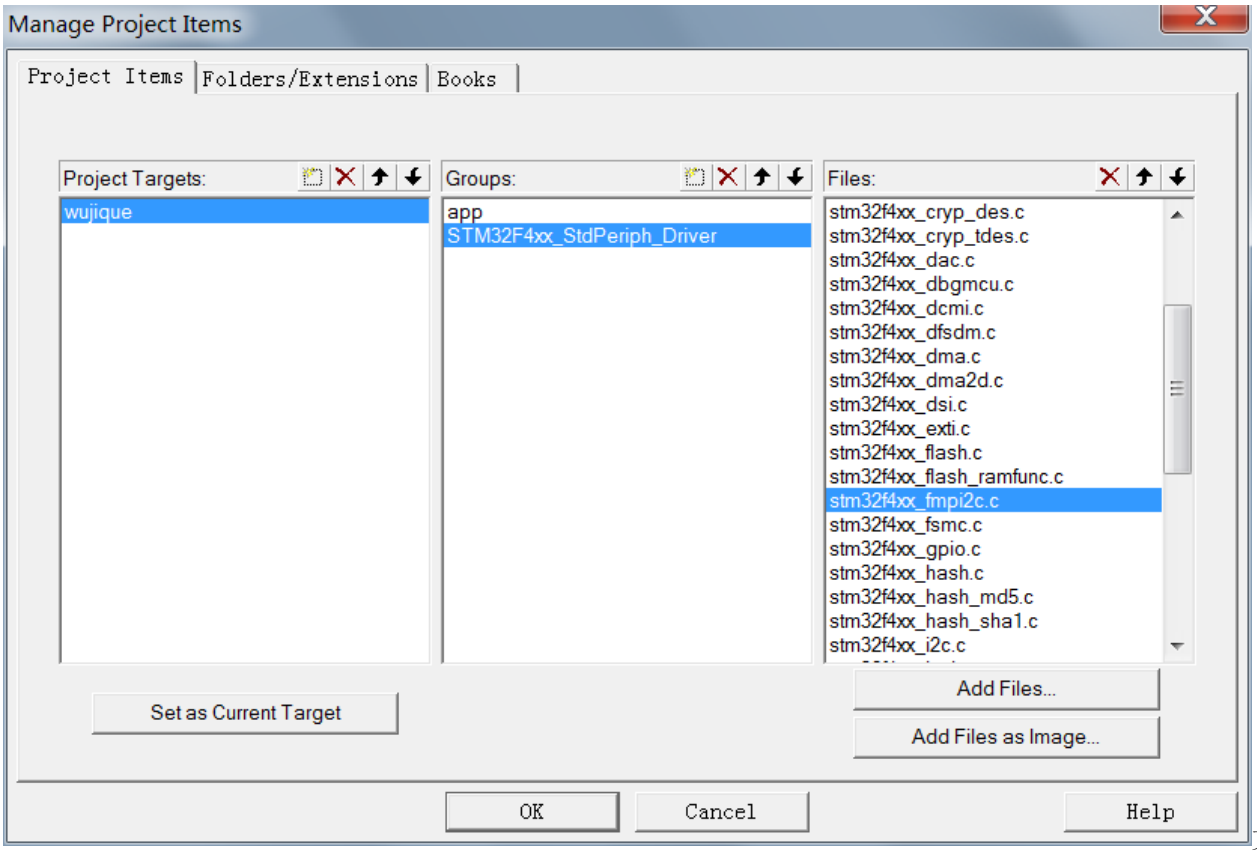
MDK

工程管理

修改文件夹结构，对 Groups 命名，并且将源文件添加到对应 Groups。（HAL 库里面有一个 stm32f4xx_fmc.c 文件是 STM32F42&STM32F43 系列才需要，F407 不需要添加）

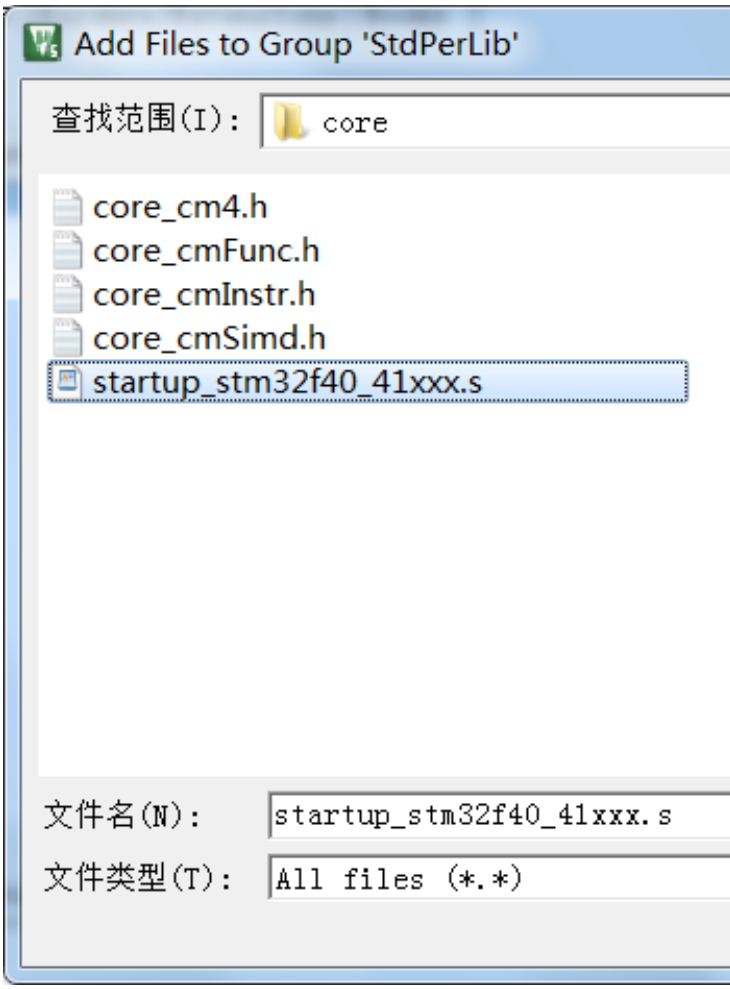

Manage Project Items
Project Items
Folders/Extensions
Books
Project Targets:
wujique
Groups:
app
STM32F4xx_StdPeriph_Driver
Files:
main.c
stm32f4xx_it.c
system_stm32f4xx.c
Set as Current Target
Add Files...
Add Files as Image...
OK
Cancel
Help

文件结构 1

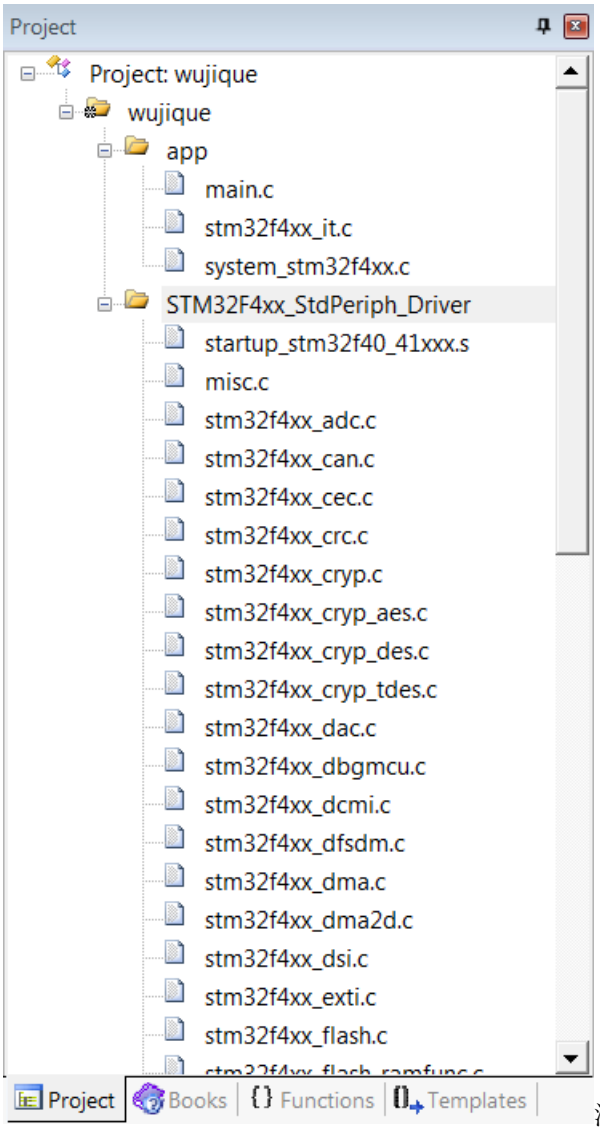


文件结构添加库文件

mdk 的源码管理比较弱，只能做一层 Groups，无法做成树状的文件目录管理。IAR 可以。

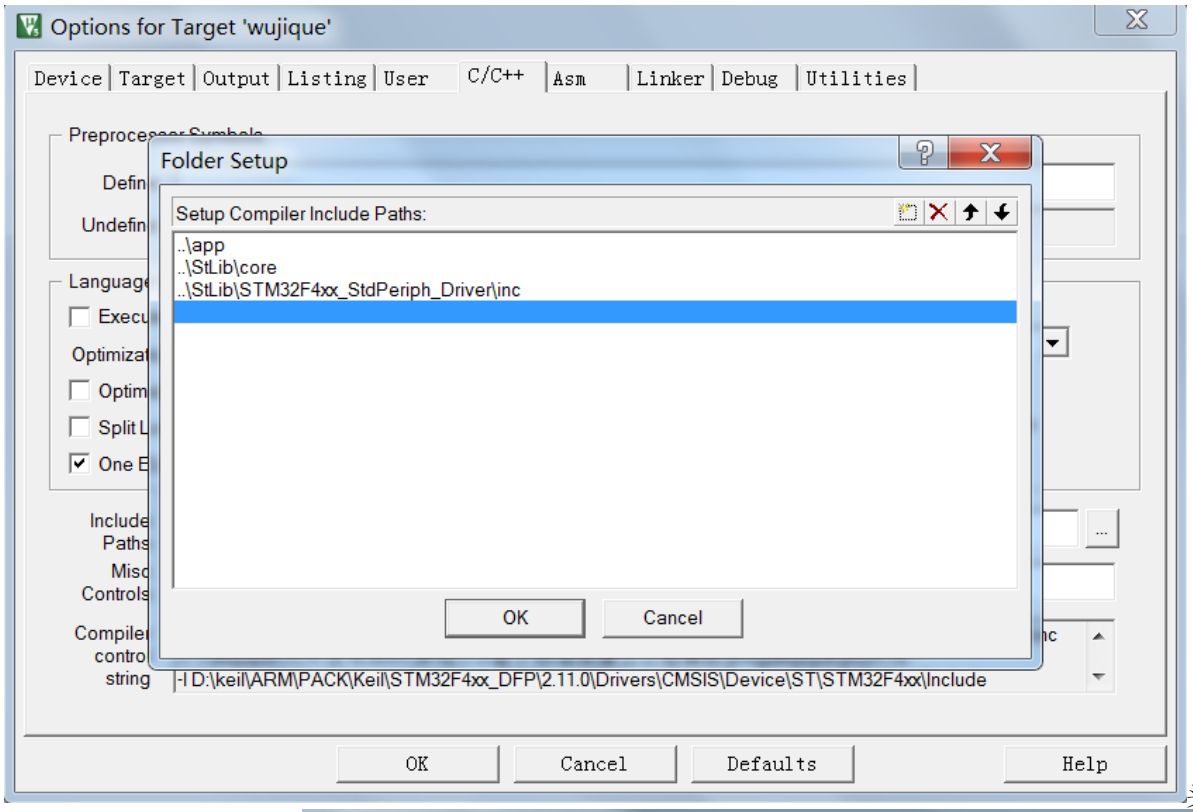


启动代码是汇编文件，添加时要注意文件类型要选择 ALL files 意添加汇编代码

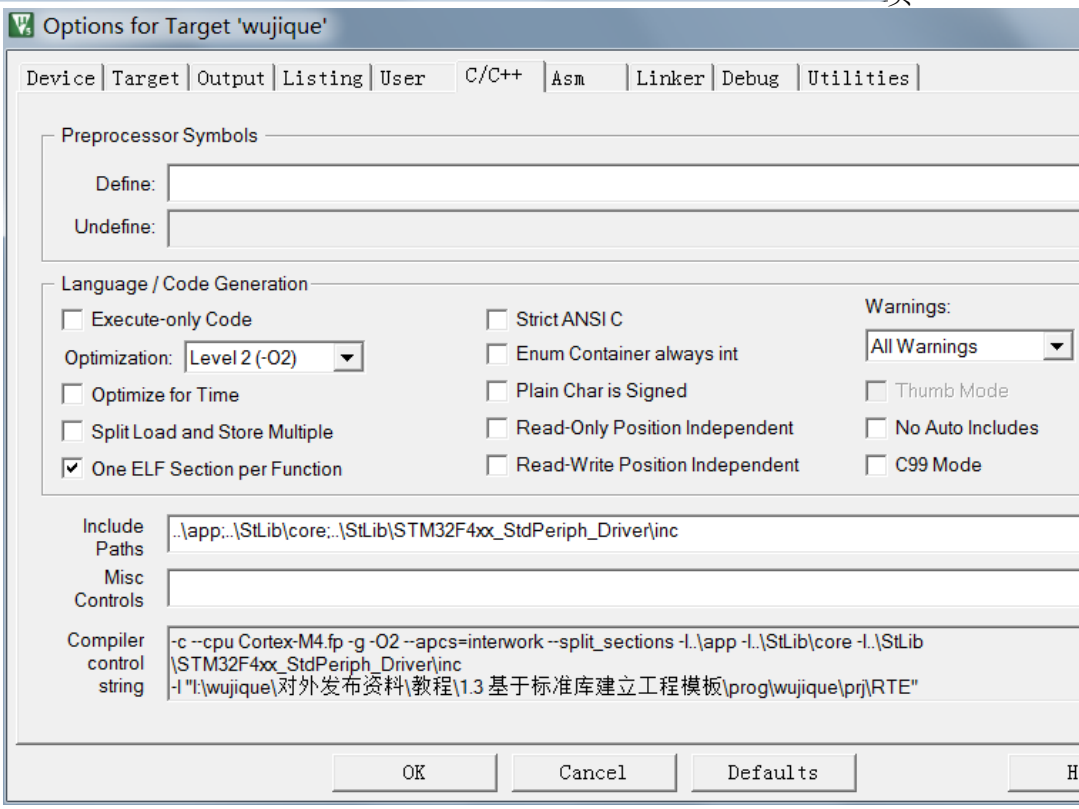


1. 添加完成后工程如下 添加完成后工程目录

2. 设置头文件路径在 option 中的 C/C++ 菜单下添加头文件路径。



头



文件路径添加完成后如下

加头文件路径后

3. 按 F7 进行编译提示错误，没定义芯片型号，根据错误提示在 stm32f4xx.h 文件夹内定义宏

*** Using Compiler 'V5.06 update 1 (build 61)', folder: 'D:\keil\ARM\ARMCC\Bin' Build target 'wujique' compiling stm32f4xx_it.c···..\app\stm32f4xx.h(124): error: #35: #error directive: "Please select first the target STM32F4xx device used in your application (in stm32f4xx.h file)" #error "Please select first the target STM32F4xx device used in your application (in stm32f4xx.h file)" ..\app\stm32f4xx_it.c: 0 warnings, 1 error

很多教程将这个宏定义在 MDK 内，个人不建议这样做，我认为所有定义应该都尽量定义在源文件内，以便后续移植修改。那天我们要讲代码换一个编译器，例如 IAR 时，可以避免出错

左键双击错误，源码窗口弹到错误的地方 120 行，可以看到，原因是我们没有定义芯片。

```
#if !defined(STM32F40_41xxx) && !defined(STM32F427_437xx) && !defined(STM32F429_439xx) &&
↪ !defined(STM32F401xx) && !defined(STM32F410xx) && \
    !defined(STM32F411xE) && !defined(STM32F412xG) && !defined(STM32F413_423xx) && !
↪defined(STM32F446xx) && !defined(STM32F469_479xx)
 #error "Please select first the target STM32F4xx device used in your application (in␣
↪stm32f4xx.h file)"
#endif /* STM32F40_41xxx && STM32F427_437xx && STM32F429_439xx && STM32F401xx &&␣
↪STM32F410xx && STM32F411xE && STM32F412xG && STM32F413_23xx && STM32F446xx &&␣
↪STM32F469_479xx */
```

往上拉，从 68 行开始看，找到对应型号，打开对应型号前面的宏。我们使用的型号是 STM32F407ZG，我们就打开 #define STM32F40_41xxx 这个宏。

```
#if !defined(STM32F40_41xxx) && !defined(STM32F427_437xx) && !defined(STM32F429_439xx) &&
↪ !defined(STM32F401xx) && !defined(STM32F410xx) && \
    !defined(STM32F411xE) && !defined(STM32F412xG) && !defined(STM32F413_423xx) && !
↪defined(STM32F446xx) && !defined(STM32F469_479xx)
  #define STM32F40_41xxx    /*!< STM32F405RG, STM32F405VG, STM32F405ZG, STM32F415RG,␣
↪STM32F415VG, STM32F415ZG,
                                STM32F407VG, STM32F407VE, STM32F407ZG, STM32F407ZE,
↪ STM32F407IG, STM32F407IE,
                                STM32F417VG, STM32F417VE, STM32F417ZG, STM32F417ZE,
↪ STM32F417IG and STM32F417IE Devices */
```

因为我们使用了标准外设库，我们还需要打开 131 行的宏

```
#if !defined  (USE_STDPERIPH_DRIVER)
/**
 * @brief Comment the line below if you will not use the peripherals drivers.
   In this case, these drivers will not be included and the application code will
   be based on direct access to peripherals registers
   */
```

(continues on next page)

(continued from previous page)

```
  #define USE_STDPERIPH_DRIVER
#endif /* USE_STDPERIPH_DRIVER */
```

重新编译，DONE，编译成功，标准库模板完成。

compiling stm32f4xx_spi.c…compiling stm32f4xx_tim.c…compiling stm32f4xx_usart.c…compiling stm32f4xx_wwdg.c…linking…Program Size: Code=1188 RO-data=424 RW-data=20 ZI-data=1652FromELF: creating hex file…".\Objects\wujique.axf"- 0 Error(s), 0 Warning(s). Build Time Elapsed: 00:00:23

7.4 晶振配置

1. 根据硬件，修改外部晶振，在 stm32f4xx.h 中，将原来宏定义 HSE_VALUE ((uint32_t)25000000) 根据实际硬件修改，屋脊雀开发板用的是 8M 晶振，改为 HSE_VALUE ((uint32_t)8000000)

```
/**
 * @brief In the following line adjust the value of External High Speed oscillator (HSE)
   used in your application

   Tip: To avoid modifying this file each time you need to use different HSE, you
        can define the HSE value in your toolchain compiler preprocessor.
  */
#if defined(STM32F40_41xxx) || defined(STM32F427_437xx)  || defined(STM32F429_439xx) ||␣
↪defined(STM32F401xx) || \
    defined(STM32F410xx) || defined(STM32F411xE) || defined(STM32F469_479xx)
 #if !defined  (HSE_VALUE)
  #define HSE_VALUE    ((uint32_t)8000000) /*!< Value of the External oscillator in Hz */
 #endif /* HSE_VALUE */
#elif defined (STM32F412xG) || defined(STM32F413_423xx) || defined(STM32F446xx)
 #if !defined  (HSE_VALUE)
  #define HSE_VALUE    ((uint32_t)8000000) /*!< Value of the External oscillator in Hz */
 #endif /* HSE_VALUE */
#endif /* STM32F40_41xxx || STM32F427_437xx || STM32F429_439xx || STM32F401xx ||␣
↪STM32F411xE || STM32F469_479xx */
```

1. 修改 PLL 配置在 system_stm32f4xx.c 文件中，371 行，原来 #define PLL_M 25，修改为 #define PLL_M 8，关于时钟的详细配置，后续章节再做介绍。

```
/************************ PLL Parameters *************************************/
#if defined(STM32F40_41xxx) || defined(STM32F427_437xx) || defined(STM32F429_439xx) ||␣
↪defined(STM32F401xx) || defined(STM32F469_479xx)
```

(continues on next page)

(continued from previous page)

```
 /* PLL_VCO = (HSE_VALUE or HSI_VALUE / PLL_M) * PLL_N */
 #define PLL_M      8
#elif defined(STM32F412xG) || defined(STM32F413_423xx) || defined (STM32F446xx)
 #define PLL_M      8
#elif defined (STM32F410xx) || defined (STM32F411xE)
 #if defined(USE_HSE_BYPASS)
  #define PLL_M      8
 #else /* !USE_HSE_BYPASS */
  #define PLL_M      16
 #endif /* USE_HSE_BYPASS */
#else
#endif /* STM32F40_41xxx || STM32F427_437xx || STM32F429_439xx || STM32F401xx ||␣
↪STM32F469_479xx */
```

7.5 结束

到此，STM32F407 开发环境配置完成，正式进入软件开发阶段。后面的教程，将基于本模板，慢慢添加各个外设的驱动。

7.6 end

CHAPTER 8

IO 口输出-流水灯-证明程序在运行

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

在第一部分软硬件准备中，我们通过串口将程序下载到芯片，验证硬件基本正常，但并不知道是程序下载进入是否能正常运行呢。这次我们用一个简单的流水灯，证明程序下载到芯片后可以正常运行。**流水灯**，就像 PC 软件上的 HelloWord 一样，每一个做单片机的工程师都会玩过。什么叫流水灯呢？就是多个 LED 排成一列，做出不同的效果，其中较典型的就是 LED 轮流点亮，像流水一样左右流动。

8.1 IO 口

IO 是 Input/Output 的简称，也即是输出输出的意思。芯片要控制外部器件，或者是从外部获取状态信息，依赖的就是 IO。一个芯片的管脚，除了电源和地，基本上全部都是 IO 口。（有些芯片的部分管脚可能只能做输出或输入中的一种）IO 最基本的功能是**输出高低电平**和**检测外部是高电平还是低电平**，也即是逻辑电路功能。至于其它的串口，SPI 等接口，也是高低电平，只不过利用电平实现了比较复杂的时序，也属于逻辑范畴。除此外，一个管脚还可以做 DAC/ADC 等其它模拟电路的功能。高低电平的电压由**芯片 IO 电压**决定，通常高电平是芯片的工作电压，低电平则是 0V。（**高低电平会有一定识别范围，TTL 电平和 CMOS 电平不一样**）以前的 51 单片机高电平是 5V，目前更多的芯片是 3.3V，还有芯片是 1.8V。

> 一个 IO 的外接电路、外接器件，要考虑两者之间的电压兼容。例如，一个 3.3V 工作电压的单片机，外接一个 5V 的器件，通常，需要使用电平转换电路。

使用一个 IO 口通常需要这样配置：

1. 使能这个 IO 口的时钟。
2. 根据需要配置为输出或者输入
3. 设置 IO 口模式，是否接上下拉电阻。
4. 作为输入，读 IO 状态；作为输出，则设置 IO 口电平。

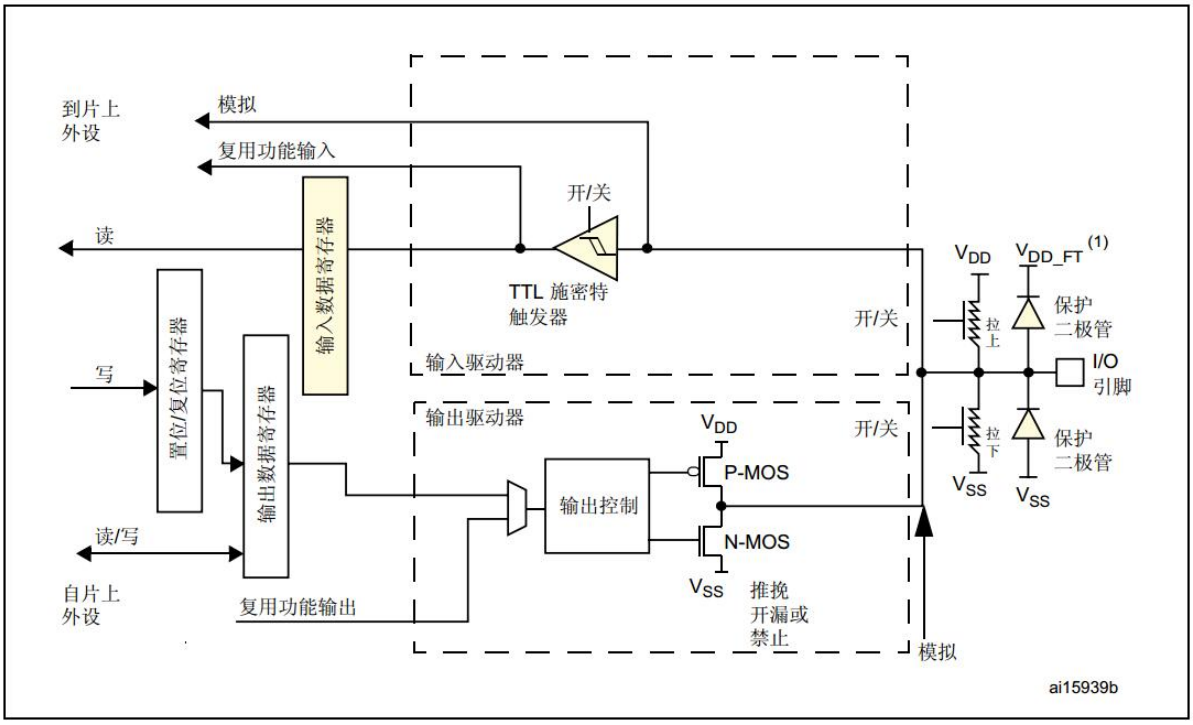
以上是基本的 IO 口操作，不同的芯片会有一点差别。

8.2 STM32 的 IO

STM32 功能强大，IO 配置也较复杂，现在我们先大概看看 IO 结构，更多功能后续慢慢了解。要了解 STM32 的 IO 口，请查阅《STM32F4xx 中文参考手册.pdf》，在第七章，通

用 IO。IO 口参考手册参考手册分 4 小节。在 7.3 节，有下面这个 STM32IO 口的结构图。从这个图，我们可以看到一个 IO 的输入输出通路、各种配置开关的位置。如果你硬件比较好，还可以看到这个 IO 的输入输出结构是怎么样的。IO 口具体如何设置，我们在例程中再说明。

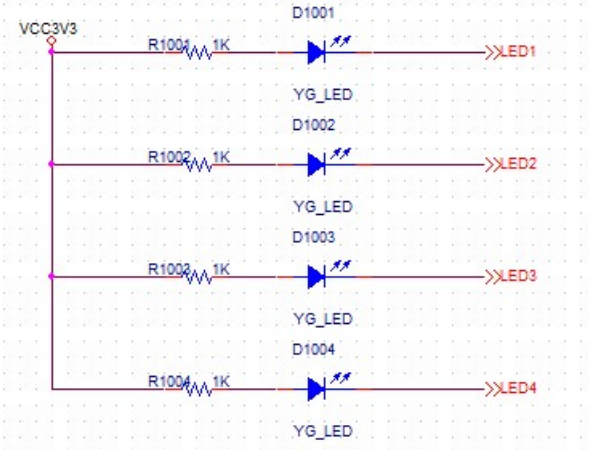
图 17. 5 V 容忍 I/O 端口位的基本结构



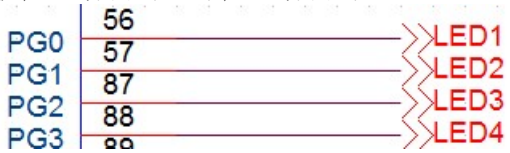
IO 口结构

8.2.1 原理图

一个 IO 口是如何点亮一个 LED 的呢？我们首先看 LED 电路原理图，一个 LED 跟一个电阻串联，一端接到电源，一端接到 IO 口。



LED 原理图



LED 原理图

LED

LED: 发光二极管，是二极管，就有正负极。当在正负极之间流过一定电流时，就能发光。电流越大，亮度越大。

在资料文件夹内有一个发光二极管的规格书。LED 的规格书中有一个很重要的参数。《黄绿 0603 (33_40mcd)_PDF_C2289_2015-07-23.pdf》

3、最大绝对标称值（环境温度=25℃）

顺向电流	I_F	20	mA
顺向峰值电流 *1	I_{FP}	100	mA
反向电压	VR	5	V
焊接温度	Tsol	回流焊：250 ° C，8sec. 手工焊：300 ° C，3sec.	
使用温度	Topr	-40° C~+85	
储存温度	Tstg	-40° C~+85	

LED 最大绝对标称值上图是规格书中的一个表，最需要关注的是第一行，顺向电流 20mA，前面说电流越大，亮度越大，但是**流过 LED 的电流有绝对标称值限制**，我们使用的这颗 LED，就不能大于 20ma。为了限制流过 LED 的电流，我们在 LED 上串接了一个 1K 的电阻。这个电阻就是通常我们所说的**限流电阻**。当 IO 输出高电平 3.3V 时，没有电流流过，LED 不发光。当 IO 输出低电平 0V 时，电流从 3.3V 电源留向 IO 口，LED 有电流流过，发光，其中电流可以简略计算：(3.3-0.6)/1K= 2.7ma。上面的电路使用低电平驱动 *LED*，*LED* 的负极接到 *IO*，这种方式叫灌电流驱动。也可以将 *LED* 正极接到 *IO* 口，电阻接到地，这样就叫拉电流驱动。不过通常我们都是使用灌电流，原因是很多单片机的灌电流能力比拉电流能力强。比如灌电流可以做到 *20ma*，拉电流可能只有 *5ma*

8.2.2 调试过程

库

ST 官方的标准外设库中，stm32f4xx_gpio.h 和 stm32f4xx_gpio.c 就是操作 IO 的库文件。在头文件中有函数声明，如下

```
/* Exported macro ------------------------------------------------------------*/
/* Exported functions --------------------------------------------------------*/

/*  Function used to set the GPIO configuration to the default reset state ****/
void GPIO_DeInit(GPIO_TypeDef* GPIOx);

/* Initialization and Configuration functions *********************************/
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
void GPIO_StructInit(GPIO_InitTypeDef* GPIO_InitStruct);
void GPIO_PinLockConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);

/* GPIO Read and Write functions **********************************************/
```

(continues on next page)

(continued from previous page)

```
uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, BitAction BitVal);
void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
void GPIO_ToggleBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);

/* GPIO Alternate functions configuration function ****************************/
void GPIO_PinAFConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_PinSource, uint8_t GPIO_AF);
```

我们就是通过调用这些函数控制 IO 口。

> 函数：C 语言中的基本组成要素。一个函数，就是一段代码的集合，通常，可以算做一个小模块。函数有函数名、输入参数，函数实体，返回值等要素组成。例如上面的第一个函数 GPIO_DeInit，他的参数是一个 GPIO_TypeDef* 指针，参数名叫做 GPIOx。函数没有返回值，所有是 void。头文件只是函数声明，函数实体在 c 文件中。

编码调试

我们在 main.c 中增加如下代码

```
/* 初始化 LED IO 口 */
        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
        GPIO_Init(GPIOG, &GPIO_InitStructure);
        /* Infinite loop */
        while (1)
        {
            GPIO_ResetBits(GPIOG, GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
        }
```

第 2 行，打开了 GPIOG 的时钟，所有的外设都需要打开时钟才能工作。第 4 行，4 个 IO 口或操作，填入 GPIO_Pin，意思就是这 4 个 IO 口同时配置，使用相同的配置。大家可以看一下定义，每个 IO 定义用一个 BIT。

bit 是指二进制中的一个位。但是程序中，我们常用的是 16 进制例如下面代码第 2 行 0x0002 0x 表示是 16 进制，值是 0x0002 转换为二进制就是 0000 0000 0000 0010 第 2 个 bit 为 1.

```
#define GPIO_Pin_0                 ((uint16_t)0x0001)  /* Pin 0 selected */
#define GPIO_Pin_1                 ((uint16_t)0x0002)  /* Pin 1 selected */
#define GPIO_Pin_2                 ((uint16_t)0x0004)  /* Pin 2 selected */
#define GPIO_Pin_3                 ((uint16_t)0x0008)  /* Pin 3 selected */
#define GPIO_Pin_4                 ((uint16_t)0x0010)  /* Pin 4 selected */
#define GPIO_Pin_5                 ((uint16_t)0x0020)  /* Pin 5 selected */
#define GPIO_Pin_6                 ((uint16_t)0x0040)  /* Pin 6 selected */
#define GPIO_Pin_7                 ((uint16_t)0x0080)  /* Pin 7 selected */
#define GPIO_Pin_8                 ((uint16_t)0x0100)  /* Pin 8 selected */
#define GPIO_Pin_9                 ((uint16_t)0x0200)  /* Pin 9 selected */
#define GPIO_Pin_10                ((uint16_t)0x0400)  /* Pin 10 selected */
#define GPIO_Pin_11                ((uint16_t)0x0800)  /* Pin 11 selected */
#define GPIO_Pin_12                ((uint16_t)0x1000)  /* Pin 12 selected */
#define GPIO_Pin_13                ((uint16_t)0x2000)  /* Pin 13 selected */
#define GPIO_Pin_14                ((uint16_t)0x4000)  /* Pin 14 selected */
#define GPIO_Pin_15                ((uint16_t)0x8000)  /* Pin 15 selected */
#define GPIO_Pin_All               ((uint16_t)0xFFFF)  /* All pins selected */
```

第 5 行，配置为输出模式。模式一共有 4 中，分别是输入、输出、功能、模拟。功能，就是用作外设功能，例如用作串口，SPI 等。模拟就是用作模拟功能的 IO，例如 ADC/DAC 等。

```
typedef enum
{
  GPIO_Mode_IN   = 0x00, /*!< GPIO Input Mode */
  GPIO_Mode_OUT  = 0x01, /*!< GPIO Output Mode */
  GPIO_Mode_AF   = 0x02, /*!< GPIO Alternate function Mode */
  GPIO_Mode_AN   = 0x03  /*!< GPIO Analog Mode */
}GPIOMode_TypeDef;
```

enum: 枚举，可以简单的认为，后续我们定义的某种变量，只会有有限个值，就可以用枚举。用枚举可以防止值越界，通常，一堆相同属性的宏定义，最好一起组合定义为枚举。typedef：类型定义。在上面代码中的意思就是，将一个 enum 定义为 GPIOMode_TypeDef 类型后续用 GPIOMode_TypeDef 定义的变量，就是这个 enum 类型。

第 6 行，配置 OType, 一共有 2 种选择

```
typedef enum
{
  GPIO_OType_PP = 0x00,
  GPIO_OType_OD = 0x01
```

(continues on next page)

(continued from previous page)

```
}GPIOOType_TypeDef;
```

第 7 行设置 IO 口速度，有四种速度选择。

```
/* Add legacy definition */
#define  GPIO_Speed_2MHz    GPIO_Low_Speed
#define  GPIO_Speed_25MHz   GPIO_Medium_Speed
#define  GPIO_Speed_50MHz   GPIO_Fast_Speed
#define  GPIO_Speed_100MHz  GPIO_High_Speed
```

第 8 行，设置上下拉模式，三种选择：

```
typedef enum
{
  GPIO_PuPd_NOPULL = 0x00,
  GPIO_PuPd_UP     = 0x01,
  GPIO_PuPd_DOWN   = 0x02
}GPIOPuPd_TypeDef;
```

第 9 行，将配置配置到 GPIOG，配置后，GPIOG_0，GPIOG_1，GPIOG_2，GPIOG_3，就是输出 IO，PP 模式，带上拉电阻。

IO 口的配置细节，可以通过查看结构体的注释了解

```
typedef struct
{
  uint32_t GPIO_Pin;              /*!< Specifies the GPIO pins to be configured.
                                       This parameter can be any value of @ref GPIO_pins_
↪define */

  GPIOMode_TypeDef GPIO_Mode;     /*!< Specifies the operating mode for the selected␣
↪pins.
                                       This parameter can be a value of @ref GPIOMode_
↪TypeDef */

  GPIOSpeed_TypeDef GPIO_Speed;   /*!< Specifies the speed for the selected pins.
                                       This parameter can be a value of @ref GPIOSpeed_
↪TypeDef */

  GPIOOType_TypeDef GPIO_OType;   /*!< Specifies the operating output type for the␣
↪selected pins.
                                       This parameter can be a value of @ref GPIOOType_
↪TypeDef */
```

(continues on next page)

(continued from previous page)

```
  GPIOPuPd_TypeDef GPIO_PuPd;      /*!< Specifies the operating Pull-up/Pull down for the␣
↪selected pins.
                                        This parameter can be a value of @ref GPIOPuPd_
↪TypeDef */
}GPIO_InitTypeDef;
```

我们原理图设计的是低电平点亮 LED，因此在 while(1) 中输出低电平。

> while(1) 是个死循环，不断重复大括号内的代码，直到遇到 break 才跳出。为什么要 while？因为 CPU 一直在运行，如果不是循环，就跑飞了。

下载程序后 LED 不闪烁，也不亮。**万用表测，电压 1.8V 左右，应该是 IO 配置不对**。根据原理图分析代码，发现代码错误，127 行，应该初始化 GPIOG，错写成 GPIOF，修正，重新编译后下载，LED 正常点亮。亮了之后我们就让他闪。

```
        while (1)
        {
            GPIO_ResetBits(GPIOG, GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
            GPIO_SetBits(GPIOG, GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
        }
```

ok，代码改好了，不断输出低电平点亮，然后输出高电平熄灭，重复，重复，灯就闪了。编译后下载进去看看效果。闪了吗？没闪。跟刚刚有什么差别？**调试硬件的时候要注意现象细节（例如某个灯冒烟等现象）**，如果没看到差别，把代码改回去对比一下。差别就是 **LED 变暗了**。为什么？我相信很多人都体会过这个段子。**原因是芯片跑得太快了，快到眼睛看不到亮灭的切换**。变暗，是因为我们尽管看不到亮灭，但是实际上 LED 亮灭是在切换的。相当于 50% 时间在亮，50% 时间灭，粗略来说，类似积分效果，平均算，效果相当于亮一半，结果就是暗了。。。。(其实严格来说相当于 50% 占空比 PWM 调光效果)。那么要看出亮灭，就需要将亮灭的时间延长，延长到你的眼睛可以看到。加上延时 1000ms 后代码如下。

```
        while (1)
        {
                GPIO_ResetBits(GPIOG, GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
                Delay(100);
                GPIO_SetBits(GPIOG, GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
                Delay(100);
        }
```

LED 闪烁了，但是闪烁的间隔跟预想不一致。程序设计 1 秒闪烁，实际大概 3 秒才闪烁。说明 Delay 函数延时不正确。原因可能有：1 SysTick_Config 配置错误，但是这个是官方的函数，**暂时**不怀疑它。2 时钟不对，要不就是晶振搞错，要不就是软件配置错误。在上一节我们修改了时钟配置，可能没修改对。经检查，在修改晶振频率时，只顾截图，未修改。晶振修改为 8M 后，一切正常。

对于错误的解决，要顺藤摸瓜。并且优先考虑相关因素，优先最新的改动。有形成可信的逻辑链。例如：闪烁时间不对，不用考虑 IO 的问题了，因为已经正常亮灭了。时钟不对，分析的流程应该是：软件配置对了吗？-> 硬件焊晶振对了吗？-> 软件用的官方库有 bug?

8.2.3 流水灯

请各位自行实现流水灯。同时请问，LED 闪烁中延时 1S 中，最短延时多少就可以看到闪烁？**可以百度电影帧率，人眼视觉残留**

8.2.4 总结

LED 闪烁起来的时候，就证明程序能跑了。

8.3 end

CHAPTER 9

串口-重要调试手段

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

很多人喜欢用 JLINK 调试程序，就个人而言，只有写汇编代码时才经常使用仿真。调试带串口的 CPU，主要还是使用串口输出调试信息。只有遇到一些很难，并且牵涉到汇编、或者寄存器异常的问题，才会使用仿真器看一下。主要有以下考虑：

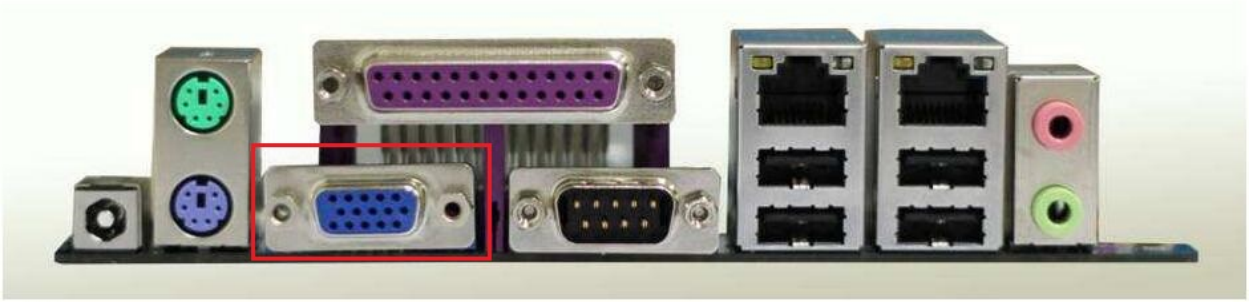
1 串口调试信息比较直观。2 仿真器单步调试会影响程序本来流程。3 大工程用仿真器效率不高，特别是做 LINUX 的时候，更加少用仿真器了。4 **程序是你写的，你的脑中该有程序怎么跑的一个构思存在，当实际与设计不一致时，你应该大概知道哪里有问题，而不是完全靠仿真器**。

9.1 串口

通常我们所说的串口，也叫 UART，RS232，TLL。(*SPI 也叫串口，例如串口屏，就是屏幕接口是 SPI (I2C)，而不是 UART*)

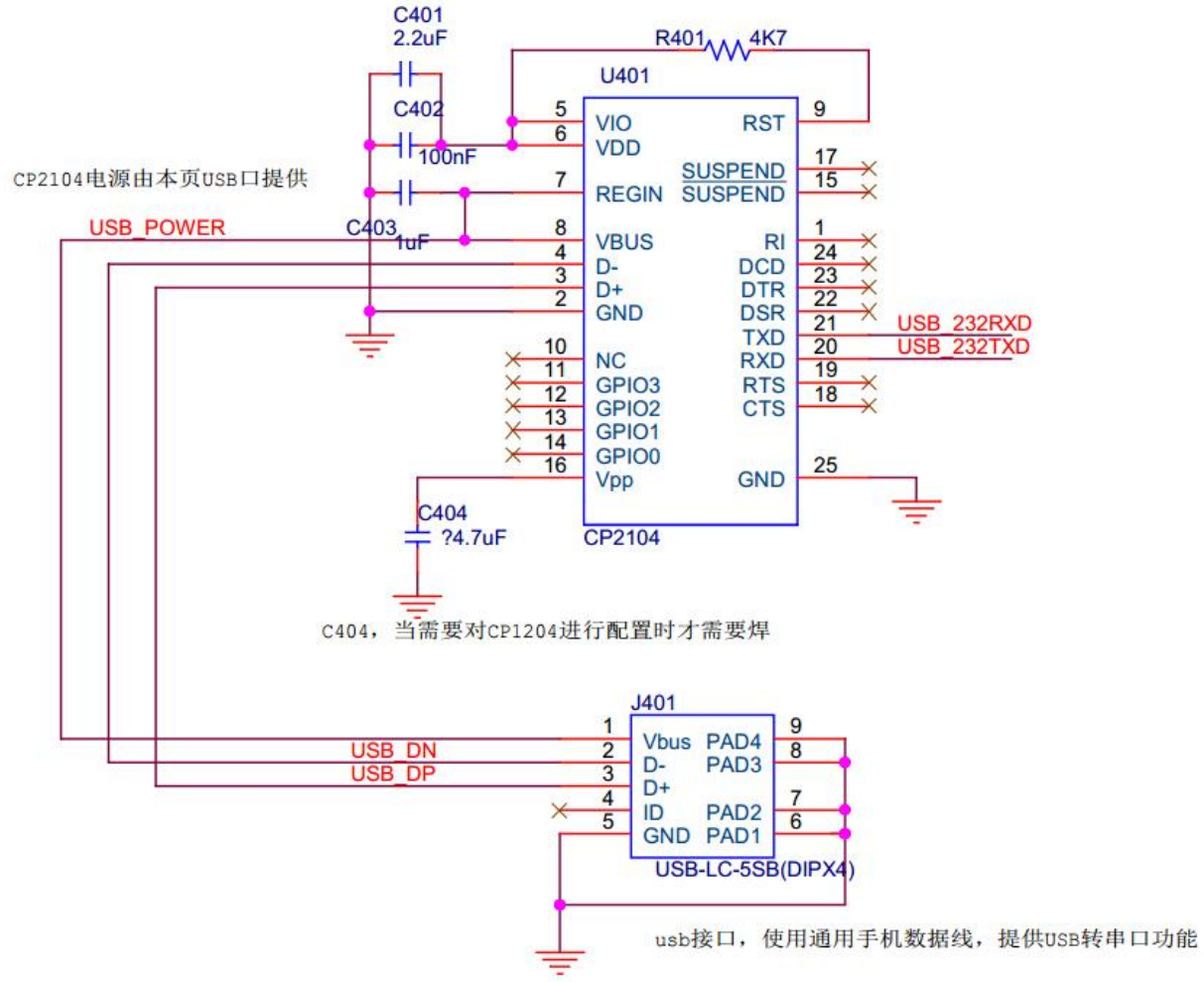
> 通用异步收发传输器（Universal Asynchronous Receiver/Transmitter），通常称作 UART，是一种异步收发传输器串行接口简称串口，也称串行通信接口或串行通讯接口（通常指 COM 接口），是采用串行通信方式的扩展接口。串行接口 (Serial Interface) 是指数据一位一位地顺序传送，其特点是通信线路简单，只要一对传输线就可以实现双向通信（可以直接利用电话线作为传输线），从而大大降低了成本，特别适用于远距离通信，但传送速度较慢。还要一种增强串口叫 USART：**USART：(Universal Synchronous/Asynchronous Receiver/Transmitter) 通用同步/异步串行接收/发送器 USART 是一个全双工通用同步/异步串行收发模块**我们常说的是 UART，USART 需要同步时钟，经常在 IC 卡控制上使用。

在以前的电脑主板上，会有一个 **DB9** 接头，下图中红框内的，这个接口就是串口。现在的主板基本上没有串口接口了，特别是笔记本，完全没有串口。



串口配置项这个 DB9 接口的信号电平，是 RS232 电平，不能直接接到单片机（STM32）的串口管脚，需要通过一个 RS232 电平转换器。很多开发板现在还提供了 DB9 接头，但是现在电脑都不带这个接口了，屋脊雀的开发板抛弃了这个大家伙。

那没有了 DB9 接口，如何使用串口呢？可以通过 USB 转串口。STM32 的串口信号接到 CP2104 芯片，CP2104 通过 USB 与电脑连接。下图就是我们的开发板底板的 USB 转串口电路。



usb

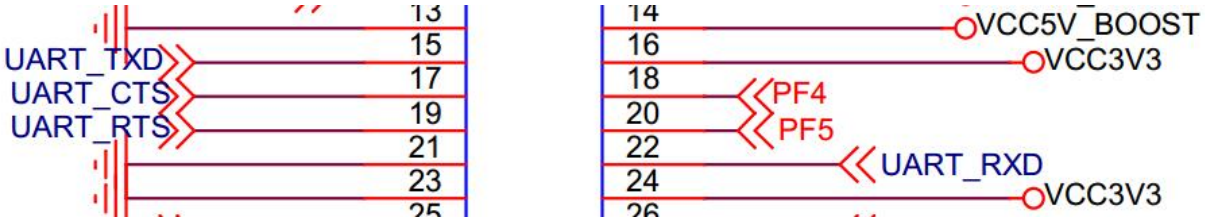
转串口

还可以通过 CMSIS DAP 调试器进行 USB 转串口。

一个完整的串口有以下信号：

> 数据：TXD（pin 3）：串口数据输出 (Transmit Data) RXD（pin 2）：串口数据输入 (Receive Data) 握手：RTS（pin 7）：发送数据请求 (Request to Send) CTS（pin 8）：清除发送 (Clear to Send) DSR（pin 6）：数据发送就绪 (Data Send Ready) DCD（pin 1）：数据载波检测 (Data Carrier Detect) DTR（pin 4）：数据终端就绪 (Data Terminal Ready) 地线：GND（pin 5）：地线其它 RI（pin 9）：铃声指示

这个是完整的通信信号，一般我们只使用数据线与地线。有一些外接高速模块会使用握手信号，也就是我们常说的**流控**。在屋脊雀 407 开发板的外扩串口上，引出流控信号 RTS、CTS，如下图：



外扩串口

串口按位（bit）发送数据，数据格式由起始位（start bit）、数据位（data bit）、奇偶校验位（parity bit）和停止位（stop bit）组成。在串口调试助手上软件可以看到数据格式配置。通常数据格式是起始位 1BIT、8bit 数据、1bit 停止位、无校验位。波特率则是串口每秒传输的 BIT 速率。可以通过波特率计算串口 1 秒钟可传输多少数据。

> 例如常用串口数据格式是 10BIT 一个字节，那么在 115200 波特率下，每秒最多传输 11520 个字节数据。每字节数据传输时间仅仅 86.80555555555556us，是一个比较快的速度了。

串口配置项

我们下面看看简单的串口通信时序，发送两个字节数据，0x55、0xaa（55aah）。使用格式：1 起始位，8 数据位，1

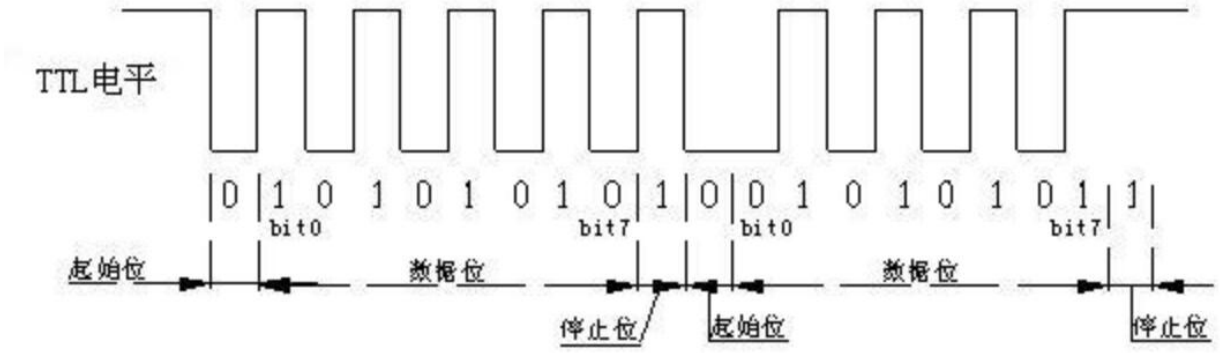
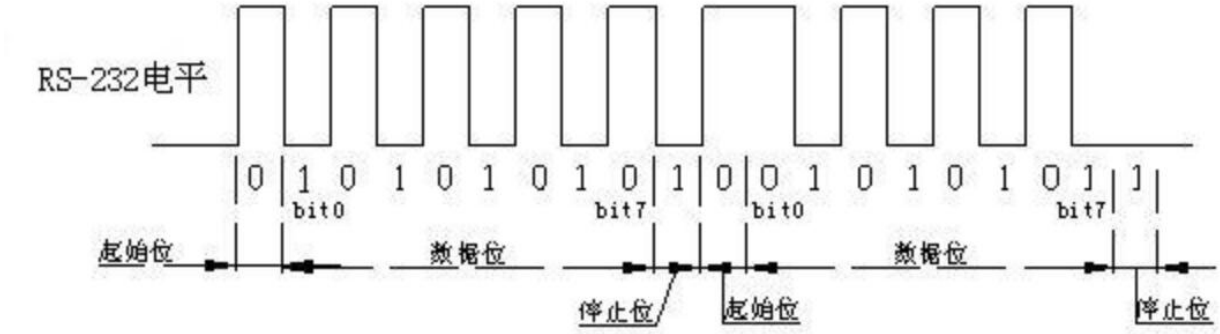


图1 TTL电平的串行数据帧格式(55aah)

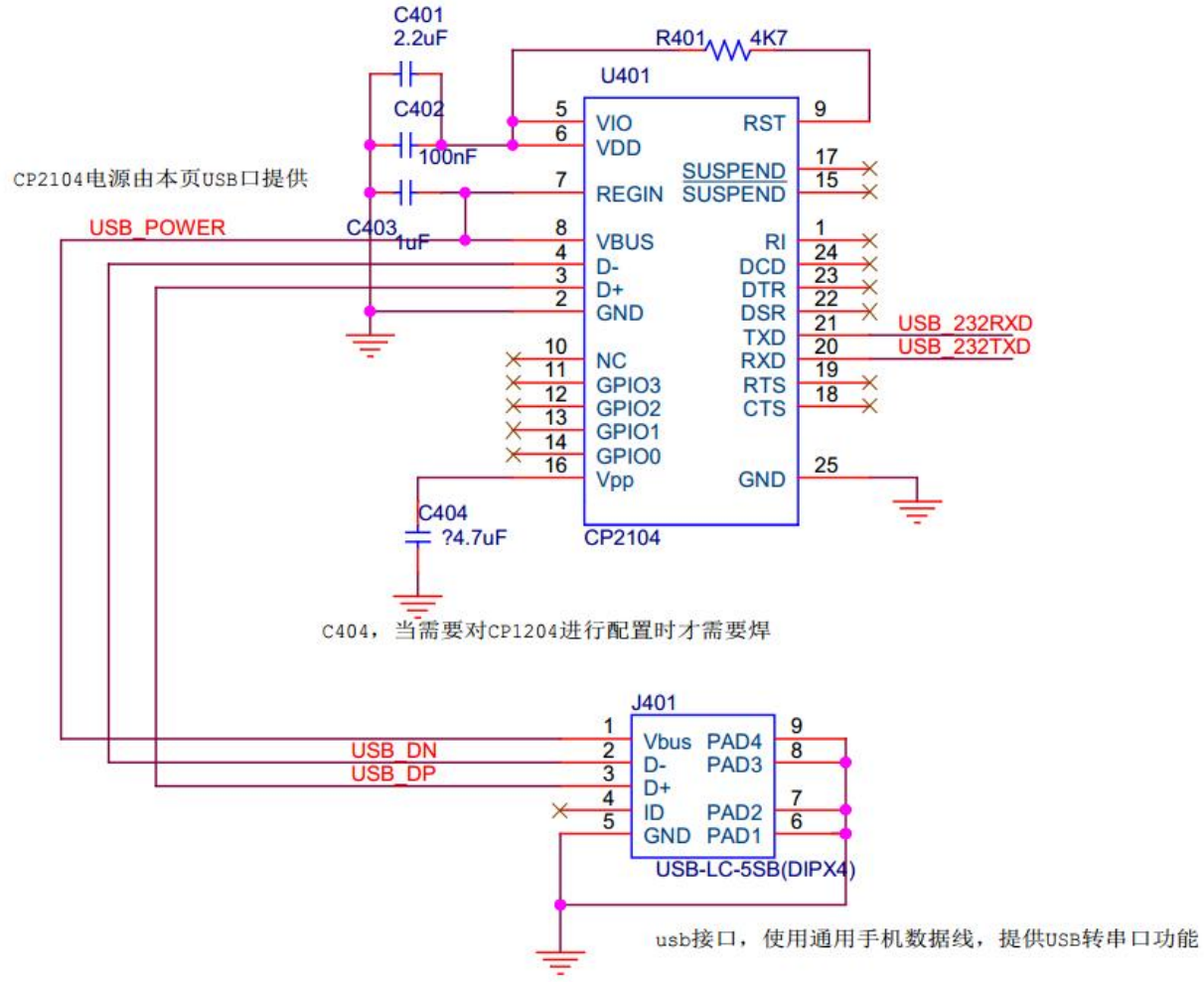


停止位。串口时序

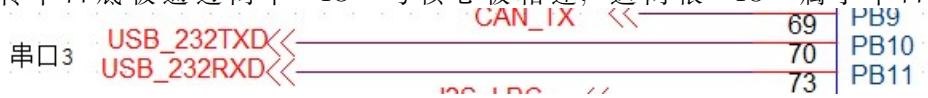
串口 RX 和 TX 的时序是一样的，一个是发送，一个是接收。

9.2 原理图

屋脊雀 F407 开发板使用了一个 USB 转串口芯片 CP2104，芯片在底板上。这个芯片相对于其他 USB 转串口芯片，更加可靠，推荐大家使用。



usb 转串口底板通过两个 IO 与核心板相连，这两根 IO 属于串口 3 的 TX 与 RX。



串口原理图

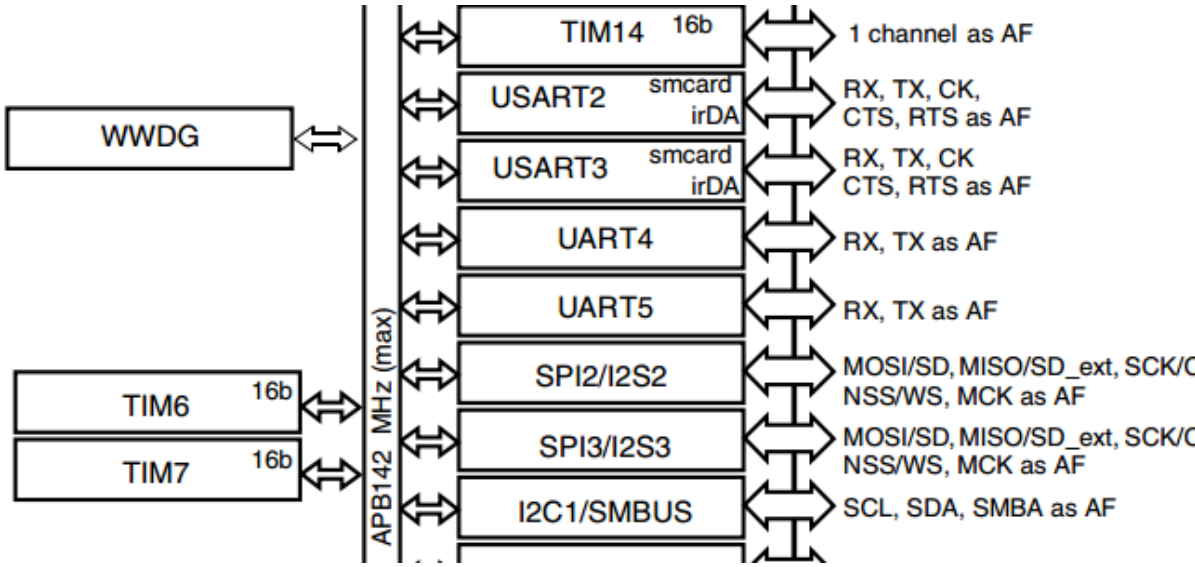
这两个 *IO* 同时从 *DAP* 调试口引出，因此，*DAP* 的串口和底板 *CP2104* 不要同时使用。

9.3 STM32 串口

STM32 芯片外设丰富，有多个串口，我们这次使用的两个管脚是 PB10 和 PB11。

通过查看《STM32F407_ 数据手册.pdf》第 56 页，管脚功能映射表可知，PB10 AF7 功能，是串口 3 的 TX，PB11 是串口 3 的 RX。***TX* 和 *RX*，都是对自己而言，*PB10* 就是 *STM32* 的发送管脚，那么就要连接到 *CP2104* 的 *RX* 管脚，以后大家画芯片原理图器件也要这样命名**

从文档《STM32F407_ 数据手册.pdf》第 17 页可以看出，UART3 挂在 APB 总线上，USART3 还支持 SM-



CARD 和 irDA 功能。

串口

串口更详细功能要从《STM32F4xx 中文参考手册.pdf》找，第 26 章

参考手册串口

关于设备的所有信息都可以从这里找到，意法半导体是非常有良心的，提供了中文版本，就算英文不好阅读起来也没有障碍。在这里写再多，也比不上官方文档，尽量不添加官方文档的内容。

> **对于 STM32 要吐槽的是：为什么接收硬缓冲只有 1 个？1 个？1 个？**上面介绍串口波特率时提到，在 115200 的波特率下，1 个字节只需要 86us，如果系统处于高负载情况下，串口中断很可能被其他中断卡住而进不了，造成数据丢失。要防止数据丢失，有以下手段：
>
> 1. 将这个串口优先级提高到最高并且可以抢占别的中断（中断嵌套）；

2. 通过人工努力优化程序运行逻辑，在时间线上，优化各个模块的运行时间，解决中断丢失问题（各中断尽量错开）。
3. 使用 DMA 模拟硬件缓冲。*以前 ARM7 芯片，通常都有 8 个或者 16 个硬 BUF*。

9.4 串口驱动设计

在设计串口驱动之前，思考几点：

问题 1. 串口驱动给谁用？需要提供什么样的接口？问题 2. 串口驱动要实现什么功能？问题 3. 串口接收发送如何设计？(对于高速运行的程序来说，串口是一个**慢设备**) 问题 4. 中断如何设计？

- 问题 1 1. 串口有可能直接给应用程序使用。应用程序通过串口与 PC 或其他设备通信。2. 可能连接一个设备模块，例如串口 WIFI 模块。在 APP 层看，只知道 WIFI，至于 WIFI 模块使用什么接口跟 CPU 连接，APP 是不知道的。因此，在这种情况下，**使用串口的是 WIFI 驱动**。
- 问题 2 串口就是实现数据收发功能，**串口驱动不应该关心收发的内容**。你觉得在串口中断中判断回车换行 (0x0d/0x0a) 合适吗？
- 问题 3 由于串口是慢设备，最好的方法就是发送接收都通过缓冲区处理，如果还需要进一步提高性能，可以考虑用硬件 DMA。
- 问题 4 所有的中断程序都是越短越好，对串口来说，接收到数据，放入缓冲区就立刻退出中断。

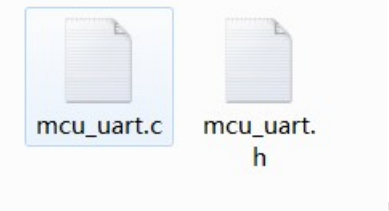
我们的串口驱动程序就根据上面几点思考编写。

9.5 环形缓冲技术

也就是常说的 RingBuf。在 stmcu 论坛有一个帖子说的非常详细，请大家移步阅读。http://www.stmcu.org/module/forum/thread-616132-1-2.html

9.6 编码

在文件夹建立一个 mcu_dev 文件夹，**用于保存 CPU 片上外设的驱动**。新建两个文件 mcu_uart.c、



mcu_uart.h，并且添加到 MDK 跟 SI 工程，**头文件搜索路径也添加**。串口驱动代码就不解释了，在源码中有完整注释。除了驱动代码外，在 main.c 中增加串口测试函数调用

```
        mcu_uart_open(3);
        while (1)
        {
                GPIO_ResetBits(GPIOG, GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
                Delay(100);
                GPIO_SetBits(GPIOG, GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
                Delay(100);
                mcu_uart_test();
        }
```

在 stm32f4xx_it.c 中增加中断处理

```
/**
  * @brief  This function handles PPP interrupt request.
  * @param  None
  * @retval None
  */
/*void PPP_IRQHandler(void)
{
}*/

/**
  * @}
  */
void USART3_IRQHandler(void)
{
        mcu_uart3_IRQhandler();
}
```

中断是什么，中断如何运行，在后面章节有说明。

- uart_printf 实现代码如下，注意 string 数组的大小，我们定义了 256，也即是一次输出调试信息字符不能超过这个 buf 的大小。

```
/*
使用串口输出调试信息
*/
s8 string[256];//调试信息缓冲，输出调试信息一次不可以大于 256

#ifdef __GNUC__
/* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf
     set to 'Yes') calls __io_putchar() */
```

(continues on next page)

(continued from previous page)

```
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */

PUTCHAR_PROTOTYPE
{
    /* Place your implementation of fputc here */
    /* e.g. write a character to the USART */
    USART_SendData(USART3, (uint8_t) ch);

    /* Loop until the end of transmission */
    while (USART_GetFlagStatus(USART1, USART_FLAG_TC) == RESET);
    return ch;
}

extern int vsprintf(char * s, const char * format, __va_list arg);
/**
 *@brief:      uart_printf
 *@details:    从串口格式化输出调试信息
 *@param[in]   s8 *fmt
               ...
 *@param[out]  无
 *@retval:
 */
void uart_printf(s8 *fmt,...)
{
    s32 length = 0;
    va_list ap;

    s8 *pt;

    va_start(ap,fmt);
    vsprintf((char *)string,(const char *)fmt,ap);
    pt = &string[0];
    while(*pt!='\0')
    {
        length++;
        pt++;
    }
```

(continues on next page)

(continued from previous page)

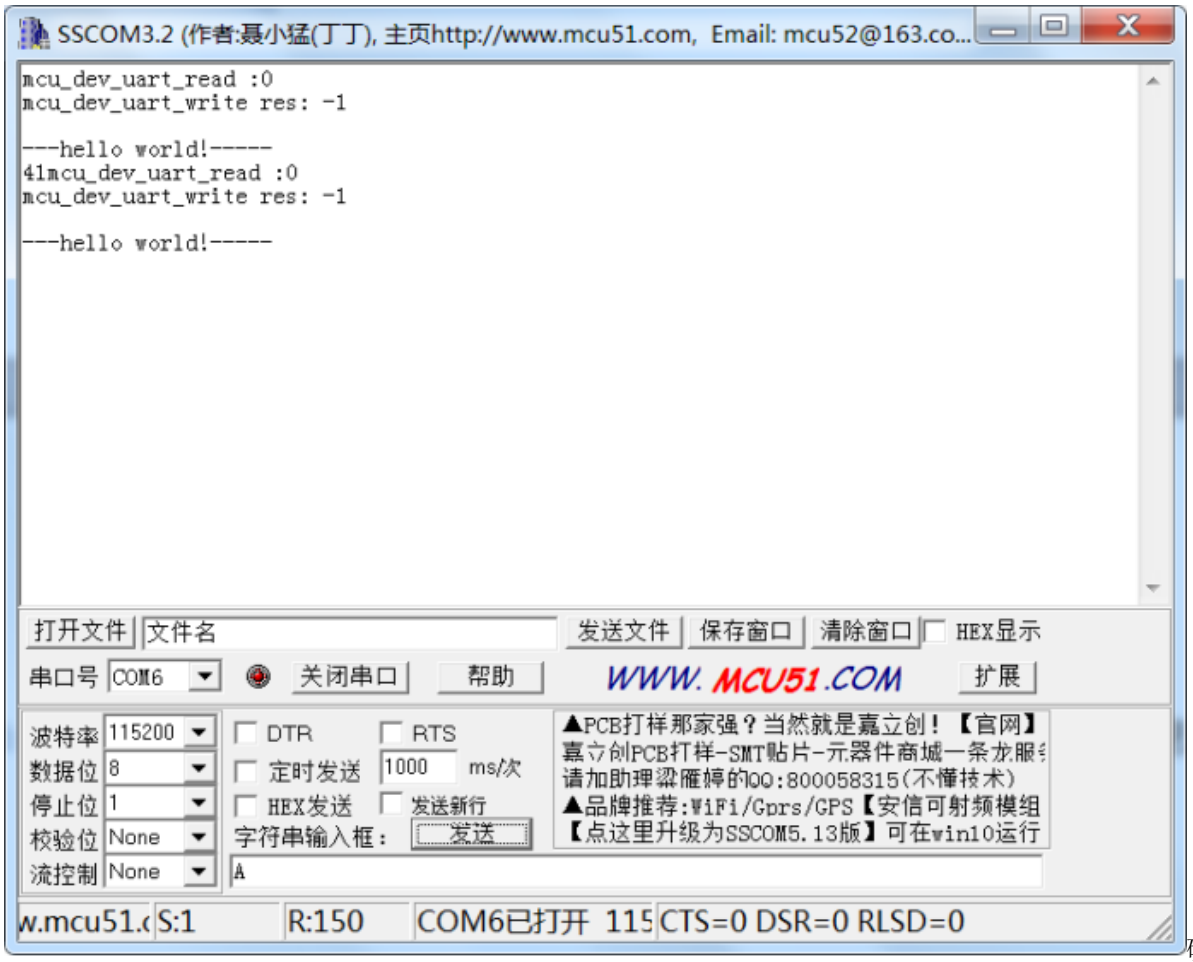
```
    mcu_uart_write(PC_PORT, (u8*)&string[0], length);  //写串口

    va_end(ap);
}
```

9.7 调试过程

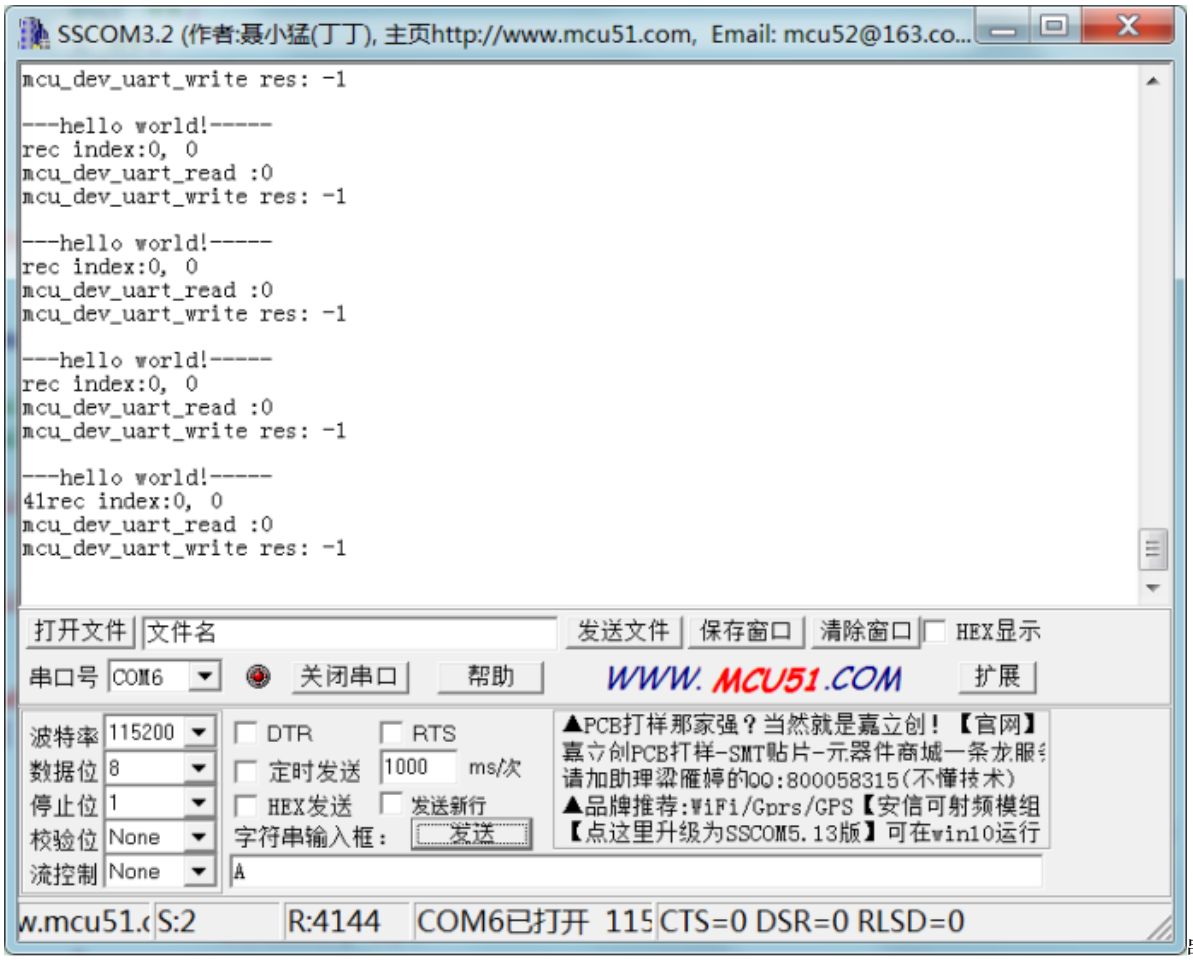
- 问题 1 PC 发送数据给开发板，开发板的 LED 就不闪，也不再发送字符，说明可能**死机**了。**因为是 PC 发送数据触发问题，所以应该是程序接收中断未处理好**。经查，在 USART3_IRQHandler 中没有添加代码，中断根本没处理，因此串口**重复进入中断，造成卡死**（其实没死机，只是一直中断出不来，无法执行 main 函数）。修改好后，PC 端发送串口数据，开发板不再死机。

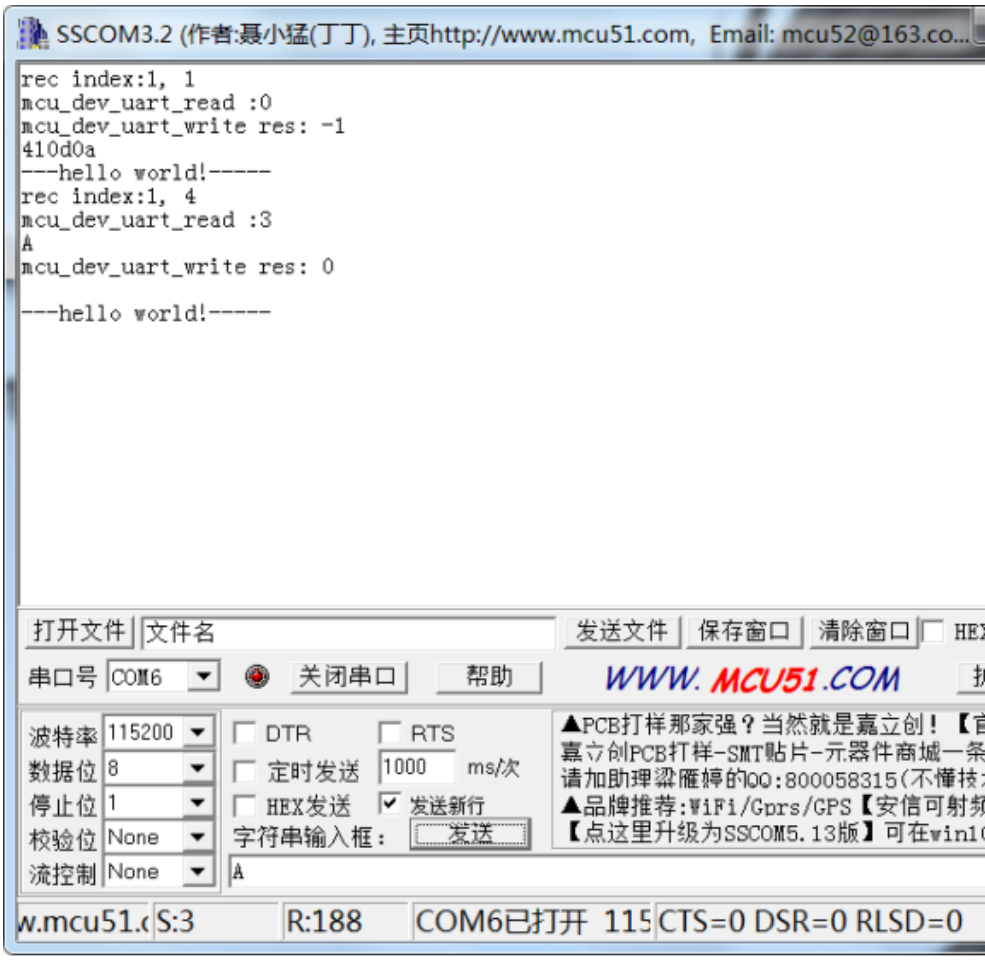
解决串口接收数据死机

- 问题 2 继续测试，发送数据，按照测试程序设计，收到 A 后，将 A 发送给电脑，但是现在收不到数据。在串口中断 mcu_uart3_IRQhandler 中**增加调试信息**，收到数据则将数据打印。从调试信息看，能收到数据。



确认串口中断能收到数据在 mcu_uart_read 函数内增加调试信息，每次调用都输出当前串口缓冲索引，发现收到数据的时候都是 0。这时候发现，在这个函数的前面调用了 mcu_uart_open，而在 OPEN 中每次都会将串口缓冲两个索引清 0。


SSCOM3.2 (作者:聂小猛(丁丁), 主页http://www.mcu51.com, Email: mcu52@163.co...
mcu_dev_uart_write res: -1
---hello world!-----
rec index:0, 0
mcu_dev_uart_read :0
mcu_dev_uart_write res: -1
---hello world!-----
rec index:0, 0
mcu_dev_uart_read :0
mcu_dev_uart_write res: -1
---hello world!-----
rec index:0, 0
mcu_dev_uart_read :0
mcu_dev_uart_write res: -1
---hello world!-----
41rec index:0, 0
mcu_dev_uart_read :0
mcu_dev_uart_write res: -1
打开文件
文件名
发送文件
保存窗口
清除窗口
HEX显示
串口号
COM6
关闭串口
帮助
WWW.MCU51.COM
扩展
波特率
115200
数据位
8
停止位
1
校验位
None
流控制
None
DTR
RTS
定时发送
1000
ms/次
HEX发送
发送新行
字符串输入框：
发送
A
▲PCB打样那家强？当然就是嘉立创！【官网】
嘉立创PCB打样-SMT贴片-元器件商城一条龙服
请加助理梁雁婷的QQ:800058315(不懂技术)
▲品牌推荐:WiFi/Gprs/GPS【安信可射频模组
【点这里升级为SSCOM5.13版】可在win10运行
w.mcu51.c
S:2
R:4144
COM6已打开 115
CTS=0 DSR=0 RLSD=0
串



口索引每次清零把 open 函数去掉口调试完成问题解决，调试串口收发已经正常。

9.8 调试信息使用

- **调试信息是需要管理的**

前面我们实现了用串口的 printf。现在让我们来定义 LOG 功能。考虑问题：

> 1.LOG 除了用串口输出，还可能用 USB 输出，LCD 显示等。2.LOG 要分等级，在发布程序时，要把 DEBUG LOG 屏蔽。

我们在 app 文件夹定义一个 wjq_log.c 和 wjq_log.h。把 uart_printf 搬到这个源文件内（这个函数后面基本不用了）。复制 uart_printf 函数并改造，增加 LOG 等级判断。LOG 等级定义如下：

```
typedef enum
{
        LOG_DISABLE = 0,
        LOG_ERR,        //错误
        LOG_FUN,        //功能（用 LOG 输出算一个功能）
```

(continues on next page)

(continued from previous page)

```
        LOG_INFO,         //信息，例如设备初始化等信息
        LOG_DEBUG,         //调试，正式程序通常屏蔽
}LOG_L;
```

- 常用 LOG 说明

输出调试信息跟变量值%d,%02x,%08x 等是常用输出格式

```
wjq_log(LOG_FUN, "mcu_dev_uart_read :%d\r\n", len);
```

%d，打印十进制格式。%02x，打印十六进制格式，2 位%08x，打印十六进制格式，8 位，打印地址时用。%s，打印字符串

输出当前函数名，文件名，代码行号，当前时间

```
wjq_log(LOG_FUN, "%s,%s,%d,%s\r\n", __FUNCTION__,__FILE__,__LINE__,__DATE__);
```

LOG 要短在中断中添加调试信息尽量短，串口是一个慢设备，输出太长的调试信息执行时间较长，会影响程序运行，特别是一些时间敏感的函数。—可以考虑将 *LOG* 做队列输出— **时刻记得 LOG 的影响**在某些临界处，添加太多调试信息会造成程序流程跟预想不一致。因此需要对各个模块调试信息进行管理，不能一下子打开所有模块的调试信息。

9.9 PC 串口工具

常用工具有 sscom32、Xshell、Docklight。特点如下：

sscom32: 简单易用，单片机常用。前面调试串口就是用 sscom32。**Xshell**：是一个强大的安全终端模拟软件，它支持 SSH1, SSH2, 以及 Microsoft Windows 平台的 TELNET 协议。可以远程登录电脑等，玩 Linux 的应该常用。同时支持串口，Linux 开发时登录命令行控制台就常用。**有个人免费版本**。**Docklight**：支持帧格式组织解析，对于做串口协议通信很有用。

如果仅仅做调试信息 LOG 输出，建议使用 Xshell，后面我们移植 UBOOT 的命令行控制台到 STM32 上，使用 Xshell 进行交互调试。

9.10 思考

1 目前只实现了一个串口的驱动，STM32 一般会有多个串口，串口驱动要如何修改？每个串口都写一份代码会很累的。2 现在程序都是单线程在跑，等后面添加了 FREERTOS 之后，驱动应该设计？需要考虑什么问题？

9.11 end

CHAPTER 10

包罗万象的小程序

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190315

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

我们通过 IO 和串口的软件开发，已经体验了嵌入式软件开发。不知道大家有没有疑惑，为什么软件能控制硬件？反正当年我学习 51 的时候，有这个疑惑。今天我们就暂停软件开发，分析单片机到底是如何**软硬件结合**的。并通过一个基本的程序，分析单片机程序的编译，运行。

10.1 软硬件结合

初学者，通常有一个困惑，就是为什么软件能控制硬件？就像当年的 51，为什么只要写 P1=0X55，就可以在 IO 口输出高低电平？要理清这个问题，先要认识一个概念：**地址空间**。

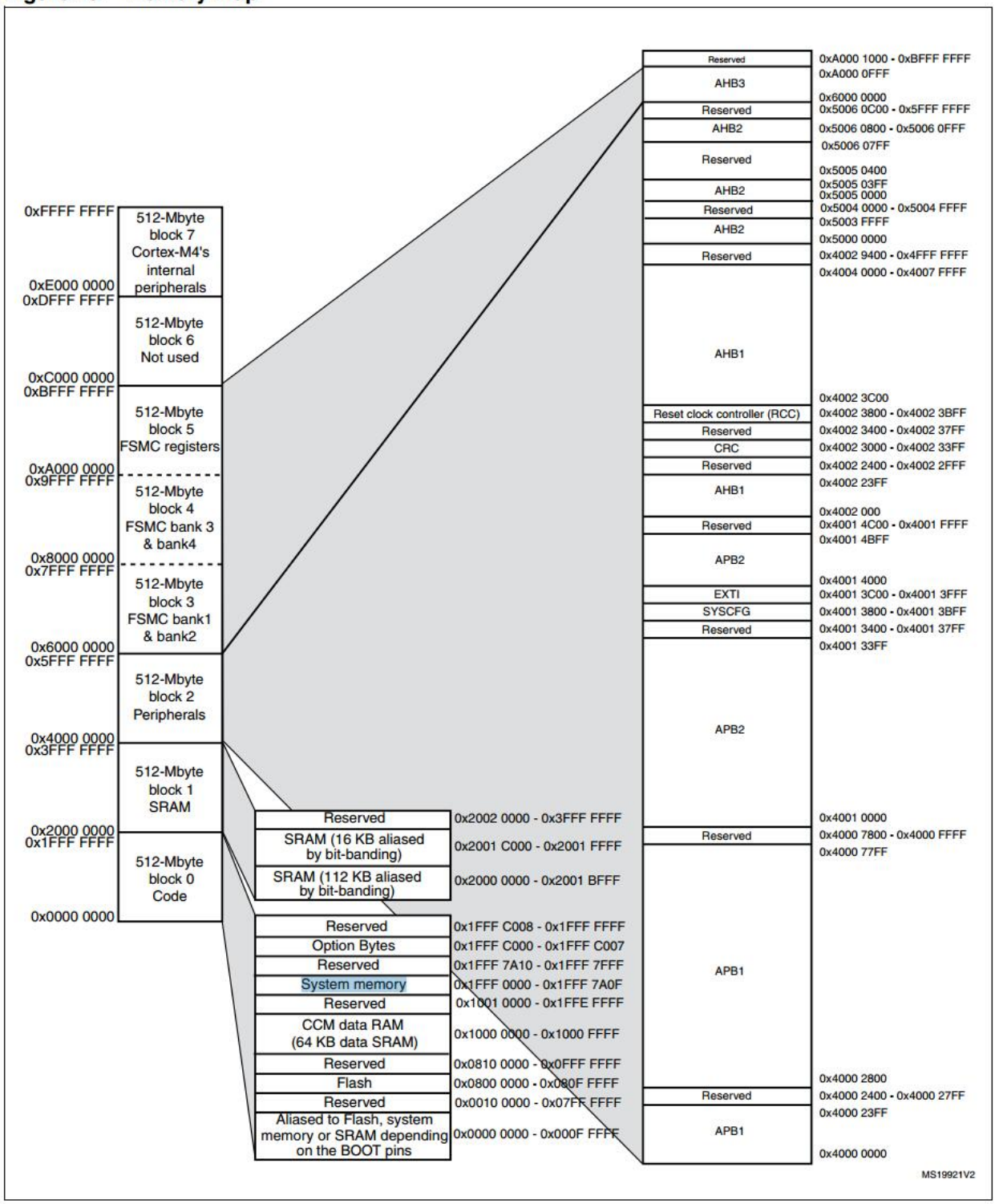
10.1.1 寻址空间

什么是地址空间呢？所谓的地址空间，就是 PC 指针的寻址范围，因此也叫寻址空间。

> 大家应该都知道，我们的电脑有 32 位系统和 64 位系统之分，为什么呢？因为 32 位系统，PC 指针就是一个 32 位的二进制数，也就是 0xffffffff，范围只有 4G 寻址空间。现在内存越来越大，4G 根本不够，所以需要扩展，为了能访问超出 4G 范围的内存，就有了 64 位系统。STM32 是多少位的？是 32 位的，因此 PC 指针也是 32 位，寻址空间也就是 4G。

我们来看看 STM32 的寻址空间是怎么样的。在数据手册《STM32F407_ 数据手册.pdf》中有一个图，这个图，就是 STM32 的寻址空间分配。所有的芯片，都会有这个图，名字基本上都是叫 Memory map，用一个新芯片，就先看这个图。

Figure 15. Memory map



stm

SRAM

- 最左边，8 个 block，每个 block 512M，总共就是 4G，也就是芯片的寻址空间。
- block 0 里面有一段叫做 FLASH，也就是内部 FLASH，我们的程序就是下载到这个地方，起始地址

是 0X800 0000，大家注意，这个只有 1M 空间。现在 STM32 已经有 2M flash 的芯片了，超出 1M 的 FLASH 放在哪里呢？请自行查看对应的芯片手册。

- 3 在 block 1 内，有两段 SRAM，总共 128K，这个空间，也就是我们前面说的内存，存放程序使用的变量。如果需要，也可以把程序放到 SRAM 中运行。407 不是有 196K 吗？
- 其实 407 有 196K 内存，但是有 64k 并不是普通的 SRAM，而是放在 block 0 内的 CCM。这两段区域不连续，而且，CCM 只能内核使用，外设不能使用，例如 DMA 就不能用 CCM 内存，否则就死机。
- block 2，是 Peripherals，也就是外设空间。我们看右边，主要就是 APB1/APB2、AHB1/AHB2，什么东西呢？回头再说。
- block 3、block4、block5，是 FSMC 的空间，FSMC 可以外扩 SRAM，NAND FALSH，LCD 等外设。

好的，我们分析了寻址空间，我们回过头看看，软件是**如何控制硬件**的。在 IO 口输出的例程中，我们配置 IO 口是调用库函数，我们看看库函数是怎么做的。例如：

```
GPIO_SetBits(GPIOG, GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
```

这个函数其实就是对一个变量赋值，对 GPIOx 这个结构体的成员 BSRRL 赋值。

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_Pin));

  GPIOx->BSRRL = GPIO_Pin;
}
```

assert_param: 这个是断言，用于判断输入参数是否符合要求 GPIOx 是一个输入参数，是一个 GPIO_TypeDef 结构体指针，所以，要用-> 获取其成员

GPIOx 是我们传入的参数 GPIOG，具体是啥？在 stm32f4xx.h 中有定义。

```
#define GPIOG                ((GPIO_TypeDef *) GPIOG_BASE)
```

GPIOG_BASE 同样在文件中有定义，如下：

```
#define GPIOG_BASE           (AHB1PERIPH_BASE + 0x1800)
```

AHB1PERIPH_BASE，AHB1 地址，有点眉目了吧？在进一步看看

```
/*!< Peripheral memory map */
#define APB1PERIPH_BASE       PERIPH_BASE
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x00010000)
```

(continues on next page)

(continued from previous page)

```
#define AHB1PERIPH_BASE       (PERIPH_BASE + 0x00020000)
#define AHB2PERIPH_BASE       (PERIPH_BASE + 0x10000000)
```

再找找 PERIPH_BASE 的定义

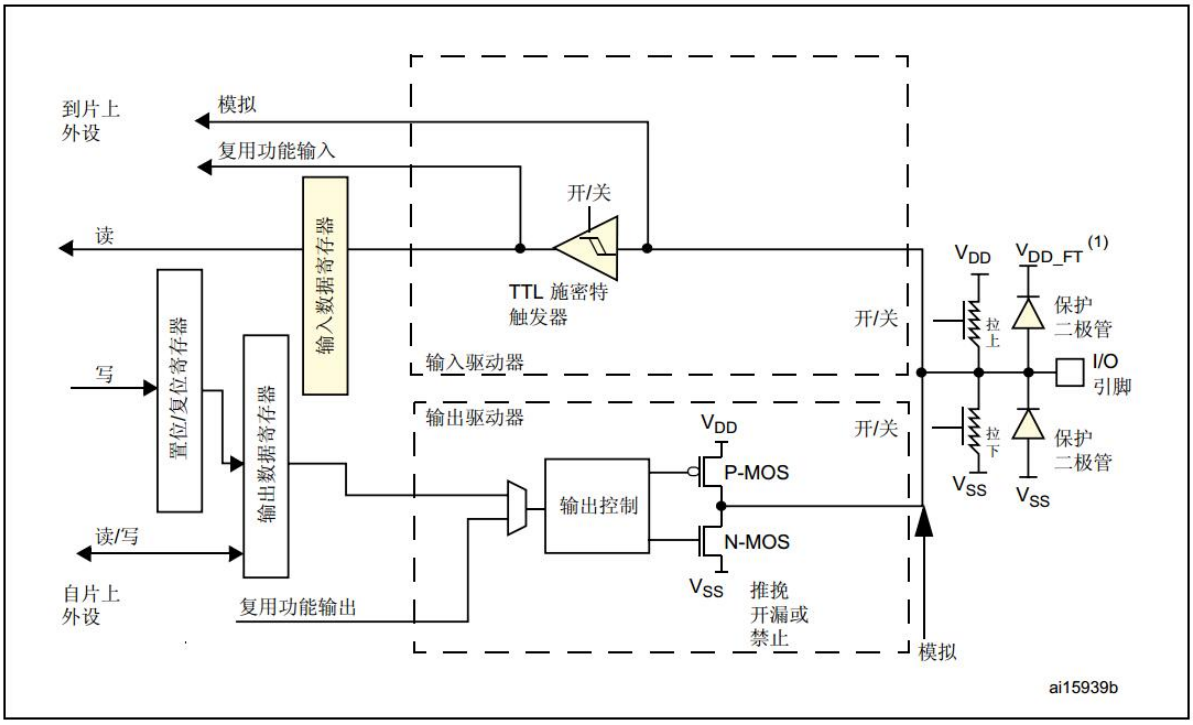
```
#define PERIPH_BASE           ((uint32_t)0x40000000)
```

到这里，我们可以看出，操作 IO 口 G，其实就是操作 0X40000000+0X1800 这个地址上的一个结构体里面的成员。说白了，就是操作了这个地方的寄存器。实质跟我们操作普通变量一样，就像下面的两句代码，区别就是变量 i 是 SRAM 空间地址，0X40000000+0X1800 是外设空间地址。

```
u32 i;
i = 0x55aa55aa;
```

这个外设空间地址的寄存器是 IO 口硬件的一部分。如下图，左边的输出数据寄存器，就是我们操作的寄存器（内存、变量），它的地址就是 0X40000000+0X1800+0x14.

图 17. 5 V 容忍 I/O 端口位的基本结构



stm

SRAM

控制其他外设也类似，就是将数据写到外设寄存器上，跟操作内存一样，就可控制外设了。

> 寄存器，其实应该是内存的统称，外设寄存器应该叫做特殊寄存器。慢慢的，所有人都把外设的叫做寄存器，其他的统称内存或 RAM。寄存器为什么能控制硬件外设呢？因为，初略的说，一个寄存器的一个 BIT，就是一个开关，开就是 1，关就是 0。通过这个电子开关去控制电路，从

而控制外设硬件。

10.2 纯软件-包罗万象的小程序

我们已经完成了串口和 IO 口的控制，但是我们仅仅知道了怎么用，对其他一无所知。程序怎么跑的？代码到底放在那里？内存又是怎么保存的？下面，我们通过一个简单的程序，学习嵌入式软件的基本要素。

10.2.1 分析启动代码

- 函数从哪里开始运行?

每个芯片都有复位功能，复位后，芯片的 PC 指针（一个寄存器，指示程序运行位置，对于多级流水线的芯片，PC 可能跟真正执行的指令位置不一致，这里暂且认为一致）会复位到固定值，一般是 0x00000000，在 STM32 中，复位到 0X08000004。因此复位后运行的第一条代码就是 0X08000004。前面我们不是拷贝了一个**启动代码文件**到工程吗？startup_stm32f40_41xxx.s，这个汇编文件为什么叫启动代码？因为里面的汇编程序，就是复位之后执行的程序。在文件中，有一段数据表，称为**中断向量**，里面保存了各个**中断的执行地址**。**复位，也是一个中断**。芯片复位时，芯片从中断表中将 Reset_Handler 这个值（**函数指针**）加载到 PC 指针，芯片就会执行 Reset_Handler 函数了。(一个函数入口就是一个指针)

```
; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     MemManage_Handler         ; MPU Fault Handler
                DCD     BusFault_Handler          ; Bus Fault Handler
                DCD     UsageFault_Handler        ; Usage Fault Handler
```

Reset_Handler 函数，先执行 SystemInit 函数，这个函数在标准库内，主要是初始芯片时钟。然后跳到 ___main 执行，___main 函数是什么函数? 是我们在 main.c 中定义的 main 函数吗? 后面我们再说这个问题。

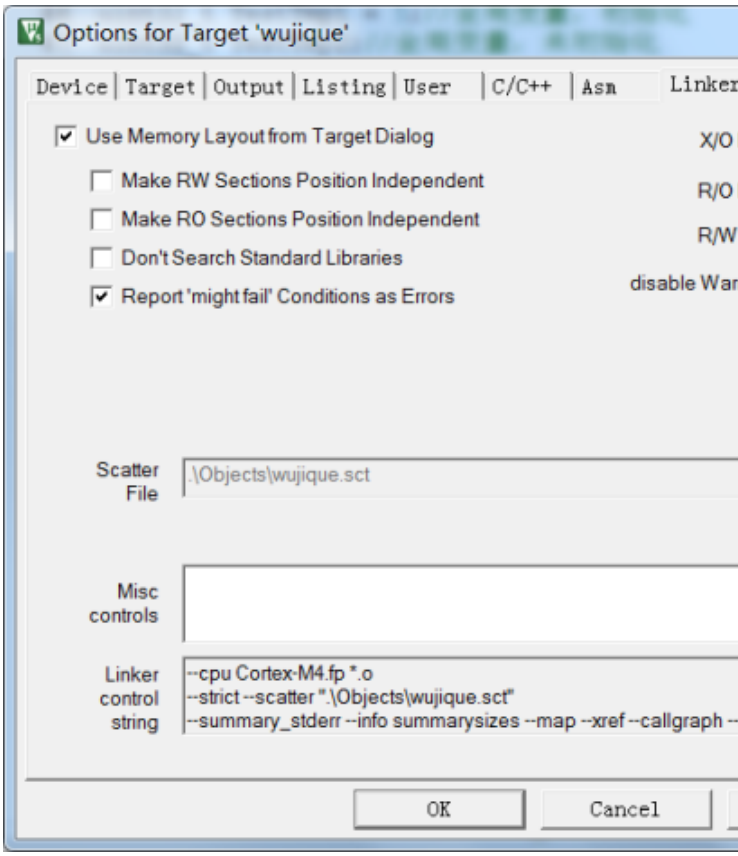
```
; Reset handler
Reset_Handler    PROC
                 EXPORT  Reset_Handler             [WEAK]
        IMPORT  SystemInit
        IMPORT  __main
```

(continues on next page)

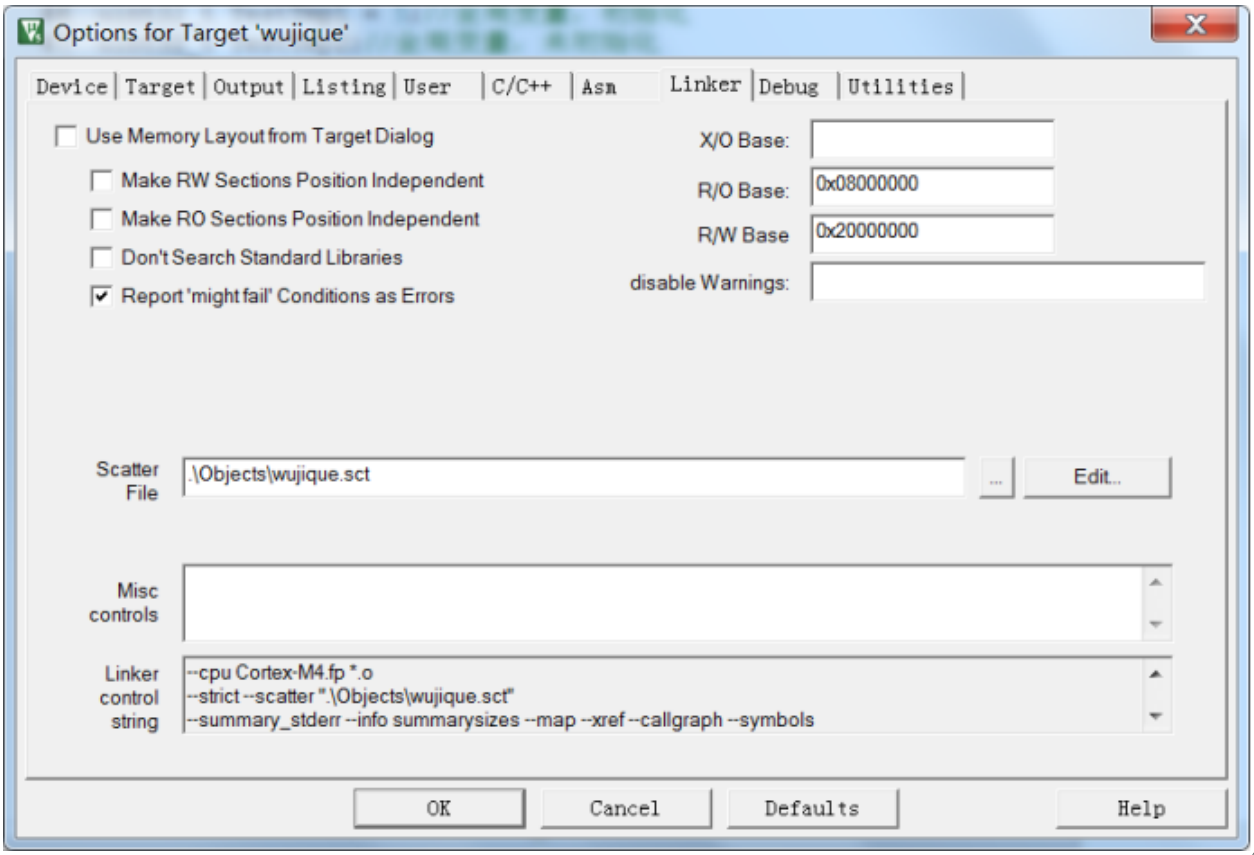
(continued from previous page)

```
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

芯片是怎么知道开始就执行启动代码的呢？或者说，我们如何把这个启动代码放到复位的位置？这就牵涉到一个一般情况下不关注的文件 wujique.sct，这个文件在 wujique\prj\Objects 目录下，通常把这个文件叫做**分散加载文件**，编译工具在链接时，根据这个文件放置各个代码段和变量。

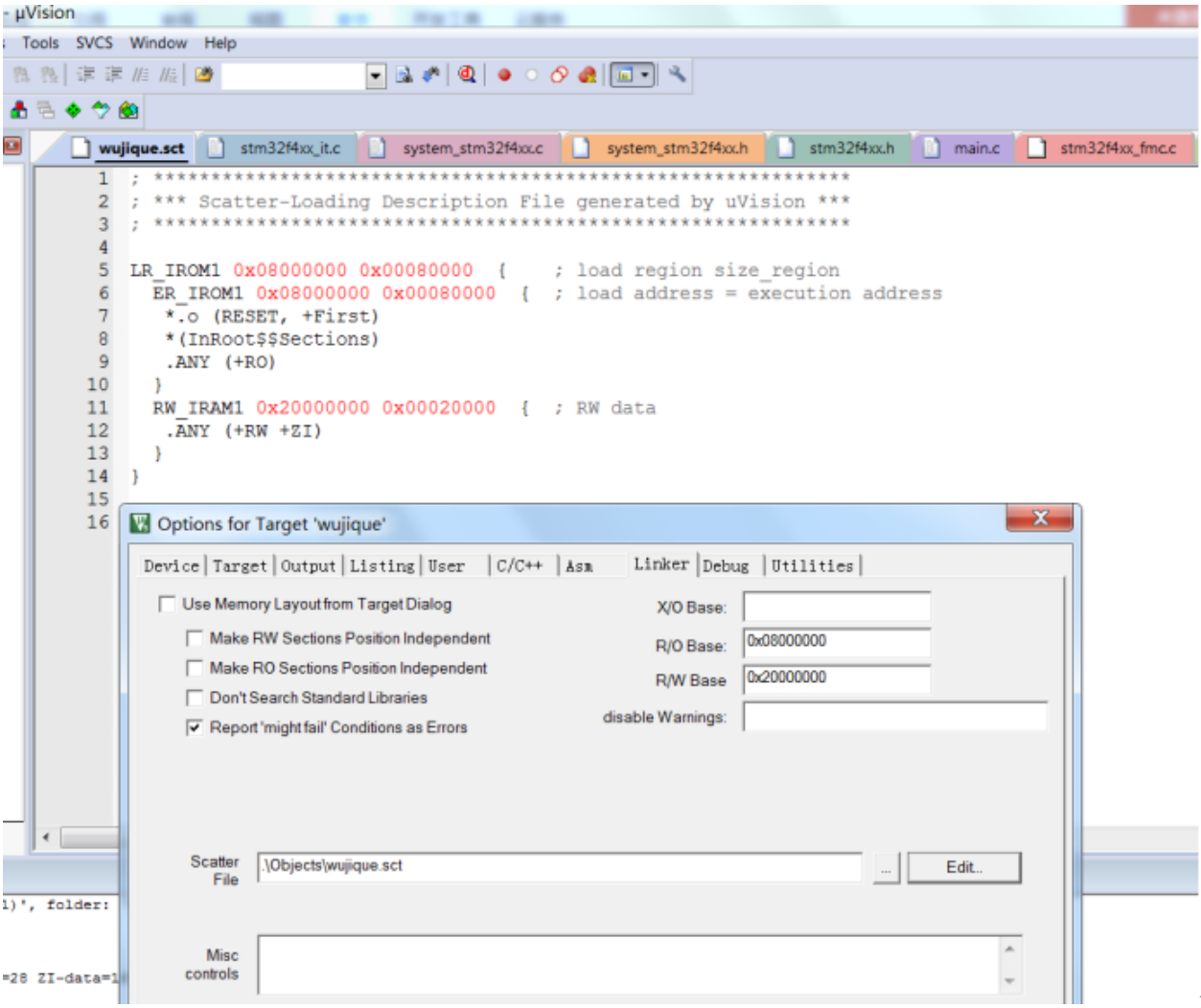


在 MDK 软件 Options 菜单 Linker 下有关于这个菜单的设置。linker 设置把 Use Memory Layout from Target Dialog 前面的勾去掉，之前不可设置的框都可以设置了。点击



Edit 进行编辑。用户自定义分散加载在代码编辑框出现了分散加载文件内容，当前文件只有基本的内容。

> 其实这个文件功能很强大，通过修改这个文件可以配置程序的很多功能，例如：1 指定 FLASH 跟 RAM 的大小于起始位置，当我们把程序分成 BOOT、CORE、APP，甚至进行驱动分离的时候，就可以用上了。2 指定函数与变量的位置，例如把函数加载到 RAM 中运行。



分散加载文件从这个基本的分散加载文件我们可以看出：

- 第 6 行 ER_IROM1 0x08000000 0x00080000 定义了 ER_IROM1，也就是我们说的内部 FLASH，从 0x08000000 开始，大小 0x00080000。
- 第 7 行 *.o (RESET, +First)* 从 *0x08000000* 开始，先放置一个.o 文件，并且用 (RESET, +First) 指定 RESET 块优先放置，RESET 块是什么？请查看启动代码，中断向量就是一个 AREA，名字叫 RESET，属于 READONLY。这样编译后，RESET 块将放在 0x08000000 位置，也就是说，中断向量就放在这个地方。DCD 是分配空间，4 字节，第一个就是 ___initial_sp，第二个就是 Reset_Handler 函数指针。也就是说，最后编译后的程序，将 Reset_Handler 这个函数的指针（地址），放在 0x800000+4 的地方。所以芯片在复位的时候，就能找到复位函数 Reset_Handler。
- 第 8 行 *(InRoot$$Sections) 什么鬼？GOOGLE 啊！回头再说。
- 第 9 行.ANY (+RO) 意思就是其他的所有 RO，顺序往后放。就是说，其他代码，跟着启动代码后面。
- 第 11 行 RW_IRAM1 0x20000000 0x00020000 定义了 RAM 大小。
- 第 12 行.ANY (+RW +ZI) 所有的 RW ZI，全部放到 RAM 里面。RW,ZI，也就是变量，这一行指定了变量保存到什么地址。

10.2.2 分析用户代码

到此，基本启动过程已经分析完。下一步开始分析用户代码，就从 main 函数开始。1 程序跳转到 main 函数后: RCC_GetClocksFreq 获取 RCC 时钟频率；SysTick_Config 配置 SysTick，在这里打开了 SysTick 中断，10 毫秒一次。Delay(5); 延时 50 毫秒。

```
int main(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;

 /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       files before to branch to application main.
       To reconfigure the default setting of SystemInit() function,
       refer to system_stm32f4xx.c file */

  /* SysTick end of count event each 10ms */
  RCC_GetClocksFreq(&RCC_Clocks);
  SysTick_Config(RCC_Clocks.HCLK_Frequency / 100);

  /* Add your application code here */
  /* Insert 50 ms delay */
  Delay(5);
```

2 初始化 IO 就不说了，进入 while(1)，也就是一个死循环，嵌入式程序，都是一个死循环，否则就跑飞了。

```
/* 初始化 LED IO 口 */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;

GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_Init(GPIOG, &GPIO_InitStructure);

/* Infinite loop */
mcu_uart_open(3);
while (1)
{
  GPIO_ResetBits(GPIOG, GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
```

(continues on next page)

(continued from previous page)

```
  Delay(100);
  GPIO_SetBits(GPIOG, GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
  Delay(100);
  mcu_uart_test();

  TestFun(TestTmp2);
}
```

3 在 while(1) 中调用 TestFun 函数，这个函数使用两个全局变量，两个局部变量。

```
/* Private functions ---------------------------------------------------------*/
u32 TestTmp1 = 5;//全局变量，初始化为 5
u32 TestTmp2;//全局变量，未初始化

const u32 TestTmp3[10] = {6,7,8,9,10,11,12,13,12,13};

u8 TestFun(u32 x)//函数，带一个参数，并返回一个 u8 值
{
        u8 test_tmp1 = 4;//局部变量，初始化
        u8 test_tmp2;//局部变量，未初始化

        static u8 test_tmp3 = 0;//静态局部变量

        test_tmp3++;

        test_tmp2 = x;

        if(test_tmp2> TestTmp1)
                test_tmp1 = 10;
        else
                test_tmp1 = 5;

        TestTmp2 +=TestTmp3[test_tmp1];

        return test_tmp1;
}
```

然后程序就一直在 main 函数的 while 循环里面执行。中断呢？对，还有中断。**中断中断，就是中断正常的程序执行流程**。我们查看 Delay 函数，uwTimingDelay 不等于 0 就死等？谁会将 uwTimingDelay 改为 0？

```
/**
  * @brief  Inserts a delay time.
  * @param  nTime: specifies the delay time length, in milliseconds.
  * @retval None
  */
void Delay(__IO uint32_t nTime)
{
  uwTimingDelay = nTime;

  while(uwTimingDelay != 0);
}
```

搜索 uwTimingDelay 变量，函数 TimingDelay_Decrement 会将变量一直减到 0。

```
/**
  * @brief  Decrements the TimingDelay variable.
  * @param  None
  * @retval None
  */
void TimingDelay_Decrement(void)
{
  if (uwTimingDelay != 0x00)
  {
    uwTimingDelay--;
  }
}
```

这个函数在哪里执行？经查找，在 SysTick_Handler 函数中运行。谁用这个函数？

```
/**
  * @brief  This function handles SysTick Handler.
  * @param  None
  * @retval None
  */
void SysTick_Handler(void)
{
  TimingDelay_Decrement();
}
```

经查找，在中断向量表中有这个函数，也即是说这个函数指针保存在中断向量表内。当发生中断时，就会执行这个函数。当然，在进出中断会有保存和恢复现场的操作。这个主要涉及到汇编，暂时不进行分析了。有兴趣自己研究研究。通常，现在我们开发程序不用关心上下文切换了。

```
__Vectors       DCD     __initial_sp               ; Top of Stack
                DCD     Reset_Handler              ; Reset Handler
                DCD     NMI_Handler                ; NMI Handler
                DCD     HardFault_Handler          ; Hard Fault Handler
                DCD     MemManage_Handler          ; MPU Fault Handler
                DCD     BusFault_Handler           ; Bus Fault Handler
                DCD     UsageFault_Handler         ; Usage Fault Handler
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     SVC_Handler                ; SVCall Handler
                DCD     DebugMon_Handler           ; Debug Monitor Handler
                DCD     0                          ; Reserved
                DCD     PendSV_Handler             ; PendSV Handler
                DCD     SysTick_Handler            ; SysTick Handler
```

10.3 余下问题

1 ___main 函数是什么函数? 是我们在 main.c 中定义的 main 函数吗? 2 分散加载文件中 *(InRoot$$Sections) 是什么？3 ZI 段，也就是初始化为 0 的数据段，什么时候初始化？谁初始化?

为什么这几个问题前面留着不说？因为这是同一个问题。顺藤摸瓜!

10.4 通过 MAP 文件了解代码构成

10.4.1 编译结果

程序编译后，在下方的 Build Output 窗口会输出信息:

```
*** Using Compiler 'V5.06 update 5 (build 528)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Build target 'wujique'
compiling stm32f4xx_it.c...
...
assembling startup_stm32f40_41xxx.s...
compiling misc.c...
...
compiling mcu_uart.c...
linking...
```

(continues on next page)

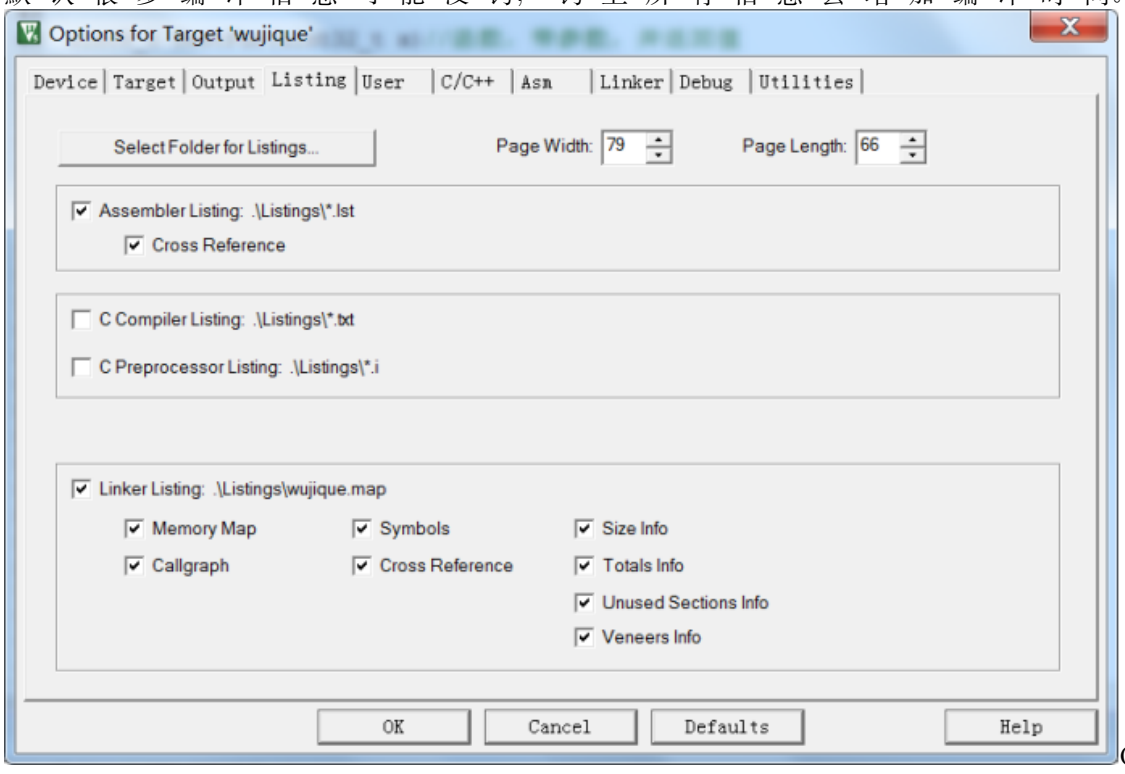
(continued from previous page)

```
Program Size: Code=9038 RO-data=990 RW-data=40 ZI-data=6000
FromELF: creating hex file...
".\Objects\wujique.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:32
```

- 编译目标是 wujique
- C 文件 compiling，汇编文件 assembling，这个过程叫编译
- 编译结束后，就进行 link，链接。
- 最后得到一个编译结果，9038 字节 code，RO 990，RW 40，ZI 6000。CODE，是代码，很好理解，那 RO、RW、ZI 都是什么?
- **FromELF，创建 hex 文件，FromELF 是一个好工具，需要自己添加到 option 中才能用**

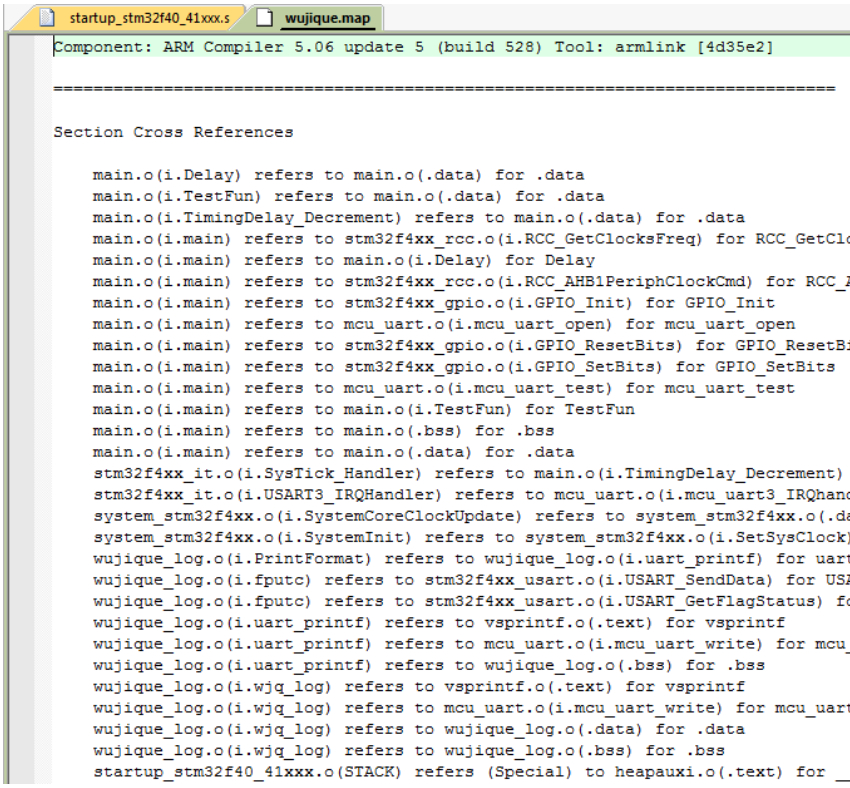
10.4.2 map 文件配置

更多编译具体信息在 map 文件中，在 MDK Options 中我们可以看到，所有信息都放在\Listings\wujique.map 默认很多编译信息可能没钩，钩上所有信息会增加编译时间。



Options 中的 MAP 设置

10.4.3 map 文件



打开 map 文件，好乱?习惯就好。我们抓重点就行了。乱的 MAP

- map 总信息

从最后看起，看到没？**最后的这一段 map 内容，说明了整个程序的基本概况**。有多少 RO？RO 到底是什么？有多少 RW?RW 又是什么？ROM 为什么不包括 ZI Data？为什么包含 RW Data?

```
==============================================================================

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      9038        554        990         40       6000     328889   Grand Totals
      9038        554        990         40       6000     328889   ELF Image Totals
      9038        554        990         40          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                10028 (   9.79kB)
    Total RW  Size (RW Data + ZI Data)              6040 (   5.90kB)
    Total ROM Size (Code + RO Data + RW Data)      10068 (   9.83kB)

==============================================================================
```

MAP 最后

- Image component sizes

往上，看看 Image component sizes，这个就比刚刚的总体统计更细了。**这**

部分内容，说明了每个源文件的概况首先，是我们自己的源码，这个程序我们的代码不多，只有 main.o，wujique_log.o，和其他一些 STM32 的库文件。

```
================================================================================

Image component sizes


      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

       236         32          0         16         16     300463   main.o
       724        150         14          6       4096       4461   mcu_uart.o
       104          6          0          0          0       1164   misc.o
        64         26        392          0       1536        848   startup_stm32f40_41xxx.o
       164          0          0          0          0       3111   stm32f4xx_gpio.o
        24          0          0          0          0       4522   stm32f4xx_it.o
       216         24          0         16          0       4466   stm32f4xx_rcc.o
       384          8          0          0          0       7437   stm32f4xx_usart.o
       276         34          0          0          0       1489   system_stm32f4xx.o
        64          8          0          1        256       2244   wujique_log.o

    --------------------------------------------------------------------------
      2264        288        438         40       5904     330205   Object Totals
         0          0         32          0          0          0   (incl. Generated)
         8          0          0          1          0          0   (incl. Padding)

    --------------------------------------------------------------------------
```

Image component sizes

第 2 部分是库里面的文件，看到没？里面有一个 ___main.o。___main 函数是不是我们写的 main 函数？明显不是，我们的 main 函数是放在 main.o 文件。这么小的一个工程，用了这么多库，你以前关注过吗？估计没有，除非你曾经将一个原本在 1M flash 上的程序压缩到能在 512K 上运行。

```
    Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member Name

       8          0          0          0          0         68   __main.o
     392          4         17          0          0         92   __printf_flags_ss_wp.o
      14          0          0          0          0         68   __printf_wp.o
       0          0          0          0          0          0   __rtentry.o
      12          0          0          0          0          0   __rtentry2.o
       6          0          0          0          0          0   __rtentry4.o
      52          8          0          0          0          0   __scatter.o
      26          0          0          0          0          0   __scatter_copy.o
      28          0          0          0          0          0   __scatter_zi.o
       6          0          0          0          0          0   _printf_a.o
       6          0          0          0          0          0   _printf_c.o
      44          0          0          0          0        108   _printf_char.o
      48          6          0          0          0         96   _printf_char_common.o
      40          0          0          0          0         68   _printf_charcount.o
       6          0          0          0          0          0   _printf_d.o
     120         16          0          0          0         92   _printf_dec.o
       6          0          0          0          0          0   _printf_e.o
       6          0          0          0          0          0   _printf_f.o
    1054          0          0          0          0        216   _printf_fp_dec.o
     764          8         38          0          0        100   _printf_fp_hex.o
     128         16          0          0          0         84   _printf_fp_infnan.o
       6          0          0          0          0          0   _printf_g.o
     148          4         40          0          0        160   _printf_hex_int_ll_ptr.o
       6          0          0          0          0          0   _printf_i.o
     178          0          0          0          0         88   _printf_intcommon.o
```

Image component 库

```
--------------------------------------------------------------------

  Code (inc. data)   RO Data    RW Data    ZI Data      Debug

    6694        266        551          0         96       3892
      18          0          0          0          0        348
      48          0          0          0          0        124

--------------------------------------------------------------------
    6774        266        552          0         96       4364

--------------------------------------------------------------------
```

第 3 部分也是库，暂时没去分析这两个是什么东西。

component 第三部分

> 库文件是什么？库文件就是别人已经别写好的代码库。在代码中，我们经常会包含一些头文件，例如：
>
> ```
> #include <stdarg.h>
> #include <stdlib.h>
> #include <string.h>
> ```
>
> 这些就是库的头文件。这些头文件保存在 MDK 开发工具的安装目录下。我们经常用的库函数有：memcpy、memcmp、strcmp 等。只要代码中包含了这些函数，就会链接库文件。

- 文件 map

再往上，就是文件 MAP 了，也就时**每个文件中的代码段（函数）跟变量在 ROM 跟 RAM 中的位置**。首先是 ROM 在 0x08000000 确实放的是 startup_stm32f40_41xxx.o 中的 RESET

```
==============================================================================

Memory Map of the image

  Image Entry point : 0x08000189

  Load Region LR_IROM1 (Base: 0x08000000, Size: 0x00002754, Max: 0x00080000, ABSOLUTE)

    Execution Region ER_IROM1 (Exec base: 0x08000000, Load base: 0x08000000, Size: 0x0000272c, Max: 0x00080000, ABSOLUTE)

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08000000   0x08000000   0x00000188   Data   RO          357    RESET               startup_stm32f40_41xxx.o
    0x08000188   0x08000188   0x00000008   Code   RO         5444  * !!!main             c_w.l(__main.o)
    0x08000190   0x08000190   0x00000034   Code   RO         5765    !!!scatter          c_w.l(__scatter.o)
    0x080001c4   0x080001c4   0x0000001a   Code   RO         5767    !!handler_copy      c_w.l(__scatter_copy.o)
    0x080001de   0x080001de   0x00000002   PAD
    0x080001e0   0x080001e0   0x0000001c   Code   RO         5769    !!handler_zi        c_w.l(__scatter_zi.o)
```

memory

```
    0x080021f4   0x080021f4   0x000000dc   Code   RO         5372    i.mcu_u
    0x080022d0   0x080022d0   0x00000048   Code   RO         5373    i.mcu_u
    0x08002318   0x08002318   0x000000b8   Code   RO         5376    i.mcu_u
    0x080023d0   0x080023d0   0x0000005c   Code   RO         5377    i.mcu_u
    0x0800242c   0x0800242c   0x00000040   Code   RO          300    i.wjq_l
```

map rom 每个文件有有多行，例如串口，4 个函数。

map rom

然后是 RAM 的，main.o 中的变量，放在 0x20000000，总共有 0x0000000c，类型是 Data、RW。串口有两种变量，data 和 bss，什么是 bss？这两个名称，是 section name，也就是段的意思。看前面 type 和 Attr，**RW Data，放在.data 段；RW Zero 放在.bss 段，RW Zero，其实就是 ZI**。到底哪些变量是 RW，哪些是 ZI？

```
Execution Region RW_IRAM1 (Exec base: 0x20000000, Load base: 0x0800272c, Size: 0x00001798, Max: 0x00020000, ABSOLUTE)

Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

0x20000000   0x0800272c   0x00000010   Data   RW          9    .data               main.o
0x20000010   0x0800273c   0x00000001   Data   RW        302    .data               wujique_log.o
0x20000011   0x0800273d   0x00000010   Data   RW       3219    .data               stm32f4xx_rcc.o
0x20000021   0x0800274d   0x00000001   PAD
0x20000022   0x0800274e   0x00000006   Data   RW       5380    .data               mcu_uart.o
0x20000028       -        0x00000010   Zero   RW          8    .bss                main.o
0x20000038       -        0x00000100   Zero   RW        301    .bss                wujique_log.o
0x20000138       -        0x00001000   Zero   RW       5378    .bss                mcu_uart.o
0x20001138       -        0x00000060   Zero   RW       5632    .bss                c_w.l(libspace.o)
0x20001198       -        0x00000200   Zero   RW        356    HEAP                startup_stm32f40_41xxx.o
0x20001398       -        0x00000400   Zero   RW        355    STACK               startup_stm32f40_41xxx.o
```

memory map ram

- Image Symbol Table

再往上就是 Image Symbol Table，就更进一步到每个函数或者变量的信息了。

```
================================================================================

Image Symbol Table

    Local Symbols

    Symbol Name                             Value     Ov Type        Size  Object(Section)

    ../clib/angel/boardlib.s                0x00000000   Number         0  boardinit1.o ABSOLUTE
    ../clib/angel/boardlib.s                0x00000000   Number         0  boardinit2.o ABSOLUTE
    ../clib/angel/boardlib.s                0x00000000   Number         0  boardinit3.o ABSOLUTE
    ../clib/angel/boardlib.s                0x00000000   Number         0  boardshut.o ABSOLUTE
    ../clib/angel/handlers.s                0x00000000   Number         0  __scatter_copy.o ABSOLUTE
    ../clib/angel/handlers.s                0x00000000   Number         0  __scatter_zi.o ABSOLUTE
    ../clib/angel/kernel.s                  0x00000000   Number         0  __rtentry.o ABSOLUTE
```

MAP 变量

例如，全局变量 TestTmp1，是 Data，4 字节，分配的位置是 0x20000004。

```
    TestTmp1                                0x20000004   Data           4  main.o(.data)
    TestTmp2                                0x2000000c   Data           4  main.o(.data)
    LogLevel                                0x20000010   Data           1  wujique_log.o(.data)
    UartBuf3OverFg                          0x20000022   Data           1  mcu_uart.o(.data)
    UartHead3                               0x20000024   Data           2  mcu_uart.o(.data)
```

MAP

TestTmp1 TestTmp3 数组放在哪里？**放在 0X080024E0 这个地方**，这可是代码区额。因为我们用 const 修饰了这个全局变量数组，告诉编译器，这个数组是不可以改变的，编译器就将这个数组保存到代码中了。程序中我们经常会使用一些大数组数据，例如字符点阵，通常有几 K 几十 K 大，不可能也没必要放到 RAM 区，整个程序运行过程这些数据都不改变，因此通过 const 修饰，将其存放到代码区。

const 的用处比较多，可以修饰变量，也可以修饰函数。更多用法自行学习

```
    __I$use$fp                              0x080024de   Number         0  usenofp.o(x$fpl$usenofp)
    TestTmp3                                0x080024e0   Data          40  main.o(.constdata)
    Region$$Table$$Base                     0x08002614   Number         0  anon$$obj.o(Region$$Table)
    Region$$Table$$Limit                    0x08002634   Number         0  anon$$obj.o(Region$$Table)
    __ctype                                 0x0800265d   Data           0  lc_ctype_c.o(locale$$data)
```

MAP

TestTmp1

```
    .data                                   0x20000000   Section
    test_tmp3                               0x20000000   Data
    uwTimingDelay                           0x20000008   Data
    .data                                   0x20000010   Section
```

那局部变量存放在哪里呢?我们找到了 test_tmp3，TestTmp1 没找到 test_tmp1/test_tmp2，为什么呢？在定义时，test_tmp3 增加了 static 定义，意思就是

静态局部变量，功能上，相当于全局变量，定义在函数内，限制了这个全局变量只能在这个函数内使用。哪 test_tmp1、test_tmp2 放在哪里呢？**局部变量，在编译链接时，并没有分配空间，只有在运行时，才从栈分配空间**。

```
u8 TestFun(u32 x)//函数，带一个参数，并返回一个 u8 值
{
        u8 test_tmp1 = 4;//局部变量，初始化
        u8 test_tmp2;//局部变量，未初始化

        static u8 test_tmp3 = 0;//静态局部变量
```

上一部分，我们留了一个问题，哪些变量是 RW，哪些是 ZI？我们看看串口变量的情况，UartBuf3 放在 bss 段，其他变量放在.data 段。为什么数组就放在 bss？bss 是英文 Block Started by Symbol 的简称。

```
__ctype                          0x0800265d   Data          0  lc_ctype_c.o(loca
TestTmp1                         0x20000004   Data          4  main.o(.data)
TestTmp2                         0x2000000c   Data          4  main.o(.data)
LogLevel                         0x20000010   Data          1  wujique_log.o(.da
UartBuf3OverFg                   0x20000022   Data          1  mcu_uart.o(.data)
UartHead3                        0x20000024   Data          2  mcu_uart.o(.data)
UartEnd3                         0x20000026   Data          2  mcu_uart.o(.data)
RCC_Clocks                       0x20000028   Data         16  main.o(.bss)
string                           0x20000038   Data        256  wujique_log.o(.bs
UartBuf3                         0x20000138   Data       4096  mcu_uart.o(.bss)
__libspace_start                 0x20001138   Data         96  libspace.o(.bss)
__temporary_stack_top$libspace   0x20001198   Data          0  libspace.o(.bss)
```

串口变量

到这里，我们可解释下面几个概念了：

Code 就是代码，函数。RO Data，就是只读变量，例如用 const 修饰的数组。RW Data，就是读写变量，例如全局变量跟 static 修饰的局部变量。ZI Data，就是系统自动初始化为 0 的读写变量，大部分是数组，放在 bss 段。RO Size 等于代码加只读变量。RW Size 等于读写变量（包括自动初始化为 0 的），这个也就是 RAM 的大小。ROM Size，也就是我们编译之后的目标文件大小，也就是 FLASH 的大小。但是？为什么会包含 RW Data 呢？因为所有全局变量都需要一个初始化的值（就算没有真正初始化，系统也会分配一个初始化空间），例如我们定义一个变量 u8 i = 8; 这样的全局变量，8，这个值，就需要保存在 FALSH 区。

```
==============================================================================

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      9038        554        990         40       6000     328889   Grand Totals
      9038        554        990         40       6000     328889   ELF Image Totals
      9038        554        990         40          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                10028 (   9.79kB)
    Total RW  Size (RW Data + ZI Data)              6040 (   5.90kB)
    Total ROM Size (Code + RO Data + RW Data)      10068 (   9.83kB)

==============================================================================
```

MAP

最后

我们看看函数的情况，前面我们不是有一个问题吗？___main 和 main 是一个函数吗？查找 main 后发现，main 是 main，放在 0x08000579

```
__ARM_fpclassify                0x0800207b   Thumb Code    48  fpclassify.o(i.__ARM_fpclassify)
_is_digit                       0x080020ab   Thumb Code    14  __printf_wp.o(i._is_digit)
main                            0x080020b9   Thumb Code   142  main.o(i.main)
mcu_uart3_IRQhandler            0x08002159   Thumb Code   144  mcu_uart.o(i.mcu_uart3_IRQhandler)
mcu_uart_open                   0x080021f5   Thumb Code   208  mcu_uart.o(i.mcu_uart_open)
mcu_uart_read                   0x080022d1   Thumb Code    64  mcu_uart.o(i.mcu_uart_read)
mcu_uart_test                   0x08002319   Thumb Code    72  mcu_uart.o(i.mcu_uart_test)
mcu_uart_write                  0x080023d1   Thumb Code    86  mcu_uart.o(i.mcu_uart_write)
```

map

main

```
__Vectors_Size                  0x00000188   Number         0  startup_stm32f
__Vectors                       0x08000000   Data           4  startup_stm32f
__Vectors_End                   0x08000188   Data           0  startup_stm32f
__main                          0x08000189   Thumb Code     8  __main.o(!!!ma
__scatterload                   0x08000191   Thumb Code     0  __scatter.o(!!
__scatterload_rt2               0x08000191   Thumb Code    44  __scatter.o(!!
```

___main 是 ___main，放在 0x08000189

___main

___main 到 main 之间发生了什么？还记得分散加载文件中的这句吗？

```
*(InRoot$$Sections)
```

___main 就在这个段内。下图是 ___main 的地址，在 0x08000189。___Vectors 就是中断向量，放在最开始。

```
__Vectors_Size                  0x00000188   Number         0  startup_stm32f40_41xxx.o ABSOLUTE
__Vectors                       0x08000000   Data           4  startup_stm32f40_41xxx.o(RESET)
__Vectors_End                   0x08000188   Data           0  startup_stm32f40_41xxx.o(RESET)
__main                          0x08000189   Thumb Code     8  __main.o(!!!main)
__scatterload                   0x08000191   Thumb Code     0  __scatter.o(!!!scatter)
__scatterload_rt2               0x08000191   Thumb Code    44  __scatter.o(!!!scatter)
__scatterload_rt2_thumb_only    0x08000191   Thumb Code     0  __scatter.o(!!!scatter)
__scatterload_null              0x0800019f   Thumb Code     0  __scatter.o(!!!scatter)
__scatterload_copy              0x080001c5   Thumb Code    26  __scatter_copy.o(!!handler_copy)
  scatterload_zeroinit          0x080001e1   Thumb Code    28    scatter_zi.o(!!handler_zi)
```

map

vectors

```
5  LR_IROM1 0x08000000 0x00080000  {    ; lo
6    ER_IROM1 0x08000000 0x00080000  {  ; lo
7     *.o (RESET, +First)
8     *(InRoot$$Sections)
9     .ANY (+RO)
10   }
```

在分散加载文件中，紧跟 RESET 的就是 *(InRoot$$Sections)。

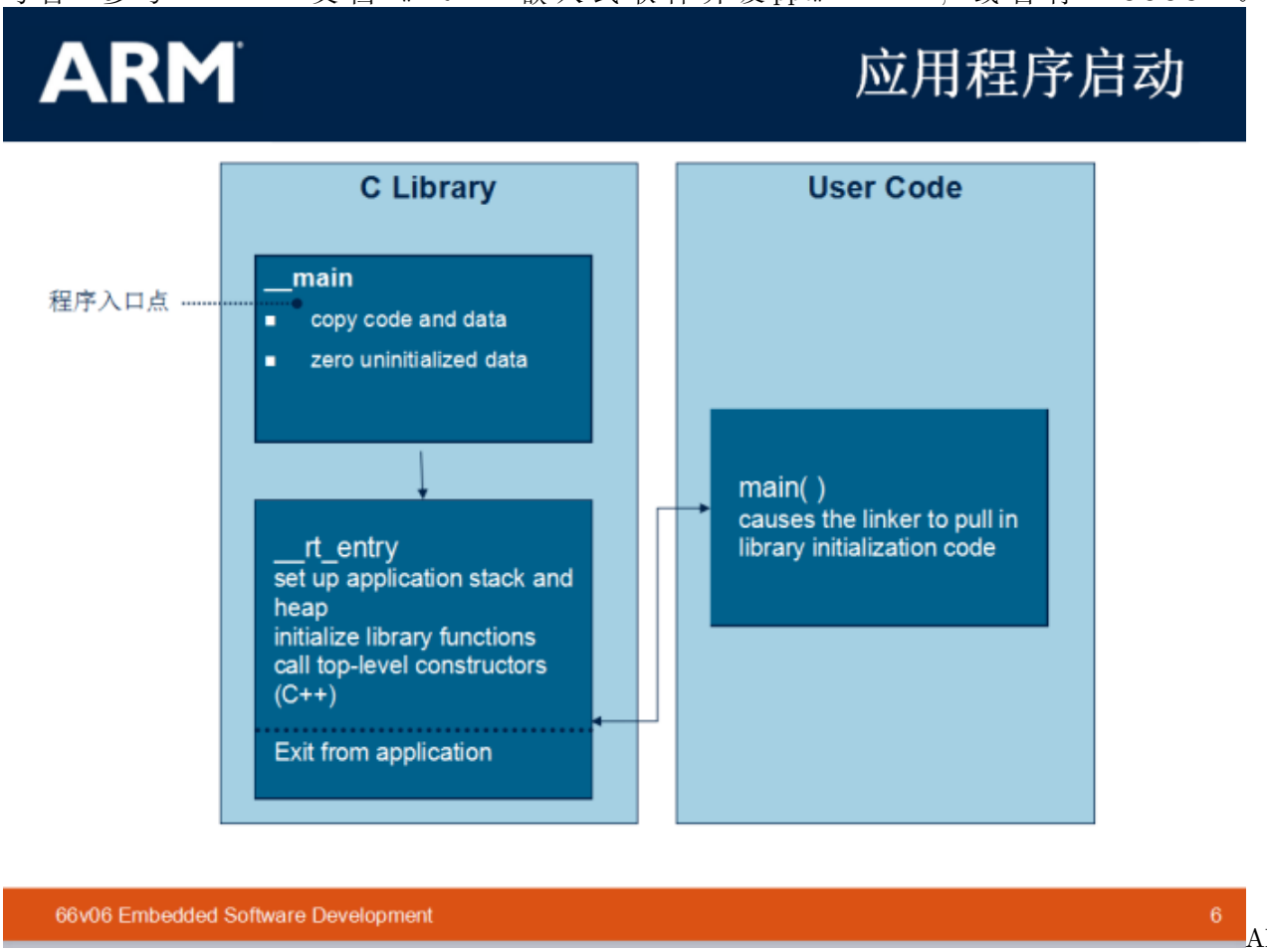
散加载文件

```
Memory Map of the image

  Image Entry point : 0x08000189

  Load Region LR_IROM1 (Base: 0x08000000, Size: 0x00000658, Max: 0x00080000, ABSOL

    Execution Region ER_IROM1 (Base: 0x08000000, Size: 0x0000063c, Max: 0x00080000

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08000000   0x00000188   Data   RO          278    RESET               startu
    0x08000188   0x00000008   Code   RO         5277  * !!!main             c_w.l
    0x08000190   0x00000034   Code   RO         5434    !!!scatter          c_w.l
    0x080001c4   0x0000001a   Code   RO         5436    !!handler_copy      c_w.l
```

而且，RESET 段正好大小 0x00000188。

大小

巧合？参考 PPT 文档《ARM 嵌入式软件开发.ppt》，或自行 GOOGLE。



ARM 应用程序启动

这一段代码都完成什么功能呢？主要完成 ZI 代码的初始化，也就是将一部分 RAM 初始化为 0。其他环境初始化。。。。**通常，我们不用管这一部分**。

- 其他再往上，就是其他信息了，例如优化了哪些东西，移除了哪些函数。

10.5 最后

到这里，一个程序，是怎么组成的，程序是如何运行的，基本有一个总体印象了。不过，对于中断，后面还会进行详细说明。

10.6 end

CHAPTER 11

IO 输入-按键检测

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

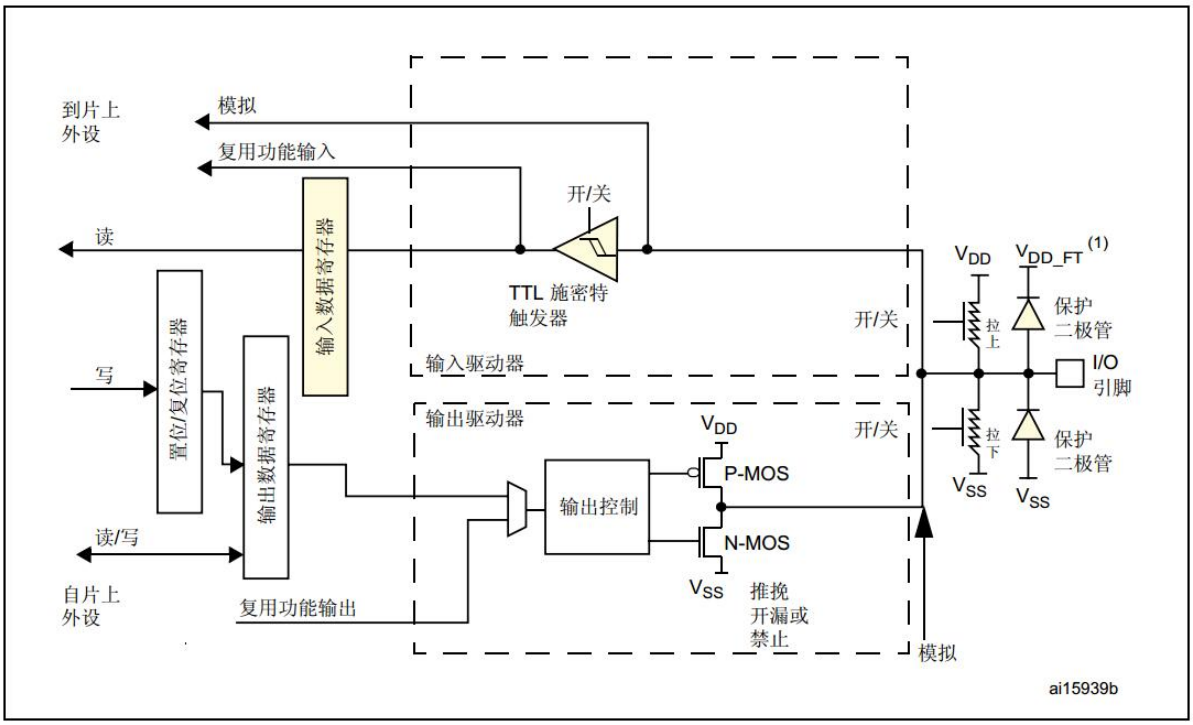
前面我们已经学习了 IO 口输出功能，现在我们就学习 IO 的输入功能。

11.1 IO 口输入

所谓的 IO 口输入，其实非常简单，就是可以从一个 IO 口读取连接在 IO 口上的电路的电平，高电平，读到 1，低电平就读到 0。下面是 IO 口结构图，前面我们

已经看过，如果配置为输入时，只要读取输入数据寄存器就可以获取 IO 口电平了。

图 17. 5 V 容忍 I/O 端口位的基本结构



IO 口结构在 IO 口输出章节，我们详细说了 IO 口配置，其中下面两个配置只是针对输出的，输入无效：

```
/**
  * @brief  GPIO Output type enumeration
  */
typedef enum
{
  GPIO_OType_PP = 0x00,
  GPIO_OType_OD = 0x01
}GPIOOType_TypeDef;
#define IS_GPIO_OTYPE(OTYPE) (((OTYPE) == GPIO_OType_PP) || ((OTYPE) == GPIO_OType_OD))

/**
  * @brief  GPIO Output Maximum frequency enumeration
  */
typedef enum
{
  GPIO_Low_Speed     = 0x00, /*!< Low speed     */
  GPIO_Medium_Speed  = 0x01, /*!< Medium speed */
  GPIO_Fast_Speed    = 0x02, /*!< Fast speed    */
  GPIO_High_Speed    = 0x03  /*!< High speed    */
```

(continues on next page)

(continued from previous page)

```
}GPIOSpeed_TypeDef;
```

11.2 按键输入

按键输入是人机交互的一个重要输入手段，最常见的按键就是电脑键盘。按键输入是 IO 口输入应用的最简单例子。然而深究起来，按键扫描并没那么简单。

11.2.1 原理图

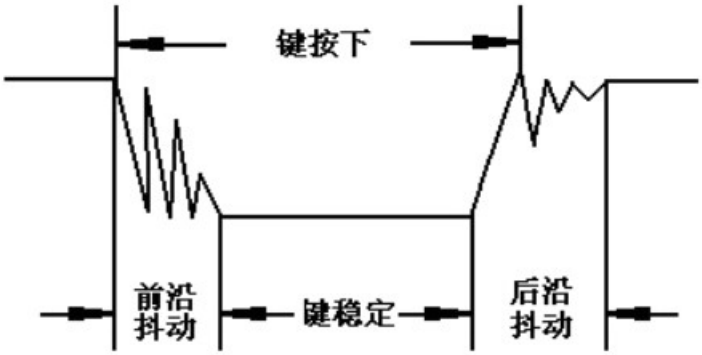
先看按键原理图 单个按键原理图按键的原理比较简单，一个按键两个脚，一个脚接都 IO 口，一个脚接到地。当按键按下，两端短路，IO 口就接到地，就是低电平。那没按下时，IO 口啥都没接，就是高电平。为啥是高电平呢？因为 IO 口在芯片内部可以配置连接一个内部上拉电阻。如果使用了没有内部上拉电阻的 IO，就只能在外部接一个电阻将 IO 口上拉到高电平。

> 上拉电阻不能太小，当按键按下时，VCC 经过电阻接到地，电流就等于 VCC/电阻，太小，漏电流会很大。普通按键按下的时间比较短，如果是一些状态开关（原理和按键类似），接地可能是一个常态，长时间漏电，费电。上拉电阻也不能太大，如果你整一个 10M 的电阻，很容易耦合干扰，IO 口电平乱跳，造成假按键动作。一般，不是低功耗的设备，4.7K 挺好，低功耗设备，1M 差不多。

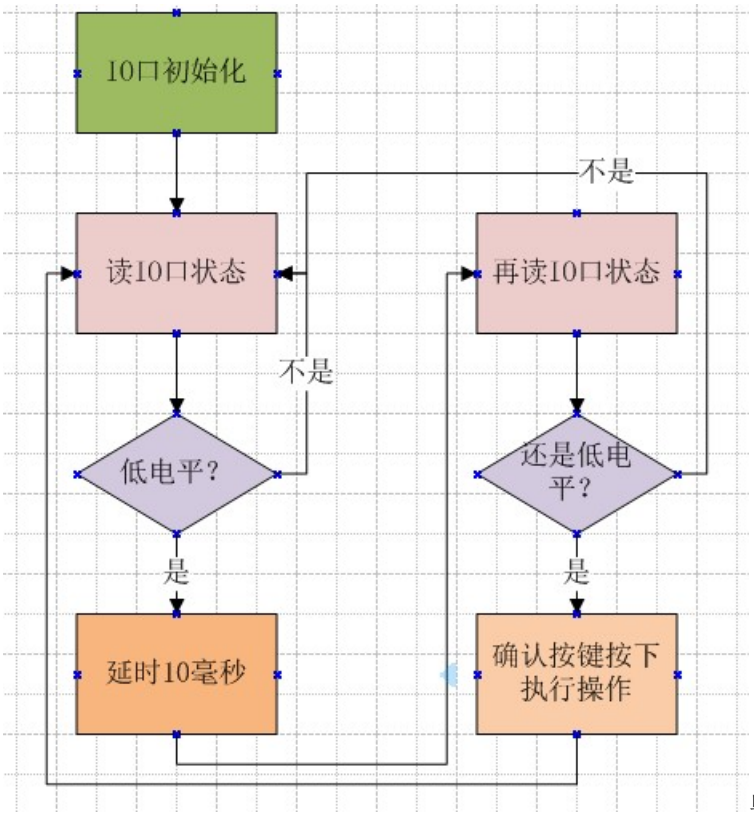
11.2.2 按键扫描方式

首先，记住流水灯章节提到的问题：**芯片跑得很快**。从一个 IO 口读取输入电平，只是一瞬间的事。第二，手可能会抖动。第三，机械按键可能会抖动。

> 用示波器抓按键按下的波形，波形可能如下图，可见状态变化时波形有抖动。



按键抖动



大概的按键流程如下：单键扫描流程这个流程只是判断按下的流程，按键松开同样要做抖动处理。

11.3 编码调试

按键属于**芯片外设备**。我们在工程目录增加一个 board_dev 文件夹，在文件夹内添加两个文件：dev_key.c、dev_key.h。代码具体见源文件。记得添加到 MDK 工程，头文件路径也要添加。我们首先调试好 IO 口输入，如下面代码，通过两个调试信息从串口查看 IO 口电平。

```
s32 dev_key_scan(void)
{
        uint8_t sta;

        sta = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
        if(sta == Bit_SET)
        {
                KEY_DEBUG(LOG_DEBUG, "key up\r\n");
        }
        else
        {
                KEY_DEBUG(LOG_DEBUG, "key down\r\n");
        }
```

(continues on next page)

(continued from previous page)

```
}
```

将这个函数放到 main.c 的 while 循环中运行，在 while(1) 之前，要调用函数 dev_key_init，对按键 IO 进行初始化，初始化为输入 IO 口。

```
        /* Infinite loop */
        mcu_uart_open(3);
        dev_key_init();
        while (1)
        {
                dev_key_scan();
                Delay(10);

        }
```

没有按下按键时，串口输出 “key up”，按住按键时输出 “key down”。由于程序一直运行，LOG 会连续不断输出 IO 口状态。IO 口调试好之后就处理防抖动。添加防抖处理的扫描函数如下，注意定义变量的时候，使用了 volatile 跟 static 关键字。

```
s32 dev_key_scan(void)
{
        volatile uint8_t sta;//局部变量，放在栈空间，进入函数时使用，退出后释放。
        static u8 key_sta = 1;//通过 static 指定 key_sta，函数退出不会释放

        sta = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
        if((sta == Bit_SET) && (key_sta == 0))
        {
                Delay(5);
                sta = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
                if(sta == Bit_SET)
                {
                        key_sta = 1;
                        KEY_DEBUG(LOG_DEBUG, "key up\r\n");
                        return 1;
                }
        }
        else if((sta == Bit_RESET) && (key_sta == 1))
        {
                Delay(5);
                sta = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
                if(sta == Bit_RESET)
```

(continues on next page)

(continued from previous page)

```
                {
                        key_sta = 0;
                        KEY_DEBUG(LOG_DEBUG, "key down\r\n");
                        return 0;
                }
        }
        else
        {
                /* 按键没变化 */

                return -1;
        }
}
```

有个要点：

> key_sta 变量定义，在函数内定义的是局部变量，但是通过一个 static 关键字修饰，即使函数退出，也不释放，下次进入函数，key_sta 的值就不会变。static 修饰函数内的局部变量，变量生命周期上，相当于全局变量。使用范围，相当于局部变量，只能在函数内使用。

这个扫描函数的流程完全按照前面的流程图处理：

> 第 6 行读 IO 口状态第 7 行判断状态是不是等于 Bit_Set，也就是高电平，并且 key_sta 等于 0，也就是说，上一次的状态是 0，状态是按下。这样做的目的是，我们只处理状态变化。第 9 行，延时第 10 行，再次读 IO 状态第 11 行，判断状态是否为高电平。第 13 行，将按键状态改为 1，高电平，松开状态。第 18 到 28 行是按下状态处理，原理跟松开一样。

main 函数中，按键按下，点亮 LED，按键松开，熄灭 LED。

```
/* Infinite loop */
        mcu_uart_open(3);
        wjq_log(LOG_INFO, "hello word!\r\n");
        dev_key_init();
        while (1)
        {
                s32 key;
                key = dev_key_scan();
                if(key == 0)
                {
                        GPIO_ResetBits(GPIOG, GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2| GPIO_
↪Pin_3);
                }
                else if(key == 1)
```

(continues on next page)

(continued from previous page)

```
                {

                        GPIO_SetBits(GPIOG, GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2| GPIO_
↪Pin_3);

                }
                Delay(1);

        }
```

重新编译下载进去后即可验证。

11.4 优化改造

到此，一个大家在其他教程中常见，基本的按键扫描就实现了。但是，能用吗？为什么？我认为结论是**不能用**。有以下问题：

1. 扫描过程去抖动用了**硬延时**，俗称死等，啥都没干，白白浪费 CPU 的时间。
2. 应用和驱动耦合在一起，耦合就是强关联。看上面程序，应用就是 main.c 中的读到按键后点亮 LED。驱动就是按键扫描，也就是 dev_key_scan 函数。应用直接通过调用驱动获取键值，属于耦合（**好的设计应该调用驱动提供的接口**）。耦合有什么不好呢？

1. 驱动只有应用要读按键了才进行按键扫描，多个应用怎么办？如果应用执行不及时，按键会丢失吧？。——**驱动受制于应用**
2. 扫描到按键直接就给应用。——**驱动拖累应用**

11.4.1 改造后关键代码-扫描

关键思路：每次进入 scan 函数，只做状态判断。 scan 函数可以放到定时器执行，等移植 RTOS 后，也可以放到线程内。现在我们先放到 main 函数的 while(1) 中，这个循环内有一个 Delay，也就相当于间隔 10 毫秒执行一次 scan。**防抖动通过多次扫描实现。 扫描到按键后，将键值写入按键缓冲，至于谁要用，什么时候用，驱动不管**。具体流程请看函数注释。

```
/**
 *@brief:      dev_key_scan
 *@details:    扫描按键
 *@param[in]   void
 *@param[out]  无
```

(continues on next page)

(continued from previous page)

```
 *@retval:
 */
s32 dev_key_scan(void)
{
        volatile u8 sta;//局部变量，放在栈空间，进入函数时使用，退出后释放。
        static u8 new_sta = Bit_SET;
        static u8 old_sta = Bit_SET;
        u8 key_value;
        static u8 cnt = 0;

        if(KeyGd != 0)
                return -1;

        /* 读按键状态 */
        sta = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
        /*
                判断跟上次读的状态是不是一样，
                原因是，保证防抖过程的状态是连续一样的。
                不明白可以想象按键状态快速变化。
                这种情况我们不要认为是有按键。
        */
        if((sta != new_sta))
        {
                cnt = 0;
                new_sta = sta;
        }
        /*
                与上次得到键值的状态比较，
                如果不一样，说明按键有变化
        */
        if(sta != old_sta)
        {
                cnt++;

                if(cnt >= DEV_KEY_DEBOUNCE)
                {
                        /* 防抖次数达到，扫描到一个按键变化 */
                        cnt = 0;
                        key_value = DEV_KEY_PRESS;
```

(continues on next page)

(continued from previous page)

```
                        /* 判断是松开还是按下 */
                        if(sta == Bit_RESET)
                        {
                                KEY_DEBUG(LOG_DEBUG, "key press!\r\n");
                        }
                        else
                        {
                                key_value += DEV_KEY_PR_MASK;
                                KEY_DEBUG(LOG_DEBUG, "key rel!\r\n");
                        }
                        /* 键值写入环形缓冲 */
                        KeyBuf[KeyW] = key_value;
                        KeyW++;
                        if(KeyW>= KEY_BUF_SIZE)
                        {
                                KeyW = 0;
                        }
                        /* 更新状态 */
                        old_sta = new_sta;
                }
        }
        return 0;
}
```

11.4.2 应用驱动分离

按键驱动改造后，应用也要同步修改，修改如下。虽然现在扫描和点灯还是放在 *main* 中，不过这个只是当前测试而已。而且，虽然放在一起，两个模块在逻辑上已经没有关系了。就算没有点灯程序，按键扫描也会按照自己的设计执行。跟点灯不再强关联。

```
/* Infinite loop */
        mcu_uart_open(3);
        wjq_log(LOG_INFO, "hello word!\r\n");
        dev_key_init();

        dev_key_open();
        while (1)
        {
                /* 驱动轮询 */
```

(continues on next page)

(continued from previous page)

```
                dev_key_scan();

                /* 应用 */
                u8 key;
                s32 res;

                res = dev_key_read(&key, 1);
                if(res == 1)
                {
                        if(key == DEV_KEY_PRESS)
                        {
                                GPIO_ResetBits(GPIOG, GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_
↪2| GPIO_Pin_3);
                        }
                        else if(key == DEV_KEY_REL)
                        {

                                GPIO_SetBits(GPIOG, GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_
↪2| GPIO_Pin_3);
                        }
                }

                Delay(1);

        }
```

11.5 总结

改造后的程序，完全可以用于实际项目。在按键驱动中，增加了 open 和 close 函数，还有设备描述符。

```
/* 按键设备符 */
s32 KeyGd = -1;
```

这都是为了后续**所有设备驱动统一管理**做准备。

11.6 end

CHAPTER 12

定时器-定时-说中断

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

定时器是芯片上一个最重要的外设。在写代码时，经常需要一段代码延时一定时间后执行；或者是一段程序间隔一定时间不断重复执行。最典型的就是嵌入式操作系统，例如 freertos，就需要一个定时器作为系统调度心跳时钟。在调试按键扫描时提到，我们可以将 scan 函数放到定时器中执行，这也是定时器驱动程序执行的一个例子。定时器是一个统称，实际上**定时**仅仅是定时器的一个功能。定时器的功能通常包括：**定时**、**PWM 输出**、**输入捕获**、**输入计数**等功能。这次，我们先调试定时功能。

12.1 定时器

定时器是什么？最简单的定时器就像一个倒计时时钟，大家的手机应该都有这个功能。设定一个时间，例如 10 分钟，10 分钟一到，闹铃就响起。那单片机的定时器跟手机的定时器有什么区别呢?

1. 我们用单片机定时器，通常定时都是 us/ms/s，很少定时几分钟的。
2. 单片机定时时间到了，不会直接响闹铃，一般只会在某个状态寄存器的某个 bit 置位，表明定时到了。如果这个定时器使能了中断，就会产生一个中断。
3. 单片机的定时器功能更多，例如重复定时，假如定时 1s，如果配置为自动重复，那么只要启动定时器，每一秒都会产生闹铃，直到定时器被停止。
4. 设置更复杂，手机要定时 10 分钟，直接拨指针或者输入 10 就可以了。单片机要定时 10S，要根据系统时钟计算，选择合适的预分频和定时计数。还要设置相关的中断开关等。

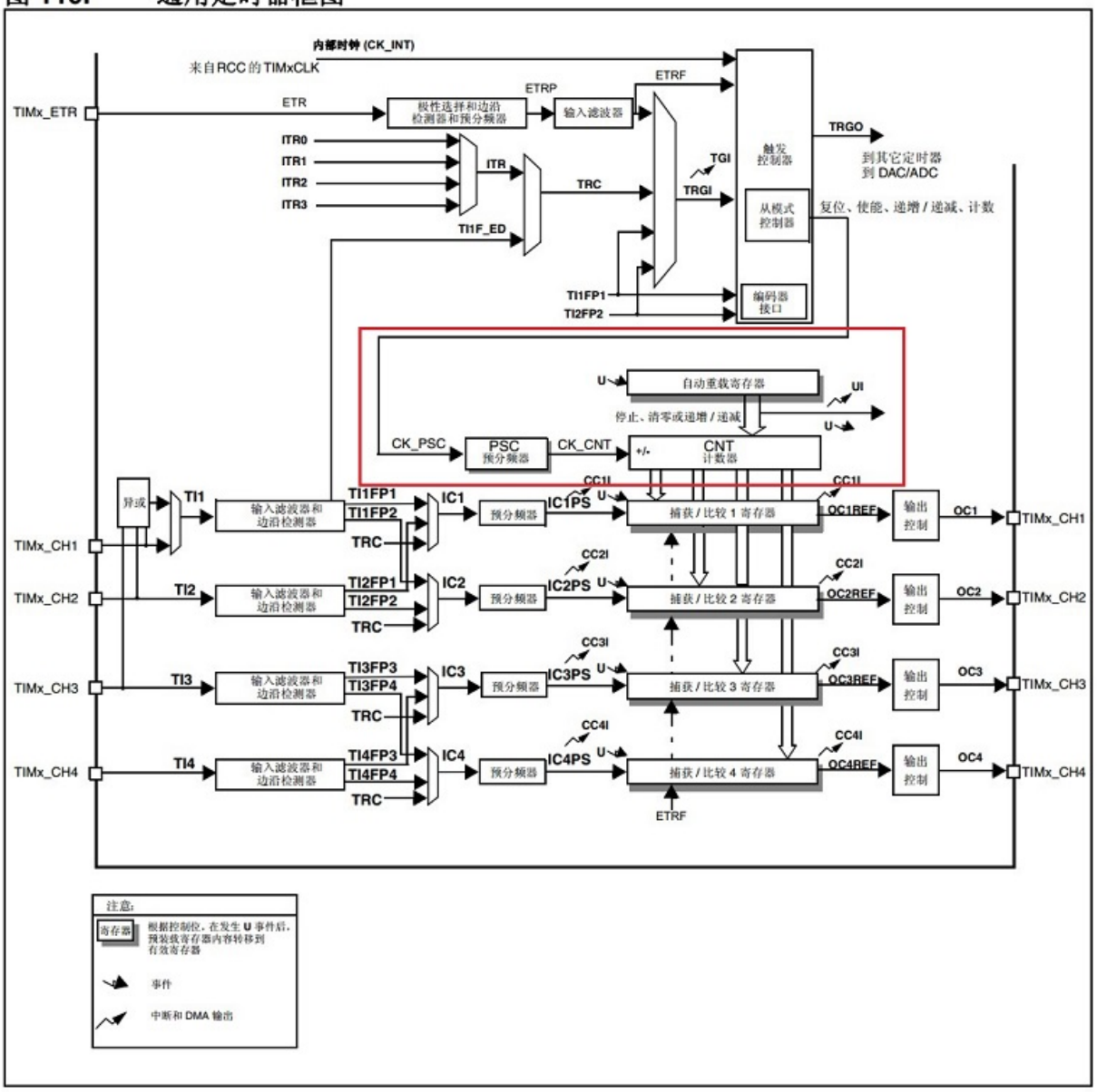
12.2 STM32 定时器

我们先看看 STM32 的定时器都有哪些功能。

12.2.1 框图

使用一个芯片的定时器，先了解他的框图。本次实验我们使用 TIM5，在参考手册第 15 章节有详细说明。下图是他的框图，看起来非常复杂，可见定时器功能是多么丰富。如果我们只是当做一个定时器用，就只要关心红框内的三个框：PSC 预分频器，也就是我们设置的预分频系数，意思就是，例如我们设置为 2，那么每经过 2 个 CK_PSC 时钟，CNT 计数器才会变化 1。自动重载寄存器，设置的值跟 CNT 设置的值一样，如果我们设置为重复定时，如果是减计数，当 CNT 达到 0，芯片会自动从重载寄存器拷贝计数值到计数器。加计数时，用 CNT 值跟重载寄存器比较，判断定时时间是否到。

图 119. 通用定时器框图



定时器框图

12.2.2 时钟

前面说到，单片机定时器跟手机定时器比，单片机设置更复杂，设置要根据系统时钟计算。那么我们就要先了解系统的时钟，通常在参考手册内会有一个时钟树。STM32F407 芯片时钟树在 6.2 章节，

6.2 时钟

可以使用三种不同的时钟源来驱动系统时钟 (SYSCLK)：

- HSI 振荡器时钟
- HSE 振荡器时钟
- 主 PLL (PLL) 时钟

器件具有以下两个次级时钟源：

- 32 kHz 低速内部 RC (LSI RC)，该 RC 用于驱动独立看门狗，也可选择提供给 RC 用于停机/待机模式下的自动唤醒。
- 32.768 kHz 低速外部晶振（LSE 晶振），用于驱动 RC 时钟 (RTCCLK)

对于每个时钟源来说，在未使用时都可单独打开或者关闭，以降低功耗。

芯

STM32F405xx/07xx 和 STM32F415xx/17xx 的定时器时钟频率由硬件情况：

1. 如果 APB 预分频器为 1，定时器时钟频率等于 APB 域的频率。
2. 否则，等于 APB 域的频率的两倍 (x2)。

片时钟树在本章节中,有针对定时器的说明

时器时钟在 system_stm32f4xx.c 文件中，时钟初始化函数 static void SetSysClock(void)，对 PCLK1 进行了 4 分频初始化，那么 168/4=42M，预分频是 2 不是 1，因此定时器的时钟为 APB 时钟倍频，42M*2=84M。

```
#if defined(STM32F40_41xxx) || defined(STM32F427_437xx) || defined(STM32F429_439xx)
||  defined(STM32F412xG) || defined(STM32F446xx) || defined(STM32F469_479xx)
    /* PCLK2 = HCLK / 2*/
    RCC->CFGR |= RCC_CFGR_PPRE2_DIV2;

    /* PCLK1 = HCLK / 4*/
    RCC->CFGR |= RCC_CFGR_PPRE1_DIV4;
#endif /* STM32F40_41xxx || STM32F427_437x || STM32F429_439xx  || STM32F412xG
|| STM32F446xx || STM32F469_479xx */
```

12.2.3 初始化

下面代码为定时器初始化，前面是宏定义，大家写代码一定要多用宏定义，用宏定义的代码，修改起来更方便。

```
#define TestTim TIM5
/*
    定时器时钟为 84M,
    Tout=((SYSTEM_CLK_PERIOD)*(SYSTEM_CLK_PRESCALER))/Ft us.

        预分频,8400 个时钟才触发一次定时器计数
```

(continues on next page)

(continued from previous page)

```
        那么一个定时器计数的时间就是 (1/84M)*8400 = 100us
*/
#define SYSTEM_CLK_PRESCALER    8400
#define SYSTEM_CLK_PERIOD       10000//定时周期

/**
 *@brief:      mcu_timer_init
 *@details:    定时器初始化
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 mcu_timer_init(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
    //打开定时器时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE);
    //复位定时器
    TIM_Cmd(TestTim, DISABLE);
    TIM_SetCounter(TestTim, 0);

    //设定 TIM5 中断优先级
    NVIC_InitStructure.NVIC_IRQChannel = TIM5_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;//抢占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;      //响应优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    TIM_TimeBaseInitStruct.TIM_ClockDivision = TIM_CKD_DIV1;
    TIM_TimeBaseInitStruct.TIM_CounterMode = TIM_CounterMode_Up;//向上计数
    TIM_TimeBaseInitStruct.TIM_Period = SYSTEM_CLK_PERIOD - 1;  //周期
    TIM_TimeBaseInitStruct.TIM_Prescaler = SYSTEM_CLK_PRESCALER-1;//分频
    TIM_TimeBaseInitStruct.TIM_RepetitionCounter = 1;
    TIM_TimeBaseInit(TestTim, &TIM_TimeBaseInitStruct);

    TIM_ITConfig(TestTim, TIM_IT_Update, ENABLE);//打开定时器中断

    TIM_Cmd(TestTim, ENABLE);//使能定时器 (启动)
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

在初始化函数内，主要分 3 部分：

- 时钟

  第 25 行打开设备时钟。

- 定时器配置

  27 行，停止定时器。28 行，复位定时器计数值。37 行，时钟分频，配置为 DIV1，也就是没分频，定时器时钟为 84M。38 行，计数方式，选择向上计数。39 行，计数周期，也就是要定时多少个计数的意思，使用向上计数，计数达到后产生事件。40 行，设置预分频系数，也就是决定了每个计数的时长。41 行，设置是否自动重加载，意思是：一个定时到了，是否自动开始下一个定时（不断重复）。42 行，执行配置。46 行，启动定时器。

- 中断

  31 行，指定配置 TIM5 中断。32 行，设置抢断优先级。33 行，设置响应优先级。34 行，使能。35 行，执行配置。44 行，打开定时器中断。

预分频系数，也就是 Prescaler。程序中定义为 8400，也就是说在 84M 时钟情况下，8400 个时钟才触发一次定时器计数，那么一个定时器计数的时间就是 (1/84M)*8400 = 100us，那么要定时 1S 钟，**周期**则要设置为 1x1000x1000/100=10000。经过 10000 次计数后，恰好就是 1S，定时器会产生一个状态事件，如果**中断使能**，则发生中断。

12.3 查询模式

如果没有使能中断，想要知道定时器是否已经到时间，需要一直查询定时器的状态寄存器。如何查询？请看 ST 提供的库 stm32f4xx_tim.c 其中有函数：

```
/**
  * @brief  Checks whether the specified TIM flag is set or not.
  * @param  TIMx: where x can be 1 to 14 to select the TIM peripheral.
  * @param  TIM_FLAG: specifies the flag to check.
  *          This parameter can be one of the following values:
  *            @arg TIM_FLAG_Update: TIM update Flag
  *            @arg TIM_FLAG_CC1: TIM Capture Compare 1 Flag
  *            @arg TIM_FLAG_CC2: TIM Capture Compare 2 Flag
  *            @arg TIM_FLAG_CC3: TIM Capture Compare 3 Flag
  *            @arg TIM_FLAG_CC4: TIM Capture Compare 4 Flag
  *            @arg TIM_FLAG_COM: TIM Commutation Flag
  *            @arg TIM_FLAG_Trigger: TIM Trigger Flag
```

(continues on next page)

(continued from previous page)

```
  *            @arg TIM_FLAG_Break: TIM Break Flag
  *            @arg TIM_FLAG_CC1OF: TIM Capture Compare 1 over capture Flag
  *            @arg TIM_FLAG_CC2OF: TIM Capture Compare 2 over capture Flag
  *            @arg TIM_FLAG_CC3OF: TIM Capture Compare 3 over capture Flag
  *            @arg TIM_FLAG_CC4OF: TIM Capture Compare 4 over capture Flag
  *
  * @note   TIM6 and TIM7 can have only one update flag.
  * @note   TIM_FLAG_COM and TIM_FLAG_Break are used only with TIM1 and TIM8.
  *
  * @retval The new state of TIM_FLAG (SET or RESET).
  */
FlagStatus TIM_GetFlagStatus(TIM_TypeDef* TIMx, uint16_t TIM_FLAG)
```

用这个函数就可以查询定时器的各种状态了。对应会有一个清标志函数：

```
void TIM_ClearFlag(TIM_TypeDef* TIMx, uint16_t TIM_FLAG)
```

必须手动用这个函数清除标志，否则，就不知道下一次定时器的到来了。

12.4 中断

在前面章节，我们曾初步了解了芯片系统中断。现在我们再通过定时器中断看看中断的细节。什么是中断呢?

> 通常我们的程序是按顺序执行（函数跳转和返回也是按顺序执行），顺序都是我们安排好的 (我们上帝之手)。当中断来临时，停止正在执行的程序，强行执行中断服务程序，中断服务程序运行结束后，在自动返回原来执行程序的位置继续执行。中断和顺序程序不一样的就是，**中断发生时间不确定**。假设按键输入设置为 IO 中断，那么什么时候产生中断？随机的，没人知道什么时候会产生按键。

下面我们通过定时器的程序大概了解一下中断的应用。大家要注意，我们这里仅仅说应用，至于一个中断的切换返回细节，暂不做讨论。

12.4.1 NVIC

在 STM32 这个芯片中，或者说 contex 这种芯片中，与中断相关的有两部分：

1. 外设本身，比如你要使用定时器的中断，必须在定时器中使能对应的中断。
2. NVIC，嵌套向量控制器。

其中外设只要打开对应中断就行了。NVIC 就比较复杂了，NVIC 是芯片控制管理所有中断的模块。在《STM32F4xx 中文参考手册.pdf》第 10 章有详细说明。使用 NVIC，主要内容是设置中断的优先级。

12.4.2 优先级

contex 内核有两个优先级：抢占和响应。抢占优先级就是：如果 A 优先级高，B 优先级低，当 A 发生中断，就算 B 正在处理中断，A 也会立刻响应。响应优先级就是：如果 A 响应优先级比 B 高，C 中断正在执行，就算 B 先来，A 也可以在 C 中断结束后，优先执行。前提是 A 和 B 的抢占优先级一样，并且不比 C 高。

通过上面分析，我们可以知道或者通常，把抢占优先级叫做主优先级，响应优先级叫做次优先级。判断两个中断谁优先级高，先比主优先级，再比次优先级。

12.4.3 优先级分组

哪么优先级怎么设置呢？contex 提供了多达 8bit 用来控制抢占优先级和响应优先级。STM32 自用了其中的低 4 位，这低 4 位如何分配给抢占和响应，需要在初始化时调用函数分配：

```
/**
  * @brief  Configures the priority grouping: pre-emption priority and subpriority.
  * @param  NVIC_PriorityGroup: specifies the priority grouping bits length.
  *   This parameter can be one of the following values:
  *     @arg NVIC_PriorityGroup_0: 0 bits for pre-emption priority
  *                                4 bits for subpriority
  *     @arg NVIC_PriorityGroup_1: 1 bits for pre-emption priority
  *                                3 bits for subpriority
  *     @arg NVIC_PriorityGroup_2: 2 bits for pre-emption priority
  *                                2 bits for subpriority
  *     @arg NVIC_PriorityGroup_3: 3 bits for pre-emption priority
  *                                1 bits for subpriority
  *     @arg NVIC_PriorityGroup_4: 4 bits for pre-emption priority
  *                                0 bits for subpriority
  * @note   When the NVIC_PriorityGroup_0 is selected, IRQ pre-emption is no more␣
↪possible.
  *         The pending IRQ priority will be managed only by the subpriority.
  * @retval None
  */
void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)
```

一共可以设置 5 中分组模式，什么意思呢？例如第一种分组模式 NVIC_PriorityGroup_0，这种分组模式，pre 优先级没有，也就是不能设置（强制设置会出现意外），4bit 都用来表示 sub 优先级，那么 sub 优先级就可以设置 0~15。如果设置为 NVIC_PriorityGroup_2，pre 优先级两位，可以设置 0~3，sub 优先级同样 2 位，也可以设置 0~3。每个系统具体如何设置中断优先级，需要根据所有中断需求合理分配。

12.4.4 中断服务函数

一旦中断产生，就需要执行中断服务程序。定时器的中断程序如下，这个函数就是中断服务函数。在函数我们判断了是不是 Update 中断，为什么要判断呢？因为定时器**中断入口**只有一个，定时器中断有多种，当中断产生时，只能通过标志区分是什么中断源。

```
/**
 *@brief:      mcu_tim5_IRQhandler
 *@details:    定时器中断处理函数
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
void mcu_tim5_IRQhandler(void)
{
    if(TIM_GetITStatus(TIM5, TIM_FLAG_Update) == SET)
    {
        TIM_ClearFlag(TIM5, TIM_FLAG_Update);

        mcu_tim5_test();

    }
}
```

在 stm32f4xx_it.c 中断响应中调用 void mcu_tim5_IRQhandler(void);

```
void TIM5_IRQHandler(void)
{
    mcu_tim5_IRQhandler();
}
```

TIM5_IRQHandler 这个函数名，是在中断向量中定义好的，**一定要一样**。

12.4.5 中断向量

前面章节我们其实已经讨论过中断向量，我们看看定时器 5 的中断向量。

```
__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     MemManage_Handler         ; MPU Fault Handler
```

(continues on next page)

(continued from previous page)

```
                DCD     BusFault_Handler        ; Bus Fault Handler
                DCD     UsageFault_Handler      ; Usage Fault Handler
                DCD     0                       ; Reserved
                DCD     0                       ; Reserved
                DCD     0                       ; Reserved
                DCD     0                       ; Reserved
                DCD     SVC_Handler             ; SVCall Handler
                DCD     DebugMon_Handler        ; Debug Monitor Handler
                DCD     0                       ; Reserved
                DCD     PendSV_Handler          ; PendSV Handler
                DCD     SysTick_Handler         ; SysTick Handler
                ...
                DCD     SDIO_IRQHandler         ; SDIO
                DCD     TIM5_IRQHandler         ; TIM5
```

定时器 5 的中断函数指针在 19 行 (文件 137 行)。

12.4.6 定时器中断流程

到此，我们已经配置好定时器了，流程大概如下：1 使能定时器时钟，并配置定时器。2 设置 NVIC 定时器中断优先级。3 打开定时器中断，启动定时器。4 定时到后，产生时间标志，同时产生中断标志。5 芯片从中断向量查找中断服务程序入口，执行中断服务程序。执行后返回。6 重复 4。

12.5 编码实验

代码见\mcu_dev 目录下的 mcu_timer.c 和 mcu_timer.h。另外，在 main 函数前调用 void mcu_timer_init(void) 函数，初始化定时器。

```
/* Infinite loop */
        mcu_uart_open(3);
        wjq_log(LOG_INFO, "hello word!\r\n");
        dev_key_init();
        mcu_timer_init();

        dev_key_open();
        while (1)
        {
                /* 驱动轮询 */
                dev_key_scan();
```

(continues on next page)

(continued from previous page)

```
        /* 应用 */
        u8 key;
        s32 res;

        res = dev_key_read(&key, 1);
        if(res == 1)
        {
                if(key == DEV_KEY_PRESS)
                {
                        GPIO_ResetBits(GPIOG, GPIO_Pin_0
                            | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                }
                else if(key == DEV_KEY_REL)
                {

                        GPIO_SetBits(GPIOG, GPIO_Pin_0
                            | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                }
        }

        Delay(1);

    }
```

编译后下载进去，串口可见运行结果，从结果看，确实每秒中进入一次定时器中断。

> hello word! tim int 1 tim int 2 tim int 3 tim int 4 tim int 1 tim int 2 tim int 3 tim int 4 tim int 1 tim int 2

12.6 总结

定时器功能就实现了。请大家自行尝试将按键扫描放到定时器中，并且将定时器定时改为 5 毫秒一次中断。

12.7 end

CHAPTER 13

定时器-PWM-蜂鸣器

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

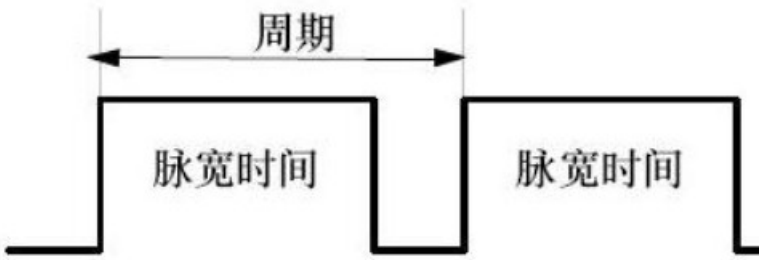
技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

上一章节我们调试了定时器定时功能。现在我们调试定时器输出 PWM 功能。

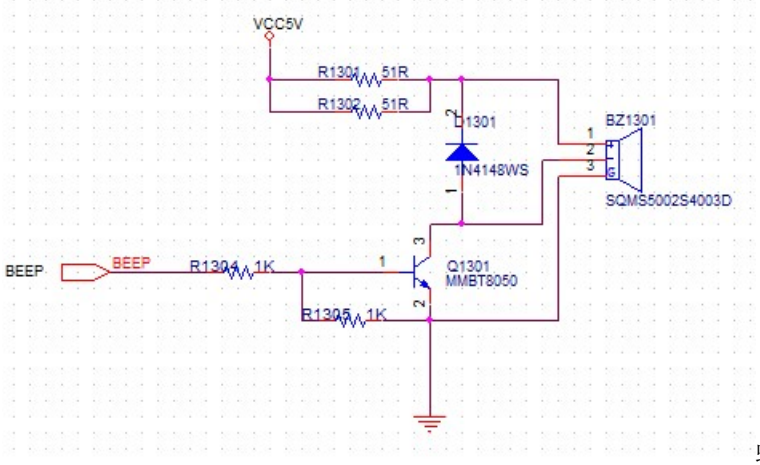
13.1 PWM



PWM(Pulse Width Modulation)是脉冲宽度调制的缩写。
波形简单的说就是高低电平不断切换。在流水灯章节我们曾经提过。一个高低电平切换就是一个**周期**，在一个周期内，高电平持续时间占周期的百分比就是常说的**占空比**。PWM 常用于控制灯光和电机。通常占空比越大，电机转速越快，LED 越亮（高电平驱动方式）。

13.2 原理图

本次我们使用一个固定频率的 50% 占空比 PWM 驱动一个电磁式蜂鸣器。



蜂鸣器电路蜂鸣器选用贴片电磁式，参数如下，从表中可以看到，只要我们输出一个 4000Hz 的频率，就能驱动蜂鸣器。

1	Rated Voltage	3.6Vo-p
2	Operating Voltage	2.0～5.0Vo-p
3	Rated Current	Max.100mA
4	Sound Output at 10cm	Min. 80dB
5	Coil Resistance	12±3Ω
6	Resonant Frequency	4000Hz
7	Operating Temperature	-30℃～+70℃
8	Store Temperature	-40℃～+85℃
9	Net Weight	Approx 0.3g
10	RoHS	Yes

蜂鸣器参数

13.3 STM32 定时器 PWM

蜂鸣器 beep 接在 PD13，在数据手册《STM32F407_ 数据手册.pdf》中可查到，**PD13 是 TIM4 的 CH2，因此要在这个 IO 上输出 PWM，需要用定时器 4，并且是在通道 2 上输出。**

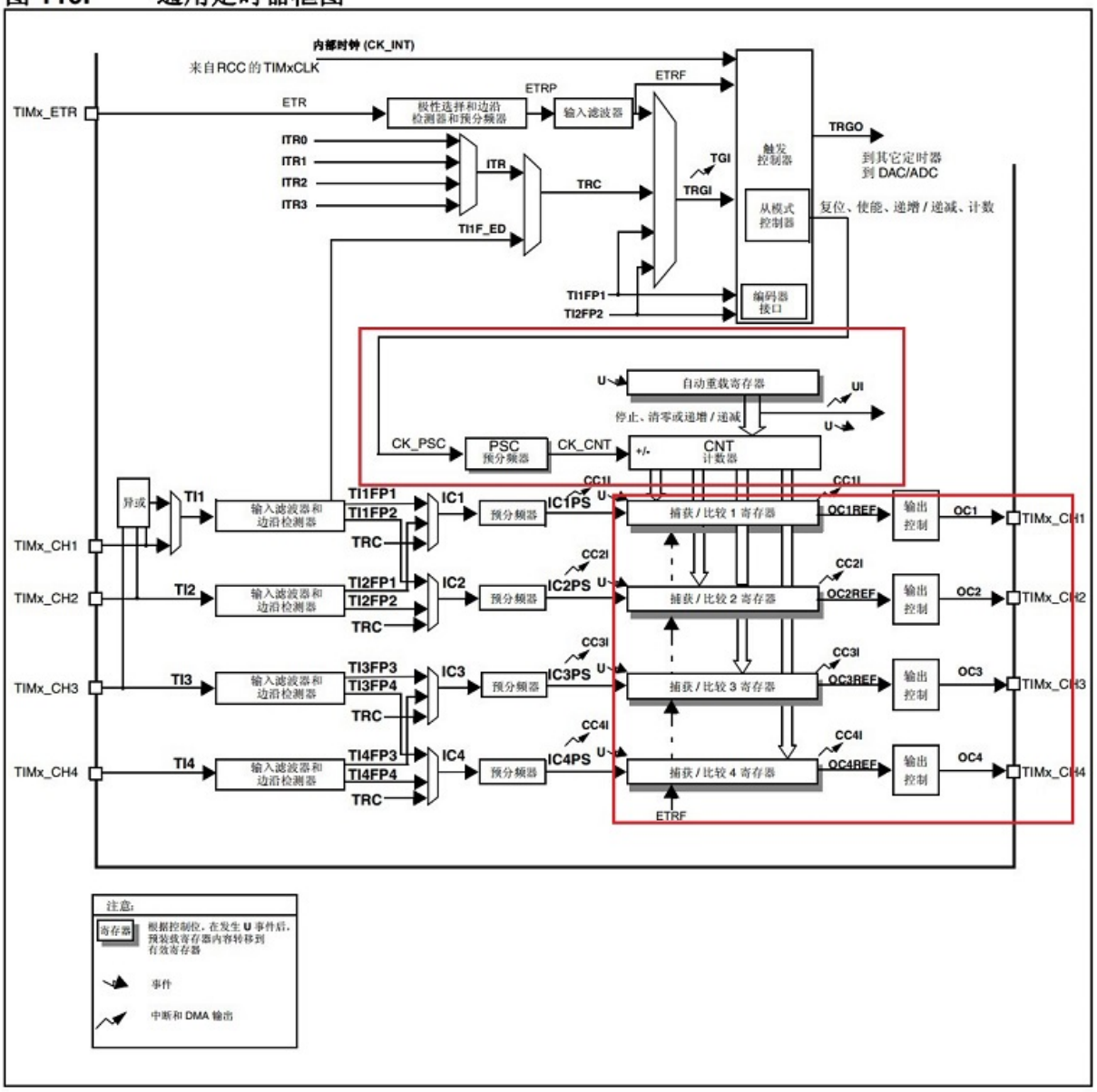
Pinouts and pin description **STM32F405xx, STM32F407xx**

Table 6. STM32F40x pin and ball definitions (continued)

Pin number					Pin name (function after reset)(1)	Pin type	I / O structure	Notes	Alternate functions	Additional functions
LQFP64	LQFP100	LQFP144	UFBGA176	LQFP176						
-	60	82	M15	101	PD13	I/O	FT		FSMC_A18/TIM4_CH2/ EVENTOUT	

定时器 PWM 上一节我们看过定时器的框图，做 PWM 功能，需要用到的功能比定时多了输出控制部分。也就是下图右下角的大红框中的内容。

图 119. 通用定时器框图



定时器框图如何使用定时器 4 在 PD13 上输出 4KHz 的 PWM？我们通过代码讲解。

13.4 编码

在 board_dev 文件夹创建两个新文件 dev_buzzer.c、dev_buzzer.h，将这两个文件夹添加到工程。

13.4.1 IO 口初始化

把一个 IO 口作为外设功能，只需要将对应 IO 口设置为 **AF 模式，并且使用配置函数配置为对应的外设功能**，具体见初始化函数注释。

```
s32 dev_buzzer_init(void)
{
        GPIO_InitTypeDef GPIO_InitStructure;

        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE); //---使能 GPIOD 时钟
    GPIO_PinAFConfig(GPIOD,GPIO_PinSource13,GPIO_AF_TIM4); //---管脚复用为 TIM4 功能

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //---复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //---速度 50MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //---推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //---上拉
    GPIO_Init(GPIOD,&GPIO_InitStructure);

    mcu_tim4_pwm_init(BUZZER_CLK_PERIOD,BUZZER_CLK_PRESCALER);

        return 0;
}
```

13.4.2 定时器初始化

在初始化函数内，调用 mcu_timer.c 文件的 mcu_tim4_pwm_init 函数初始化定时器 4。

```
void mcu_tim4_pwm_init(u32 arr,u32 psc)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_OCInitTypeDef TIM_OCInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4,ENABLE);//---TIM4 时钟使能

    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
    TIM_TimeBaseStructure.TIM_Prescaler = psc - 1; //---定时器分频
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //---向上计数模式
    TIM_TimeBaseStructure.TIM_Period= arr - 1; //---自动重装载值
    TIM_TimeBaseInit(TIM4,&TIM_TimeBaseStructure);//---初始化定时器 4

    //----初始化 TIM4 PWM 模式
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; //---PWM 调制模式 1
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //---比较输出使能
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low; //---输出极性低
```

(continues on next page)

(continued from previous page)

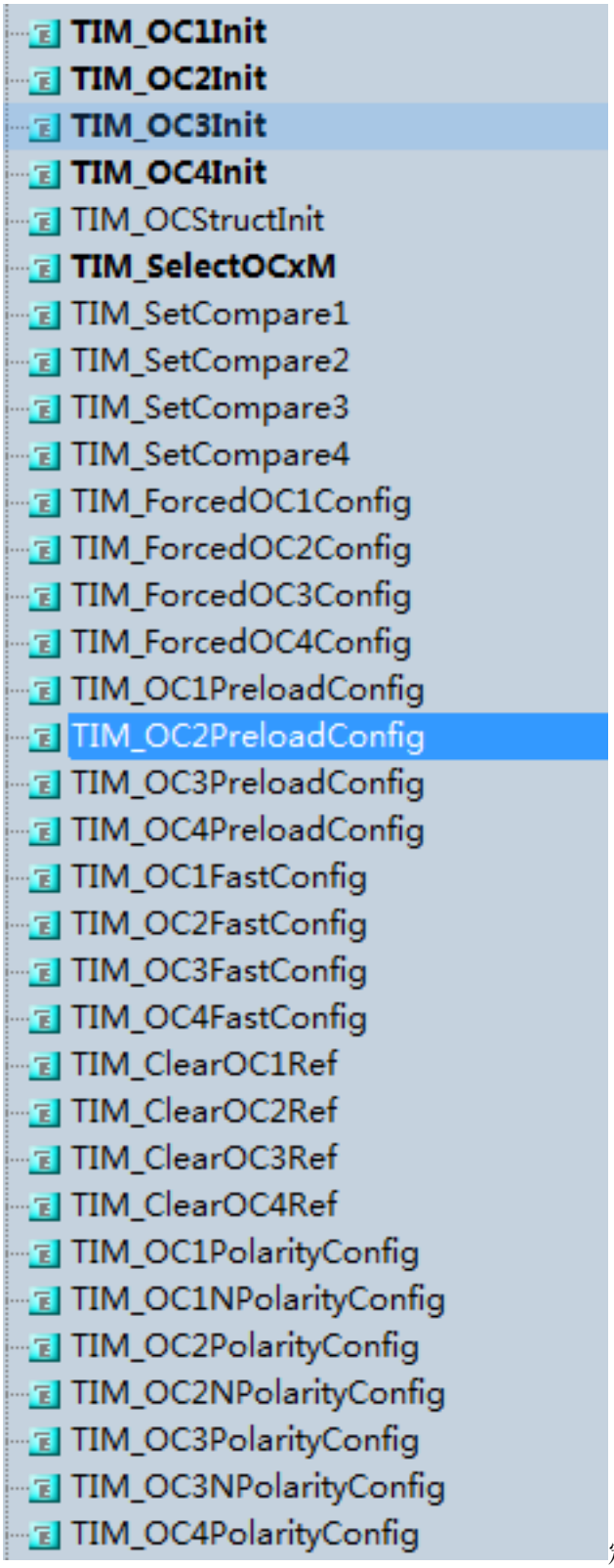
```
    /* 默认配置的是通道 2*/
    TIM_OC2Init(TIM4, &TIM_OCInitStructure); //---初始化外设 TIM4
    TIM_SetCompare2(TIM4, arr/2);//---占空比 50%
    TIM_OC2PreloadConfig(TIM4, TIM_OCPreload_Enable); //---使能预装载寄存器
    TIM_ARRPreloadConfig(TIM4,ENABLE);

}
```

相对前面配置定时功能，多了 PWM 配置。

15 行，配置定时器输出模式，定时器输出有以下 6 种模式选择。

```
/** @defgroup TIM_Output_Compare_and_PWM_modes
  * @{
  */

#define TIM_OCMode_Timing                  ((uint16_t)0x0000)
#define TIM_OCMode_Active                  ((uint16_t)0x0010)
#define TIM_OCMode_Inactive                ((uint16_t)0x0020)
#define TIM_OCMode_Toggle                  ((uint16_t)0x0030)
#define TIM_OCMode_PWM1                    ((uint16_t)0x0060)
#define TIM_OCMode_PWM2                    ((uint16_t)0x0070)
```

16 行，比较输出使能，也就是打开 PWM 输出功能。17 行，输出极性低，这个所谓的输出极性，也就是控制比较寄存器比设置的计数小的时候，输出 0 还是输出 1。或者简单的理解就是，先输出低电平还是先输出高电平。20-23 行，执行配置，这个地方要注意，不同的输出通道设置，需要使用不同的函数：我们用的是通道 2，那么用的就是 TIM_OC2Init、TIM_SetCompare2、TIM_OC2PreloadConfig。在库文件中可以看到下面这些函数，定时器有 4 个通道，就有 4 套配



置函数。定时器输出设置函数

IO 口和定时器都配置好后，只要启动定时器，就可以输出 PWM 了。

13.4.3 打开和关闭的逻辑

经过初始化后，输出 PWM，蜂鸣器就会响。应用上不可能让蜂鸣器一直响，所以要提供 OPEN 和 CLOSE 接口。如下：

```
/**
 *@brief:      dev_buzzer_open
 *@details:    打开蜂鸣器
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 dev_buzzer_open(void)
{
        GPIO_InitTypeDef GPIO_InitStructure;

        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE); //---使能 GPIOD 时钟
    GPIO_PinAFConfig(GPIOD,GPIO_PinSource13,GPIO_AF_TIM4); //---管脚复用为 TIM4 功能
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //---复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //---速度 50MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //---推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //---上拉
    GPIO_Init(GPIOD,&GPIO_InitStructure);

    TIM_Cmd(TIM4, ENABLE); //---使能 TIM4

        return 0;
}
/**
 *@brief:      dev_buzzer_close
 *@details:    关闭蜂鸣器
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 dev_buzzer_close(void)
{
        GPIO_InitTypeDef GPIO_InitStructure;

    TIM_Cmd(TIM4, DISABLE); //---关闭定时器 TIM4
```

(continues on next page)

(continued from previous page)

```
        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE); //---使能 GPIOD 时钟

        /* 关闭蜂鸣器时，要将 IO 改为普通 IO，并且输出低电平，否则蜂鸣器会造成大电流 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; //---复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //---速度 50MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //---推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //---上拉
    GPIO_Init(GPIOD,&GPIO_InitStructure);
        GPIO_ResetBits(GPIOD, GPIO_Pin_13);

        return 0;
}
```

是不是跟你想的不一样？只要停止或者启动定时器不就行了吗？功能上，确实是只要停止定时器，PWM 就没有输出，蜂鸣器就不响；打开定时器，输出 PWM，定时器就会响。但是实际上会有问题，请看注释。

> 因为我们关闭 PWM 输出，是直接停止定时器。那么在停止定时器的时候，**PWM 输出的电平是随机的**，如果正好在输出高电平时停止定时器，蜂鸣器电流上的三极管就一直处于导通状态，此时电流会增大 100ma。所以关闭蜂鸣器后要将 IO 转为输出模式，并输出低电平。打开蜂鸣器时再将 IO 设置为 PWM 模式。

13.5 调试

在 main 函数中增加如下代码，初始化后，按下按键蜂鸣器响，松开按键蜂鸣器关。

```
/* Infinite loop */
        mcu_uart_open(3);
        uart_printf("hello word!\r\n");
        mcu_timer_init();
        dev_key_init();
        dev_buzzer_init();
        while (1)
        {
                s32 key;
                key = dev_key_scan();
                if(key == 0)
                {
                        GPIO_ResetBits(GPIOG, GPIO_Pin_0
```

(continues on next page)

(continued from previous page)

```
                        | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                dev_buzzer_open();
            }
            else if(key == 1)
            {

                GPIO_SetBits(GPIOG, GPIO_Pin_0
                        | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                dev_buzzer_close();

            }
            Delay(1);

    }
```

- 调试过程 1 代码编写完成，编译，无错误，下载进去后，蜂鸣器不响。手头暂时没有示波器。咋办？目前并**没有确定是软件问题还是硬件问题**，因此，**首先要确定硬件没有问题**。前面我们已经调通了 IO 口输出与定时器中断，可以**利用定时器中断操作 IO 口翻转电平输出一个 4K 频率的 PWM**。经验证，蜂鸣器响，硬件无问题。2 经查询，发现在设置 PWM 通道时错误，参考的代码用的是通道 1，而我们用的是通道 2。**初始化 PWM 时，每一个带 OC1 字符的函数或者定义都改为 OC2 即可**。也即是上面 mcu_tim4_pwm_init 代码中的 20、21、22 行。3 目前蜂鸣器是能发出声音了，但是因为暂时没有示波器，无法保证频率是否准确，占空比是否是 50%，**实际项目中必须用示波器验证**。

13.6 总结

PWM 应用非常广泛，除了控制蜂鸣器，还可以控制 LED 灯，或者控制 LCD 背光。只要学会一种，其他都一样。

13.7 end

CHAPTER 14

定时器-捕获-触摸按键

前面章节介绍了定时器的定时与输出 PWM 功能，定时器还有很多功能。《STM32F4xx 中文参考手册.pdf》中定时器就有说明。本次我们就使用定时器的**输入捕获功能**，实现一个简单的**电容触摸按键**。

14.1 输入捕获

假如我们要检测一个 PWM 波形的高低电平时间，会怎么做呢？如果没有输入捕获，我们可以使用一个普通定时器和一个 IO 输入中断实现。流程大概如下：

- 将一个 IO 口配置为中断，上升沿触发。
- 配置一个定时器，向下计数模式，例如倒计时 1000 毫秒
- IO 产生中断。
- 在 IO 中断中，停止定时器，并且获取剩余计数值，例如是 600, 说明时间已经过去 400 毫秒。同时配置 IO 为下降沿触发。再启动定时器。
- IO 产生下降沿中断。

- 在 IO 中断中，停止定时器，并且获取剩余计数值，例如是 300，说明时间已经过去 300 毫秒。那么 600-300 = 300，就是高电平时间。也就完成了一次捕获中断，低电平同理。

定时器的输入捕获与此类似，只不过定时器硬件完成了判断 IO 口上升下降沿并保存时间值的功能。

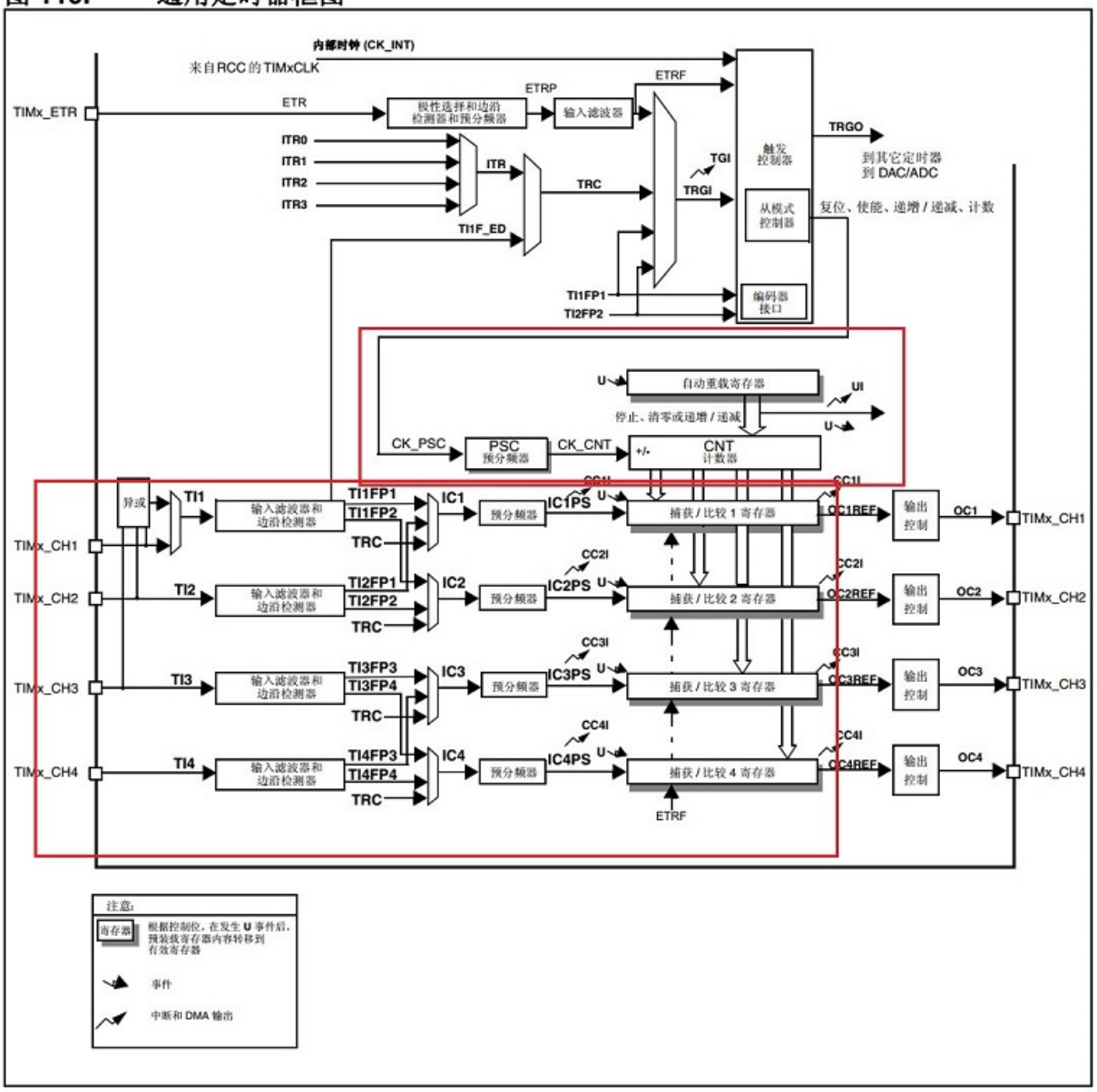
- 首先打开定时器，向上计数模式。
- IO 口（定时器通道）产生一个上升沿。
- 定时器在上升沿时将定时器的计数 CNT，保存到 CCRx1。
- IO 口（定时器通道）产生一个下降沿。
- 定时器在下降沿时将定时器的计数 CNT，保存到 CCRx2。上升沿到下降沿时长可能超出定时器计数时限，需要做溢出处理。
- 那么 CCRx1 跟 CCRx2，就是定时器捕获到的值，通过这两个值。可算出高电平时间，低电平同理。

不同的芯片输入捕获功能基本相同，细节上可能有所差别。

14.2 STM32 捕获中断

输入捕获，是定时器的功能，前面我们已经用了 STM32 定时器**定时**、**输出**两个功能。输入捕获是定时器的输入功能，除了输入捕获，还有输入计数等其他输入功能。输入捕获用到下图红框内的功能。

图 119. 通用定时器框图

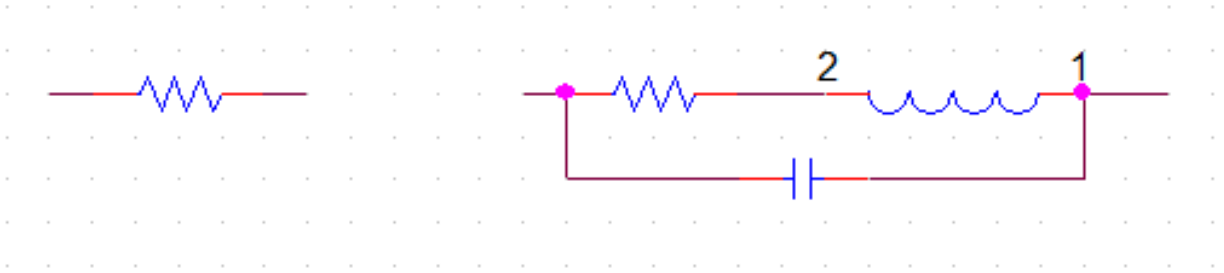


定时器框图

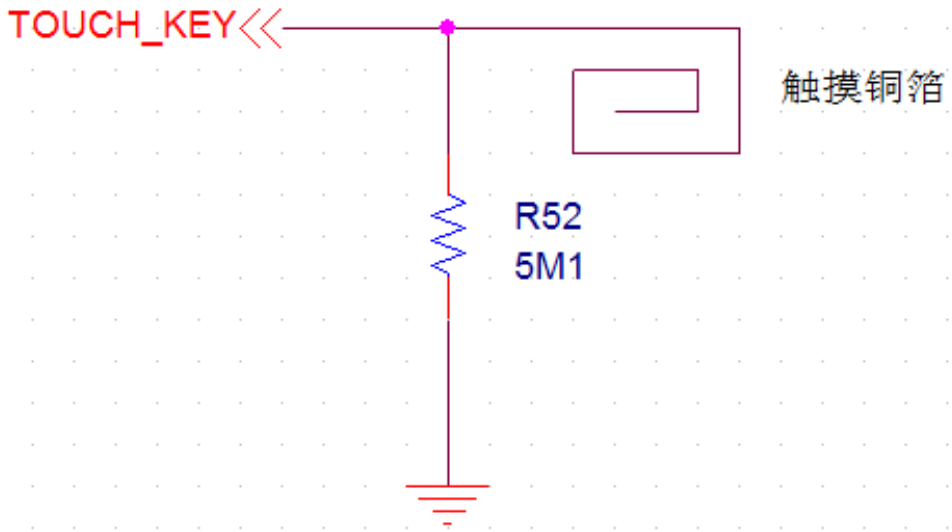
这看《STM32F4xx 中文参考手册.pdf》，不再累赘。然后通过例程代码进一步熟悉。

14.3 电容触摸按键原理

所谓电容式，就是利用电容的充放电检测是否触摸。学过基本电子学的同学应该都知道，一个真正的器件，会有一个等效电路，而不仅仅是我们说的这个器件功能本身（理想电路）。例如一个电阻，等效电路与理想电路如下:

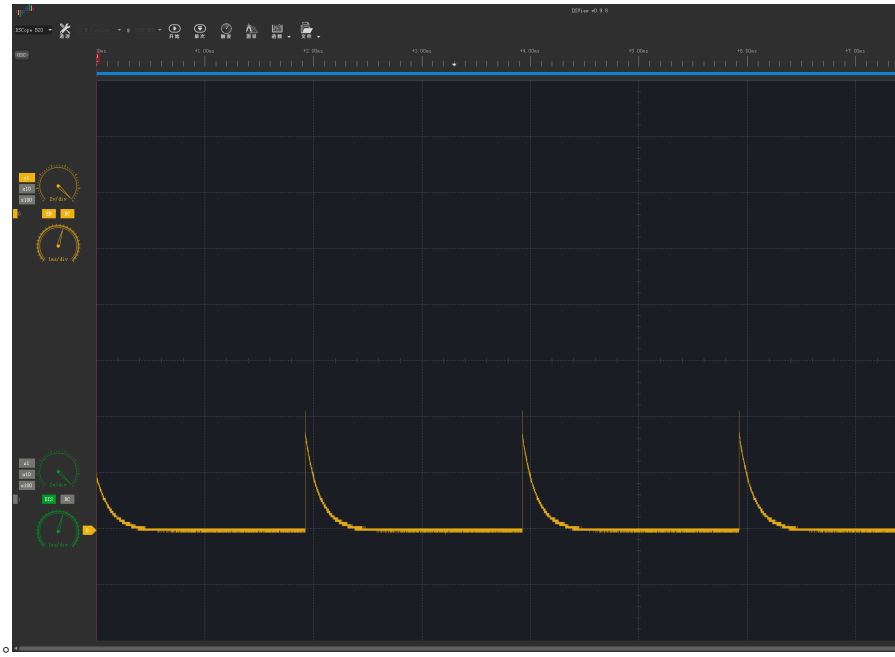


电阻理想电路左边是理想电阻，右边是等效电阻，除了电阻，还包含了寄生电感跟寄生电容。寄生性能在低速电路通常不用考虑，在高速电路就是一个重要指标了。人，就是一个大电阻，也有寄生电容。当人一接触 PCB 板上的触摸铜箔，就改变了整个触摸电路的寄生电容。触摸电路的放电时间就会改变。



- 电容触摸按键电路
  容触摸按键电路
- 充放电波形下图是触摸按键充放电波形，有几个特点：

1. 电容很小，在 IO 口高电平 3.3V 充电，很快就会充满，所以低电平到高电平非常陡峭。
2. 放电电阻较大，我们用的是 5.1M，所以放电时间比充电时间长很多很多。



3. 当手指触摸铜箔，电容增大，放电时间变长。放电波形

- 检测流程
- 先用 IO 口输出高电平，电阻跟触摸铜箔的寄生电容就会充电。
- 然后将 IO 口改为输入捕获，这时刚刚充满电的电容就会通过电阻放电，放电结束后（电压达到 IO 低电平识别范围）就触发输入捕获中断
- 我们将得到的数据进行分析处理，即可区分是否有触摸

14.4 编码

下面进入编码设计。

14.4.1 驱动设计

触摸按键驱动分两部分：

1. 输入捕获相关的，放到定时器驱动，为 touchkey 算法提供时间流。
2. 触摸算法处理部分，单独做一个 dev_touchkey 驱动，提供触摸按键 api 给 APP 使用。

对于这两个驱动的分割，有如下考虑：

1. **输入捕获就是输入捕获，捕获到时间流后，就上传给上一个模块**。至于这个时间流的具体功能，是触摸按键呢？还是其他，例如磁条卡磁道时间流。定时器会知道吗？不知道，也不应该知道，知道也不应该管。

2. **触摸模块根据时间流数据处理后得到触摸事件**。我们现在用的是定时器捕获，如果改为普通定时器加 IO 中断。触摸驱动要识别这两者吗？**不需要，而且要兼容**。无论定时捕获还是定时 +IO 中断，对于触摸驱动来说，就是个黑盒子，只要给我时间流数据就行了。
3. 有一种触摸按键，是直接用芯片处理的，从芯片处就已经得到了触摸按键事件。假设现在方案是定时捕获，我们完成了驱动，量产了。然后要修改为 IC 方案。怎么样的驱动设计，改动最小？

14.4.2 定时器配置

在 mcu_timer 驱动中增加初始化定时器输入捕获内容。硬件使用 PA3 作为捕获输入，查看数据手册，PA3 是 TIM2 的 CH4，还是 TIM5 的 CH4，是 TIM9 的 CH2。

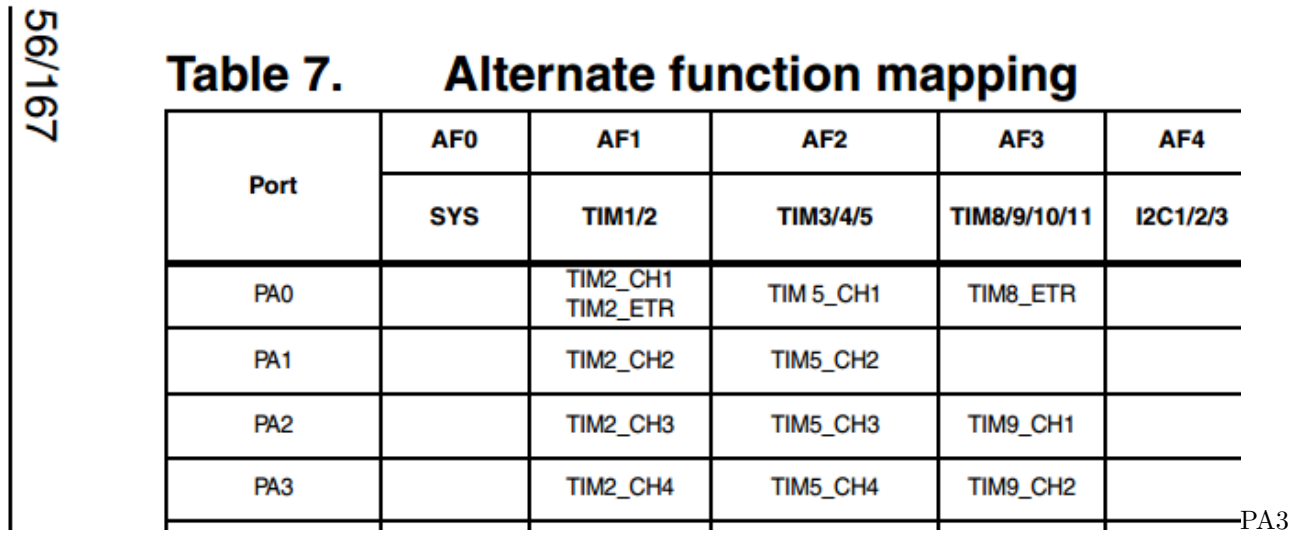


Table 7. Alternate function mapping

Port	AF0	AF1	AF2	AF3	AF4
	SYS	TIM1/2	TIM3/4/5	TIM8/9/10/11	I2C1/2/3
PA0		TIM2_CH1 TIM2_ETR	TIM 5_CH1	TIM8_ETR	
PA1		TIM2_CH2	TIM5_CH2		
PA2		TIM2_CH3	TIM5_CH3	TIM9_CH1	
PA3		TIM2_CH4	TIM5_CH4	TIM9_CH2	

PA3 对应定时器通道我们使用 TIM2 的 CH4 作为捕获定时器输入。不使用捕获中断，捕获值通过查询获取。主要有两个函数，一个是初始化定时器捕获，一个是查询获取定时器捕获的值。代码都是对库的调用，具体配置请看源码。

```
/**
 *@brief:      mcu_timer_cap_init
 *@details:    初始化定时器捕获，不使用中断
 *@param[in]   u32 arr
               u16 psc
 *@param[out]  无
 *@retval:
 */
void mcu_timer_cap_init(u32 arr,u16 psc)
{

        TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
        TIM_ICInitTypeDef TIM2_ICInitStructure;
```

(continues on next page)

(continued from previous page)

```
        //初始化 TIM2
        RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE); // 时钟使能
        TIM_TimeBaseStructure.TIM_Period = arr; //设定计数器自动重装值
        TIM_TimeBaseStructure.TIM_Prescaler =psc; //预分频器
        TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
        TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //TIM 向上计数
        TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure); // 初始化定时器 2

        //初始化通道 4
        TIM2_ICInitStructure.TIM_Channel = TIM_Channel_4; //选择输入端 IC4 映射到 TIM2
        TIM2_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Falling; //下降沿捕获
        TIM2_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
        TIM2_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1; //配置输入分频，不分频
        TIM2_ICInitStructure.TIM_ICFilter = 0x00;//配置输入滤波器 不滤波
        TIM_ICInit(TIM2, &TIM2_ICInitStructure);//初始化 TIM2 IC4

        TIM_ClearITPendingBit(TIM2, TIM_IT_CC4|TIM_IT_Update); //清除中断标志
        TIM_SetCounter(TIM2,0);

        TIM_Cmd(TIM2,ENABLE); //使能定时器 2
}
/**
 *@brief:      mcu_timer_get_cap
 *@details:    查询获取定时去捕获值
 *@param[in]   void
 *@param[out]  无
 *@retval:     捕获值，超时则返回最大值
 */
u32 mcu_timer_get_cap(void)
{

        while(TIM_GetFlagStatus(TIM2, TIM_IT_CC4) == RESET)//等待捕获上升沿
        {
                if(TIM_GetCounter(TIM2) > 0xffffffff-1000)
                        return TIM_GetCounter(TIM2);//超时了，直接返回 CNT 的值
        }
        return TIM_GetCapture4(TIM2);
}
```

16~21 行，是对定时器片配置。24~29 行，是对定时器输入的配置，还记得 PWM 实验吗？调用

的是 OC 接口，现在调用的是 IC 接口，也就是 input config 的意思吧。

14.4.3 触摸按键处理

创建 dev_touchkey 设备驱动。因为目前没有系统，驱动设计为：定时轮询 + 缓冲区模式。有系统的时候也可以使用这种模式，也可以考虑修改为线程 + 邮箱的模式。

- 驱动伪代码流程

1. dev_touchkey_init 初始化
2. 在 main 函数的 while(1) 中轮询 dev_touchkey_task 函数
3. dev_touchkey_task 中首先调用 dev_touchkey_resetpad 函数对按键充电。
4. 充电完成后执行 dev_touchkey_iocap，将 IO 转为定时器输入捕获通道。
5. 调用 mcu_timer_cap_init 函数配置定时器捕获。
6. 调用 mcu_timer_get_cap 获取捕获的值。
7. 通过函数 dev_touchkey_scan 处理捕获值，如果确定状态变化，将事件写入 TouchKeyBuf。

下面代码是触摸电容的检测流程，得到时间流后，再用 scan 函数处理时间流。

```
/**
 *@brief:      dev_touchkey_task
 *@details:    触摸按键线程，常驻任务
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 dev_touchkey_task(void)
{
        volatile u32 i = 0;
        u32 cap;

        if(TouchKeyGd != 0)
                return -1;
        //IO 输出 1，对电容充电
        dev_touchkey_resetpad();
        //延时一点，充电
        for(i=0;i++;i<0x12345);
        //将 IO 口设置为定时去输入捕获通道
        dev_touchkey_iocap();
        //开定时器捕获，如果预分频 8，一个定时器计数是 100ns 左右 ，这个值要通过调试，
```

(continues on next page)

(continued from previous page)

```
        mcu_timer_cap_init(0xffffffff, 8);
        cap = mcu_timer_get_cap();
        TOUCHKEY_DEBUG(LOG_DEBUG, "\r\n%08x---", cap);

        dev_touchkey_scan(cap);

        return 0;
}
```

scan 处理流程，**设计思想可以参考按键扫描**（很多数据处理流程都可以参考按键处理）。具体流程看代码吧。

- 应用流程

  调用 dev_touchkey_read 读取触摸按键事件。

```
/**
 *@brief:       dev_touchkey_test
 *@details:     触摸按键测试程序
 *@param[in]    无
 *@param[out]   无
 *@retval:
 */
s32 dev_touchkey_test(void)
{
        u8 tmp;
        s32 res;

        //dev_touchkey_open();

        res = dev_touchkey_read(&tmp, 1);
        if(1 == res)
        {
                if(tmp == DEV_TOUCHKEY_TOUCH)
                {
                        wjq_log(LOG_FUN, "touch key test get a touch event!\r\n");
                }
                else if(tmp == DEV_TOUCHKEY_RELEASE)
                {
                        wjq_log(LOG_FUN, "touch key test get a release event!\r\n");
                }
        }
```

(continues on next page)

(continued from previous page)

```
        return 0;

}
```

- 缺陷

1. 使用查询方式获取捕获值，查询就相当于死等，浪费 CPU 时间，需要改为中断模式。
2. 充电使用硬延时，同样浪费 CPU 时间，可以改为定时器。请大家尝试优化，屋脊雀会再最后提供的整体软件上优化。

14.4.4 调试

- 第一步 **先调试获取时间流功能，获取到的时间流要能够反映触摸变化**。

测试程序如下，在 main 函数 while 循环中调用即可。

```
/**
 *@brief:       dev_touchkey_task
 *@details:     触摸按键线程，常驻任务
 *@param[in]    void
 *@param[out]   无
 *@retval:
 */
s32 dev_touchkey_task(void)
{
        u32 i = 0;
        u32 cap;

        uart_printf("touchkey touch\r\n");
        //IO 输出 1，对电容充电
        dev_touchkey_resetpad();
        //延时一点，充电
        for(i=0;i++;i<0x12345);
        //开定时器捕获，如果预分频 84，一个定时器计数是 1us 左右 ，这个值要通过调试，
        uart_printf("touchkey touch start cap.\r\n");
        mcu_timer_cap_init(0xffffffff, 84);
        cap = mcu_timer_get_cap();
        uart_printf("cap value:%08x\r\n", cap);
}
```

后面驱动进行系统系统整合，这个 task 函数要放到一个定时器，或者是在 while 主循环中定时轮询。如果有操作系统，就单独创建一个线程，或者放到守护线程中运行。而且要整改这个函数，

因为现在里面有死等，系统整合后不使用死等，避免占系统时间片资源，可以通过将这个 task 整改为**分步骤执行**来实现。首先充电，然后退出 task。第二次进入 task，就是启动 cap，然后退出。第三次进来就是读捕获数据，然后循环重复第一步。

串口调试信息如下，从中看出手不触摸的时候，捕获值为 0X3B，手放上去时，0X54。两者之间差别不大，而且数值都较小。需要加快定时器计数，将两者的捕获值拉大，**增强识别度**。

—hello world!—— touchkey touch touchkey touch start cap. cap value:0000003b

—hello world!—— touchkey touch touchkey touch start cap. cap value:00000054

—hello world!—— touchkey touch touchkey touch start cap. cap value:00000054

—hello world!—— touchkey touch touchkey touch start cap. cap value:0000003b

将 dev_touchkey_task 中 mcu_timer_cap_init 函数的预分频改为 8，将定时器速度直接加快 10 倍

```
mcu_timer_cap_init(0xffffffff, 8);
```

修改后调试信息如下，可以看出，捕获值已经放大，识别度由原来的 20 左右变为 160，考虑干扰的情况，160 上下浮 50，识别度已经算可以了。如果还需要优化，可以通过修改硬件放电电阻阻值，或者增加旁路电容。

—hello world!—— touchkey touch touchkey touch start cap. cap value:00000232

—hello world!—— touchkey touch touchkey touch start cap. cap value:000002ac

—hello world!—— touchkey touch touchkey touch start cap. cap value:00000300

—hello world!—— touchkey touch touchkey touch start cap. cap value:00000234

—hello world!—— touchkey touch touchkey touch start cap. cap value:00000231

注：以上测试数据在第一版硬件上测试

- 第二步 **对数据进行处理并识别**

数据处理识别的方案：

时间流 cap，连续 N 次，与上一轮稳定状态的平均值比较，偏差超过门限，则认为是一次变化，同时保存平均值做为新的稳定状态。根据偏差方向（变大还是变小），判断是触摸还是松开，（上电初始化后第一次变化丢弃）。

源码如下，将 dev_touchkey_scan 函数添加到 dev_touchkey_task 获取到 cap 之后。同时修改 main 函数，将原来的延时改为 10 毫秒，加快触摸按键扫描速度。具体见代码。

```
#define DEV_TOUCHKEY_GATE 50//确认状态变化的门限值，根据硬件性能调节本参数到合适灵敏度即可。
#define DEV_TOUCHKEY_DATA_NUM 4//一轮稳定状态时间流个数，可以通过修改这个调节触摸扫描时间
static u16 TouchKeyLastCap = 0;//最后一次稳定的 CAP 平均值
/**
 *@brief:      dev_touchkey_scan
 *@details:    扫描触摸捕获的数据流
```

(continues on next page)

(continued from previous page)

```
 *@param[in]   u32
 *@param[out]  无
 *@retval:
 */
s32 dev_touchkey_scan(u32 cap)
{
        static u16 average = 0;//平均值
        static u8 cap_cnt = 0;//有效捕获计数
        static u8 last_dire = DEV_TOUCHKEY_IDLE;//上一个值的方向, 1 位变大, 触摸, 2 为变小,
松开

        //uart_printf("--%08x-%04x-", cap, TouchKeyLastCap);
        if(cap > TouchKeyLastCap + DEV_TOUCHKEY_GATE)
        {
                if(last_dire != DEV_TOUCHKEY_TOUCH)
                {
                        cap_cnt = 0;
                        average = 0;
                        last_dire = DEV_TOUCHKEY_TOUCH;
                }

                cap_cnt++;
                average = average + cap;
                //uart_printf("b-");
        }
        else if(cap < TouchKeyLastCap - DEV_TOUCHKEY_GATE)
        {
                if(last_dire != DEV_TOUCHKEY_RELEASE)
                {
                        cap_cnt = 0;
                        average = 0;
                        last_dire = DEV_TOUCHKEY_RELEASE;
                }
                cap_cnt++;
                average = average + cap;
                //uart_printf("s-");
        }
        else
        {
                //uart_printf("i-");
```

(continues on next page)

(continued from previous page)

```
                cap_cnt = 0;
                average = 0;
                last_dire = DEV_TOUCHKEY_IDLE;
        }

        //uart_printf("\r\n");
        if(cap_cnt >= DEV_TOUCHKEY_DATA_NUM)
        {

                if(DEV_TOUCHKEY_RELEASE == last_dire)
                {
                        uart_printf("\r\n----------------------rel\r\n");
                }
                else if(DEV_TOUCHKEY_TOUCH == last_dire)
                {
                        uart_printf("\r\n----------------------touch\r\n");
                }

                if(TouchKeyLastCap == 0)
                {
                        uart_printf("\r\n--------------------init\r\n");
                }

                TouchKeyLastCap = average/DEV_TOUCHKEY_DATA_NUM;
                cap_cnt = 0;
                average = 0;
        }

}
```

我们从源码分析，

1 TouchKeyLastCap 初始化为 0。2 第 18 行到第 49 行代码，判断新的 cap 值跟 TouchKeyLastCap 的偏差，如果大于门限，认为是触摸，并且进行记录；小于，认为是松开，同时进行记录；否则认为是没变化，清记录。3 记录大于 DEV_TOUCHKEY_DATA_NUM 次后，确认是变化，根据变化方向，判断触摸还是松开，同时更新平均值。

本处理流程，在初始化时没有预先获取一个稳定状态做为非触摸状态。因此在上电的时候无论是触摸还是非触摸，在后续都可以正常识别。触摸着按键上电，init 之后松开，正确识别为 rel，后续识别正常。

—hello world!—– ————————touch ————————-init ————————rel ———————— –touch ————————rel ————————touch ————————rel

上电时没有触摸，init 之后触摸，正常识别为 touch，后续识别正常。

—hello world!—— ——————touch ————————-init ——————touch ——————rel ——————touch ——————rel

- 第三步 **识别结果保存及接口处理**

经过上一步，已经能正常获取触摸事件。但是对于一个驱动来说，与上层的接口及数据交互方案是一个重要的驱动设计内容。在 TOUCHKEY 驱动中，我们使用环形缓冲区的设计。

1. 在 dev_touchkey_scan 函数中识别到事件后。将事件写入缓冲区。
2. APP 通过 dev_touchkey_read 接口读取事件。

dev_touchkey_scan 函数修改增加如下代码

```
                if(TouchKeyLastCap == 0)
                {
                        //uart_printf("\r\n--------------------init\r\n");
                }
                else
                {
                        //uart_printf("\r\n--------------------chg\r\n");
                        if(last_chg != last_dire)//防止重复上报
                        {
                                //uart_printf("\r\n--------report\r\n");
                                TouchKeyBuf[TouchKeyWrite++] = last_dire;
                                if(TouchKeyWrite >= DEV_TOUCHKEY_BUF_SIZE)
                                TouchKeyWrite = 0;
                        }

                        last_chg = last_dire;

                }
```

实现读函数如下：

```
/**
 *@brief:       dev_touchkey_read
 *@details:     读设备，获取触摸事件
 *@param[in]    u8 *buf
                u32 count
 *@param[out]   无
 *@retval:
 */
```

(continues on next page)

(continued from previous page)

```
s32 dev_touchkey_read(u8 *buf, u32 count)
{
        u32 cnt = 0;

        while(1)
        {
                if(TouchKeyWrite ==  TouchKeyRead)
                        break;

                if(cnt >= count)
                        break;

                *(buf+cnt) = TouchKeyBuf[TouchKeyRead++];
                if(TouchKeyRead >= DEV_TOUCHKEY_BUF_SIZE)
                        TouchKeyRead = 0;

                cnt++;
        }

        return cnt;

}
```

dev_touchkey_test 就是一个应用。将 dev_touchkey_test 放到 main 函数的 while 循环中执行。

```

/**
 *@brief:       dev_touchkey_test
 *@details:     触摸按键测试程序
 *@param[in]    无
 *@param[out]   无
 *@retval:
 */
s32 dev_touchkey_test(void)
{
        u8 tmp;
        s32 res;

        res = dev_touchkey_read(&tmp, 1);
        if(1 == res)
```

(continues on next page)

(continued from previous page)

```
        {
                if(tmp == DEV_TOUCHKEY_TOUCH)
                {
                        uart_printf("touch key test get a touch event!\r\n");
                }
                else if(tmp == DEV_TOUCHKEY_RELEASE)
                {
                        uart_printf("touch key test get a release event!\r\n");
                }
        }

}
```

测试结果

> —hello world!— touch key test get a touch event! touch key test get a release event! touch key test get a touch event! touch key test get a release event! touch key test get a touch event! touch key test get a release event! touch key test get a touch event! touch key test get a release event!

到此，touchkey 设备驱动基本完成。当然，可以优化的地方还很多。

14.5 思考

1. 我们已经实现了发现触摸就上报，触摸离开也上报。长按呢？如何处理？**该谁处理？**
2. 对于一个应用来说，你是触摸按键还是机械按键，它并不关心。APP，触摸按键，机械按键，三者之间的关系关系如何处理？接口如何处理？

14.6 end

CHAPTER 15

I2C-收音机-功放

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

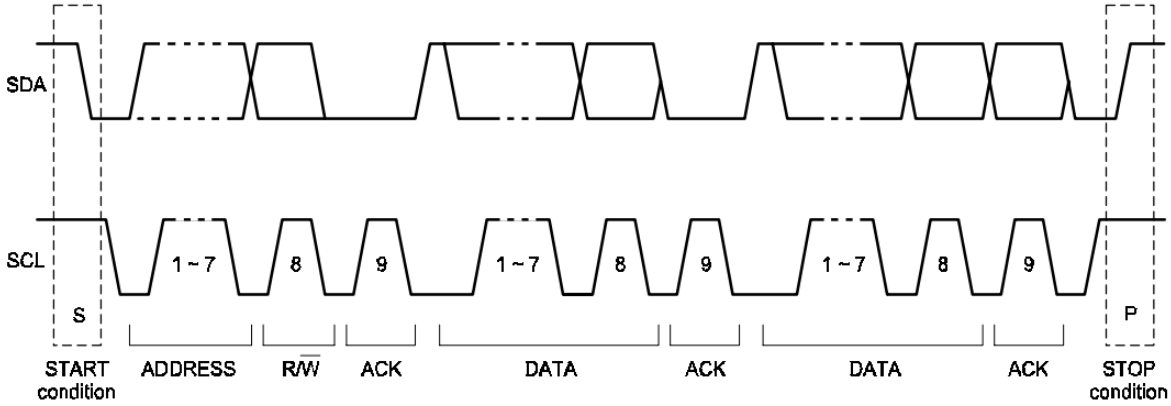
资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

前面已经调试了 IO 口，定时器，串口。本章节我们调试 I2C。别的教程都是用 I2C 控制 EPROM，实际上 EPROM 现在用的已经比较少了。如果只是随便存一点数据，STM32 内部 FLASH 就可以使用。如果要存较多数据，例如字库，一般都使用 SPI FLASH。我们用 I2C 做一点好玩的，控制 TEA5767，一块飞利浦的收音机芯片，玩一波电波情缘。

15.1 I2C 接口

关于 I2C 接口，看文档《I2C 总线协议.pdf》，周立功写的。在第 12 页有一个时序图如下图，我们就从这个图了解



I2C。I2C 时序

1. I2C 通信使用两根线 SDA 和 SCL，SCL 是时钟线，SDA 是数据线，控制时钟的是主机。
2. I2C 通信过程看图底部英文标识：（1）首先发送起始信号 START，然后发送地址 ADDRESSS，接着是读写位 R/W，主机释放 SDA 线，从机使用 SDA 线返回应答位 ACK。地址有 7 位，紧接着的第 8 位是数据方向位 R/ W，0 表示发送（写），1 表示请求数据（读）。两者传输时正好组成一个字节。（2）如果是读，主机释放 SDA，由从机控制，但是时钟还是由主机控制。从机在时钟控制下，从 SDA 线上返回数据，一个字节后，主机控制 SDA 线发送 ACK 信号，如此循环直到读结束。（3）如果是写，主机控制 SDA 线发送数据，一个字节后，释放 SDA 线，从机返回 ACK 信号，如此循环直到写结束。（4）主机发送结束信号。
3. I2C 总线可挂载多个 I2C 设备，通过地址区分，地址有 7 位地址或 10 位地址，常见芯片通常是 7 位。
4. 多个芯片通信时，I2C 会进行仲裁（这部分个人不熟悉，需要了解可以认真看文档，或者直接找飞利浦的文档看）。

15.2 STM32 I2C

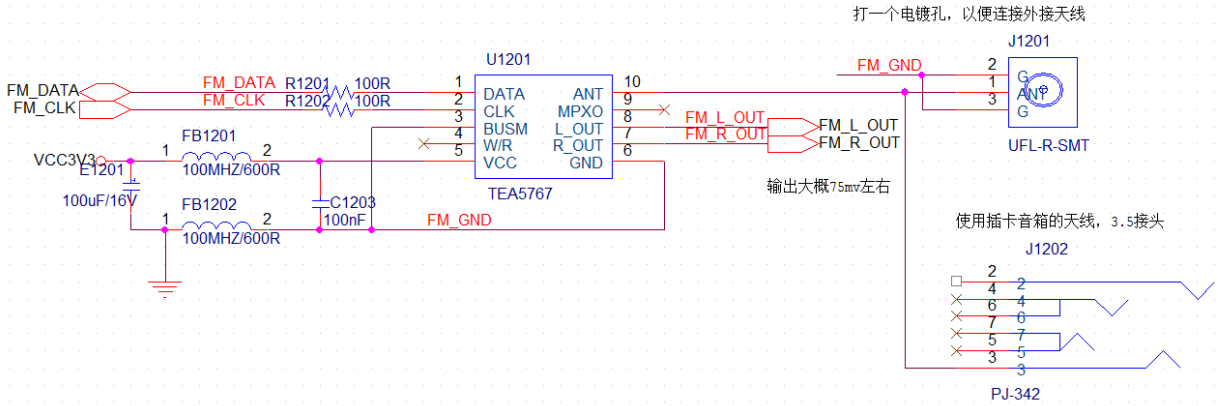
ST 的 I2C 复杂且不好用，口碑不好，大家都在用软件模拟。听说是为了避开飞利浦专利。本次我们使用 IO 口模拟 I2C，**硬件 I2C 在摄像头中会使用**。

15.3 收音机模块

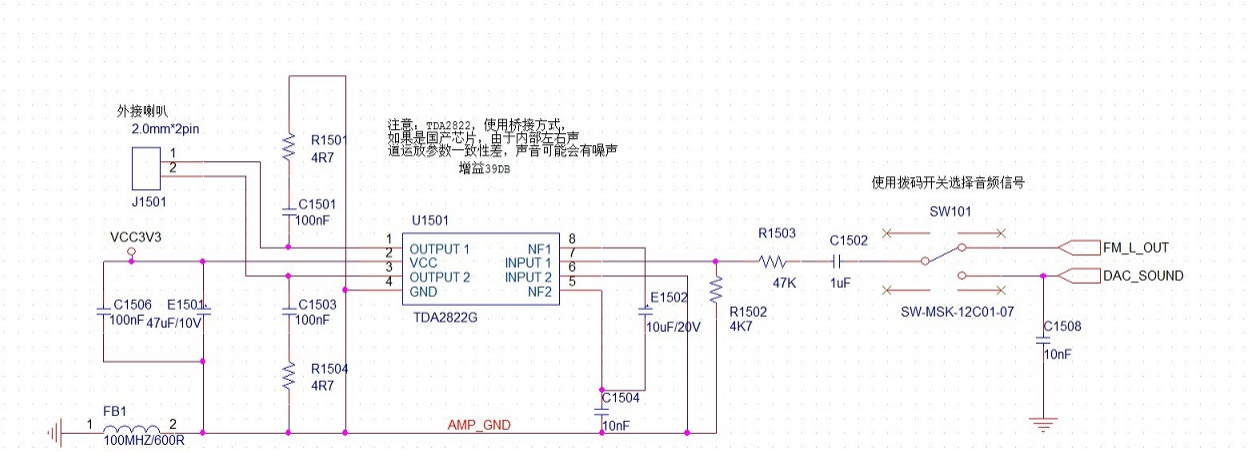
板载的收音机模块 TEA5767 是飞利浦的。**使用总线操控芯片，通常也就是操作芯片里面的寄存器**对于大部分芯片这个说法都是合适的。TEA5767 的功能，通过他的寄存器了解。请参考微控设计网 DC 版主翻译整理的《TEA5767HN 低功耗立体声收音机接收器.pdf》

15.4 原理图

TEA5767 模块通过 I2C 接口控制，输出双声道信号。天线使用 3.5 插卡音箱拉杆天线。收音功能对电源要求较高，在电源端安排了一个 100UF 的钽电容，电源跟地串了磁珠。I2C 信号线也串了 100R 电阻。



TEA5767 电路 TEA5767 输出声音信号只有 20 多 mv 左右，不能直接推动喇叭，需要功放放大，我们使用的功放是



TDA2822。运放电路电路使用 TDA2822 桥接方式，放大倍数较大。输入端使用电阻分压，降低输入信号，防止放大过渡。除了 FM 信号，另外一路 DAC_SOUND 也使用功放，通过拨动开关选择哪路音频信号输入到功放。

15.5 编码

建立两个驱动：mcu_i2c 和 dev_tea5767。权且认为模拟 I2C 属于片上设备吧，放到 mcu_dev 目录。tea5767 属于板上设备，代码放到 board_dev。代码请阅读源文件。

15.5.1 I2C 关键代码

请从 GIT 上下载最新的代码，此处的代码只是教程，不是最新 I2C 主要流程如下面函数，其他 IO 口初始化函数请自行查看代码。

```
s32 mcu_i2c_transfer(u8 addr, u8 rw, u8* data, s32 datalen)
{
    s32 i;
    u8 ch;

    //发送起始
    mcu_i2c_start();
    //发送地址 + 读写标志
    //处理 ADDR
    if(rw == MCU_I2C_MODE_W)
    {
        addr = ((addr<<1)&0xfe);
        //uart_printf("write\r\n");
    }
    else
    {
        addr = ((addr<<1)|0x01);
        //uart_printf("read\r\n");
    }

    //uart_printf("i2c addr:%02x\r\n", addr);
    mcu_i2c_writebyte(addr);
    mcu_i2c_wait_ack();

    i = 0;
    while(i < datalen)
    {
        //数据传输
        if(rw == MCU_I2C_MODE_W)//写
        {
            ch = *(data+i);
            //uart_printf("i2c:w:%02x\r\n", ch);
            mcu_i2c_writebyte(ch);
            mcu_i2c_wait_ack();

        }
```

(continues on next page)

(continued from previous page)

```
        else if(rw == MCU_I2C_MODE_R)//读
        {
            ch = mcu_i2c_readbyte();
            mcu_i2c_ack();
            *(data+i) = ch;
            //uart_printf("i2c:r:%02x\r\n", ch);
        }
        i++;
    }

    //发送结束
    mcu_i2c_stop();
    return 0;
}
```

1. 输入参数 addr 是七位地址，不包含读写位。
2. 参数 rw 为读写标志。
3. 首先发送 start 信号
4. 根据读写标志处理 addr，发送 addr 后等待 ack。
5. 进入数据传输，读写传输流程分开。
6. 数据传输结束后发送 stop 信号。

mcu_i2c_transfer 的实现，可见流程是完全按照 I2C 波形设计的。值得注意的是函数参数的设计：1 地址用 7 位，这个是根据实际设计，I2C 地址就是 7 位的，很多代码将地址设计为 8 位，将读写标志也包含，个人认为不符合要求。2 读写标志单独做一个参数，如此无论读写，都只是用一个函数。

15.5.2 TEA5767 代码设计

略

TEA5767 实际应用场景并不多，有兴趣的可自行研究。

15.6 调试

- 首先调试 I2C
- 程序跑起来后，在等待 I2C 的 ACK 处超时。检查硬件，发现两个调试用的电阻没焊上。
- 焊上后，还是不行，上逻辑分析仪，没抓到波形。

- 加一个简单的测试程序，定时翻转两个 IO 口电平，用逻辑分析仪抓波形，抓不到波形。

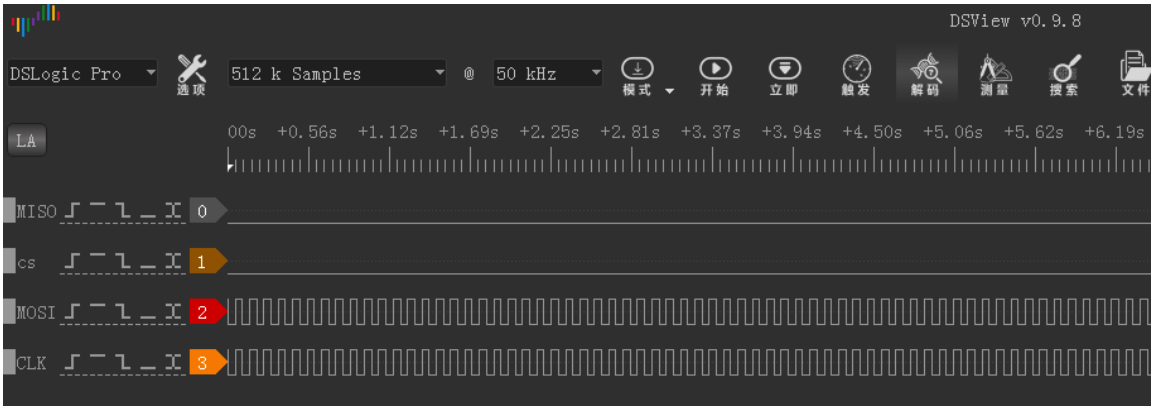
```
00248:      while(1)
00249:      {
00250:          uart_printf("test \r\n");
00251:          mcu_i2c_scl(1);
00252:          mcu_i2c_sda(1);
00253:          Delay(5);
00254:          mcu_i2c_scl(0);
00255:          mcu_i2c_sda(0);
00256:          Delay(5);
00257:      }
```

I2C 无波形

- 查看初始化代码，发现早上测试 SPI 的时候，把 I2C 初始化屏蔽了，自己坑自己了，打开如下。

```
00139:      mcu_dev_uart_open(3);
00140:      //mcu_timer_init();
00141:      //dev_buzzer_init();
00142:
00143:      mcu_spi_init(); //初始化 SPI 控制器
00144:      dev_spiflash_init();//初始化 spi flash
00145:
00146:      mcu_i2c_init();
00147:      dev_tea5767_init();
00148:
00149:    /* Infinite loop */
00150:    while (1)
00151:    {
```

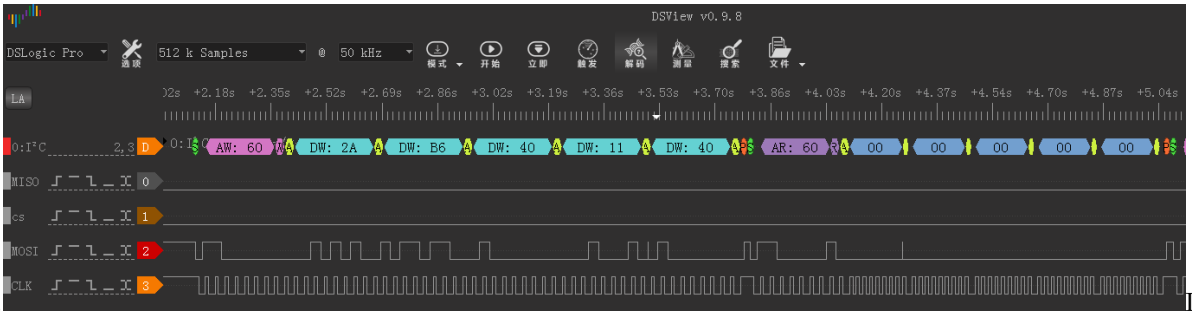
I2C 测试波形



- 抓到正常翻转波形。

翻转波形

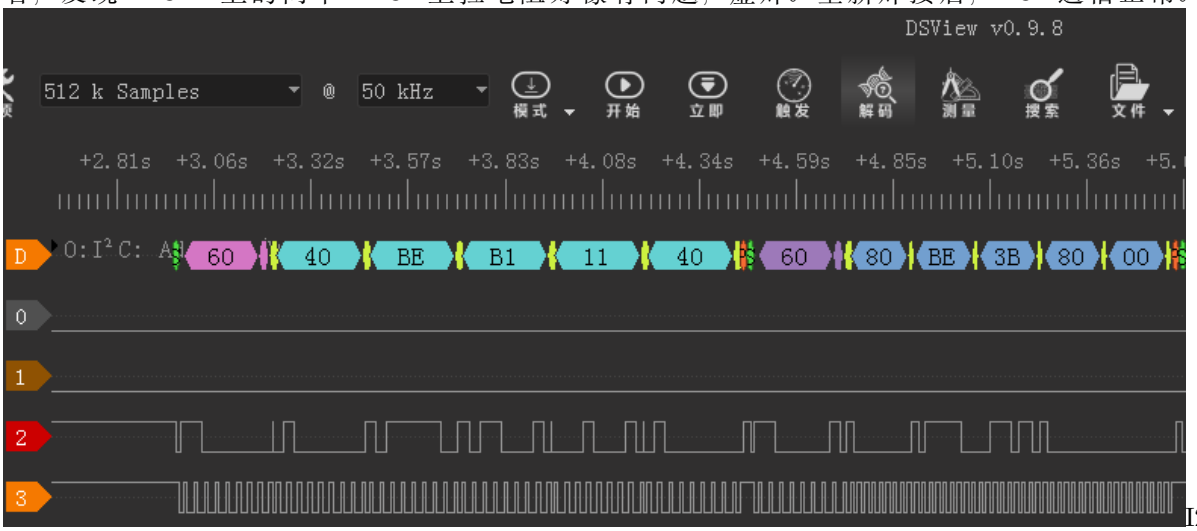
- 恢复程序，调试信息不再输出超时，但是读回来的数据全部是 0x00，肯定不对，用逻辑分析仪抓到的协议



也全部是 0X00。I

读数据不对

- 逻辑分析仪能抓到发送波形，程序应该没什么问题。问题应该是芯片或者 I2C 接口，再查看，发现 PCB 上的两个 I2C 上拉电阻好像有问题，虚焊。重新焊接后，I2C 通信正常。



I2C

虚焊

15.6.1 TEA5767

- 控制通过 I2C 控制 TEA5767，读写不带寄存器地址。读，则连续读出 5 个字节，写，同样一次性写五个字节，但是五个字节数据意义不一样。具体见《TEA5767HN 低功耗立体声收音机接收器.pdf》。
- 寻台 TEA 的操作主要是寻台，提供两个函数 dev_tea5767_auto_search 和 dev_tea5767_search。

  dev_tea5767_auto_searc 芯片自动寻台，但是寻台并不是很准，寻台成功后，要判断信号强度。以免错寻。

  dev_tea5767_search 程序寻台，直接设置一个频率，延时后读信号强度，不符合要求则再设置下一个频率。

15.6.2 测试函数

在 main.c 中进行测试，首先要初始化 I2C 接口，再打开 tea5767，然后强制设置一个频率 (请设置当地 FM 电台频率)，在循环内，当按键按下时，开始寻台。

```
    mcu_i2c_init();

    //mcu_timer_init();
    dev_key_init();
    dev_buzzer_init();

    dev_tea5767_open();
    dev_tea5767_setfre(97100);

    while (1)
    {
            s32 key;
            key = dev_key_scan();
            if(key == 0)
            {
                    GPIO_ResetBits(GPIOG, GPIO_Pin_0
                      | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                    //dev_buzzer_open();
                    dev_tea5767_search(1);
            }
            else if(key == 1)
            {

                    GPIO_SetBits(GPIOG, GPIO_Pin_0
                       | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                    dev_buzzer_close();

            }
            Delay(5);

            /* 测试触摸按键 */
            dev_touchkey_task();
            dev_touchkey_test();
    }
```

1. 板载的 TEA5767 毕竟是一个小模组而已，性能无法和收音机相比。
2. 电脑电源会带来干扰，降低收音机灵敏度，用充电宝供电并且断开与电脑所有连接，效果会提升不少。
3. 配套的天线只能做功能测试，如果效果不好，可以在天线尾端接一段导线，并且将导线挂到高处。导线并不是越长越好，太长反而会引入其他干扰。按照 FM 的波长，天线总长 65 厘米左右，实测接一段 60 厘米的导线效果不错。

4. 空旷处（窗户边）肯定比室内效果要好。
5. 网络、摄像头、USB、SD 卡、TFT LCD 屏等，在运行时，都会发射干扰，降低收音机灵敏度。如要解决这个问题，需要增加屏蔽措施，考虑毕竟只是一块开发板，决定不做如此复杂，而且经过测试，在收音机信号良好的情况下，干扰影响不大。
6. 通过 WM8978 播放收音比 TDA2822 效果要好（工作室没能力调音，TDA2822 单声道，WM8978 立体声）。

15.7 思考

多 I2C 控制器和多个 I2C 外设之间的交叉组合，如何编写驱动？在此提前说一下：要有一点面向对象思想，I2C 控制器是一个对象，I2C 设备是一个对象。更重要的是，I2C 控制器驱动（代码）也是一个对象，I2C 设备的驱动（代码）当然也可以认为是一个对象。

GITHUB 仓库最新代码已经实现，请自行查阅

15.8 end

CHAPTER 16

DAC-波形-声音的真相

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

人机交互的电子产品经常需要语音提示。如果没有语音外设，可以通过一个 DAC 输出波形，经简单放大后就能发出声音。如果音源干净清晰，电路设计好，音质还是非常不错的。台系（华邦等）的语音芯片通常就是 DAC 输出音乐。大家小时候用的音乐贺卡，就是用这些芯片制作的。

16.1 DAC 是什么？

DAC 是数字模拟转换器（英语：Digital to analog converter，英文缩写：DAC），是一种将数字信号转换为模拟信号（以电流、电压或电荷的形式）的设备。上面的定义比较抽象，在单片机来说，形象的说法是：

> 给一个在 DAC **位数**范围内的值，这个值就是数字量，DAC 就根据**参考电压**，将其转换为电压值，在指定的管脚上输出一个**电压**，也即是**模拟量**。

1. **参考电压 Vref**：DAC 转换后输出的最高电压，DAC 输出范围 0~Vref。精度也是根据参考电压计算。
2. **DAC 位数**：DAC 的关键性能，位数即是 DAC 精度，也是 DAC 的输出步进。通常有 8 位、10 位、12 位等。例如 12 位，即是说可以将输出精确到：Vref/0xfff。一个 12 位的 DAC 在 3.3V 参考电压下，输出可以精确到 0.805mv。将 0X01 送到 DAC，管脚将输出 0.805mv；将 0x02 送到 DAC，管脚将输出 1.61mv；将 0xfff 送到 DAC，将输出 Vref。

16.2 STM32 DAC

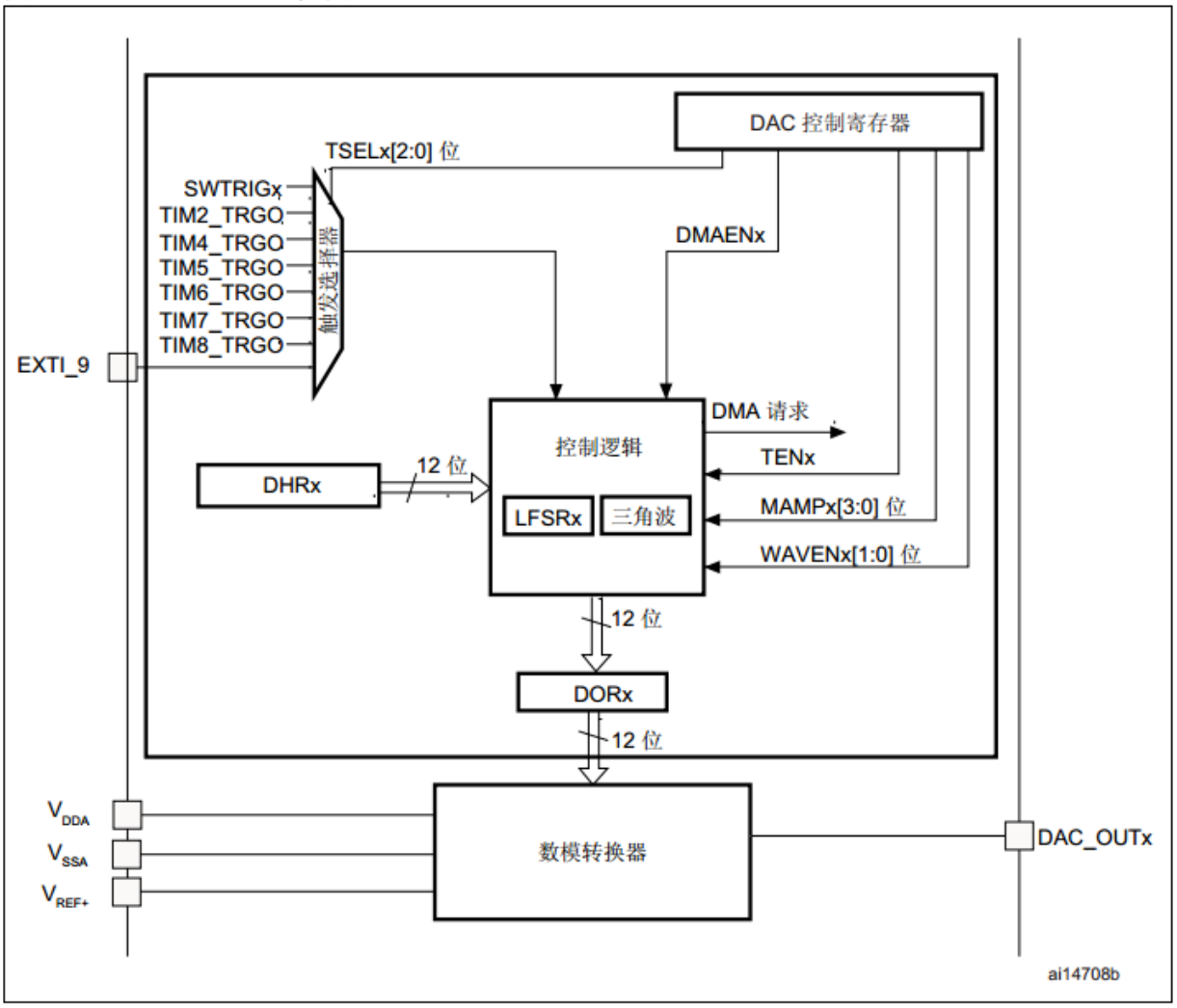
查看《STM32F4xx 中文参考手册.pdf》STM32F4 系列 DAC 功能特性如下：

DAC 特性

功能框图如下，从图可以看出：

- DAC 可以用软件触发、定时器触发、外部 IO 触发。
- DAC 可以有 DMA。
- 最下方的数模转换器，在控制逻辑控制之下，根据输入电压，在 DAC_OUT 上输出 DAC 电压。

图 54. DAC 通道框图



DAC 通道框图硬件上使用 PA5 作为 DAC 输出测试。在《STM32F407_ 数据手册.pdf》管脚描述表格中有说明 PA5 是 DAC2 的输出管脚。

									EVENTOUT	
21	30	41	P4	51	PA5	I/O	TTa	(4)	SPI1_SCK/ OTG_HS_ULPI_CK / TIM2_CH1_ETR/ TIM8_CHIN/ EVENTOUT	ADC12_IN5/DAC2_OUT
									SPI1_MISO /	

PA5DAC

16.3 声音

声音是一种波。在电子上，波，就是不同电压值在时间上的序列。因此，在 DAC 管脚上，一直持续输出不同的电压值，即可形成一列波，这列波通过放大，通过喇叭转换，震动空气，就变成了声波。通常的 CD 音乐采样频率时 44.1K，属于高保真。但是实际上，只要 8K 的采样频率，声音还原质量就很好了。儿童玩具、声音贺卡的声音通常就是 8K。8K 采样频率，每个样点间隔就是 1s/8k=125us。因此，**将一个 8K 采样的声音文件，每 125us 读取一个声音文件里面的样点，在 dac 上输出，就可以还原声音了**。

16.4 编码调试

调试分三步：

1. 先调试 DAC 输出正确电压。
2. 再调试播放一段内嵌在程序的声音。
3. 最后调试播放一个 WAV 声音文件（这一步暂时不做，等文件系统跟 SD 卡驱动做好后再调试，反正是纯软件调试，不影响验证硬件）。

16.4.1 DAC 调试

首先要让 DAC 能输出指定电压值。添加 mcu_dac.c 和 mcu_dac.h 到工程。

- 初始化

```
/**
 *@brief:      mcu_dac_open
 *@details:    打开 DAC 控制器
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 mcu_dac_open(void)
{
        GPIO_InitTypeDef GPIO_InitStructure;
    DAC_InitTypeDef DAC_InitType;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);//----使能 PA 时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);//----使能 DAC 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;//---模拟模式
```

(continues on next page)

(continued from previous page)

```
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;//---下拉
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);//---初始化 GPIO

    DAC_InitType.DAC_Trigger=DAC_Trigger_None;  //---不使用触发功能 TEN1=0
    DAC_InitType.DAC_WaveGeneration=DAC_WaveGeneration_None;   //---不使用波形发生
    DAC_InitType.DAC_LFSRUnmask_TriangleAmplitude=DAC_LFSRUnmask_Bit0;
    DAC_InitType.DAC_OutputBuffer=DAC_OutputBuffer_Disable ;        //---输出缓存关闭
    //DAC_InitType.DAC_LFSRUnmask_TriangleAmplitude = DAC_TriangleAmplitude_4095; //噪声
生成器
        DAC_Init(DAC_Channel_2,&DAC_InitType); //---初始化 DAC 通道 2

        DAC_Cmd(DAC_Channel_2, ENABLE); //---使能 DAC 通道 2
    DAC_SetChannel2Data(DAC_Align_12b_R, 0); //---12 位右对齐数据格式   输出 0

                return 0;
}
```

上面函数是打开 DAC 代码，其实也就是初始化配置 DAC。和前面的定时器输出和定时器输入一样，除了使用 DAC 外设，还需要用 IO 口。

> 16~20 行，将 IO 口 PA5 配置为模拟功能。22 行，配置 DAC 不使用触发。23 行，不使用波形发生器，DAC 可以产生三角波等波形。24 行设置屏蔽/幅值选择，只有用波形发生器才有用。25 行禁止输出缓存。27 行执行配置。

- 输出电压

```
/**
 *@brief:      mcu_dac_output
 *@details:    设置 DAC 输出值
 *@param[in]   u16 vol,  电压，单位 MV，0-Vref
 *@param[out]  无
 *@retval:
 */
s32 mcu_dac_output_vol(u16 vol)
{

    u32 temp;

    temp = (0xfff*vol)/3300;
```

(continues on next page)

(continued from previous page)

```
    MCU_DAC_DEBUG(LOG_DEBUG, "\r\n---test dac data:%d-----\r\n", temp);

    DAC_SetChannel2Data(DAC_Align_12b_R, temp);//12 位右对齐数据格式
                return 0;
}
```

13 行是电压计算，原理是：

> 配置值/电压 = 0XFFF/3.3V 配置值是我们要写到 DAC 的值，电压就是输入参数，单位是 mv。3300 也就是 3300mv；0xFFF，因为我们的 DAC 是 12 位，也即是说，当我们设置 DAC 为 0XFFF 时，DAC 输出 3.3V。

17 行调用函数将配置值写到 DAC。

- 测试程序

```
s32 mcu_dac_test(void)
{
    uart_printf("\r\n---test dac!-----\r\n");

    mcu_dac_open();
    mcu_dac_output_vol(1500);//1.5v
    while(1);
}
```

程序设计输出 1.5V，测试输出管脚，电压为 1.492V，基本准确，偏差 0.01V，这个偏差有可能是基准电压，也就是我们的 3.3V 有偏差。实测 3.3V，只有 3.28V，偏差 0.02V。如果要求不是很严格的场合，基本算正常。

16.4.2 播放语音调试

在写语音播放代码之前要记住以下几点：

1. 是 app 调用 DAC 声音驱动播放声音，还是 DAC 声音驱动去找语音数据。
2. 根据 1，请问是 APP 提供声音数据给 DAC 驱动还是 DAC 声音去找声音数据?
3. 要播放一个保存在 SD 卡中的 8K 采样频率的 WAV 文件，请问：SD 卡，8K 采样，WAV，这三个参数跟 DAC 声音驱动是否有关?

对于这几点，个人看法如下：

> DAC 声音驱动只实现将一定格式的声音数据转换为声音。格式包含什么呢? 采样频率，单声道还是多声道，多少位，这三个参数是 DAC 需要的。文件格式是 WAV 还是 PCM 还是 MP3，跟 DAC 声音驱动没关系，至于你是放在 SD 卡还是 U 盘，那更加没关系了。这些事情，应该由**语音播放中间层**处理。

那么 DAC sound 驱动要提供什么接口呢?

> init—-初始化设备 open—-打开设备，意味则要用这个设备 close—-关闭设备 setting—设置，采样频率，声道，位宽（当然，对于 DACsound 来说只支持单声道，位宽也是固定的）提供一个控制接口—-控制启动播放，暂停，停止，查询状态最后一个接口就是填充数据，如何填充? 请思考。

以上的问题在本节暂时不处理，后面等我们做完 *WM8978* 的驱动，两个声音驱动一起分析，对于一个声音驱动应该做成什么样子，就更加清晰明了了。 **现在先使用最快的速度编写一套代码，让硬件发出声音，以便硬件改版，软件架构问题后续慢慢优化**。

- 语音播放流程

播放 DAC 语音，就是使用 DAC 和 IO 口还有定时器的配合。

1. 初始化 DAC 和 IO。
2. 定时器设置为 125us 中断一次。
3. 在定时器中断中读取语音数据，并用 DAC 输出电压。
4. 重启定时器，循环 3，直到语音播放结束。

- 声音数据准备

现在还没有调试 WAV 解码，也没有完成 SD 卡文件系统。只好将一段声音内嵌到代码内，这样也可以避免其他模块干扰，只验证 DAC 播放语音功能。

如何将一段声音内嵌到代码？

- 代码驱动说明

在 board_dev 文件夹创建 dacsound 驱动源码文件：dev_dacsound.c、dev_dacsound.h

在 mcu_timer 驱动中增加定时器 3 初始化和中断处理函数，定时 125us。定时器前面已经学习，不再累赘在中断中调用 dev_dacsound_timerinit 函数输出 DAC 电压。

调用 dev_dacsound_open 初始化 dacsound 功能。调用 dev_dacsound_play 开始播放，函数内开启了定时器。然后进入主要处理函数 dev_dacsound_timerinit，这个函数在定时中断中调用，125us 执行一次。

```
s32 dev_dacsound_timerinit(void)
{
    u8 data1 = 0, data2 = 0;
    s16 data = 0;
    u16 tmp;

        data1 = BeepData[soundindex++];
        data2 = BeepData[soundindex++];
        /* 要注意，读到的数据是 S16，正负值 */
        data = (s16)((data2 << 8) | data1);
        tmp = (data+0X7FFF)>>4;//12 位 DAC
```

(continues on next page)

(continued from previous page)

```
        //uart_printf("%04x ", tmp);
        mcu_dac_output(tmp);

        if(soundindex >= BEEP_DATA_LEN)
        {
                uart_printf("dac sound play finish!");
                /* 停止定时器 */
                mcu_tim3_stop();
        }
}
```

处理过程并不复杂，读取两个字节数据，组成一个 16 位数据，丢到 DAC。需要注意的是:

1. **声音数据是 s16，也就是最高位是正负标志。但是我们的 DAC 可不支持负数，因此需要将波形直流电平（波形水平中间线，类似 X 轴），由 0V 抬高，抬高多少呢？抬高到最高电压的一半，也就是 0X7FFF，我们直接加上 0X7FFF 的偏移。**
2. 我们的 DAC 是 12 位的，数据是 16 位的，数据右移 4 位匹配。
3. **本算法有音频失真，请问原因是什么？最新处理方法请查看 github 上持续更新的代码。**

还要记得在 stm32f4xx_it.c 添加中断入口

```
void TIM3_IRQHandler(void)
{
    mcu_tim3_IRQhandler();
}
```

- 测试修改 main.c，第 3 行打开 dacsound，第 12 行，当按下按键时，播放语音。

```
/* Infinite loop */
        mcu_uart_open(3);
        wjq_log(LOG_INFO, "hello word!\r\n");
        mcu_i2c_init();
        dev_key_init();
        //mcu_timer_init();
        dev_buzzer_init();
        dev_tea5767_init();
        dev_dacsound_init();

        dev_key_open();
        dev_dacsound_open();
```

(continues on next page)

(continued from previous page)

```
        //dev_tea5767_open();
        //dev_tea5767_setfre(105700);

        while (1)
        {
                /* 驱动轮询 */
                dev_key_scan();

                /* 应用 */
                u8 key;
                s32 res;

                res = dev_key_read(&key, 1);
                if(res == 1)
                {
                        if(key == DEV_KEY_PRESS)
                        {
                                //dev_buzzer_open();
                                dev_dacsound_play();
                                GPIO_ResetBits(GPIOG, GPIO_Pin_0
                                | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                                //dev_tea5767_search(1);
                        }
                        else if(key == DEV_KEY_REL)
                        {
                                //dev_buzzer_close();
                                GPIO_SetBits(GPIOG, GPIO_Pin_0
                                | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                        }
                }

                Delay(1);

                /* 测试触摸按键 */
                //dev_touchkey_task();
                //dev_touchkey_test();
        }
```

现在应该能听到声音了。

16.5 思考

现在我们仅仅是验证了 DAC 播放语音功能。如何解码 WAV？dacsound 提供什么接口？使用什么机制播放？这些问题我们等 WM8978 调试之后优化完成。

---

end

CHAPTER 17

SPI-SPI FLASH

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

前面我们调试了第一条总线——I2C 总线，在嵌入式领域还有另外一条常用总线——SPI。现在我们就来调试 SPI 和 SPI FLASH 。

17.1 SPI 总线

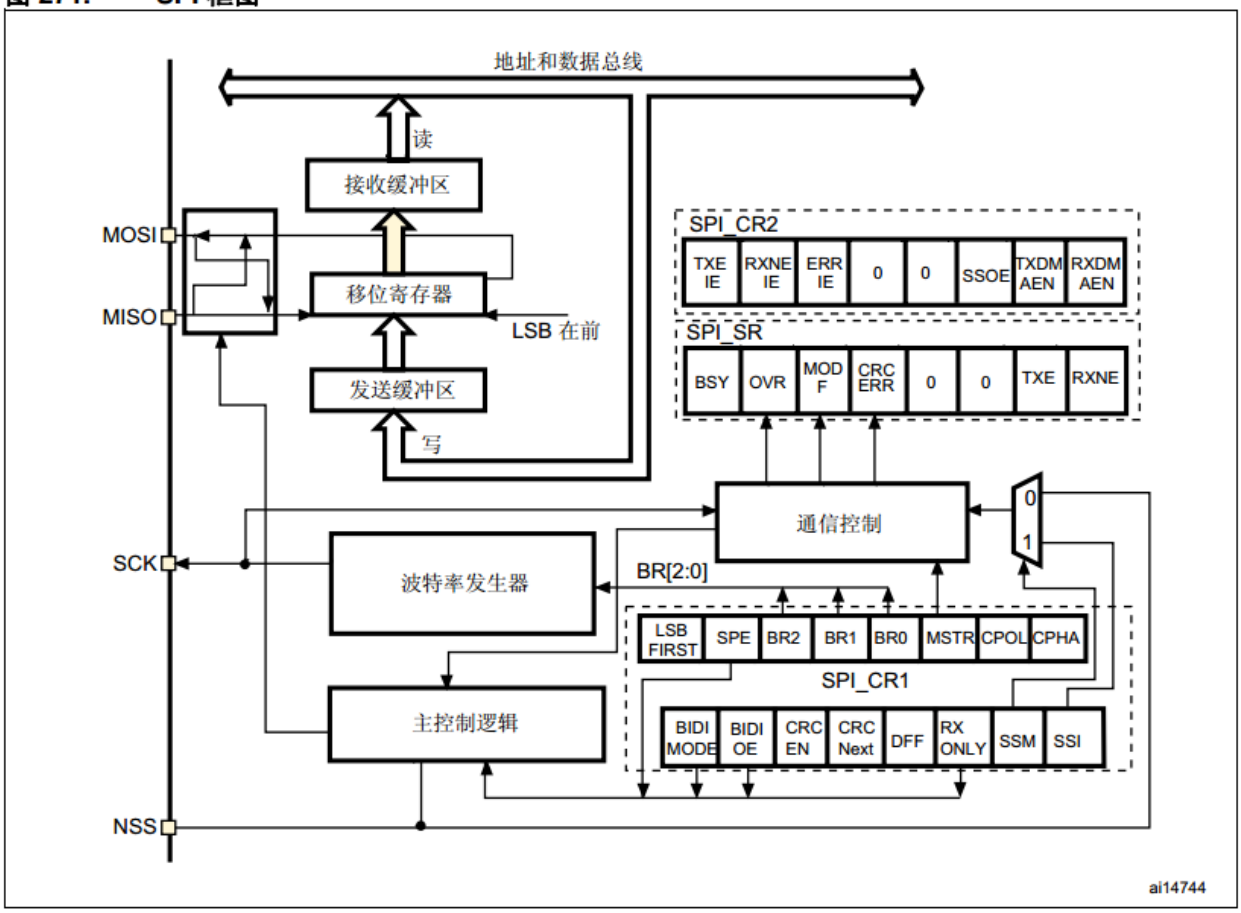
SPI 是串行外设接口（Serial Peripheral Interface）的缩写。大概资料，百度百科 https://baike.baidu.com/item/SPI%E6%8E%A5%E5%8F%A3/2527392

1. SPI 通常使用 4 根线连接。(1) MOSI – 主器件数据输出，从器件数据输入（master out slave in）(2) MISO – 主器件数据输入，从器件数据输出（master in slave out）(3) SCLK –时钟信号，由主器件产生。(4) NSS – 从器件使能信号，由主器件控制，通常叫 CS、片选。
2. SPI 根据时钟极性 CPOL 与时钟相位 CPHA 的不同，有 4 钟工作模式。
3. SPI 使用多个片选管脚就可连接多个从设备。

17.2 STM32 SPI 控制器

在《*STM32F4xx 中文参考手册.pdf*》"*27 串行外设接口 (SPI)*" 章节有详细说明。STM32 的 SPI 控制器框图如下：

图 271. SPI 框图



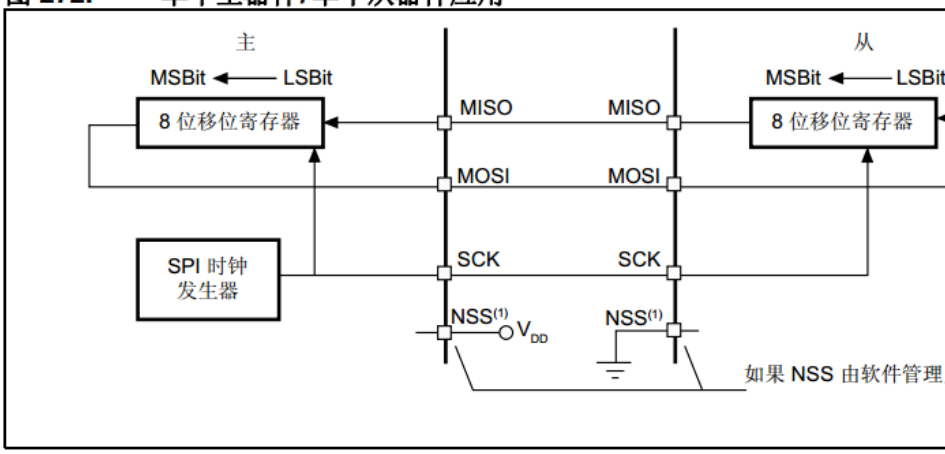
SPI 控制器从中可以看出：

1 移位寄存器只有一个，因为接受和发送是同时进行的。发送从右边出去，接收从左边进来。2 4 根引脚，MOSI、MISO、SCK、NSS（CS）

SPI 的大概工作过程就是：

1. 主设备将对应从设备 CS 拉低，使能从设备。
2. 主设备输出时钟信号，主设备移位寄存器的数据按 BIT 从 MOSI 上输出，从设备收到 BIT 后，保存到自己的移位寄存器，**同时**将自己移位寄存器中的数据从 MISO 上输出，主设备收到后保存在移位寄存器中。如此循环，8 个 BIT 传输结束，进行读写操作，然后进行下一个字节的传输。

图 272. 单个主器件/单个从器件应用



3. 传输结束,将从设备片选拉高,结束。
   传输四种工作模式见参考手册图 273。
4. SPI 时钟在《STM32F407_ 数据手册.pdf》有描述，SPI3 最高可以达到 21Mbit/s。

2.2.23 Serial peripheral interface (SPI)

The STM32F40x feature up to three SPIs in slave and master modes in full-duplex and simplex communication modes. SPI1 can communicate at up to 37.5 Mbits/s, SPI2 and SPI3 can communicate at up to 21 Mbit/s. The 3-bit prescaler gives 8 master mode frequencies and the frame is configurable to 8 bits or 16 bits. The hardware CRC generation/verification supports basic SD Card/MMC modes. All SPIs can be served by the DMA controller.

The SPI interface can be configured to operate in TI mode for communications in master mode and slave mode.

SPI 最快速度在表格 11 中可以看到 APB1 的时钟最快 42M，SPI3 就挂载 APB1 上，为了达到最快的 21M，必须使用 2 分频。

Table 11. General operating conditions

Symbol	Parameter	Conditions	Min	Max	Unit
f_{HCLK}	Internal AHB clock frequency	VOS bit in PWR_CR register = 0(1)	0	144	MHz
		VOS bit in PWR_CR register= 1	0	168	
f_{PCLK1}	Internal APB1 clock frequency		0	42	
f_{PCLK2}	Internal APB2 clock frequency		0	84	
V_{DD}	Standard operating voltage		1.8(2)	3.6	V
V_{DDA}(3)(4)	Analog operating voltage (ADC limited to 1.2 M samples)	Must be the same potential as V_{DD}(5)	1.8(2)	3.6	V
	Analog operating voltage (ADC limited to 1.4 M samples)		2.4	3.6	
V_{BAT}	Backup operating voltage		1.65	3.6	V

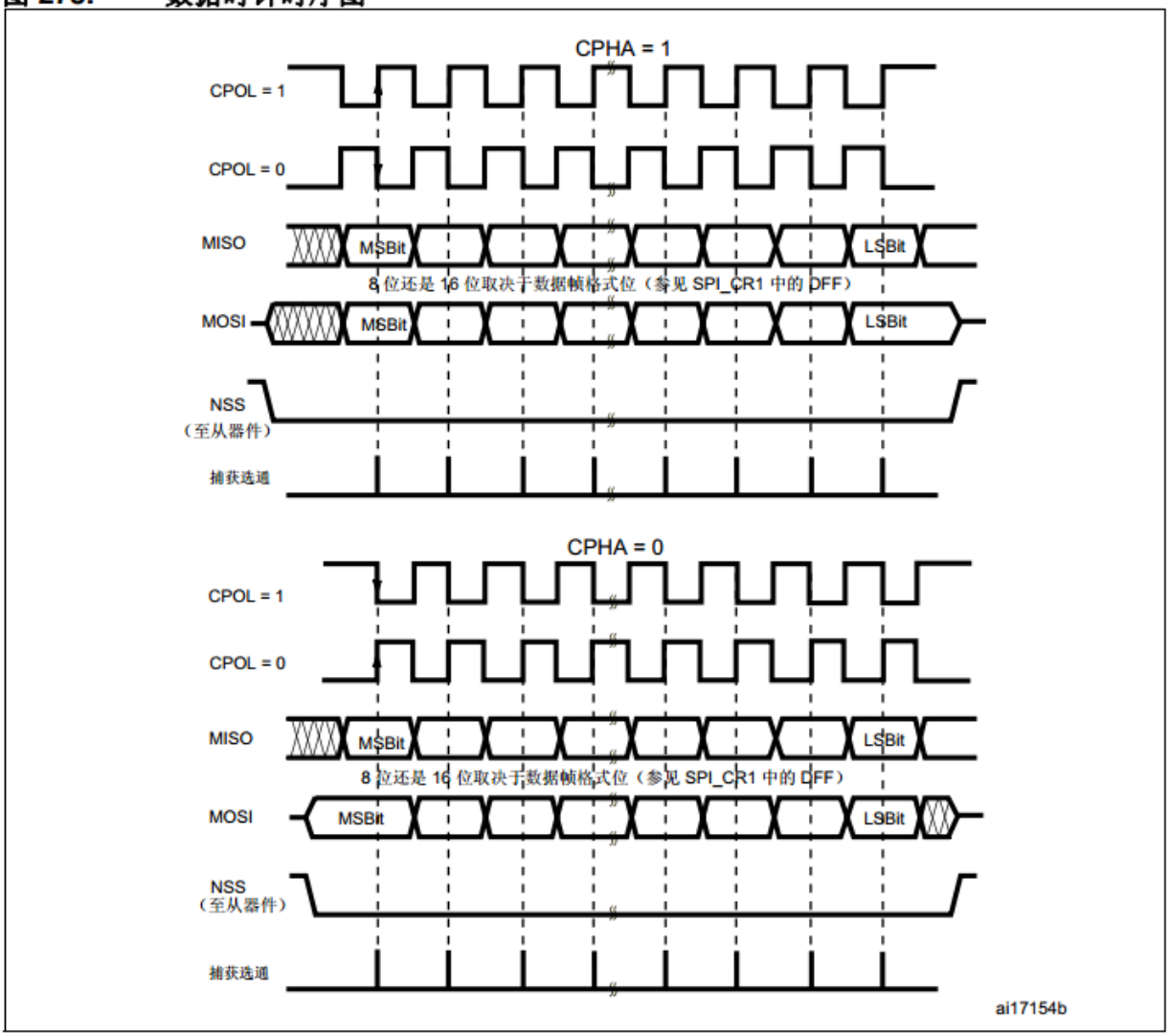
SPI

分频

前面说到，SPI 有 4 中工作模式，在 STM32 的参考手册也有说明。

1. CPOL 是时钟极性，如果为 1，则是先输出低再输出高。0 则相反。
2. CPHA 是时钟相位，如果为 1，则是 180 度。0 则是 0 度。通俗的说，如果是 1，就在时钟的第二个边沿采样。如果是 0，就在时钟的第一个边沿采样。

图 273. 数据时钟时序图



1. 所示的时序为 SPI_CR1 寄存器中的 LSBFIRST 位复位时的时序。

时钟图

从上图也可以看到一个标准的 SPI 通信时序是怎么样的。主设备输出时钟，并在 MOSI 上输出数据。从设备在 MISO 上输出数据。在通信期间，NSS(CS) 保持低电平。

17.3 SPI FLASH

底板使用的 SPI FLASH 是 MX25L3206EZNI-12G 。我们就通过这个芯片的规格书《MX25L3206E_DS_EN.pdf》学习 SPI FLAHS。

1. 从名字通常能看出容量大小，32Mbit/8=4Mbyte，我们通常使用 Byte，因此这个芯片只有 4M，并不是 32M，如果别人跟你说 32M 的 FLASH，要搞清楚单位，很可能是 32Mbit。
2. 文档开头会描述性能，软件需要关心的是: (1) 工作模式，本芯片支持 Mode 0 和

Mode3。（2）1024 个 sector，每个 sector 有 4K。每个 sector 都可以单独擦除。（3）64 个 BLOCK，每个 BLOCK 大小 64K，也就是说，一个 BLOCK 有 16 个 SECTOR。BOLOCK 也可以整体擦除。（4）可以 page 编程，一个 page 有 256 字节。

FEATURES

GENERAL

- Single Power Supply Operation
  - 2.7 to 3.6 volt for read, erase, and program operations
- Serial Peripheral Interface compatible -- Mode 0 and Mode 3
- 33,554,432 x 1 bit structure or 16,777,216 x 2 bits (Dual Output mode) structure
- 1024 Equal Sectors with 4K byte each
  - Any Sector can be erased individually
- 64 Equal Blocks with 64K byte each
  - Any Block can be erased individually
- Program Capability
  - Byte base
  - Page base (256 bytes)
- Latch-up protected to 100mA from -1V to Vcc +1V

SPI

FLASH 特性

3. 之后是 PERFORMANCE 跟 SOFTWARE FEATURES，主要是一些参数特性，例如擦除时间等。一般来说不用太关注。除非系统有要求。比较影响性能的也就是擦除时间。有些厂家的会比较慢。其实相对 CPU 速度来说，擦除 FLASH 是一个很慢的过程。部分 FLASH 会有额外性能，例如有 **OTP 区**，有**加密区**等等。

4. FLASH 的组织需要关注一下，特别是在 FLASH 进行替代的时候，组织模式一定要一样。也就是说 BLOCK、SECTOR、PAGE 的分布要一致。有些芯片只有前面 2 个 BLOCK 可以页操作，后续的只能 sector 操作。

MEMORY ORGANIZATION

Table 1. Memory Organization

Block	Sector	Address Range	
63	1023	3FF000h	3FFFFFh
	:	:	:
	1008	3F0000h	3F0FFFh
62	1007	3EF000h	3EFFFFh
	:	:	:
	992	3E0000h	3E0FFFh
: :	: :	: :	: :
0	15	00F000h	00FFFFh
	:	:	:
	3	003000h	003FFFh
	2	002000h	002FFFh
	1	001000h	001FFFh
	0	000000h	000FFFh

SPI flash 组织

5. 之后需要认真关注的是命令。命令就是操作 FLASH 时主机发给 FLASH 的指令。如何使用后面会说明。

COMMAND DESCRIPTION

Table 4. COMMAND DEFINITION

SPI flash 命令

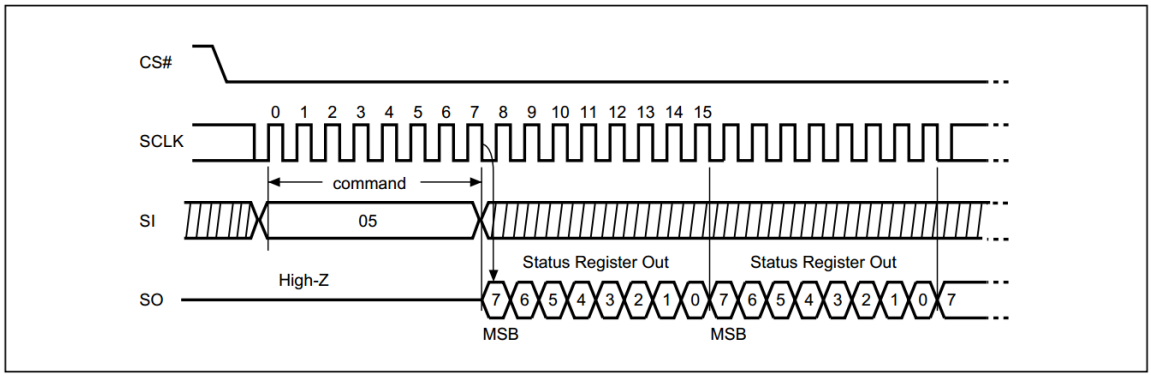
6. 时序分析前面几个是 SPI 时序，一般不看，遇到问题挂示波器才会对比一下。

Timing Analysis

Figure 7. Serial Input Timing

SPI falsh 时序章节但是到后面就是命令流程了，也就是说明命令如何使用，FLASH 如何操作，本处挑两个说说

Figure 13. Read Status Register (RDSR) Sequence (Command 05)

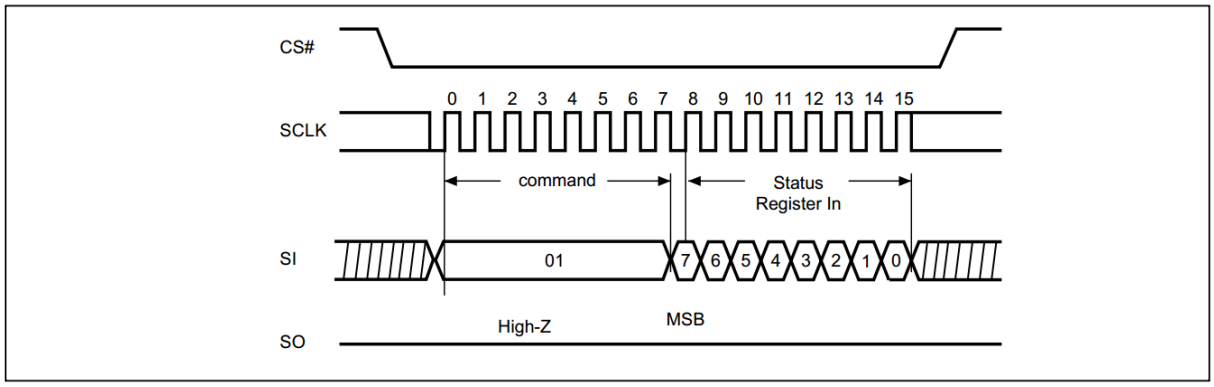


SPI

falsh 读状态上图是读 FLASH 的状态，操作过程就是：

1. 主机拉低 CS 信号，使能 FLASH。
2. 主机将命令 05 发送给 FLASH，这时候 SO 属于高阻状态，说明 FLASH 是不回数据的，主机也不需要 FLASH 回。主机会读到一个 0XFF。
3. 主机继续发送时钟，但是 SI 线上发送什么数据无所谓，通常我们发送 0XFF。此时 FLASH 就会回数据给主机了。

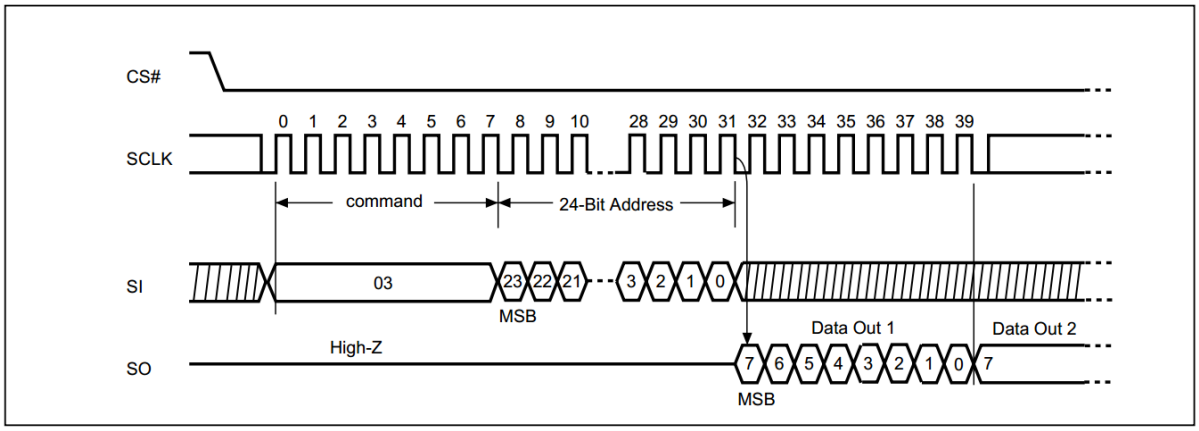
Figure 14. Write Status Register (WRSR) Sequence (Command 01)



SPI

falsh 写状态上图是写状态的，大家应该能看懂了，整个过程 FLASH 没有在 SO 线上返回数据。

Figure 15. Read Data Bytes (READ) Sequence (Command 03)



SPI

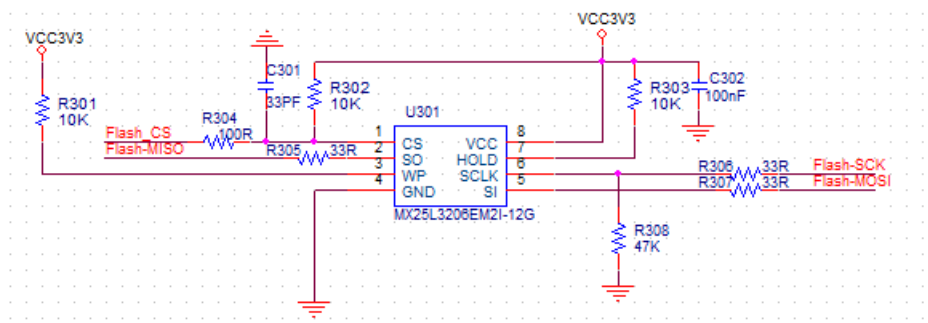
falsh 读数据上图是读数据，与读状态不同的是，CPU 要多发送 24bit 地址。在 FLASH 的规格书中，图 29 说明了如何编程和擦除，其实也就是前面命令时序的组合。

Figure 29. Program/ Erase flow with read array data SPI falsh 擦除流程

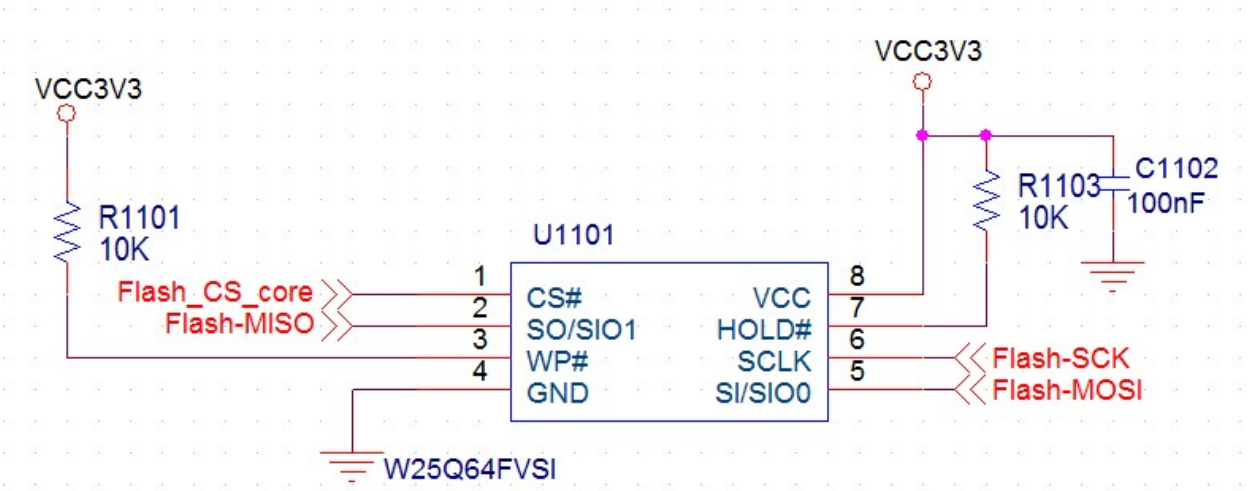
1. 规格书最后就是一些性能参数，封装等信息了。基本与编程无关。
2. 除了以上特性之外，所有 FLASH 都共一个特性：**FLASH 上用于存储数据的每一个 BIT，只能由 1 改写为 0**。例如原来 FLASH 上某个 BYTE=0xff，可以将其改写为 0x00-0xff 的任何值。如果某个 BYTE=0x0f，就只能修改为 0x00-0x0f 之间的值。擦除可以将一个 page、sector、block 一次性全部改写为 0XFF。因此，通常在写之前都会进行读出 flash、改写数据、擦除 flash、写回 flash，一共四步操作。完整的流程比较消耗时间。根据特性可以进行优化，例如，知道数据是 0XFF，就可以直接改写。
3. 每次操作 FLASH，都是先发一个命令，再进行数据通信。**在发命令前，要有一个 CS 的下降沿**。

17.4 原理图

屋脊雀 F407 硬件在底板和核心板上，各配置了一片 SPI FLASH。底板配置的是 MX25L3206EM2I，底板的电路在 SPI 信号上进行了阻容滤波处理。



底板 SPI-FLASH 原理图核心板配置的是 W25Q64FVSI(或 JVSI)，核心板 FLASH 离 CPU 较近，没有加阻容滤波处理，这样也能减少空间，毕竟核心板器件还是比较密集。



核心板 SPIFLASH 原理图 **这两个 SPI FLASH 都是接在 SPI3 控制器上，SPI3 控制器还是外扩接口的 SPI 控制器，也即是说，一个 SPI 上可能接有 3 个设备或更多**。为什么要配置两片 SPI FALSH? 当然不是堆硬件，是为了模拟一个情景：**多个 SPI 总线上挂载多个器件**，如果你写的 SPI 跟 SPI FLASH 驱动不能适应这样的场景，我觉得不是一个好程序。我们提供的源码，就是要处理这种情况。

17.5 驱动设计

需要设计两个驱动，一个是 cpu 上的 SPI 控制器的驱动；另外一个则是板上外设 FLASH 的驱动。分别命名为 mcu_spi 和 dev_flash。具体代码见例程。在做驱动前我们要认识以下概念：

- **SPI 控制器**：STM32 上的 SPI3 就是一个控制器。在程序中就是一些参数或者寄存器。**SPI 驱动**：为了用 SPI3，我们会写一段代码，这段代码就是 SPI 驱动。

  spi 驱动是为了配套 SPI 控制器，在 STM32 中有多个 SPI 控制器，SPI 驱动有几套?

- **SPI FLASH 设备**：设备就是实物，我们的硬件有两个 SPI FLASH 设备。**SPI FLASH 驱动**：我们写的代码，操作 (读写)FLASH 的代码，就是驱动。

  我们有两个 FLASH 设备，写几个驱动?

17.5.1 SPI 驱动要怎么设计

1. SPI 能干什么。

SPI 属于全双工总线。**发送时钟信号，在发送数据的同时会收到数据**。这个特性反映在 SPI 驱动上就是

> 发送出去一个字节，就会收到一个字节。接收一个字节，就需要发送一个字节。**因此我们认为，SPI_WRITE 或者 SPI_READ 这样单独读写的接口不符合 SPI 特性**

SPI 可以运行在不同的频率和模式。SPI 的 CS 可以控制。

1. 用 SPI 的程序想要 SPI 干什么?

我们可能用 SPI 控制 LCD，SPI FLASH，RF24L01 等设备。对于这些设备，有只写操作的，也有读写都要操作的。对于 CS 管脚，在占用 SPI 控制器时，有可能要变化电平，例如 SPI FALSH，在发送命令时就需要一个 CS 下降沿。

1. SPI 驱动和 SPI 设备的关系

在我们的硬件上，只用一个 SPI 控制器 SPI3，配合 3 根 CS 线。一套 SPI 驱动如何控制三个设备呢? **千万不要将 CS 的控制放到 SPI FALSH 驱动中。CS 属于 SPI 控制器的一部分。**

如果不将 CS 控制和 SPI 控制器绑定，配合多个 CS 时，编写代码很容易造成 SPI 控制器冲突。

要控制多个 SPI 设备，在接口传入一个参数表明操作哪个接口即可。

因此接口我们设置如下：

s32 mcu_spi_init(void); s32 mcu_spi_open(SPI_DEV dev, SPI_MODE mode, u16 pre); s32 mcu_spi_close(SPI_DEV dev); s32 mcu_spi_transfer(SPI_DEV dev, u8 *snd, u8 *rsv, s32 len); s32 mcu_spi_cs(SPI_DEV dev, u8 sta); 从上到下分别是：初始化，打开（占用），关闭（释放），传输，CS 控制。

我们看看初始化代码

```
s32 mcu_spi_init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_InitTypeDef SPI_InitStruct;

    //初始化片选，系统暂时设定为 3 个 SPI，全部使用 SPI3
    //DEV_SPI_3_1，核心板上的 SPI FLASH
        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
        GPIO_SetBits(GPIOB,GPIO_Pin_14);

    //DEV_SPI_3_2，底板的 SPI FLASH
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
```

(continues on next page)

(continued from previous page)

```
    GPIO_Init(GPIOG, &GPIO_InitStructure);
        GPIO_SetBits(GPIOG,GPIO_Pin_15);

        //DEV_SPI_3_3, 核心板外扩 SPI
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_Init(GPIOG, &GPIO_InitStructure);
        GPIO_SetBits(GPIOG,GPIO_Pin_6);

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5;//---PB3~5
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//---复用功能
    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(GPIOB, &GPIO_InitStructure);//---初始化

    //配置引脚复用映射
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource3, GPIO_AF_SPI3); //PB3 复用为 SPI3
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource4, GPIO_AF_SPI3); //PB4 复用为 SPI3
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource5, GPIO_AF_SPI3); //PB5 复用为 SPI3

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI3, ENABLE);// ---使能 SPI3 时钟
    // 复位 SPI 模块
    SPI_I2S_DeInit(SPI_DEVICE);

    SPI_InitStruct.SPI_Direction = SPI_Direction_2Lines_FullDuplex;//---双线双向全双工
    SPI_InitStruct.SPI_Mode = SPI_Mode_Master;//---主模式
    SPI_InitStruct.SPI_DataSize = SPI_DataSize_8b;//---8bit 帧结构
    SPI_InitStruct.SPI_CPOL = SPI_CPOL_High;//----串行同步时钟的空闲状态为低电平
    SPI_InitStruct.SPI_CPHA = SPI_CPHA_2Edge;//---数据捕获于第 1 个时钟沿
    SPI_InitStruct.SPI_NSS = SPI_NSS_Soft; //---SPI_NSS_Hard; 片选由硬件管理, SPI 控制器不
管理
    SPI_InitStruct.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;  //---预分频
    SPI_InitStruct.SPI_FirstBit = SPI_FirstBit_MSB;//---数据传输从 MSB 位开始
    SPI_InitStruct.SPI_CRCPolynomial = 7;//---CRC 值计算的多项式

    SPI_Init(SPI_DEVICE, &SPI_InitStruct);

    //SPI_SSOutputCmd(SPI_DEVICE, DISABLE);
```

(continues on next page)

(continued from previous page)

```
    DevSpi3Gd = -1;
    return 0;
}
```

6~30 行，初始化使用 SPI3 控制器的 3 个 SPI 接口的 CS 脚，初始化为输出，并且输出高电平。34~44，初始化 SPI 控制器 3 个引脚，也就是把 IO 配置为 AF 功能，并且配置为 SPI3 的 AF 功能。46，打开 SPI 时钟。48，复位 SPI3 控制器。50~60，配置 SPI3 控制器，每个配置什么意思在代码注释都说明了。

根据硬件定义 3 个 SPI 设备号，3 个 SPI 都使用 SPI3 控制器

```
typedef enum{
        DEV_SPI_NULL = 0,

        DEV_SPI_3_1 = 0X31,//核心板上的 SPI 使用 SPI3，定义为 SPI_3_1
        DEV_SPI_3_2,        //底板板上的 SPI 使用 SPI3，定义为 SPI_3_2
        DEV_SPI_3_3,                //外扩的 SPI 定义为 SPI_3_3
}SPI_DEV;
```

传输是最关键函数看代码，入口参数 3 个：

snd 和 rsv 是指针，发送数据从 snd 取，接收到的数据保存到 rsv；len 是数据长度。snd 或者 rsv 其中一个可以为 NULL，但是不能全部为 0，如果 rsv 为 NULL，则意味着只是想发送数据，接收到的数据丢弃；如果 snd 为空，则意味着只想接收数据，**函数会默认发送 0XFF**：实际应用中，有外设是不允许默认发送 0XFF 的，例如触摸控制芯片 XPT2046，如果在读数据时发送 0XFF，会立刻启动 XTP2046AD 转换，造成读到的触摸参数错乱。最新的 SPI 通信代码请从 GIT 上提取（上面的代码会持续更新）

```
s32 mcu_spi_transfer(u8 *snd, u8 *rsv, s32 len)
{
    s32 i = 0;
    s32 pos = 0;
    u32 time_out = 0;
    u16 ch;

    if( ((snd == NULL) && (rsv == NULL)) || (len < 0) )
    {
        return -1;
    }

    /* 忙等待 */
    time_out = 0;
```

(continues on next page)

(continued from previous page)

```
    while(SPI_I2S_GetFlagStatus(SPI_DEVICE, SPI_I2S_FLAG_BSY) == SET)
    {
        if(time_out++ > MCU_SPI_WAIT_TIMEOUT)
        {
            return(-1);
        }
    }

    /* 清空 SPI 缓冲数据, 防止读到上次传输遗留的数据 */
    time_out = 0;
    while(SPI_I2S_GetFlagStatus(SPI_DEVICE, SPI_I2S_FLAG_RXNE) == SET)
    {
        SPI_I2S_ReceiveData(SPI_DEVICE);
        if(time_out++ > 2)
        {
            return(-1);
        }
    }

    /* 开始传输 */
    for(i=0; i < len; )
    {
        // 写数据
        if(snd == NULL)/* 发送指针为 NULL, 说明仅仅是读数据 */
        {
            //uart_printf("--1--");
            SPI_I2S_SendData(SPI_DEVICE, 0xff);
        }
        else
        {
            ch = (u16)snd[i];
            SPI_I2S_SendData(SPI_DEVICE, ch);
            //uart_printf("s%02x ", ch);
        }
        i++;

        // 等待接收结束
        time_out = 0;
        while(SPI_I2S_GetFlagStatus(SPI_DEVICE, SPI_I2S_FLAG_RXNE) == RESET)
        {
```

(continues on next page)

(continued from previous page)

```
            time_out++;
            if(time_out > MCU_SPI_WAIT_TIMEOUT)
            {
                return -1;
            }
        }
        // 读数据
        if(rsv == NULL)/* 接收指针为空，读数据后丢弃 */
        {
            //uart_printf("--2--");
            SPI_I2S_ReceiveData(SPI_DEVICE);
        }
        else
        {
            ch = SPI_I2S_ReceiveData(SPI_DEVICE);
            rsv[pos] = (u8)ch;
            //uart_printf("r%02x ", ch);
        }
        pos++;

    }

    return i;
}
```

1. 13 到 21，等待 SPI 控制器不忙。
2. 23-32，读走 SPI 控制器里面的缓存数据。
3. 进入 FOR 循环开始传输。
4. 判断输入参数 snd，如果为空（0），就说明只是想读数据，没数据发送。那么我们就自作主张，发送一个 0XFF 过去。
5. 发送的同时，控制器就在接收数据了，53 到 60 行等待接收完毕。
6. 62 行开始是接收数据，如果接收缓冲为空 (0)，我们就仅仅将数据读出来丢弃掉。

17.5.2 SPI FALSH 驱动

SPI 驱动编写要考虑两方面：

SPI FLASH 的功能实现。SPI FALSH 驱动架构。

1. 功能

FLASH 功能包括，擦除，读，写，芯片 ID 等。其中各操作还分 page，sector，BLOCK,chip；例如擦，可以擦 1 个 sector，也可以擦 1 个 BLOCK，或者是整片擦除。读就没有要求，可以从任何地址读。写也没有要求，可以从任何地址写。但是为了方便 APP 使用，特别是文件系统，我们会封装 sector 擦、读、写函数。

2. 架构

设计 SPI FLASH 架构前要考虑的问题是：

1. 多个 FLASH 挂在多个 SPI 上。我们的硬件就是两片 FLASH 挂在两个 SPI 接口上（同一个硬件 SPI 控制器）。
2. FALSH 型号自动识别，就算硬件更换了 Flash，驱动也不需要修改。

最终的 SPI FLASH 驱动可以做到，**一套 SPI FLASH 驱动，可以处理多个挂在不同 SPI 上的不同 flash 芯片**。要实现这个目标，主要要实现下面 3 个小目标：

1. 驱动就是一个软件模块。
2. 对上提供操作接口。
3. 对下要求提供挂参数：哪个 SPI？什么芯片？

因此我们设计驱动如下：

- 接口

```
extern s32 dev_spiflash_readmorebyte(DevSpiFlash *dev, u32 addr, u8 *dst, u32 len);
extern s32 dev_spiflash_write(DevSpiFlash *dev, u8* pbuffer, u32 addr, u16 wlen);
extern s32 dev_spiflash_sector_erase(DevSpiFlash *dev, u32 sector_addr);
extern s32 dev_spiflash_sector_read(DevSpiFlash *dev, u32 sector, u8 *dst);
extern s32 dev_spiflash_sector_write(DevSpiFlash *dev, u32 sector, u8 *src);
extern s32 dev_spiflash_init(void);
extern s32 dev_spiflash_open(DevSpiFlash *dev, char* name);
extern s32 dev_spiflash_test(void);
```

各接口功能，看名称就知道啥意思了。

- 芯片参数定义

```
/*SPI FLASH 信息 */
typedef struct
{
        char *name;
        u32 JID;
        u32 MID;
```

(continues on next page)

(continued from previous page)

```
        /* 容量，块数，块大小等信息 */
        u32 sectornum;//总块数
        u32 sector;//块大小
        u32 structure;//总容量

}_strSpiFlash;

/*
        常用的 SPI FLASH 参数信息
*/
_strSpiFlash SpiFlashPraList[]=
{
        {"MX25L3206E", 0XC22016, 0XC215, 1024, 4096, 4194304},
        {"W25Q64JVSI", 0Xef4017, 0Xef16, 2048, 4096, 8388608}
};
```

对于各种芯片，抽象定义一个结构体，结构体成员包含了芯片信息。然后定义一个列表数组，表明当前支持的芯片型号。当前我们只做了两个型号。大家可以自己丰富这个表格。

- 设备树定义

```
/*SPI FLASH 设备定义 */
typedef struct
{
        char *name;//设备名称
        SPI_DEV spi;//挂载在哪条 SPI 总线
        _strSpiFlash *pra;//设备信息
}DevSpiFlash;

/*
        设备树定义
*/
#define DEV_SPI_FLASH_C 2//总共有两片 SPI FLASH

DevSpiFlash DevSpiFlashList[DEV_SPI_FLASH_C]=
{
        /* 有一个叫做 board_spiflash 的 SPI FLASH 挂在 DEV_SPI_3_2 上，型号未知 */
        {"board_spiflash", DEV_SPI_3_2, NULL},
        /* 有一个叫做 board_spiflash 的 SPI FLASH 挂在 DEV_SPI_3_1 上，型号未知 */
        {"core_spiflash",  DEV_SPI_3_1, NULL},
};
```

设备树的意义就是**告诉驱动，什么东西挂在什么地方**。这样驱动就可以跟硬件剥离，也就能兼容更多硬件。**具体驱动设计请看源码，后续会对文档进行更详细更新**我们抽一个 SPI FLASH 驱动函数分析。

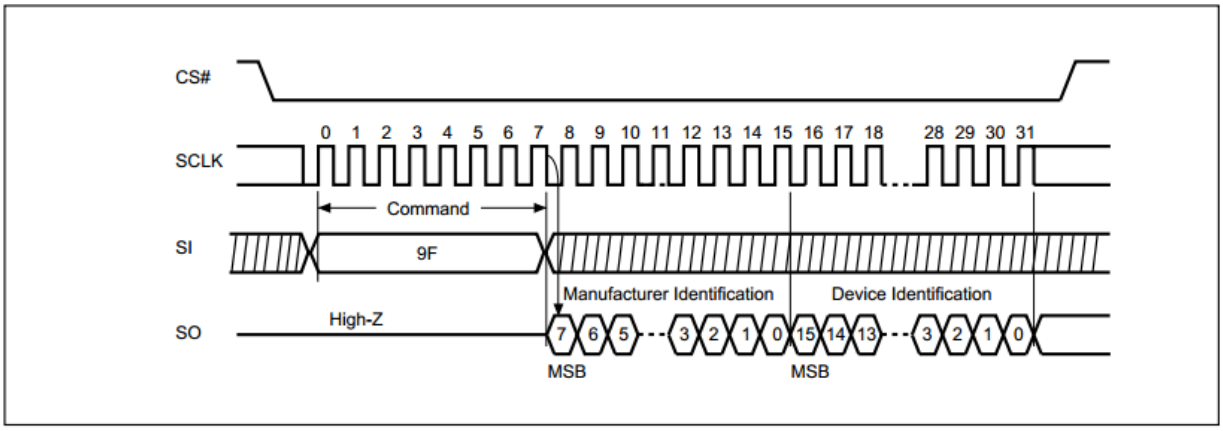
```
/**
 *@brief:      dev_spiflash_readJTD
 *@details:    读 FLASH JTD 号
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
static u32 dev_spiflash_readJTD(DevSpiFlash *dev)
{
    u32 JID;
    s32 len = 1;
    u8 command = SPIFLASH_RDJID;
    u8 data[3];

        mcu_spi_cs(dev->spi, 0);
    len = 1;
    mcu_spi_transfer(dev->spi, &command, NULL, len);
    len = 3;
    mcu_spi_transfer(dev->spi, NULL, data, len);
        mcu_spi_cs(dev->spi, 1);

    JID = data[0];
    JID = (JID<<8) + data[1];
    JID = (JID<<8) + data[2];

    return JID;
}
```

以上是读 FLASH JTD 函数

> 15 行，拉低 CS 脚，使能对应的 SPI FLASH 设备。17 行，写数据，第 3 个参数为空，我们只是写命令，1 个字节数据而已。19 行，读数据，读 3 个字节，第二个参数为空，函数将默认发送 0XFF。20 行，拉高 CS 脚。

我们看看 FLASH 规格书中读 JTD 的时序图，大家对比代码好好体会体会。

Figure 25. Read Identification (RDID) Sequence (Command 9F)



readjtd

- 驱动和设备分离的好处这样的驱动架构有什么好处呢？请看测试程序，测试程序传入一个名称，就可以操作对应 FLASH 了。不用关心 FLASH 是什么型号，挂在什么地方。更加不会去操作 CS 管脚。

```
void dev_spiflash_test_fun(char *name)
{
    u32 addr;
    u16 tmp;
    u8 i = 1;
    u8 rbuf[4096];
    u8 wbuf[4096];
    u8 err_flag = 0;

        DevSpiFlash dev;

        s32 res;

    wjq_log(LOG_FUN, ">:-------dev_spiflash_test-------\r\n");
    res = dev_spiflash_open(&dev, name);
        wjq_log(LOG_FUN, ">:-------%s-------\r\n", dev.name);
        if(res == -1)
        {
                wjq_log(LOG_FUN, "open spi flash ERR\r\n");
                while(1);
        }
    i = 0;
    for(tmp = 0; tmp < 4096; tmp++)
    {
        wbuf[tmp] = i;
```

(continues on next page)

(continued from previous page)

```
        i++;
    }
    //sector 1 进行擦除，然后写，校验。
    wjq_log(LOG_FUN, ">:-------test sector erase-------\r\n", addr);

    addr = 0;
    dev_spiflash_sector_erase(&dev, addr);
    wjq_log(LOG_FUN, "erase...");

    dev_spiflash_sector_read(&dev, addr, rbuf);;//读一页回来
    wjq_log(LOG_FUN, "read...");

    for(tmp = 0; tmp < dev.pra->sector; tmp++)
    {
        if(rbuf[tmp] != 0xff)//擦除后全部都是 0xff
        {
            wjq_log(LOG_FUN, "%x=%02X ", tmp, rbuf[tmp]);//擦除后不等于 0XFF，坏块
            err_flag = 1;
        }
    }

    dev_spiflash_sector_write(&dev, addr, wbuf);
    wjq_log(LOG_FUN, "write...");

    dev_spiflash_sector_read(&dev, addr, rbuf);
    wjq_log(LOG_FUN, "read...");

    wjq_log(LOG_FUN, "\r\n>:test wr..\r\n");

    for(tmp = 0; tmp < dev.pra->sector; tmp++)
    {
        if(rbuf[tmp] != wbuf[tmp])
        {
            wjq_log(LOG_FUN, "%x ", tmp);//读出来的跟写进去的不相等
            err_flag = 1;
        }
    }

    if(err_flag == 1)
        wjq_log(LOG_FUN, "bad sector\r\n");
```

(continues on next page)

(continued from previous page)

```
    else
        wjq_log(LOG_FUN, "OK sector\r\n");

        dev_spiflash_close(&dev);
}
```

测试

在 main 中初始化，按下按键则进行测试。

```
/* Infinite loop */
        mcu_uart_open(3);
        wjq_log(LOG_INFO, "hello word!\r\n");
        mcu_i2c_init();
        mcu_spi_init();
        dev_key_init();
        //mcu_timer_init();
        dev_buzzer_init();
        dev_tea5767_init();
        dev_dacsound_init();
        dev_spiflash_init();

        dev_key_open();
        //dev_dacsound_open();
        //dev_tea5767_open();
        //dev_tea5767_setfre(105700);

        while (1)
        {
                /* 驱动轮询 */
                dev_key_scan();

                /* 应用 */
                u8 key;
                s32 res;

                res = dev_key_read(&key, 1);
                if(res == 1)
                {
```

(continues on next page)

(continued from previous page)

```
                        if(key == DEV_KEY_PRESS)
                        {
                                //dev_buzzer_open();
                                //dev_dacsound_play();
                                dev_spiflash_test();
                                GPIO_ResetBits(GPIOG, GPIO_Pin_0
                                  | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                                //dev_tea5767_search(1);
                        }
                        else if(key == DEV_KEY_REL)
                        {
                                //dev_buzzer_close();
                                GPIO_SetBits(GPIOG, GPIO_Pin_0
                                  | GPIO_Pin_1 | GPIO_Pin_2| GPIO_Pin_3);
                        }
                }

                Delay(1);

                /* 测试触摸按键 */
                //dev_touchkey_task();
                //dev_touchkey_test();
        }
```

17.6 调试

- 1

代码写好后，调用 FLASH 测试程序就死机。第一行调试信息都没有输出。**一进函数就死，基本上都是堆栈溢出问题，俗称栈爆了**。一般都是局部变量申请太大，造成堆栈溢出，或者是进函数前堆栈已经临界，函数没申请多少局部变量也会造成溢出。测试函数申请了 8K 局部变量造成死机，根本原因是我们没有根据工程实际情况初始化堆栈。

```
void dev_spiflash_test(void)
{
    u32 addr;
    u16 tmp;
    u8 i = 1;
    u8 rbuf[4096];
```

(continues on next page)

(continued from previous page)

```
    u8 wbuf[4096];
    u8 err_flag = 0;
```

堆栈在启动代码 startup_stm32f40_41xxx.s 的开头配置，默认仅仅配置了 0X400 字节栈。堆的默认配置也不大，都需要根据工程实际情况修改。

```
; Amount of memory (in bytes) allocated for Stack
; Tailor this value to your application needs
; <h> Stack Configuration
;   <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Stack_Size      EQU     0x00000400

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp

; <h> Heap Configuration
;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Heap_Size       EQU     0x00000200

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

此处我们暂时将栈放大到 16K，以便测试程序运行。修改后测试程序正常运行。**类似的死机问题还有一个，而且经常会出现，那么就是一退出函数就死机。这样的问题通常都是因为在函数内操作内存越界，例如野指针啊，或者是写数组超出数组范围**。

- 2

调试一个新 IC 外设，一般先调通能读芯片 ID。读 ID 失败，检查程序是否有笔误，如有，修正。检查后，还是不行，怀疑硬件问题。将 SPI 通信的 4 条 PIN 全部改为 IO，全部输出高电平，用万用表检测。然后全部输出低电平，用万用表检测。同时检测 FLASH 其他管脚，发现 WP 电平不对。检查原理图，发现原理图上 WP 脚连接有误，应该连接到 VCC，但是原理图原来增加了 1 个下拉电阻用于调试，在焊接时不应该直接焊上，样板错误焊上了。去掉电阻再量电压，1.7V，半高电平，估计芯片有问题了。换另外一块样板，上电后测试，电平正常。

- 3

恢复调试过程对驱动跟测试程序的修改，上电，正常读出 FLASH ID。

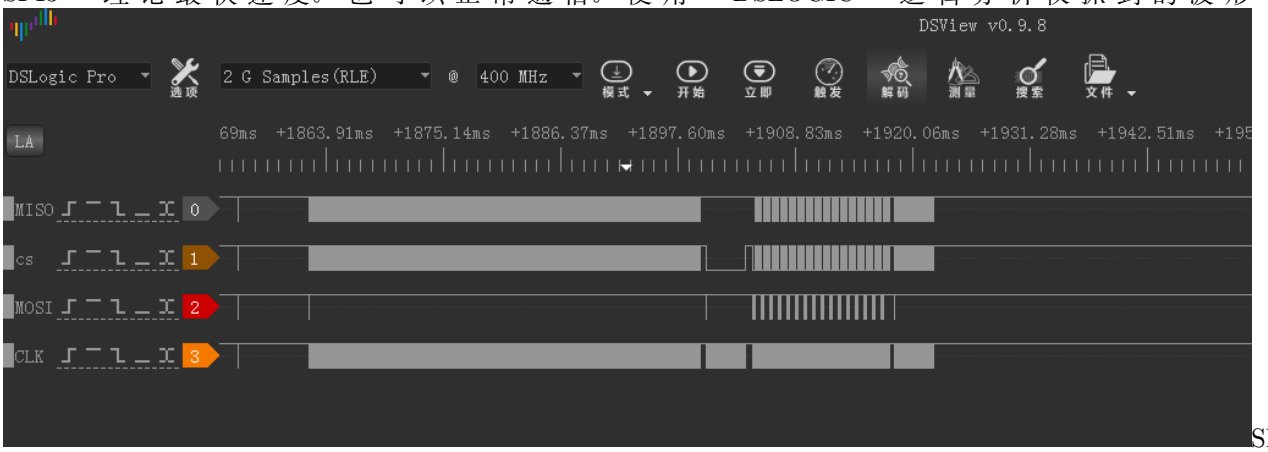
- 4

对 sector 1 进行擦，写，读操作，测试 FLASH。测试结果:

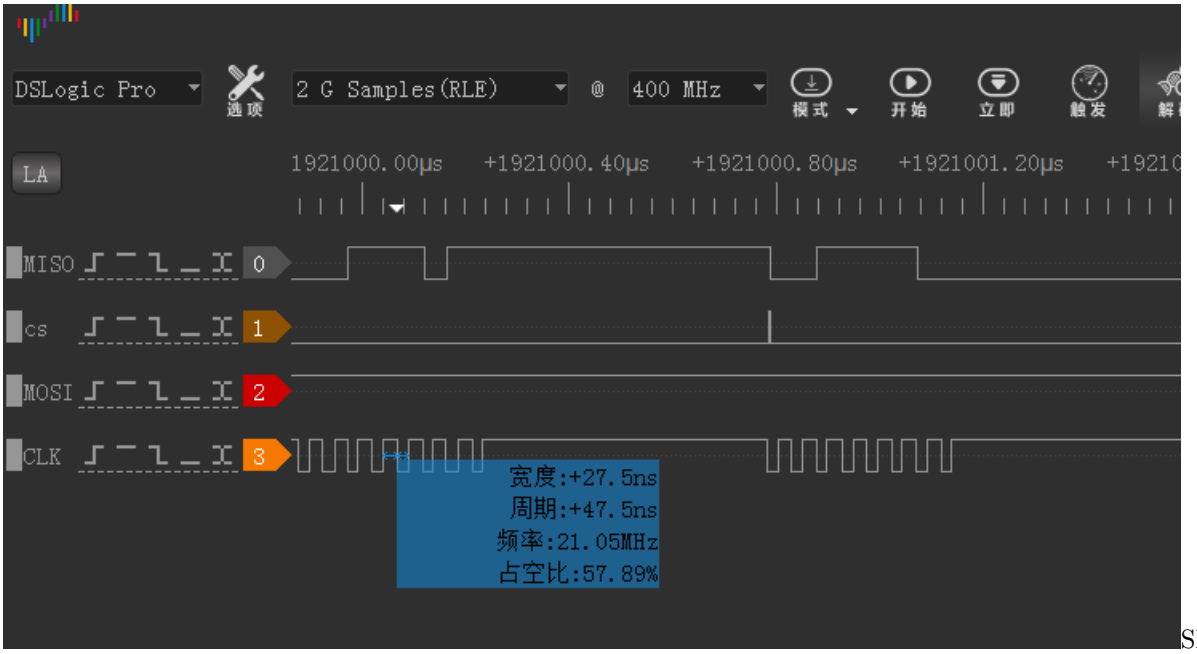
> hello word! board_spiflash jid:0xc22016 board_spiflash mid:0xc215 core_spiflash jid:0xef4017 core_spiflash mid:0xef16 :——-dev_spiflash_test——- spi flash type:MX25L3206E :——-board_spiflash——- :——-test sector erase——- erase…read…write…read…:test wr.. OK sector :——-dev_spiflash_test——- spi flash type:W25Q64JVSI :——-core_spiflash——- :——-test sector erase——- erase…read…write…read…:test wr.. OK sector

17.6.1 时序确认

经测试，SPI 最快设置为 SPI_BaudRatePrescaler_4，也就是 PCLK（84M）/4=21M，SPI3 理论最快速度。也可以正常通信。使用 DSLOGIC 逻辑分析仪抓到的波形



SPI falsh 时序波形放大后可以看到时钟频率时 21.05M，但是同时也发现一个问题，在两个字节之间的间隔，很大，浪费通信时间，程序需要优化。



SPI falsh 时序波形间隔大当然，系统目前还没有跑其他设备，例如 I2S，USB，网口等，等全部跑起来，可能就因为干扰，SPI 只能降速了。

17.7 思考

到此我们实现了一个基本算有点**软件架构**的 SPI FLASH 驱动。也为后面其他驱动编写做了一定铺垫。大家可以了解一些 **LINUX 设备驱动的软件架构思想**。宋宝华的书《**LINUX 设备驱动开发详解基于最新的 LINUX4.0 内核**》第 12 章。对于 SPI 驱动，还要进一步改善。例如：后面我们会加上用 IO 口模拟 SPI 功能，模拟 SPI 跟硬件 SPI 如何统一接口？一个 SPI LCD，既可以接到外扩的 SPI3 上，也可以接到模拟 SPI 上。LCD 驱动要如何编写，才能灵活用于两种 SPI?

这些，我们都会实现。~~ 这些，我们都已经实现。 本例程附带的代码，只是为了教程服务，虽然能用，但是架构不是最好的 如果要用于实际项目，请从 GITHUB 上下载最新代码，最新代码有很好的代码架构设计

17.8 end

CHAPTER 18

SDIO-TF CARD

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

- **本 SDIO 例程，测试过程中会将 SD 卡的数据直接删除，相当于将 SD 卡格式化，请使用一张没有数据的 SD 卡进行测试。数据丢失后果自负。**

SD 卡是我们日常使用的电子设备，例如用在相机、个人数据存储。部分手机也会使用 TF 卡作扩展存储使用。

SD 存储卡是一种基于半导体快闪记忆器的新一代记忆设备，由于它体积小、数据传输速度快、可热插拔等优良的特性，被广泛地于便携式装置上使用，例如数码相机、个人数码助理（外语缩写 PDA）和多媒体播放器等。

SD 卡标准使用 SDIO 接口通信，也支持 SPI 接口。在日常使用情景，例如相机、读卡器等，都是使用 SDIO 通信，SDIO 使用 4 个数据线，速度比 SPI 要快。SD 卡使用 SDIO 口通信过程遵循 SD 卡规范，这个规范不简单，涉及到较多的命令交互和状态转换。通常芯片厂家都会提供相关例程，用户不需要重新开发 SDIO 通信程序。本例程通过移植 ST 官方的例程实现与 SD 卡通信。

关于 SD 卡协议细节，请自行学习研究。

18.1 SDIO 接口

SDIO：安全数字输入输出接口。STM32F407，原生支持。关于 SDIO 接口，在《STM32F4xx 中文参考手册.pdf》

SDIO 主要特性

SD/SDIO MMC 卡主机接口 (SDIO) 提供 APB2 外设总线与多媒体卡 (MMC)、SD 卡、SDIO 卡以及 CE-ATA 设备之间的接口。

多媒体卡协会网站 *www.mmca.org* 中提供了由 MMCA 技术委员会发布的多媒体卡系统规范。

SD 卡协会网站 *www.sdcard.org* 中提供了 SD 存储卡和 SD I/O 卡系统规范。

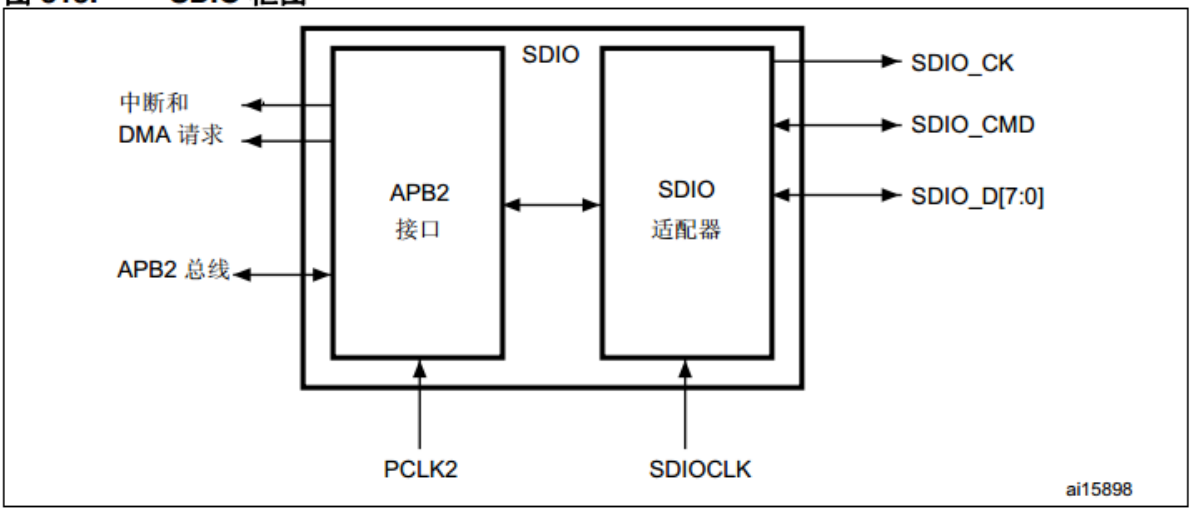
CE-ATA 工作组网站 *www.ce-ata.org* 中提供了 CE-ATA 系统规范。

SDIO 具有以下特性：

- 完全兼容*多媒体卡系统规范版本 4.2*。卡支持三种不同数据总线模式：1 位（默认）、4 位和 8 位
- 完全兼容先前版本的多媒体卡（向前兼容性）
- 完全兼容 *SD 存储卡规范版本 2.0*
- 完全兼容 *SD I/O 卡规范版本 2.0*：卡支持两种不同数据总线模式：1 位（默认）和 4 位
- 完全支持 CE-ATA 功能（完全符合 *CE-ATA 数字协议版本 1.1*）
- 对于 8 位模式，数据传输高达 48 MHz
- 数据和命令输出使能信号，控制外部双向驱动程序。

文档有说明。特性

图 313. SDIO 框图



下图是 SDIO 框图，图

这是一个总框图，主要说明外部连接。对于 SDIO 适配器，并没有详细说明。我猜测原因是，其实 SDIO 是一个比较复杂的设备，类似 USB。通常这种复杂外设，我们都不会自己开发驱动，都是用官方提供的例程。

1. 总线的通信基于命令和数据传输。其实这是大部分外设的相同之处。
2. SDIO 接口总共有 6 根线（4 位位宽），其中命令只通过 CMD 线传输。因此在**调试的时候如果命令正常，读写数据不正常，说明仅仅是数据线问题**。

SDID 接口除了用于控制 SD 卡等卡之外，也可以控制 SDIO 接口的其他模块，例如 SDIO 接口的 WIFI 模块

18.2 SDIO 接口存储卡

常见的卡有下面两种 SD 卡，俗称大卡。

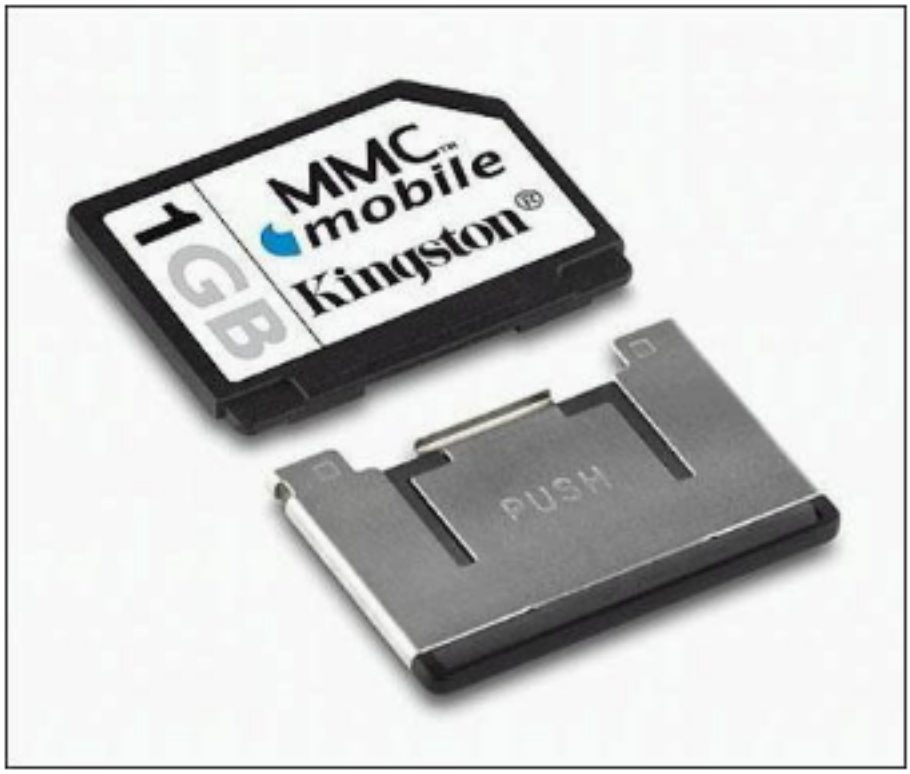
SD 卡是由松下电器、东芝和 SanDisk 联合推出，1999 年 8 月发布。

SD 卡 TF 卡，俗称小卡，2004 年标准协会更名为 micro SD 卡。

TF 卡

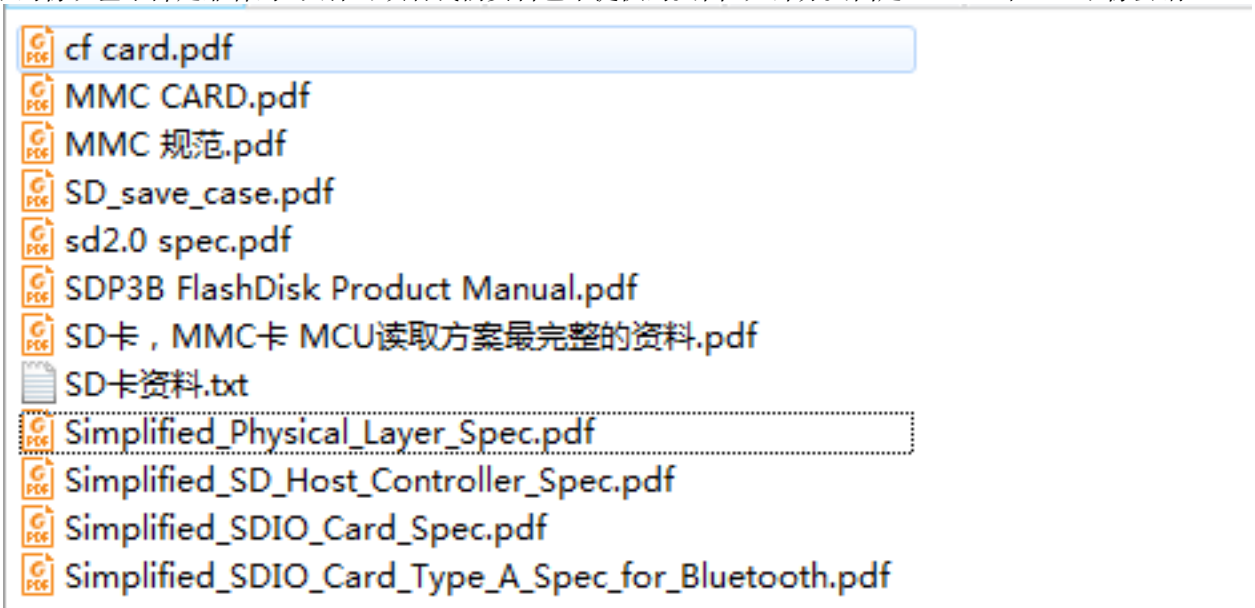
其实在 SD 卡之前，最先出现的是 MMC 卡。下图这种就是 MMC 卡，只有 SD 卡一半大小，通常我们都用一个卡尾拼成 SD 卡用。

MMC(Multi-Media Card，多媒体卡) 由西门子公司 Siemens 和 SanDisk 于 1997 年推出。



MMC 卡

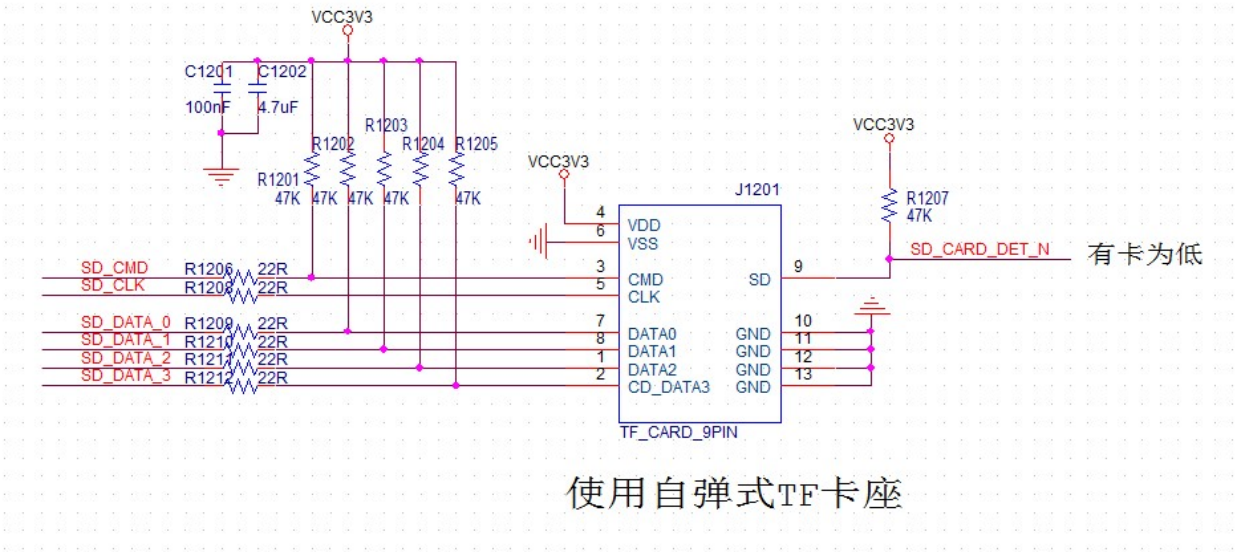
这些卡的协议基本都是兼容的。具体可以看我们资料包中提供的文档。大部分文档是 sandisk 和 SD 卡协会编



写的。协议资料话说这些资料全英文，一时半会儿看不懂。

18.3 原理图

屋脊雀 4074 开发板选用 TF 卡座，减少体积，这也是当前电子产品的趋势。



原理图

SDIO 接口使用 6 根管脚。CLK 是通信时钟 CMD 是命令串行通信线。DATA 数据线有四根。右边的 SD_CARD_DET_N 是卡插入检测 SDIO 接口，CLK 不需要上拉电阻，其他 5 根 IO 需要加上拉电阻。

18.4 驱动设计

前面说到，SDIO 是一个复杂的协议，SD 卡、TF 卡是一个较复杂设备。要完全弄清楚，非常不容易，更加不要说自己写一套了。通常，这种复杂的协议，我们不自己开发，都是芯片厂提供驱动代码。

前面几个章节的调试，我们直接参考标准库接口，再根据参考手册，就可以进行编码了。但是类似 USB，SD 卡，网络等较复杂的设备，通常做法是从官方提供的库入手。特别是 USB 通信，个人基本不可能写一套库出来，也没这个必要。

18.5 移植调试

在移植之前，先对官方例程进行分析学习。

18.5.1 例程分析

在标准库下面有 SD 卡例程，路径如下

STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Project\STM32F4xx_StdPeriph_Examples\SDIO\SDIO_uSDCard

在目录下有一个 readme.txt 文件。**看东西，先从 readme 入手**。从 readme 中可以看出，STM32F407 芯片的 SDIO 的驱动在 STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Utilities\STM32_EVAL\STM3240_41_G_EVAL 目录下，名字叫 stm324xg_eval_sdio_sd.c 和 stm324xg_eval.c。其中 stm324xg_eval.c 仅仅提供了较底层的初始化。看来主要代码都在 stm324xg_eval_sdio_sd.c 里面。

```
******************************************************************************
***
  * @file    stm324xg_eval.c
  * @author  MCD Application Team
  * @version V1.1.2
  * @date    19-September-2013
  * @brief   This file provides
  *            - set of firmware functions to manage Leds, push-button and
COM ports
  *            - low level initialization functions for SD card (on SDIO)
and
  *              serial EEPROM (sEE)
  *          available on STM324xG-EVAL evaluation board(MB786) RevB from
  *          STMicroelectronics.
```

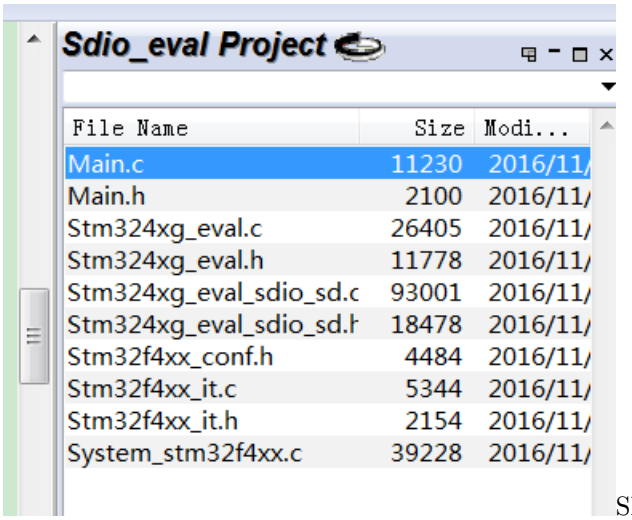
stm324xg_eval_sdio

例程还包括以下文件

路径:STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Project\STM32F4xx_StdPeriph_Examples\SDIO\SDIO_uSDCard

- SDIO/SDIO_uSDCard/system_stm32f4xx.c STM32F4xx system clock configuration file
- SDIO/SDIO_uSDCard/stm32f4xx_conf.h Library Configuration file
- SDIO/SDIO_uSDCard/stm32f4xx_it.c Interrupt handlers
- SDIO/SDIO_uSDCard/stm32f4xx_it.h Interrupt handlers header file
- SDIO/SDIO_uSDCard/main.c Main program
- SDIO/SDIO_uSDCard/main.h Main program header file

下一步，我们就浏览这六个文件，熟悉其中的流程与调用关系。使用 SI 创建一个 SDIO_EVAL 工程，将以上提到的文件添加到工程，用 SI 分析程序相当方便。**为了防止无意中修改库文件，我们把文件拷贝一份**。



SD 卡例程源文件从 MAIN 函数入手，程序首先对 SD 进行初始化。然后进行块测试，分别进行擦块，单块测试，多块测试。

```
/*------------------------------ SD Init ----------------- */
if((Status = SD_Init()) != SD_OK)
{
  STM_EVAL_LEDOn(LED4);
}

while((Status == SD_OK) && (uwSDCardOperation != SD_OPERATION_END)
      && (SD_Detect()== SD_PRESENT))
{
  switch(uwSDCardOperation)
  {
    /*-------------------------- SD Erase Test --------- */
    case (SD_OPERATION_ERASE):
    {
      SD_EraseTest();
      uwSDCardOperation = SD_OPERATION_BLOCK;
      break;
    }
    /*-------------------- SD Single Block Test --------- */
    case (SD_OPERATION_BLOCK):
    {
      SD_SingleBlockTest();
      uwSDCardOperation = SD_OPERATION_MULTI_BLOCK;
      break;
    }
```

(continues on next page)

(continued from previous page)

```
      /*-------------------- SD Multi Blocks Test --------- */
      case (SD_OPERATION_MULTI_BLOCK):
      {
        SD_MultiBlockTest();
        uwSDCardOperation = SD_OPERATION_END;
        break;
      }
    }
  }
```

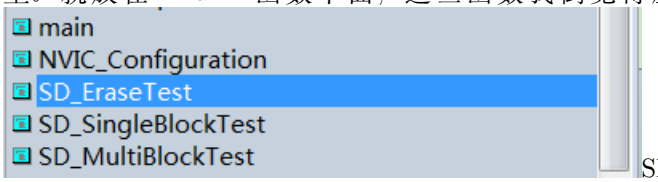
从函数 SD_Error SD_Init(void) 跟下去。函数首先调用了 SDIO 底层初始化，然后读卡，进行 Power On 了，上电成功就初始化。先看看 SDIO 底层初始化了什么。

```
SD_Error SD_Init(void)
{
  __IO SD_Error errorstatus = SD_OK;

  /* SDIO Peripheral Low Level Init */
   SD_LowLevel_Init();

  SDIO_DeInit();

  errorstatus = SD_PowerON();
```

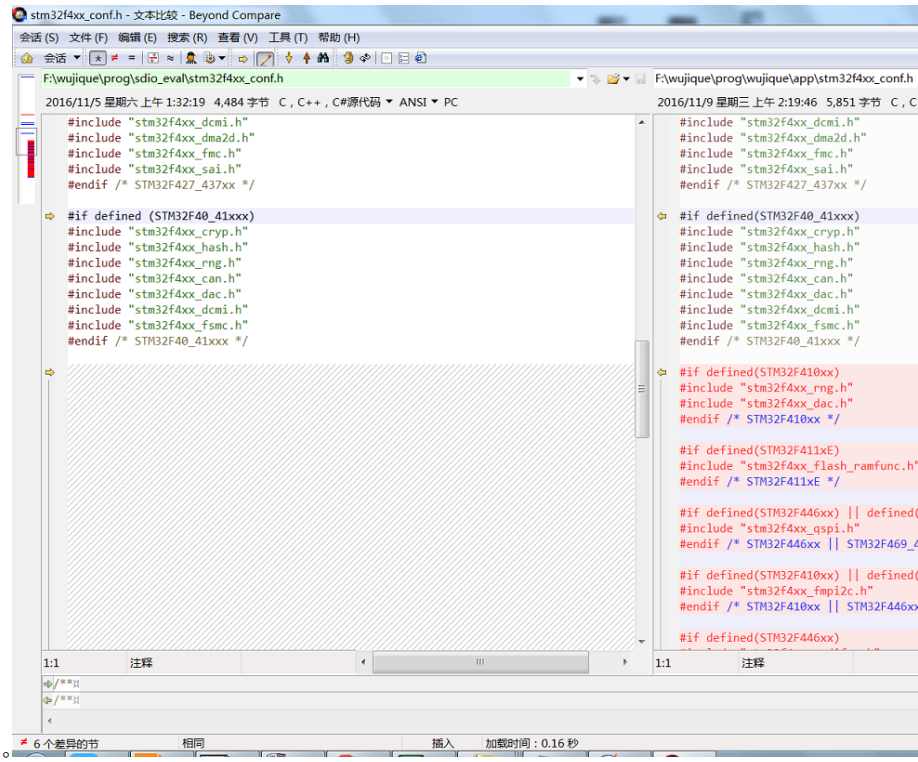
void SD_LowLevel_Init(void) 函数就在前面提到过的 stm324xg_eval.c 文件内，也就是官方的 DEMO 板的初始化文件。我个人觉得这些代码应该放到 mcu_sdio 驱动内。这个文件与 SD 卡有关的也就四个函数，前面两个是 SDIO 口初始化，后面两个是 SDIO 使用 DMA 的初始化。

```
void SD_LowLevel_DeInit(void);
void SD_LowLevel_Init(void);
void SD_LowLevel_DMA_TxConfig(uint32_t *BufferSRC, uint32_t BufferSize);
void SD_LowLevel_DMA_RxConfig(uint32_t *BufferDST, uint32_t BufferSize);
```

回到 SD_Init(void)，后续调用的函数全部都在本文件了。再回到 main，看测试程序在哪里。就放在 main 函数下面，这些函数我倒觉得应该放在 SD 卡驱动里面。当然，个人意见。

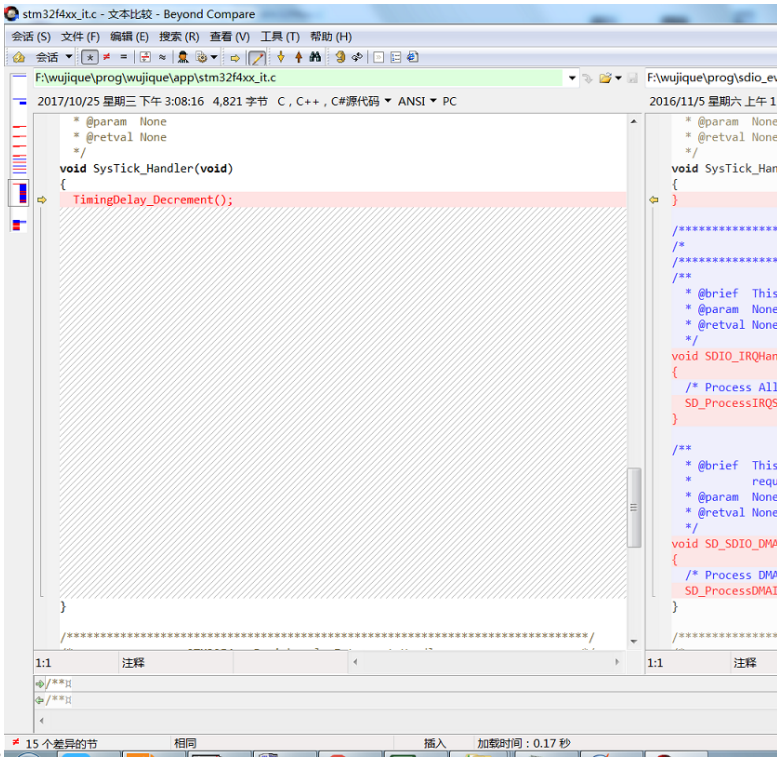


SD 卡测试函数其他几个文件，前面例程已经在使用，我们就对比一下看看差异。



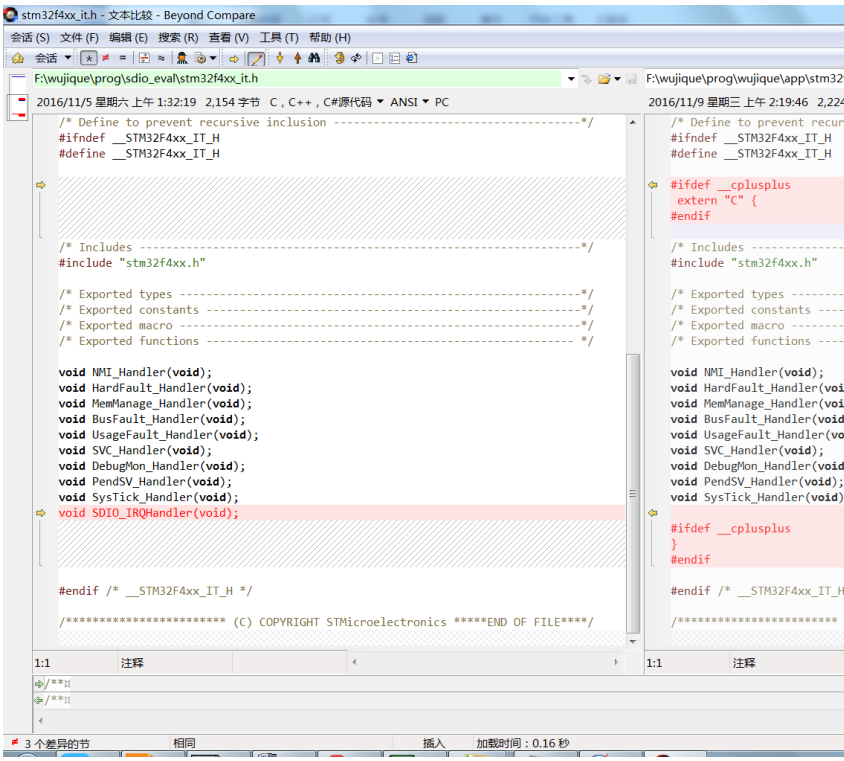
stm32f4xx_conf.h，有差别，但是与 SD 卡无关。

差异



stm32f4xx_it.c，多了两个中断处理，移植的时候记得拷贝。

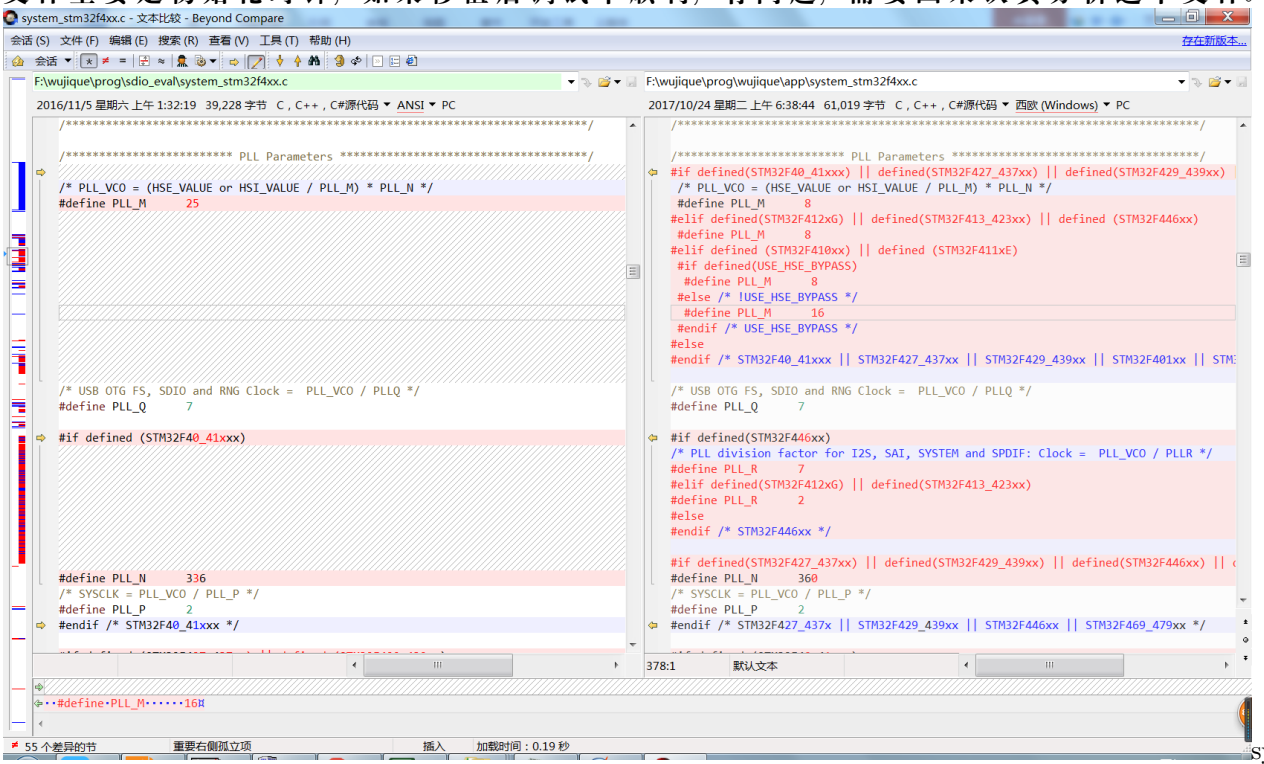
差异



stm32f4xx_it.h 多一个 SDIO 中断的声明，无关紧要

差异

system_stm32f4xx.c，差别较大，但是很多都是为了兼容其他芯片的条件编译。**但是这个文件主要是初始化时钟，如果移植后调试不顺利，有问题，需要回来认真分析这个文件。**



system_stm32f4

差异

OK，下一步我们就开始移植了。

18.5.2 移植调试过程

1. 建立一个 mcu_sdio 驱动，将 stm324xg_eval.c 里的四个函数作为 mcu_sdio 驱动，拷贝到 mcu_sdio.c。stm324xg_eval.h 里面与 SD 卡相关的定义也要拷贝到 mcu_sdio.h 里面。
2. stm324xg_eval_sdio_sd.c 跟 stm324xg_eval_sdio_sd.h 我们就直接使用。作为 board_dev 驱动。
3. 将 main.c 里面的 SD 卡测试程序拷贝到 stm324xg_eval_sdio_sd.c 最后，作为驱动测试程序使用。原来的测试程序使用了一些官方硬件上的 LED，我们全部改为串口调试信息输出。
4. 有一个需要特别关注的地方就是 static void NVIC_Configuration(void) 函数，这个函数的第一行就设置了 NVIC 的分组，也就是中断优先级分组，这个是 ARM 核相关。关于 NVIC，前面章节有说明。
5. 把 stm32f4xx_it.c 里面的两个中断入口拷贝到我们的 stm32f4xx_it.c 文件。
6. 将新文件添加到工程，编译。21 个错误，一个一个解决。挑几个看看..\mcu_dev\mcu_sdio.c(67): error: #20: identifier "SD_DETECT_GPIO_CLK" is undefined 因为 mcu_sdio.c 没有包含 mcu_sdio.h。增加 #include "mcu_sdio.h"。其实在我的个人认识当中，h 文件应该是提供给外部使用。**哪些仅仅是驱动内部使用的定义，最好是定义到 C 文件内，不让其他文件看到。减少不必要的耦合**。
7. 重新编译，还有一个错误..\board_dev\stm324xg_eval_sdio_sd.h(39): error: #5: cannot open source input file "stm324xg_eval.h" : No such file or directory 改为包含 mcu_sdio.h。
8. 又有 38 个错误了。..\board_dev\stm324xg_eval_sdio_sd.h(128): error: #20: identifier "uint32_t" is undefined 应该是没包含 stm32 的头文件，官方 stm324xg_eval_sdio_sd.h 包含了 #include "stm324xg_eval.h"，然后 stm324xg_eval.h 包含了 #include "stm32f4xx.h" 和 #include "stm32_eval_legacy.h"，我们在 mcu_sdio.h 中包含 #include "stm32f4xx.h"。
9. 编译没有错误了。
10. 开始做硬件移植，在 mcu_sdio.h 第 20 行，例程用 PH13 做 SD 卡检测，我们都没有 H 口。我们用的是 PC13.
11. 在 mcu_sdio.c 的初始化中，进行了初始化，个人认为这些**跟硬件相关的，还是放到一处用宏定义较好，方便移植修改**。数据线，和我们的硬件一样，不需要修改。

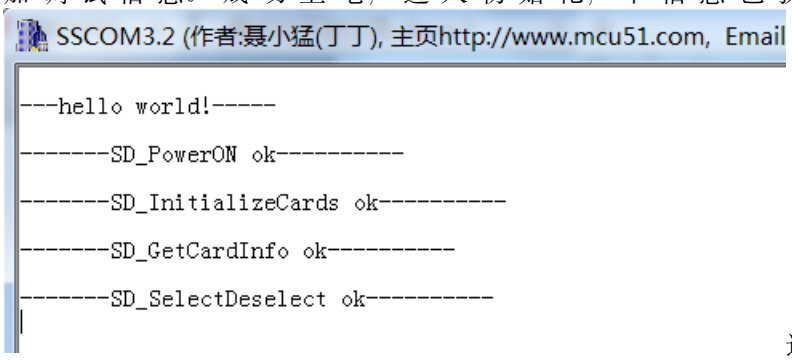
```
/* Configure PC.08, PC.09, PC.10, PC.11 pins: D0, D1, D2, D3 pins */
命令线，也跟我们一样。
/* Configure PD.02 CMD line */
时钟线也一样。
/* Configure PC.12 pin: CLK pin */
```

1. 那么就是检测脚不一样，我们看看这个管脚怎么用的。搜索后发现有一个 SD_Detect 函数在使用。

```
/**
 * @brief  Detect if SD card is correctly plugged in the memory slot.
 * @param  None
 * @retval Return if SD is detected or not
 */
uint8_t SD_Detect(void)
{
  __IO uint8_t status = SD_PRESENT;

  /*!< Check GPIO to detect SD */
  if (GPIO_ReadInputDataBit(SD_DETECT_GPIO_PORT, SD_DETECT_PIN) != Bit_RESET)
  {
    status = SD_NOT_PRESENT;
  }
  return status;
}
```

1. SDCardState SD_GetState(void) 和测试程序会使用 SD_Detect。到这里看出，检测管脚就是一个普通 IO 口，没有使用中断等其他功能，直接修改为我们的管脚即可。

2. 重新编译，插上 TF 卡，下载程序运行。初始化不成功。在初始化函数添加调试信息。成功上电，进入初始化，卡信息也获取成功了，选卡也成功。



选卡成功我们先看一下卡信息，卡信息结构体如下：

```
/**
  * @brief SD Card information
  */
typedef struct
{
  SD_CSD SD_csd;
  SD_CID SD_cid;
  uint64_t CardCapacity;  /*!< Card Capacity */
  uint32_t CardBlockSize; /*!< Card Block Size */
```

(continues on next page)

(continued from previous page)

```
  uint16_t RCA;
  uint8_t CardType;
} SD_CardInfo;
```

CSD 跟 CID 是卡信息，包含比较多信息。例如卡什么卡，什么版本，V1.0，还是 2.0 等。本处我们先把容量 Capacity 跟 Block Size 打印出来，看跟我们的卡是不是一致。调试代码如下，要注意的地方是容量的处理：如果直接使用 uart_printf 打印容量，打印出来一个 10，不正确，要拆分为两部分打印。

```
  /*------------------ Read CSD/CID MSD registers ------------------*/
  errorstatus = SD_GetCardInfo(&SDCardInfo);
    uart_printf("\r\n-------SD_GetCardInfo ok----------\r\n");

        uint32_t *p;
        p = (uint32_t *)&(SDCardInfo.CardCapacity);
        uart_printf("\r\n-------CardCapacity:%08X----------\r\n", *(p+1));
    uart_printf("\r\n-------CardCapacity:%08X----------\r\n", *(p+0));
    uart_printf("\r\n-------CardBlockSize:%d ----------\r\n", SDCardInfo.CardBlockSize);

    uart_printf("\r\n-------RCA:%d ----------\r\n", SDCardInfo.RCA);
    uart_printf("\r\n-------CardType:%d ----------\r\n", SDCardInfo.CardType);
```

卡信息调试 LOG:

——-dev_sdio_test———- ——-SD_PowerON ok———- ——-SD_InitializeCards ok ———- ——-SD_csd.DeviceSize:15271———- ——-SD_GetCardInfo ok———- ——- CardCapacity:00000001———- ——-CardCapacity:DD400000———- ——-CardBlockSize:512 ———- ——-RCA:2 ———- ——-CardType:2 ———-

还有，刚刚等了很久发现，初始化返回结果 4，数据超时。

1. 查看 SD_Error SD_EnableWideBusOperation(uint32_t WideMode)，分析大概流程后加上调试信息。应该是陷入 static SD_Error SDEnWideBus(FunctionalState NewState) 超时。源码 1103 行

```
    else if (SDIO_BusWide_4b == WideMode)
    {
      errorstatus = SDEnWideBus(ENABLE);
      uart_printf("SDEnWideBus:%d\r\n", errorstatus);
```

经查，在 FindSCR 处，源码 2438 行。

```
   uart_printf("SDIO_GetResponse ok\r\n");
  /*!< Get SCR Register */
  errorstatus = FindSCR(RCA, scr);
```

(continues on next page)

(continued from previous page)

```
    uart_printf("FindSCR:%d\r\n", errorstatus);
```

FindScr 函数内有个 while，估计程序就是卡在这里，源码 2760 行

```
while (!(SDIO->STA & (SDIO_FLAG_RXOVERR | SDIO_FLAG_DCRCFAIL
  | SDIO_FLAG_DTIMEOUT | SDIO_FLAG_DBCKEND | SDIO_FLAG_STBITERR)))
{
    if (SDIO_GetFlagStatus(SDIO_FLAG_RXDAVL) != RESET)
    {
      *(tempscr + index) = SDIO_ReadData();
      index++;
    }
}
```

1. 我想骂人，我刚刚说 SD 卡读不到数据，硬件说他看下，回来跟我说有 3 个跳线电阻没焊。我想杀人。看来以后要**先学会怀疑别人，再证明自己清白**。
2. 硬件修改后，SD 卡初始化成功，但是测试失败。

   ——-dev_sdio_test———- ——-SD_PowerON ok———- ——-SD_InitializeCards ok ———- ——-SD_csd.DeviceSize:15271———- ——-SD_GetCardInfo ok———- ——-CardCapacity:00000001———- ——-CardCapacity:DD400000———- ——-CardBlockSize:512 ———- ——-RCA:2 ———- ——-CardType:2 ———- ——-SD_SelectDeselect ok——— - SDIO_GetResponse ok FindSCR:0 2453 mdResp1Error:0 2468 CmdResp1Error:0 SDEnWideBus:0 ——-SD_EnableWideBusOperation:0———- ——-SD_Init ok———- ——-SD_EraseTest….———-

查后，发现卡在等待传输结束的等待上。SD_Error SD_WaitReadOperation(void) 函数第一个等待就过不去。前面读写已经没问题，**唯一的区别就是这里使用了 DMA**。这里的 while(1)，就是等待中断或者 DMA 标志。源码 349 行

```
__IO SD_Error TransferError = SD_OK;
__IO uint32_t TransferEnd = 0;//sdio 中断中会赋值 0x01;
__IO uint32_t DMAEndOfTransfer = 0;//DMA 中断中会赋值 0x01;
SD_CardInfo SDCardInfo;
```

网上百度，发现很多人说 ST 的 DMA BUG，其中一个 BUG 是，在读写之前要发 CMD16 设置 BLOCK 大小，我们使用的库已经修改了这个 BUG，源码 1335 行

```
  /*!< Set Block Size for Card */
  SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) BlockSize;
  SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
  SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
```

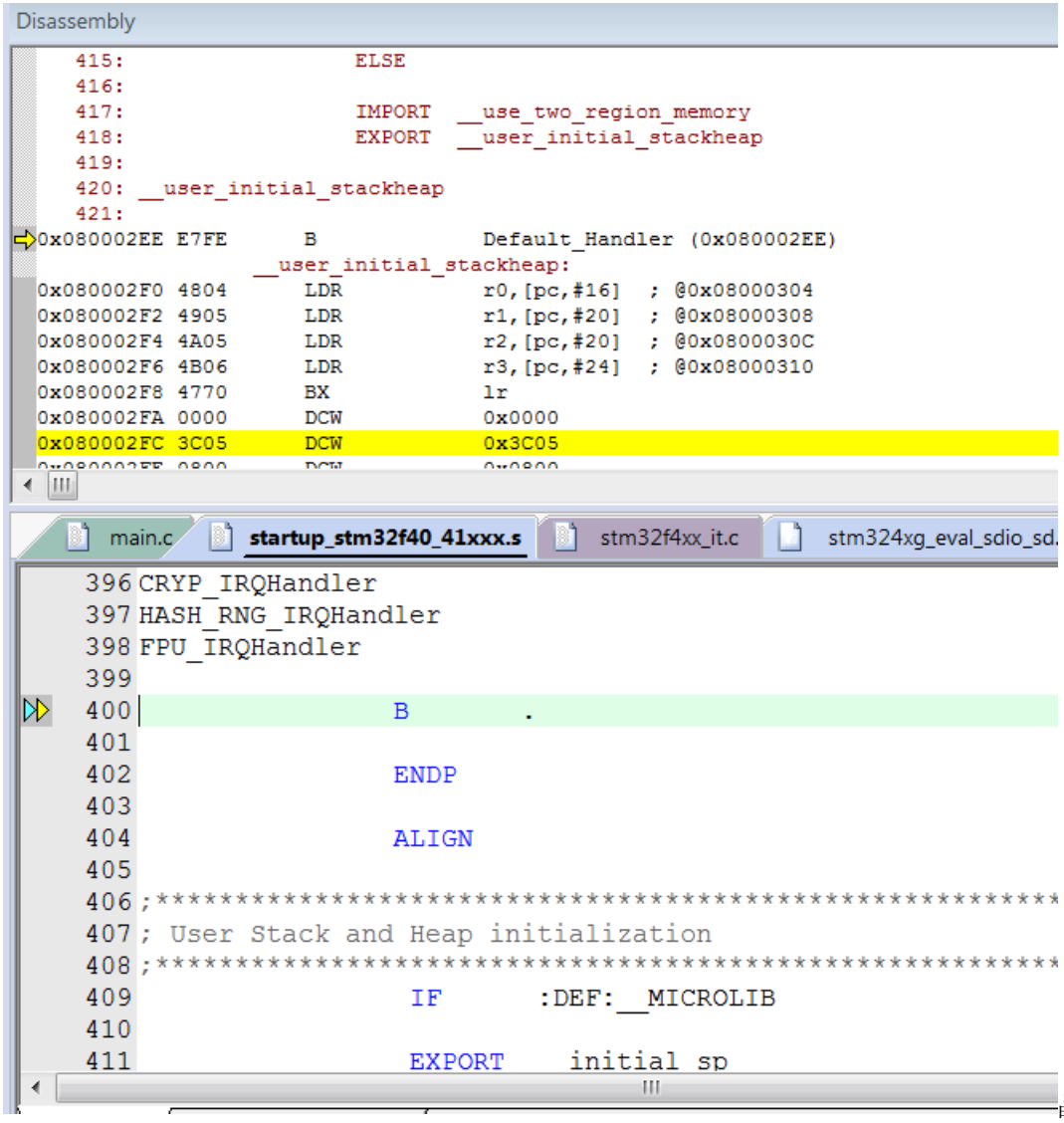
(continues on next page)

(continued from previous page)

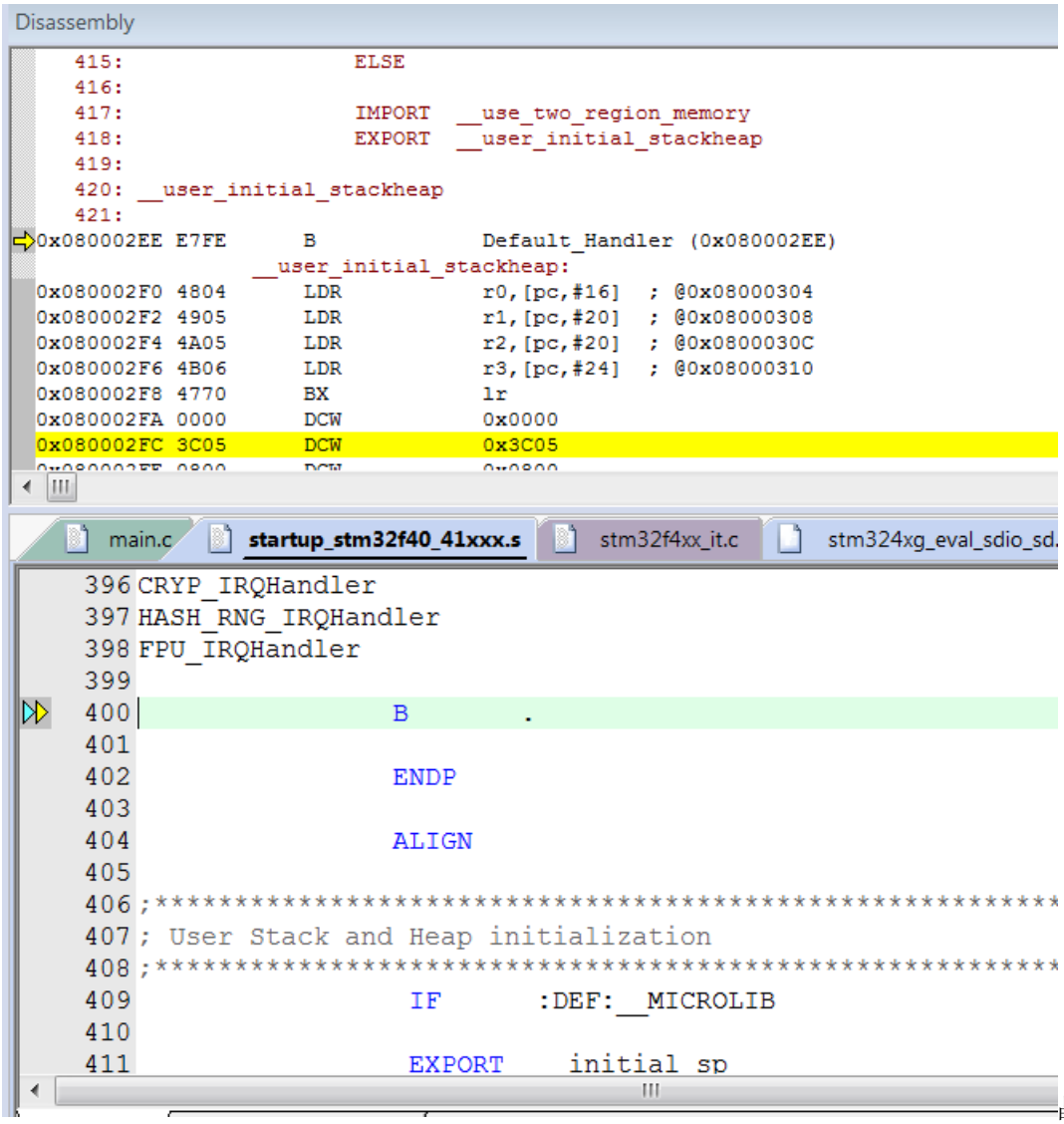
```
SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
SDIO_SendCommand(&SDIO_CmdInitStructure);
```

还有就是中断处理函数 SD_Error SD_ProcessIRQSrc(void)，以前没有处理出错信息，现在已经处理了。(从这里可以学一点，我们自己写的中断处理处理函数，最好也响应错误中断）本处是 DMA 传输，**DMA 传输一般都要求字节对齐，否则会出错或者是死机**。我们看下到底是**死机**了还是一直在**等待**。

1. 既然可能死机，我们就使用 **CMSIS DAP 调试**一下，发现死在中断入口了。



晶振未修改对



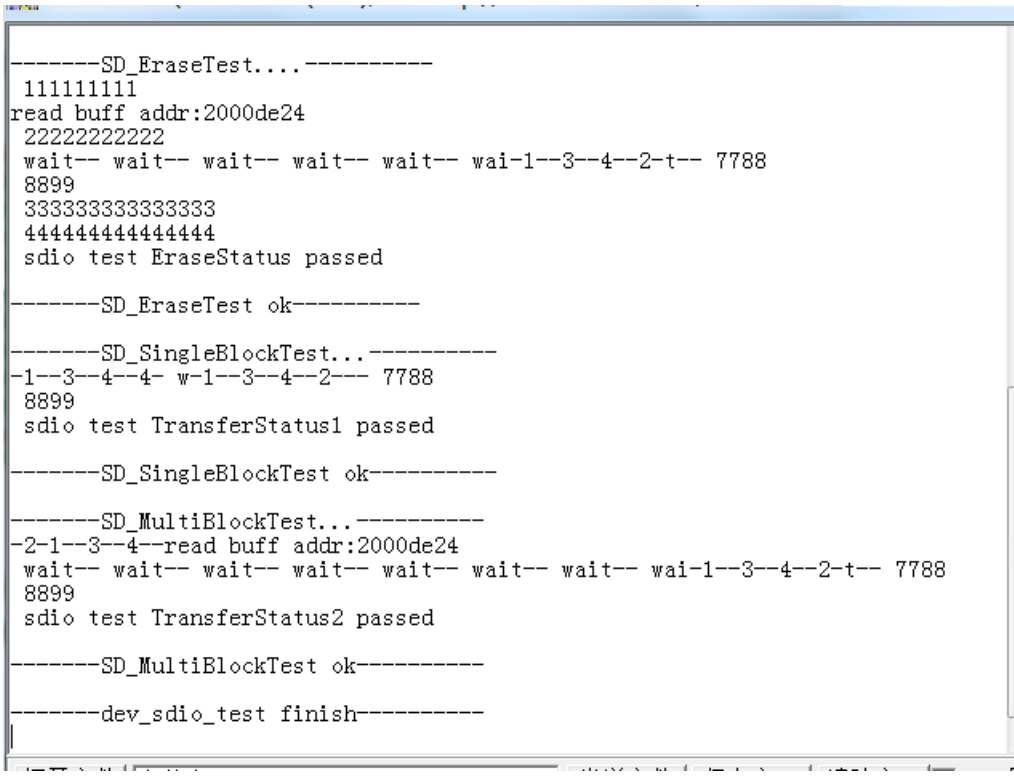
晶振未修改对说明有一个中断源一直在进中断，或者是我们没有处理。我们在 DMA 中断中增加了调试信息，但是却没有输出，很奇怪。源码 2065 行

```
void SD_ProcessDMAIRQ(void)
{
  uart_printf("-2-");
  if(DMA2->LISR & SD_SDIO_DMA_FLAG_TCIF)
  {
    DMAEndOfTransfer = 0x01;
    DMA_ClearFlag(SD_SDIO_DMA_STREAM, SD_SDIO_DMA_FLAG_TCIF|SD_SDIO_DMA_FLAG_FEIF);
  }
}
```

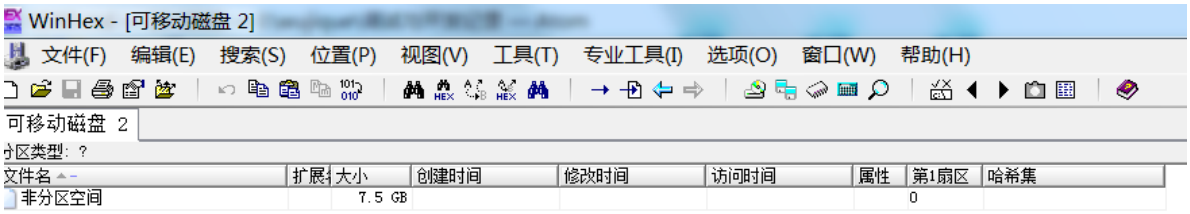
搜索中断入口函数 void SD_SDIO_DMA_IRQHANDLER(void); 发现，这个并没有在中断向量中定义，而是在 mcu_sdio.h 中用宏定义。真正的中断句柄是 DMA2_Stream3_IRQHandler，在移植的时候我们并没有

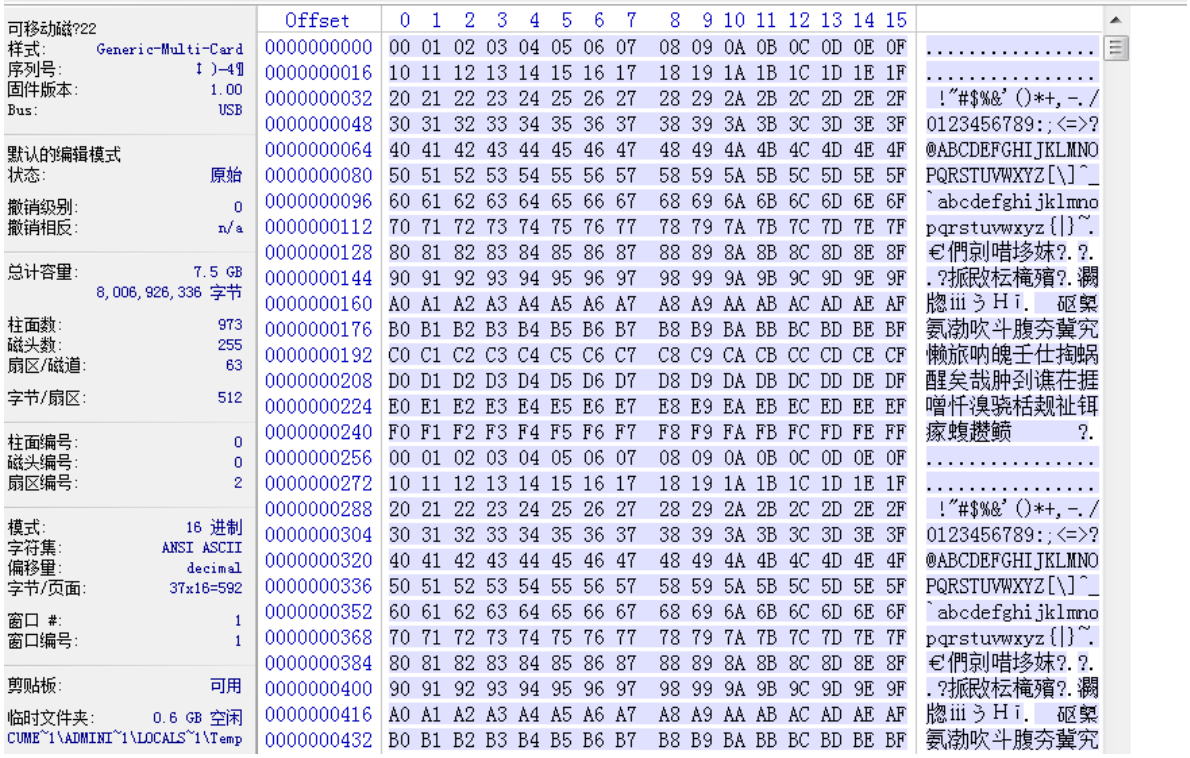
处理，而且也不了解这样做要如何处理。先改回 DMA2_Stream3_IRQHandler 试试。mcu_sdio.h 第 46 行

```
#ifdef SD_SDIO_DMA_STREAM3
 #define SD_SDIO_DMA_STREAM              DMA2_Stream3
 #define SD_SDIO_DMA_CHANNEL             DMA_Channel_4
 #define SD_SDIO_DMA_FLAG_FEIF           DMA_FLAG_FEIF3
 #define SD_SDIO_DMA_FLAG_DMEIF          DMA_FLAG_DMEIF3
 #define SD_SDIO_DMA_FLAG_TEIF           DMA_FLAG_TEIF3
 #define SD_SDIO_DMA_FLAG_HTIF           DMA_FLAG_HTIF3
 #define SD_SDIO_DMA_FLAG_TCIF           DMA_FLAG_TCIF3
 #define SD_SDIO_DMA_IRQn                DMA2_Stream3_IRQn
 #define SD_SDIO_DMA_IRQHANDLER          DMA2_Stream3_IRQHandler
#elif defined SD_SDIO_DMA_STREAM6
```



1. 测试通过。SDIO 硬件测试完成。测试通过程序测试通过，那到底是不是真的测试成功了呢？我们可以用 winhex 软件查看 TF 卡内容，开头的几个数据块，已经被我们改为 0x00–00XFF 顺序增加的数据了，说明操作是成功的。





18.6 总结

调试 SD 卡这一段写了很多，主要是让大家看看平时调试的方法，很多时候就是

- 加调试信息，顺藤摸瓜。

遗留问题，SDIO 驱动用了一个很大的 BUF，这个要修改优化。目前我们只是为了验证硬件，没有挂载文件系统，后续再加上。

18.7 end

CHAPTER 19

I2S-wm8978-音乐播放

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

前面章节我们通过 DAC 播放声音，声音质量只能做到 8K，再高的采样频率，CPU 就比较吃力了。内置的 DAC 精度也没有那么高，一些高级的随身播放器使用的独立 DAC 芯片通常达到 24 位。为了获取较高的音乐质量，屋脊雀板载了一片 WM8978，这个芯片也算是各家开发板的标配了。由于我们还没有调试文件系统，因此我们本次只完成以下功能：

1 将前面调通的收音机通过 WM8978 播放。2 内嵌一段声音文件，通过 I2S 发送到 WM8978 播放。

功能 1 验证 WM8978 是否可用。功能 2 进一步验证 I2S 跟 WM8978 是否可用。

> 在开发阶段，硬件还不是稳定状态，调试软件时要尽量通过少的功能验证硬件。如果一开始上来就直接做从 SD 卡播放 MP3 文件，一旦遇到问题，需要排除的模块就太多了。

19.1 I2S

百度百科：

> I2S(Inter—IC Sound) 总线，又称集成电路内置音频总线，是**飞利浦公司**为数字音频设备之间的音频数据传输而制定的一种总线标准，该总线专门用于音频设备之间的数据传输，广泛应用于各种多媒体系统。它采用了沿独立的导线传输时钟与数据信号的设计，通过将数据和时钟信号分离，避免了因时差诱发的失真，为用户节省了购买抵抗音频抖动的专业设备的费用。

在飞利浦公司的 I2S 标准中，既规定了硬件接口规范，也规定了**数字音频数据的格式**。

I2S 有 3 个主要信号

1. 串行时钟 SCLK，也叫位时钟（BCLK），即对应数字音频的每一位数据，SCLK 都有 1 个脉冲。SCLK 的频率 =2× 采样频率 × 采样位数。
2. 帧时钟 LRCK，(也称 WS)，用于切换左右声道的数据。LRCK 为“1”表示正在传输的是右声道的数据，为“0”则表示正在传输的是左声道的数据。LRCK 的频率等于采样频率。
3. 串行数据 SDATA，就是用二进制补码表示的音频数据。

- 有时为了使系统间能够更好地同步，还需要另外传输一个信号 MCLK，称为主时钟，也叫系统时钟（Sys Clock），是采样频率的 256 倍或 384 倍。

I2S 格式的信号无论有多少位有效数据，数据的最高位总是出现在 LRCK 变化（也就是一帧开始）后的第 2 个 SCLK 脉冲处。这就使得接收端与发送端的有效位数可以不同。如果接收端能处理的有效位数少于发送端，可以放弃数据帧中多余的低位数据；如果接收端能处理的有效位数多于发送端，可以自行补足剩余的位。这种同步机制使得数字音频设备的互连更加方便，而且不会造成数据错位。随着技术的发展，在统一的 I2S 接口下，出现了多种不同的数据格式。根据 SDATA 数据相对于 LRCK 和 SCLK 的位置不同，分为左对齐（较少使用）、I2S 格式（即飞利浦规定的格式）和右对齐（也叫日本格式、普通格式）。

19.2 STM32 I2S

stm32 没有单独的 I2S 接口，与 SPI 共用，资料也在 SPI 章节。

19.2.1 特性

- 全双工通信
- 半双工通信（仅作为发送器或接收器）
- 主模式或从模式操作
- 8 位可编程线性预分频器，可实现精确的音频采样频率（从 8 kHz 到 192 kHz）
- 数据格式可以是 16 位、24 位或 32 位
- 数据包帧由音频通道固定为 16 位（可容纳 16 位数据帧）或 32 位（可容纳 16 位、24 位、32 位数据帧）
- 可编程的时钟极性（就绪时的电平状态）
- 从发送模式下的下溢标志、接收模式下的上溢标志（主模式和从模式），以及接收和发送模式下的帧错误标志（仅从模式）
- 发送和接收使用同一个 16 位数据寄存器
- 支持的 I^2S 协议：
  - I^2S Phillps 标准
  - MSB 对齐标准（左对齐）
  - LSB 对齐标准（右对齐）
  - PCM 标准（在 16 位通道帧或扩展为 32 位通道帧的 16 位数据帧上进行短帧和长帧同步）

I2S

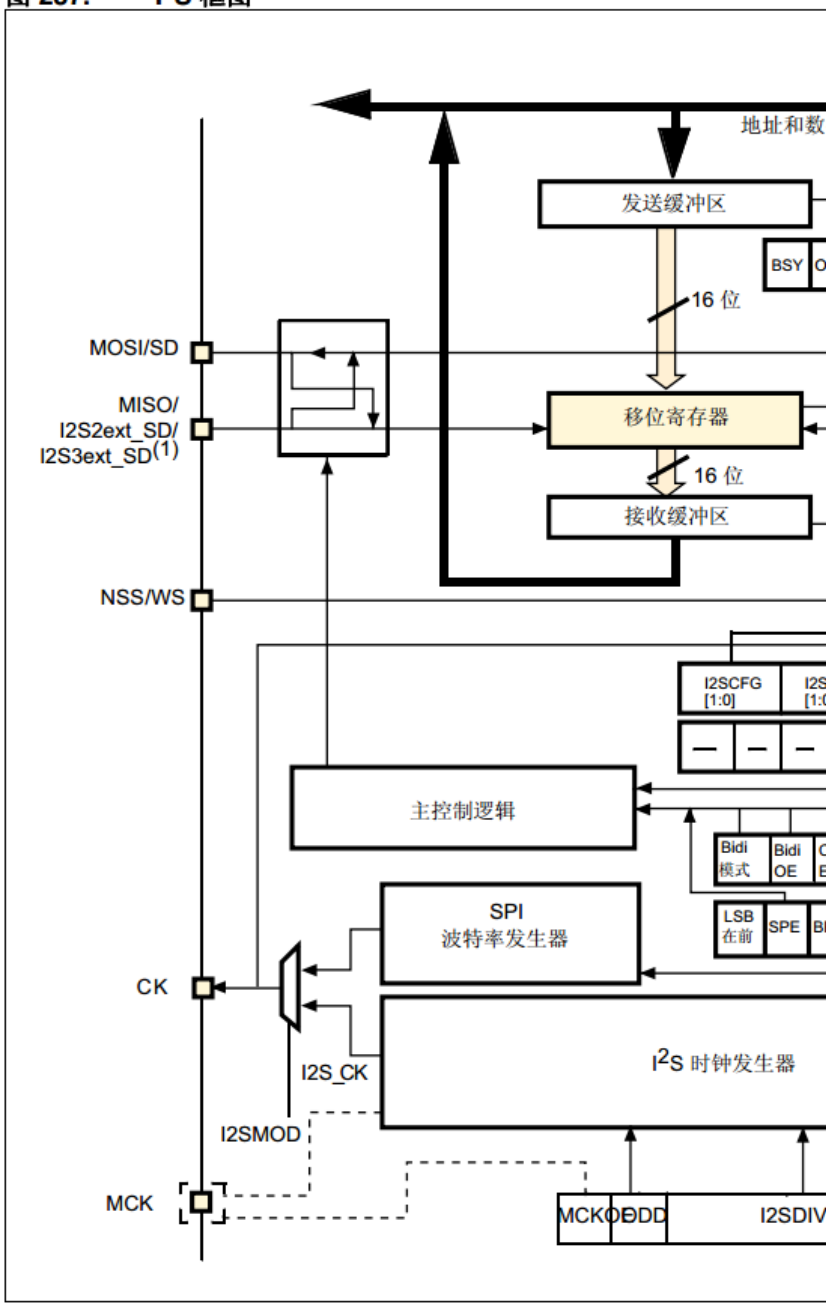
  - 数据方向始终为 MSB 在前
  - 用于发送和接收的 DMA 功能（16 位宽）
  - 可输出主时钟以驱动外部音频元件。比率固定为 256 × F_S（其中 F_S 为音频采样频率）
  - 两个 I^2S（I2S2 和 I2S3）均有专用的 PLL (PLLI2S)，可生成更为精确的时钟。
  - I^2S（I2S2 和 I2S3）时钟可由 I2S_CKIN 引脚上的外部时钟提供。

特性 I2S

特性

19.2.2 结构框图

图 287. I²S 框图



1. I2S2ext_SD 和 I2S3ext_SD 为扩展 SD 引脚，用于控制 I²S 全双工

从下面框图也可以看出,I2S 管脚和 SPI 管脚有复用。框图

19.2.3 全双工

I2S 为了支持全双工，在全双工模式下，除了 I2S2(I2S3) 之外，还用到一个额外的 I2S，那就是 I2S2_ext(I2S3_ext)。

I2S2_ext 和 I2S3_ext 只能工作在全双工模式下，而且只能工作在从模式。怎么理解呢？如果从我们的硬件与 WM8978 通信上来说：1 WM8978 有另个功能，一个是 STM32 输出数据到 WM8978 的 DAC，播放语音。另外一个是 STM23 从 WM8978 读数据，录音。2 I2S，就是用在播音时的通信，这是一个完整的 I2S。3 I2SX_ext 用在录音时的通信，因为这个 I2SX_ext 没有时钟，因此要配合 I2S 使用，也就是说 I2SX_ext 仅仅是一个从机接收数据的功能。

19.3 WM8978

WM8978 是一颗低功耗、高性能的立体声多媒体数字信号编解码器。该芯片内部集成了 24 位高性能 DAC&ADC，可以播放最高 192K@24bit 的音频信号，并且自带 EQ 调节，支持 3D 音效等功能。不仅如此，该芯片还结合了立体声差分麦克风的前置放大与扬声器、耳机和差分、立体声线输出的驱动，减少了应用时必需的外部组件，直接可以驱动耳机（16Ω@40mW）和喇叭（8Ω/0.9W），无需外加功放电路。

- 接口 MCU 可以通过 I2C 或者 SPI 控制 WM8978，屋脊雀使用 I2C 控制。I2S 接口用于传输数据，可以双向传输，播放音乐时 MCU 通过 I2S 发送数据到 WM8978，录音时 MCU 通过 I2S 读取 WM8978 的数据。
- 输入 WM8978 支持双 MIC，硬件上我们只使用一个 MIC，同时接到两路 MIC 输入。LINE 输入不使用，留出测试点。AUX 输入接入收音机音源。
- 输出喇叭通过 2.0 插座引出，配套 8 欧姆 2W 音腔喇叭。耳机输出通过 3.5 音频座接耳机。OUT3/OUT4 不使用。

19.3.1 I2S 传输

在 WM8978 资料的 70 页，DIGITAL AUDIO INTERFACES 对音频接口有详细说明。音频接口有 4 根管脚：

- ADCDAT:adc data Output
- DACDAT:dac data Input
- LRC:Data Left/Right alignment clock
- BCLK:bit clock, for synchronisation

LCR 和 BCLK 是时钟信号，如果 WM8978 是主设备，则是输出；从设备，则是输入；通常我们用 WM8978 做从设备。

WM8978 支持 5 中数据格式：

- Left justified
- Right justified
- I2S
- DSP mode A

- DSP mode B

在文档中有这五个模式的数据传输时序图。

第 3 个格式，I2S，就是我们通常说的飞利浦格式。后面我们就是用这种数据格式，我们看看他的时序图。

In I²S mode, the MSB is available on the second rising edge of BCLK following a LRC transition. The other bits up to the LSB are then transmitted in order. Depending on word length, BCLK frequency and sample rate, there may be unused BCLK cycles between the LSB of one sample and the MSB of the next.

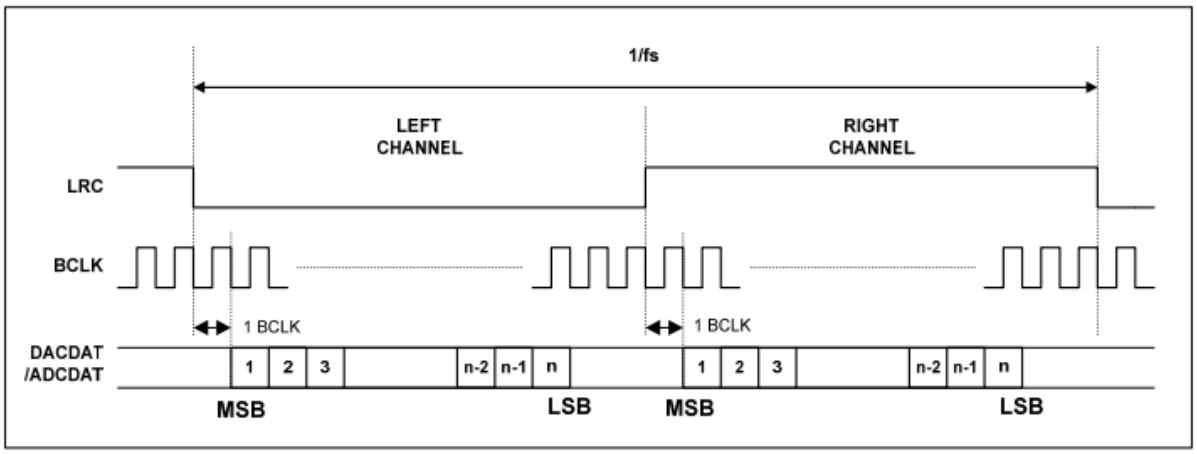


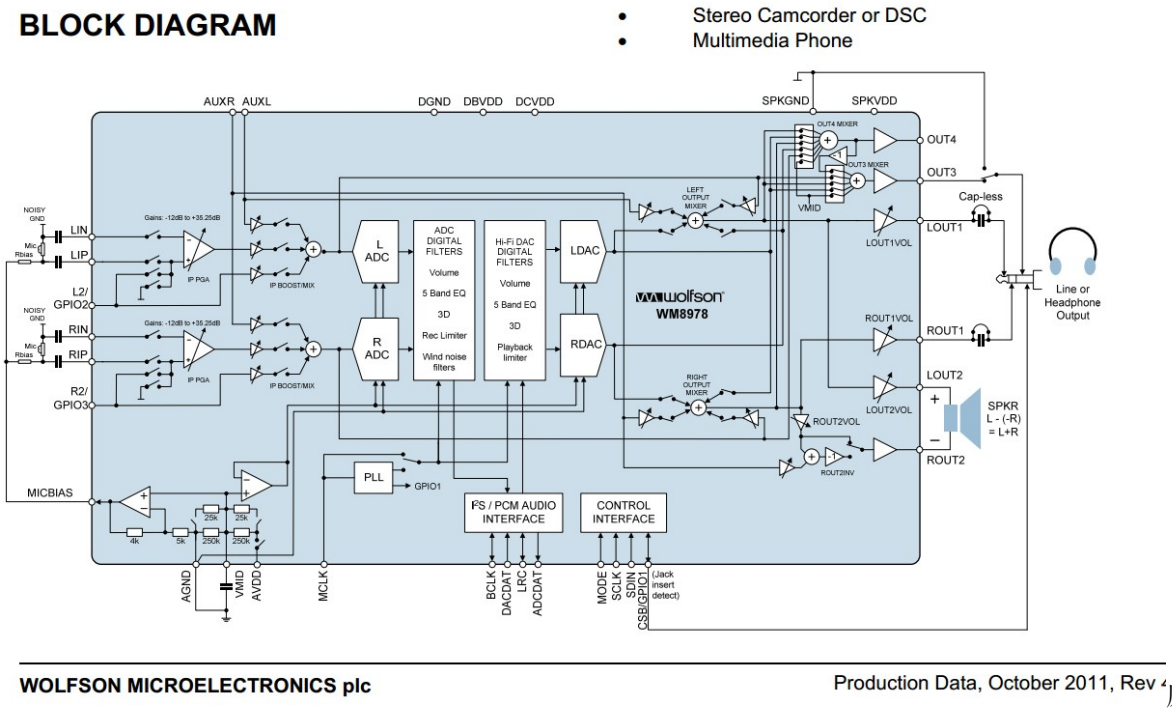
Figure 38 I²S Audio Interface (assuming n-bit word length)

飞利浦格式

LRC，控制数据左右声道。BCLK，位时钟 DACDAT/ADCDAT，数据输入输出。

19.3.2 控制

WM8978 怎么用？知道 WM8978 能做什么才知道怎么用。规格书《WM8978_v4.5.pdf》第一页就有 WM8978

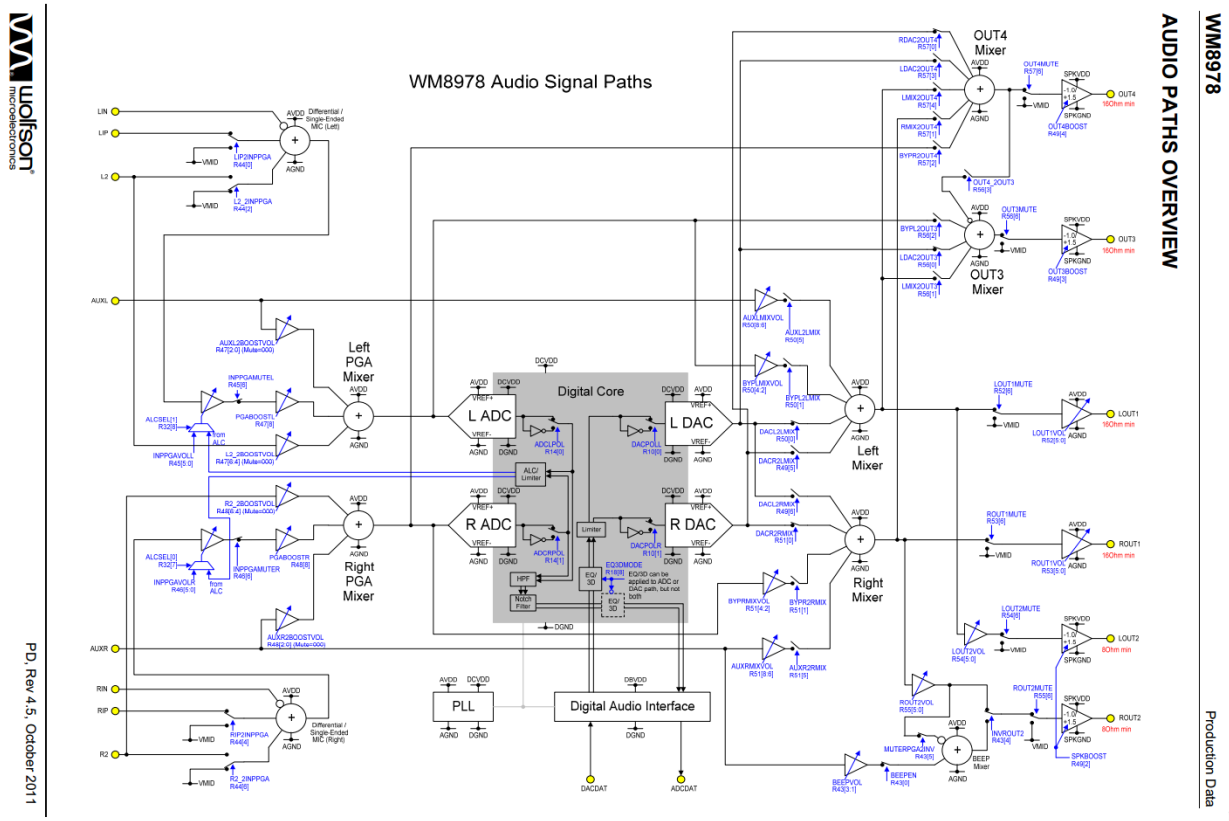


的框图。原理图一句话说明 WM8978 的功能：

> 将输入进行一定音效处理后输出。

- 有多少路输入？LINE 输入，MIC 输出，I2S 输入，AUX 输入。从框图可以看到：I2S 输入数字数据，首先进行音效处理，再通过 DAC 转换为模拟量后送到输出端。LINE/AUX/MIC，经过几个电子开关后送到输出端，同时还送到 ADC 进行采样，然后经过音效处理模块后，又通过 I2S 送出，其实也就是录音功能。
- 有多少路输出？喇叭，耳机，OUT3，OUT4。

我们通过 I2C 写 WM8978 的寄存器控制 WM8978，实际就是控制这个框图中的各个开关、混音器、音量控制、DACADC 功能。更详细的控制可以在规格书第 14 页看到，一整页大图说明了音频通路。



原理图在规格书第 89 页，有所有的寄存器说明。

WM8978 的 IIC 接口比较特殊：

1. 只支持写，不支持读数据；
2. 寄存器长度为 7 位，数据长度为 9 位。
3. 寄存器字节的最低位用于传输数据的最高位（也就是 9 位数据的最高位，7 位寄存器的最低位）
4. WM8978 的 IIC 地址固定为：0x1A。

19.4 DMA

本次实验用到 DMA，在编码前我们先学习学习 DMA。

19.4.1 DMA 是什么

按照国际惯例，百度 GOOGLE

> DMA(Direct Memory Access，直接内存存取) 是所有现代电脑的重要特色，它允许不同速度的硬件装置来沟通，而不需要依赖于 CPU 的大量中断负载。否则，CPU 需要从来源把每一片段的资料复制到暂存器，然后把它们再次写回到新的地方。在这个时间中，CPU 对于其他的工作来说就无法使用。

从这个就可以看出，DMA 就是两个硬件（内存或其他存储结构）直接通信，通信过程不需要 CPU 干预。

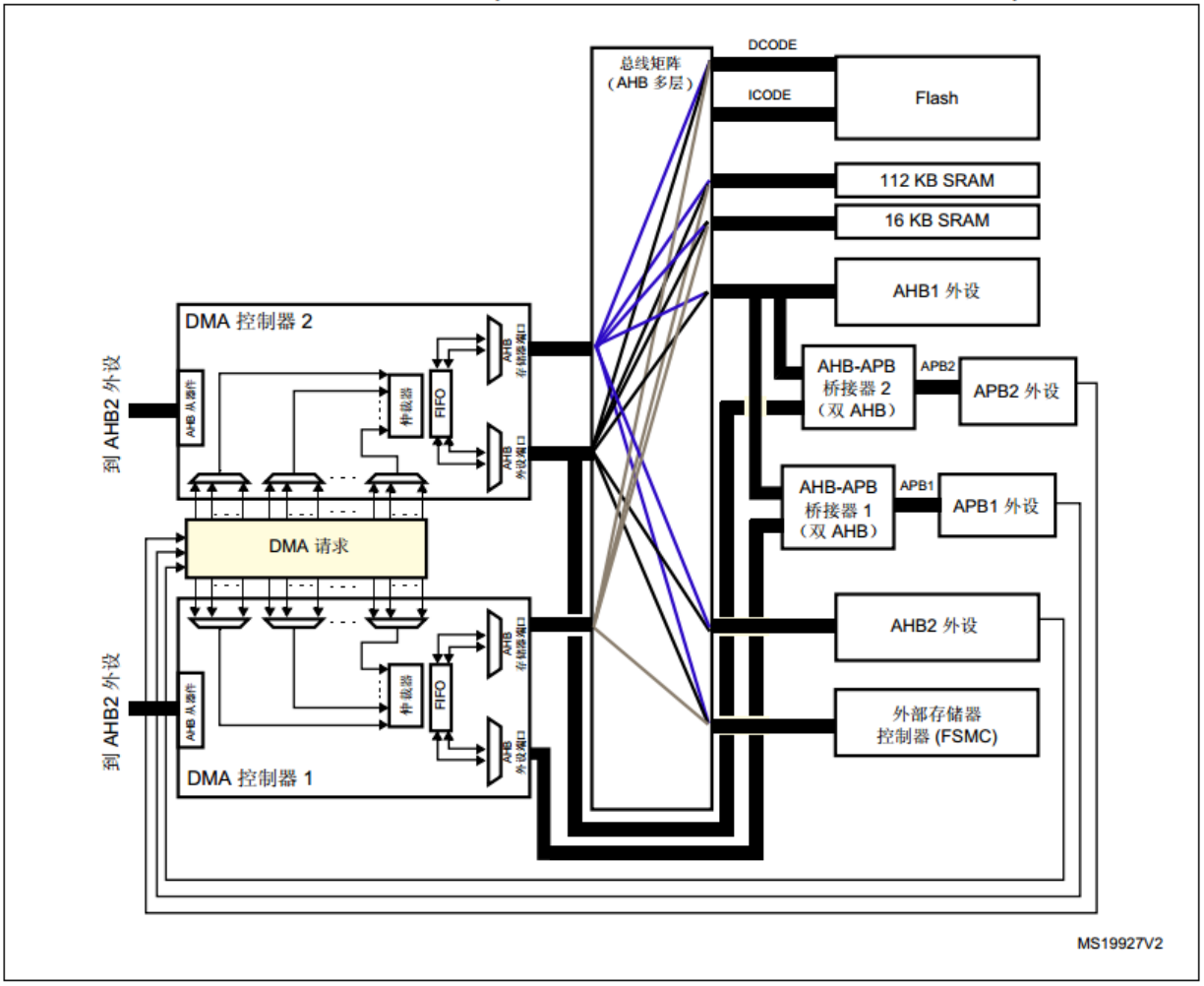
或者说，DMA 是一个只有 MOV 指令的 CPU。

对于单片机来说，就两个外设通信，不需要内核干预。举例：要将一段内存里面的数据通过串口发送出去。通常我们就是编写一段程序，一个字节一个字节将数据发送到串口。在这个过程中，CPU 是一直参与传输的。如果用了 DMA，只要配置好 DMA 后，DMA 就会自己将数据从内存传输到串口。传输过程，CPU 可以去做其他事。

19.4.2 STM32 DMA

在《STM32F4xx 中文参考手册.pdf》第九章有详细介绍。特性比较丰富，大家自己看文档，我们看看 DMA 的框图。

图 25. 两个 DMA 控制器的系统实现 (STM32F405xx/07xx 和 STM32F415xx/17xx)



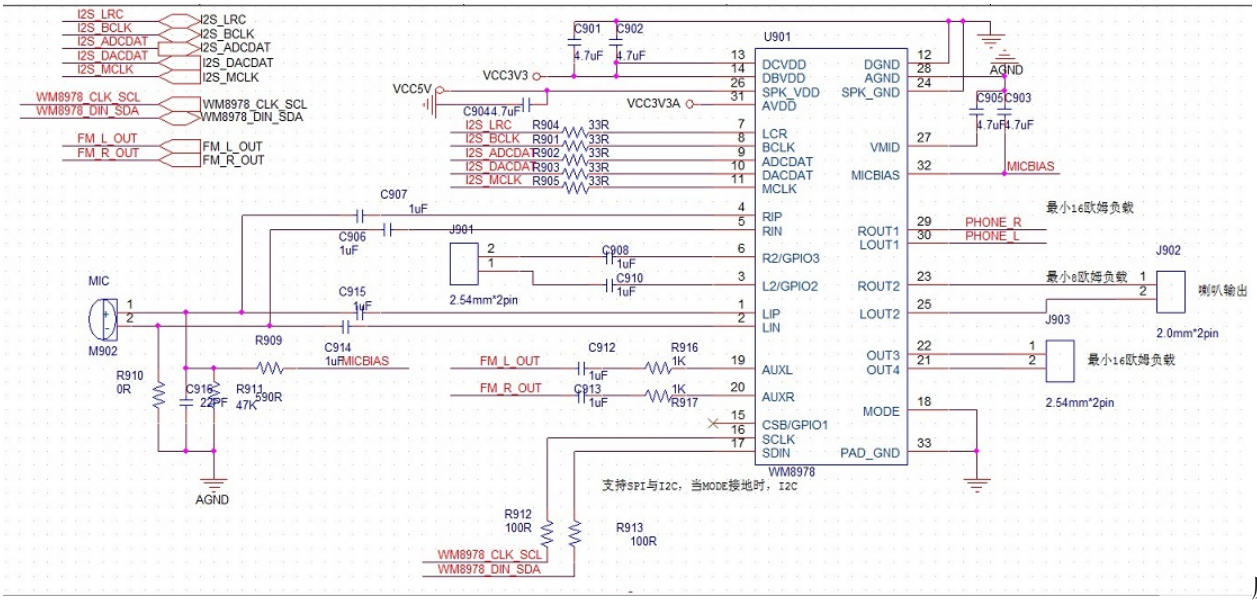
1. DMA1 控制器 AHB 外设端口与 DMA2 控制器的情况不同，不连接到总线矩阵，因此，仅 DMA2 数据流能够执行存储器到存储器的传输。

DMA 框图 STM32 有 2 个 DMA 控制器，但是 DMA2 有点特殊，只能存储器到存储器。

DMA 的具体应用，后面编码时我们会详细分析。

19.5 原理图

下图是 WM8978 的原理图。

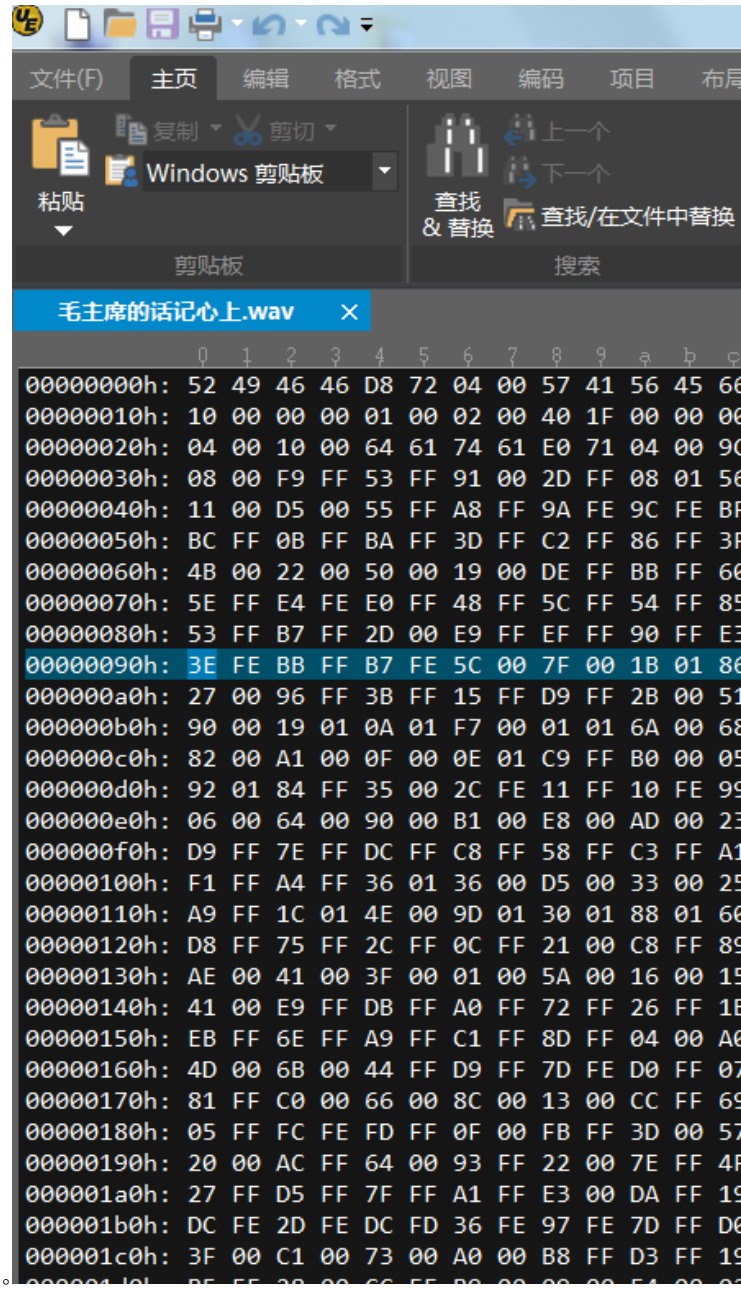


原理图细心的朋友可能发现，前面我们说 I2S 通信，只要 4 根管脚，这里怎么多了一个 MCLK 呢?

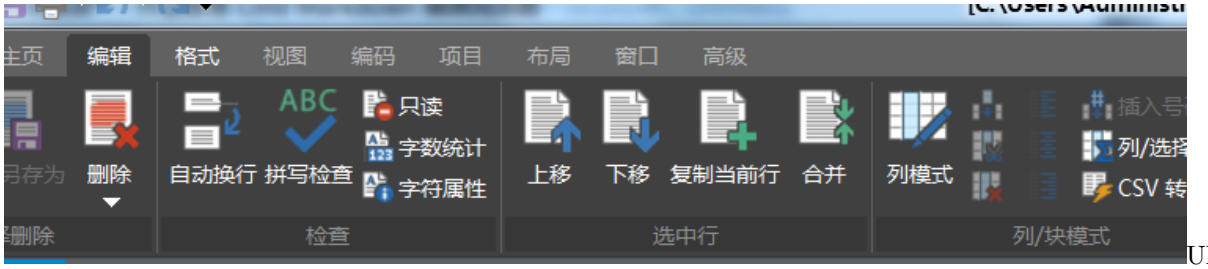
> 有时为了使系统间能够更好地同步，还需要另外传输一个信号 MCLK，称为主时钟，也叫系统时钟（Sys Clock），是采样频率的 256 倍或 384 倍。

19.6 WAV 转换成数组

这次我们只是验证 I2S 硬件，不做 WAV 解码，直接将声音数据转换成一个数组编译到程序里面。如何转换也是一个技巧，学会后处理数据就很简单。



1. 直接将一个 wav 文件拖到 UltraEdit 就可以查看二进制。文件二进制格式
2. 全选，然后右键，选择使用十六进制格式复制。
3. 新建一个 txt，粘贴到 txt，保存。
4. 用 UE 打开刚刚创建的 txt。进入编辑菜单下的列模式，下图右边列模式



UE

列操作

5. 通过列删除删除右边多余数据，通过列添加添加 0x 前缀和逗号，将数据修改为数组的格式。

```
0x52 ,0x49 ,0x46 ,0x46 ,0xD8 ,0x72 ,0x04 ,0x00 ,0x57 ,0x41 ,0x56 ,0x45 ,0x66 ,0x6D ,0x74 ,0x20,
0x10 ,0x00 ,0x00 ,0x00 ,0x01 ,0x00 ,0x02 ,0x00 ,0x40 ,0x1F ,0x00 ,0x00 ,0x00 ,0x7D ,0x00 ,0x00,
0x04 ,0x00 ,0x10 ,0x00 ,0x64 ,0x61 ,0x74 ,0x61 ,0xE0 ,0x71 ,0x04 ,0x00 ,0x9C ,0x00 ,0xF6 ,0xFF,
0x08 ,0x00 ,0xF9 ,0xFF ,0x53 ,0xFF ,0x91 ,0x00 ,0x2D ,0xFF ,0x08 ,0x01 ,0x56 ,0x00 ,0xCA ,0x01,
0x11 ,0x00 ,0xD5 ,0x00 ,0x55 ,0xFF ,0xA8 ,0xFF ,0x9A ,0xFE ,0x9C ,0xFE ,0xBF ,0xFE ,0x84 ,0xFE,
0xBC ,0xFF ,0x0B ,0xFF ,0xBA ,0xFF ,0x3D ,0xFF ,0xC2 ,0xFF ,0x86 ,0xFF ,0x3F ,0xFF ,0x50 ,0xFF,
0x4B ,0x00 ,0x22 ,0x00 ,0x50 ,0x00 ,0x19 ,0x00 ,0xDE ,0xFF ,0xBB ,0xFF ,0x60 ,0xFF ,0x15 ,0xFF,
0x5E ,0xFF ,0xE4 ,0xFE ,0xE0 ,0xFF ,0x48 ,0xFF ,0x5C ,0xFF ,0x54 ,0xFF ,0x85 ,0xFF ,0xEB ,0xFF,
0x53 ,0xFF ,0xB7 ,0xFF ,0x2D ,0x00 ,0xE9 ,0xFF ,0xEF ,0xFF ,0x90 ,0xFF ,0xE3 ,0xFE ,0x64 ,0xFF,
0x3E ,0xFE ,0xBB ,0xFF ,0xB7 ,0xFE ,0x5C ,0x00 ,0x7F ,0x00 ,0x1B ,0x01 ,0x86 ,0x00 ,0x49 ,0x00,
0x27 ,0x00 ,0x96 ,0xFF ,0x3B ,0xFF ,0x15 ,0xFF ,0xD9 ,0xFF ,0x2B ,0x00 ,0x51 ,0x00 ,0xDE ,0x00,
0x90 ,0x00 ,0x19 ,0x01 ,0x0A ,0x01 ,0xF7 ,0x00 ,0x01 ,0x01 ,0x6A ,0x00 ,0x68 ,0x01 ,0xA0 ,0x00,
0x82 ,0x00 ,0xA1 ,0x00 ,0x0F ,0x00 ,0x0E ,0x01 ,0xC9 ,0xFF ,0xB0 ,0x00 ,0x05 ,0x01 ,0x78 ,0x00,
0x92 ,0x01 ,0x84 ,0xFF ,0x35 ,0x00 ,0x2C ,0xFE ,0x11 ,0xFF ,0x10 ,0xFE ,0x99 ,0xFE ,0xCA ,0xFE,
0x06 ,0x00 ,0x64 ,0x00 ,0x90 ,0x00 ,0xB1 ,0x00 ,0xE8 ,0x00 ,0xAD ,0x00 ,0x23 ,0x00 ,0xB0 ,0xFF,
0xD9 ,0xFF ,0x7E ,0xFF ,0xDC ,0xFF ,0xC8 ,0xFF ,0x58 ,0xFF ,0xC3 ,0xFF ,0xA1 ,0xFF ,0xE9 ,0xFF,
0xF1 ,0xFF ,0xA4 ,0xFF ,0x36 ,0x01 ,0x36 ,0x00 ,0xD5 ,0x00 ,0x33 ,0x00 ,0x25 ,0x00 ,0xB7 ,0x00,
0xA9 ,0xFF ,0x1C ,0x01 ,0x4E ,0x00 ,0x9D ,0x01 ,0x30 ,0x01 ,0x88 ,0x01 ,0x60 ,0x00 ,0x2B ,0x00,
0xD8 ,0xFF ,0x75 ,0xFF ,0x2C ,0xFF ,0x0C ,0xFF ,0x21 ,0x00 ,0xC8 ,0xFF ,0x89 ,0x00 ,0x02 ,0x00,
0xAE ,0x00 ,0x41 ,0x00 ,0x3F ,0x00 ,0x01 ,0x00 ,0x5A ,0x00 ,0x16 ,0x00 ,0x15 ,0x01 ,0x83 ,0x00,
0x41 ,0x00 ,0xF9 ,0xFF ,0xDB ,0xFF ,0xA0 ,0xFF ,0x72 ,0xFF ,0x26 ,0xFF ,0x1B ,0x00 ,0x78 ,0xFF
```
wav

数组

6. 修改完成后添加数组名跟大小括号。
7. 拷贝到工程的 c 源文件，就可以将语音编译进 bin，直接下载到芯片使用。

19.7 编码调试

调试步骤：1 通过 I2C 控制 WM8978，将收音机声音从喇叭播出。2 通过 I2C 控制 WM8978，通过 MIC 采集声音从喇叭或耳机输出。3 通过 I2S 播放一段内嵌语音。

19.7.1 WM8978 控制-播放收音机声音和 MIC 功能

在 board_dev 文件夹创建 WM8978 驱动文件。I2C 接口在前面调试收音机时已经调通。根据 WM8978 的 I2C 特殊性，封装两个读写函数，开辟个数组，用来记录写到 wm8978 寄存器的值，读的时候就从这个数组回读。

```
s32 dev_wm8978_writereg(u8 reg, u16 vaule);
s32 dev_wm8978_readreg(u8 addr, u16 *data);
```

其他 WM8978 控制主要有如下：

```
s32 dev_wm8978_set_phone_vol(u8 volume)
s32 dev_wm8978_set_spk_vol(u8 volume)
s32 dev_wm8978_set_mic_gain(u8 gain)
s32 dev_wm8978_set_line_gain(u8 gain)
s32 dev_wm8978_set_aux_gain(u8 gain)
s32 dev_wm8978_inout(WM8978_INIPUT In, WM8978_OUTPUT Out)
```

函数名就可以看出功能，从上到下分别是：

> 耳机音量设置喇叭音量设置 MIC 输入增益设置 LINE 输入增益设置 AUX 增益设置输入输出通道配置

当然，上面只是基本功能，复杂音效暂时不处理。在 main 函数中添加如下代码，初始化 WM8978 后，将收音机设置到指定频率，然后打开 wm8978，就可以从喇叭或者耳机听到收音机声音。

```
/* Infinite loop */
        mcu_uart_open(3);
        wjq_log(LOG_INFO, "hello word!\r\n");
        mcu_i2c_init();
        mcu_spi_init();
        dev_key_init();
        //mcu_timer_init();
        dev_buzzer_init();
        dev_tea5767_init();
        dev_dacsound_init();
        dev_spiflash_init();
        dev_wm8978_init();

        //dev_dacsound_open();
        dev_key_open();
        dev_wm8978_open();
        dev_tea5767_open();
        dev_tea5767_setfre(105700);

        while (1)
        {
                /* 驱动轮询 */
                dev_key_scan();

                /* 应用 */
                u8 key;
```

(continues on next page)

(continued from previous page)

```
                s32 res;

                res = dev_key_read(&key, 1);
                if(res == 1)
                {
                        if(key == DEV_KEY_PRESS)
                        {
                                //dev_buzzer_open();
                                //dev_dacsound_play();
                                //dev_spiflash_test();
                                //dev_sdio_test();
                                dev_wm8978_test();
                                GPIO_ResetBits(GPIOG, GPIO_Pin_0 | GPIO_Pin_1
                                        | GPIO_Pin_2| GPIO_Pin_3);
                                //dev_tea5767_search(1);
                        }
```

其中，初始化 WM898 代码如下，先初始化 I2S，再初始化默认的 WM8978 配置：

```
s32 dev_wm8978_init(void)
{
        mcu_i2s_init();//初始化 I2S 接口
        dev_wm8978_setting_init();//配置 WM8978 初始化状态
        return 0;
}
```

初始化 WM8978 的默认配置，就是写 WM8978 的寄存器。

```
static s32 dev_wm8978_setting_init(void)
{
        s32 ret = -1;

        ret = dev_wm8978_writereg(0,0x00);          // 复位 WM8978
        if(ret == -1)                   // 复位失败
                return ret;

        dev_wm8978_writereg(1,0x1B);

        dev_wm8978_writereg(2,0x1B0);
        dev_wm8978_writereg(3, 0x000C);         // 使能左右声道混合
        dev_wm8978_writereg(6, 0x0000);         // 由处理器提供时钟信号
```

(continues on next page)

(continued from previous page)

```
        dev_wm8978_writereg(43, 0x0010);        // 设置 ROUT2 反相，驱动扬声器所必须
        dev_wm8978_writereg(49, 0x0006);        // 扬声器 1.5x 增益，开启热保护

        dev_wm8978_inout(WM8978_INPUT_NULL,
                                                WM8978_OUTPUT_NULL);

        dev_wm8978_set_mic_gain(50);
        dev_wm8978_set_phone_vol(40);
        dev_wm8978_set_spk_vol(40);
        dev_wm8978_set_aux_gain(3);
        return ret;
}
```

然后就是打开 WM8978，其实就是配置默认输入输出通道

```
s32 dev_wm8978_open(void)
{
        dev_wm8978_inout(WM8978_INPUT_DAC|WM8978_INPUT_AUX
                                        |WM8978_INPUT_LMIC|WM8978_INPUT_RMIC,
                WM8978_OUTPUT_PHONE|WM8978_OUTPUT_SPK);

        return 0;
}
```

程序运行后，就可以听到收音机的声音从 WM8978 的喇叭跟耳机输出。同时，MIC 也正常工作了，测试 MIC 时，可以屏蔽掉下面两句

```
        dev_tea5767_open();
        dev_tea5767_setfre(105700);
```

也就是关收音机。同时请注意：MIC 距离喇叭较近，有可能发生啸叫。如发生啸叫，可以调小喇叭音量，降低 MIC 增益，或则关闭喇叭输出，改为耳机监听 MIC 输入。

- 到此我们基本验证了 WM8978 是能工作的，下一步就验证 I2S 播放音乐。

19.7.2 I2S 播放音频数据

I2S 驱动，参考官方例程。I2S 的关键是框架设计，也即是怎么使用 I2S 传输数据？软件设计要考虑限制条件：

> 首先就是**速度快**，通常播放 44K 采样频率时，一秒钟就需要传输 4422(双声道 16BIT)=166K 数据，毫无疑问，必须使用 **DMA 传输**。并且需要使用 DMA 双缓冲机制，双缓冲就可以做到播

放时读数据。第二，播放语音数据来源通常是 SD 卡或者 U 盘，读文件系统数据通常速度并不是很快，因此缓冲需要开辟大一点，否则语音将断断续续。

- 初始化初始化分两部分，IO 口初始化，I2S 控制器初始化。

```
/**
 *@brief:      mcu_i2s_init
 *@details:    初始化 I2S 接口硬件
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
void mcu_i2s_init (void)
{
        GPIO_InitTypeDef GPIO_InitStructure;

        /*
                LRC         PB12
                BCLK        PB13
                ADCDAT        PC2
                DACDAT       PC3
                MCLK        PC6
        */
        // 初始化时钟
        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB|RCC_AHB1Periph_GPIOC, ENABLE);

        GPIO_PinAFConfig(GPIOB,        GPIO_PinSource12,               GPIO_AF_SPI2);
        GPIO_PinAFConfig(GPIOB,        GPIO_PinSource13,               GPIO_AF_SPI2);
        GPIO_PinAFConfig(GPIOC, GPIO_PinSource2,               GPIO_AF6_SPI2);
        GPIO_PinAFConfig(GPIOC,        GPIO_PinSource3,               GPIO_AF_SPI2);
        GPIO_PinAFConfig(GPIOC,        GPIO_PinSource6,               GPIO_AF_SPI2);

        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;                        // 复用模式
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;        // 速度等级
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                // 推挽输出
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;        //        无上下拉
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12|GPIO_Pin_13;
        GPIO_Init(GPIOB, &GPIO_InitStructure);

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_6;
        GPIO_Init(GPIOC,&GPIO_InitStructure);
```

(continues on next page)

(continued from previous page)

```
}
/**
 *@brief:      mcu_i2s_config
 *@details:    I2S 配置
 *@param[in]   u32 AudioFreq   频率
               u16 Standard    标准
               u16 DataFormat  格式
 *@param[out]  无
 *@retval:
 */
void mcu_i2s_config(u32 AudioFreq, u16 Standard,u16 DataFormat)
{
        I2S_InitTypeDef I2S_InitStructure;
        // 配置 IIS PLL 时钟
        RCC_I2SCLKConfig(RCC_I2S2CLKSource_PLLI2S);
        RCC_PLLI2SCmd(ENABLE);// 使能 PLL
        while( RCC_GetFlagStatus(RCC_FLAG_PLLI2SRDY) == 0 );// 等待配置完成

        RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);// 初始化 IIS 时钟

        SPI_I2S_DeInit(SPI2);

        I2S_InitStructure.I2S_AudioFreq = AudioFreq;//设置音频采样频率
        I2S_InitStructure.I2S_Standard = Standard;        //I2S Philips 标准
        I2S_InitStructure.I2S_DataFormat = DataFormat;        //数据长度 16 位
        I2S_InitStructure.I2S_CPOL = I2S_CPOL_Low;        // 空闲状态电平位低
        I2S_InitStructure.I2S_Mode = I2S_Mode_MasterTx;//主机发送
        I2S_InitStructure.I2S_MCLKOutput = I2S_MCLKOutput_Enable;//主时钟输出
        I2S_Init(SPI2, &I2S_InitStructure);

        I2S_Cmd(SPI2, ENABLE);        // 使能 IIS
}
```

mcu_i2s_init 就是 IO 口初始化，很简单，只要将对应 IO 设置为 AF 功能就行了。我们用的是 I2S2，所以设置为 SPI2 功能即可。

mcu_i2s_config 就是 I2S 控制器的初始化，主要有 5 个配置：采样频率，通信格式，数据长度，CPOL，主机模式，MCLK 时钟输出。

通过查看 I2S_InitTypeDef 结构体，也可以知道应该如何配置。

```
typedef struct
{

  uint16_t I2S_Mode;         /*!< Specifies the I2S operating mode.
                                This parameter can be a value of @ref I2S_Mode */

  uint16_t I2S_Standard;     /*!< Specifies the standard used for the I2S communication.
                                This parameter can be a value of @ref I2S_Standard */

  uint16_t I2S_DataFormat;   /*!< Specifies the data format for the I2S communication.
                                This parameter can be a value of @ref I2S_Data_Format */

  uint16_t I2S_MCLKOutput;   /*!< Specifies whether the I2S MCLK output is enabled or not.
                                This parameter can be a value of @ref I2S_MCLK_Output */

  uint32_t I2S_AudioFreq;    /*!< Specifies the frequency selected for the I2S␣
↪communication.
                                This parameter can be a value of @ref I2S_Audio_Frequency␣
↪*/

  uint16_t I2S_CPOL;         /*!< Specifies the idle state of the I2S clock.
                                This parameter can be a value of @ref I2S_Clock_Polarity */
}I2S_InitTypeDef;
```

- DMA 配置 DMA 前面我们大概了解了一下，其实 DMA 功能有点复杂。下面我们通过程序学习学习。

```
/**
 *@brief:      mcu_i2s_dam_init
 *@details:    初始化 I2S 使用的 DMA 通道，双缓冲模式
 *@param[in]   u16 *buffer0
               u16 *buffer1
               u32 len
 *@param[out]  无
 *@retval:
 */
void mcu_i2s_dma_init(u16 *buffer0,u16 *buffer1,u32 len)
{
    NVIC_InitTypeDef   NVIC_InitStructure;
    DMA_InitTypeDef  DMA_str;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1,ENABLE);        //使 IIS DMA 时钟
```

(continues on next page)

(continued from previous page)

```
    DMA_DeInit(DMA1_Stream4);        //恢复初始 DMA 配置

    DMA_str.DMA_Channel = DMA_Channel_0;  //IIS DMA 通道
    DMA_str.DMA_PeripheralBaseAddr = (u32)&SPI2->DR;        //外设地址
    DMA_str.DMA_Memory0BaseAddr = (u32)buffer0;                //缓冲区 0
    DMA_str.DMA_DIR = DMA_DIR_MemoryToPeripheral;        //存储器到外设模式
    DMA_str.DMA_BufferSize = len;                //数据长度
    DMA_str.DMA_PeripheralInc = DMA_PeripheralInc_Disable;        //外设非增量模式
    DMA_str.DMA_MemoryInc = DMA_MemoryInc_Enable;        //存储器增量模式
    DMA_str.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;//外设数据长度 16 位
    DMA_str.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;//存储器数据长度 16 位
    DMA_str.DMA_Mode = DMA_Mode_Circular;        //循环模式
    DMA_str.DMA_Priority = DMA_Priority_High;        //高优先级
    DMA_str.DMA_FIFOMode = DMA_FIFOMode_Disable; //不使用 FIFO
    DMA_str.DMA_FIFOThreshold = DMA_FIFOThreshold_1QuarterFull;        //FIFO 阈值
    DMA_str.DMA_MemoryBurst = DMA_MemoryBurst_Single;        //外设突发单次传输
    DMA_str.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;        //存储器突发单次传输
    DMA_Init(DMA1_Stream4, &DMA_str);        //        初始化 DMA
    //配置缓冲区 1
    DMA_DoubleBufferModeConfig(DMA1_Stream4,(uint32_t)buffer0, DMA_Memory_0);
    //配置缓冲区 1
    DMA_DoubleBufferModeConfig(DMA1_Stream4,(uint32_t)buffer1, DMA_Memory_1);

    DMA_DoubleBufferModeCmd(DMA1_Stream4,ENABLE);        //开启双缓冲模式
    DMA_ITConfig(DMA1_Stream4,DMA_IT_TC,ENABLE);        //开启传输完成中断

    SPI_I2S_DMACmd(SPI2, SPI_I2S_DMAReq_Tx,ENABLE);        //IIS TX DMA 使能.

    NVIC_InitStructure.NVIC_IRQChannel = DMA1_Stream4_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;//抢占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;  //响应优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

1. DMA 通道 18 行

DMA 通道设置。STM32 有 2 个 DMA，每个 DMA 的通道和 stream 在参考手册中都有。

表 35. DMA1 请求映射

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
通道 1	I2C1_RX		TIM7_UP		TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
通道 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
通道 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
通道 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
通道 5	UART8_TX(1)	UART7_TX(1)	TIM3_CH4 TIM3_UP	UART7_RX(1)	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UART8_RX(1)	TIM3_CH3
通道 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2		TIM5_UP	
通道 7		TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

1. 这些请求仅在 STM32F42xxx 和 STM32F43xxx 上可用。

表 36. DMA2 请求映射

外设请求	数据流 0	数据流 1	数据流 2	数据流 3	数据流 4	数据流 5	数据流 6	数据流 7
通道 0	ADC1		TIM8_CH1 TIM8_CH2 TIM8_CH3		ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
通道 1		DCMI	ADC2	ADC2		SPI6_TX(1)	SPI6_RX(1)	DCMI
通道 2	ADC3	ADC3		SPI5_RX(1)	SPI5_TX(1)	CRYP_OUT	CRYP_IN	HASH_IN
通道 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
通道 4	SPI4_RX(1)	SPI4_TX(1)	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
通道 5		USART6_RX	USART6_RX	SPI4_RX(1)	SPI4_TX(1)		USART6_TX	USART6_TX
通道 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
通道 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX(1)	SPI5_TX(1)	TIM8_CH4 TIM8_TRIG TIM8_COM

1. 这些请求在 STM32F42xxx 和 STM32F43xxx 上可用。

19.8

1. DMA_PeripheralBaseAddr 19 行外设地址，不一定是真正的外设地址。如果你是外设到内存，或者内存到外设，这里就是一个外设的地址。这里的外设就是前面表格中说的片上设备，SPI、串口、SDIO 等等。

如果你是 memory 到 memory，那这个就是 RAM 地址或者 FLASH 地址，或外部 RAM SRAM 等地址。

1. DMA_Memory0BaseAddr 20 行 memory 地址，为什么带个 0 呢？因为某种情况下，可以用双缓冲。我们这次做 I2S 播放音乐的时候我们就用双缓冲。

其他情况通常都是单缓冲。

1. DMA_DIR 21 行 DMA 方向，有 3 种：DMA_DIR_PeripheralToMemory

DMA_DIR_MemoryToPeripheral DMA_DIR_MemoryToMemory

好了，到这里就有点疑问了，配合前面设置的两个地址来看，如果是 P-M，也就是 perpheral to memory，那肯定就是 DMA_PeripheralBaseAddr 取数据，发到 DMA_Memory0BaseAddr。如果是 M-P，那就是反着来。那 M-M 呢?我没找到文档哪里写，实测，是从 DMA_PeripheralBaseAddr 取数据，发到 DMA_Memory0BaseAddr。只有 DMA2 控制器能够执行存储器到存储器的传输。

1. DMA_BufferSize 22 行准确的说应该是传输长度。如果配置不准确，传输数据就或出错，或者是一直在 DMA 传输，永远不结束。这个长度，跟后面配置字长有关：如果 DMA_PeripheralDataSize 跟 DMA_MemoryDataSize 相等，那就没话说了，该是多长就是多长。如果两者不相等，DMA_BufferSize 是指 DMA_PeripheralDataSize 数据宽度的数据数。假如 DMA_PeripheralDataSize 是半字，DMA_MemoryDataSize 是 BYTE，DMA_BufferSize=10，就表示 DMA 要传输 10 个半字，20 个 BYTE。

具体看《STM32F4xx 中文参考手册.pdf》**9.3.10 可编程数据宽度、封装/解封、字节序** 6. DMA_PeripheralInc 23 行 DMA_MemoryInc 24 行地址是否自加。我们是将内存的一堆数据发送到 I2S 控制器，因此，内存自加，I2S 控制器不自加。

每次都是发送到 &SPI2->DR 这个寄存器地址。7. DMA_PeripheralDataSize 25 行 DMA_MemoryDataSize 26 行字长设置，当我把两者都设置为 halfword 时，发现如果传输奇数个半字，会一直处于发送，DMA 不会停止。因此把 DMA_PeripheralDataSize 设置为 byte，那么传输的字节数就是 2* 实际要传输的半字，那么肯定是 2 的倍数，就能正常。关于这个在文档中也有提及，只是还看不太明白。

这次 I2S DMA，都用半字，因此需要保证传输数据是 2 的整数倍。8. DMA_Mode 27 行是否循环。所谓的循环就是当一次传输结束后，自动重新启动一次 DMA 传输，配置和前一次传输一样。摄像头就用循环，只要配置一次，启动 DMA 后，一直持续不断的更新数据到屏幕上。需要注意的是：

使用存储器到存储器模式时，不允许循环模式和直接模式。9. DMA_Priority 28 行

优先级 10. DMA_FIFOMode 29 行 FIFOThreshold 30 行是否使用 FIFO，也就是是否使用直接模式。但是，我们的配置是 disable，那么就是直接模式?

存储器到存储器不是不允许直接模式吗？

1. DMA_MemoryBurst 31 行 DMA_PeripheralBurst 32 行 9.3.11 单次传输和突发传输 “为确保数据一致性，形成突发的每一组传输都不可分割：在突发传输序列期间，AHB 传输会锁定，并且 AHB 总线矩阵的仲裁器不解除对 DMA 主总线的授权。”

   从文档看来，是保证数据完整性的，也就是说不要传到一半被别的设备打断造成数据错误?

2. DMA_Init(DMA1_Stream4, &DMA_InitStructure); 33 行

   执行配置，第一个参数要选对 Stream0，用哪个 stream，根据表格选。

3. 35 行 ~39 行

   配置双缓冲

4. 之后后面就是使能 DMA 中断，打开 SPI/I2S DMA，配置 NVIC 中断控制器。配置完成后，当需要时启动 DMA 传输即可，传输完成则产生一个中断。

- DMA 中断

当 DMA 中断产生时，判断 DMA 缓冲的标志，然后设置 BUF 标志。在主循环中查询这个 BUF 标志，根据标志填充数据到对应缓冲。

```
/**
 *@brief:      mcu_i2s_dma_process
 *@details:    I2S 使用的 DMA 中断处理函数
 *@param[in]   void
 *@param[out]  无
 *@retval:

 位 19 CT: 当前目标 (仅在双缓冲区模式下) (Current target (only in double buffer mode))
此位由硬件置 1 和清零，也可由软件写入。
0: 当前目标存储器为存储器 0 (使用 DMA_SxM0AR 指针寻址)
1: 当前目标存储器为存储器 1 (使用 DMA_SxM1AR 指针寻址)
只有 EN 为 “0” 时，此位才可以写入，以指示第一次传输的目标存储区。在使能数据流
后，此位相当于一个状态标志，用于指示作为当前目标的存储区。
 */
void mcu_i2s_dma_process(void)
{
        if(DMA1_Stream4->CR&(1<<19))
        {
                /* 当前目标存储器为 1，我们就设置空闲 BUF 为 0*/
                fun_sound_set_free_buf(0);
        }
        else
        {
                fun_sound_set_free_buf(1);
        }
}
```

- 双缓冲机制

1. 启动播放时，将两个缓冲都填上音源数据。
2. I2S 开始传输，传输完一个缓冲，DMA 产生中断，根据使用的缓冲设置要写缓冲索引。
3. 语音播放程序判断是否需要填充，填充哪个缓冲。**千万不要在 DMA 中断函数内填充缓冲，通过文件系统读几 K 数据，卡太久中断，会出问题的**。

```
/**
 *@brief:      fun_sound_get_buff_index
 *@details:    查询当前需要填充的 BUF
```

(continues on next page)

(continued from previous page)

```
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
static s32 fun_sound_get_buff_index(void)
{
        s32 res;

        res = SoundBufIndex;
        SoundBufIndex = 0xff;
        return res;
}
/**
 *@brief:      fun_sound_set_free_buf
 *@details:    设置空闲缓冲索引

 *@param[in]   u8 *index
 *@param[out]  无
 *@retval:
 */
s32 fun_sound_set_free_buf(u8 index)
{
        SoundBufIndex = index;
        return 0;
}
```

具体填充数据请看源代码。

19.8.1 调试

WM8978 I2S 功能先写一个简单的 dev_wm8978_test 测试函数，首先初始化 WM8978，设置 I2S 的格式（MCU 跟 WM8978 都要配置），初始化 I2S DMA，然后预填充两个 BUF，启动 DMA，进行 while 循环，在循环中不断查询，需要就填充数据。在 main 函数中，当按下按键时，就调用 dev_wm8978_test 播放。

> 调试的时候，左声道有点杂音，仅仅是久不久有一点杂音，一开始就以为是 I2S 通信被干扰了。分析过程：首先将填充左声道的 BUFF 数据全部设置为 0，更新程序后发现左声道竟然还有声音。然后将两个声道的数据全部填充为 0，竟然还会有一点点背景音。不科学，从而知道不是 I2S 干扰，应该是填缓冲有问题，追查下去发现缓冲填错了。当 I2S 使用完缓冲 1 的数据，就会切换到缓冲 2，此时应该填充缓冲 1，程序却填充缓冲 2 了。修正后，填充数据 0，WM8978 静音: 填充音源数据，左声道没杂音。

19.9 思考

wm8978 的基本功能就调试通了。请思考：我们前面做了一个 DAC 播放语音，现在又做了一个更加高级的 WM8978。这两个驱动要提供什么接口？提供给谁？或者从上到下分析，APP 要播放语音，要什么接口？硬件有两个声卡设备，怎么操作？对于语音播放来说，仅仅实现 DAC 播放或者是 WM8978 播放时远远不够的。等硬件验证完，文件系统也移植好的时候，我们会做一个语音播放管理程序。

19.10 END

CHAPTER 20

FSMC-TFT LCD 调试记录

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190402

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

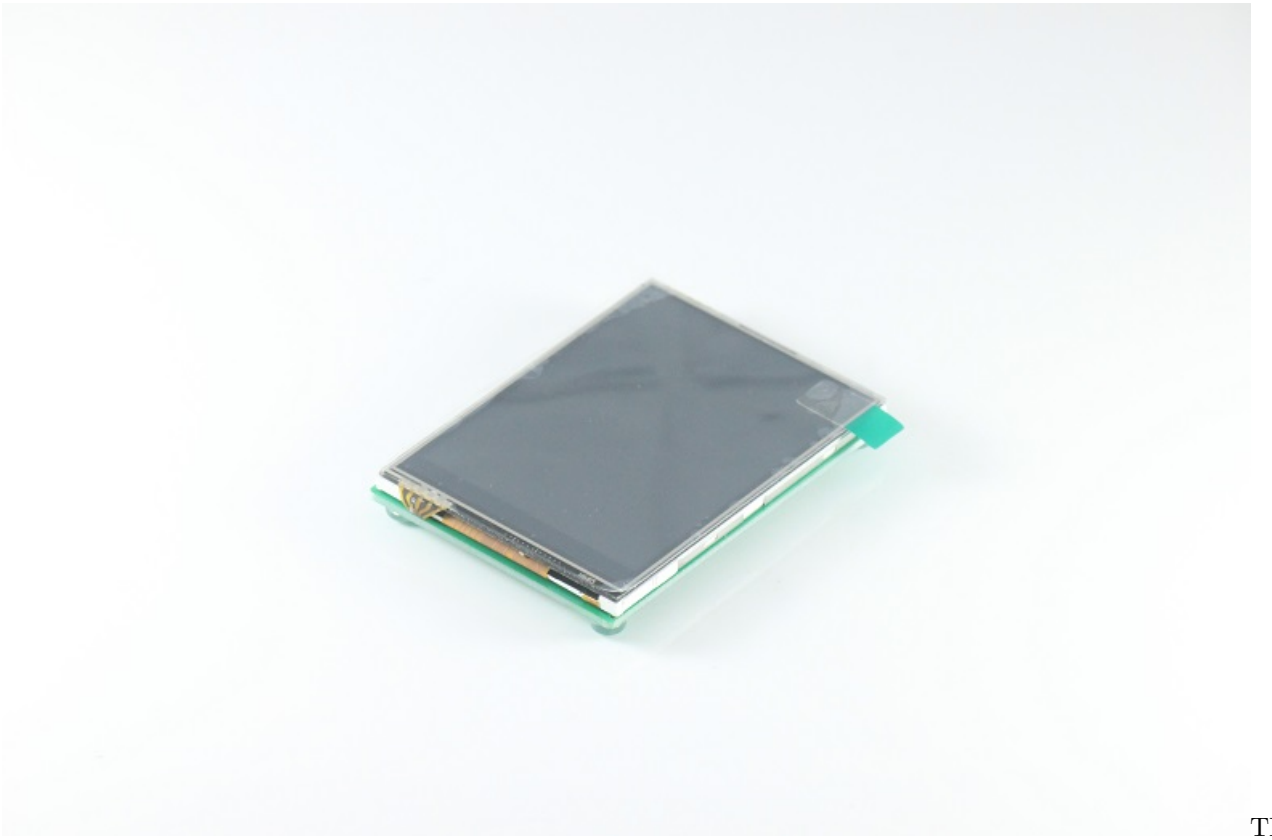
QQ 群：767214262

电子设备人机交互包含输入输出，液晶显示是输出主要方法。屋脊雀 F407 开发板在核心板上有 TFT LCD 液晶屏接口，可以接我们接口的 8080 并口总线液晶屏。

开发阶段先调试 LCD 显示简单的英文字符，图片与汉字显示暂时不处理。

20.1 LCD 液晶屏

tft-lcd 是薄膜晶体管液晶显示器英文 thin film transistor-liquid crystal display 的缩写。



TFT LCD

本次调试的是屋脊雀设计的模块

> 带四线电阻触摸屏 2.8 寸标准 37PIN 并口，支持 8080 或 6800 接口。分辨率 320*240 16 位真彩色（65k 色）。液晶驱动芯片为 ILI9341。双接口接口：FPC30 和 DIP30。默认使用 FPC 排线连接。

本次仅仅编写显示驱动，触摸屏驱动下一章再讨论。

20.1.1 LCD 组成

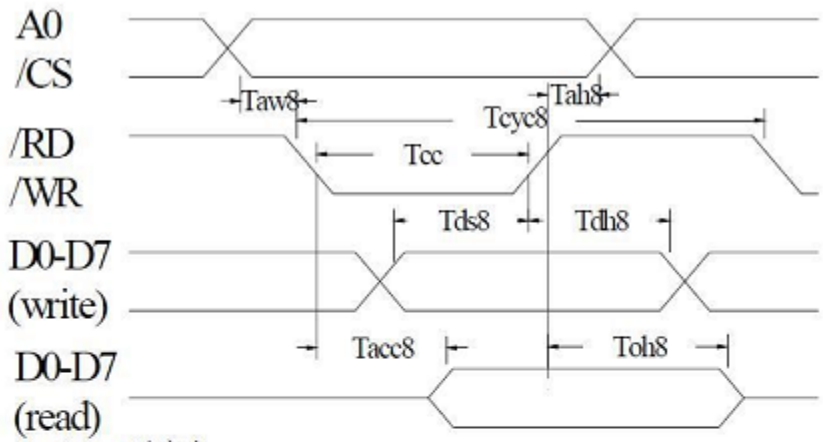
我们常说的 LCD，其实是 LCD 模组，包含驱动芯片和液晶面板。驱动芯片跟液晶面板之间有很多线连接，通过这些线控制液晶面板。**我们开发 LCD 驱动，其实就是开发驱动芯片的驱动**。常用的 LCD 驱动 IC 非常多，但是从功能上分析，基本类似。学会一种就可以触类旁通。下面我们通过 9341 驱动学习 LCD 驱动芯片的功能。

20.1.2 时序

并口有两种时序模式：英特尔的 8080 时序、摩托罗拉的 6800 时序。并口不仅仅用于 TFT LCD，其它 SDRAM 等芯片也使用并口时序通信。

- 8080 时序

Intel8080时序读、写操作时序图



Intel8080 时序表：

符号	参数说明	Vdd=4.5V～5.5V		Vdd=2.7V～4.5V		单位
		最小	最大	最小	最大	
Tcyc8	系统周期时间	550	—	550	—	ns
Taw8	地址建立时间	0	—	0	—	ns
Tah8	地址保持时间	10	—	10	—	ns
Tcc	读写脉冲宽度	120	—	150	—	ns
Tds8	写数据建立时间	120	—	120	—	ns
Tdh8	写数据保持时间	5	—	5	—	ns
Tacc8	读数据建立时间	—	50	—	80	ns
Toh8	读数据保持时间	10	50	10	80	ns

Intel8080 接口信号的组合功能：

/CS	A0	/RD	/WR	功能
1	X	X	X	禁止操作
0	0	0	1	读状态标志位
0	0	1	0	写指令参数和显示数据
0	1	0	1	读显示数据和光标指针
0	1	1	0	写指令代码

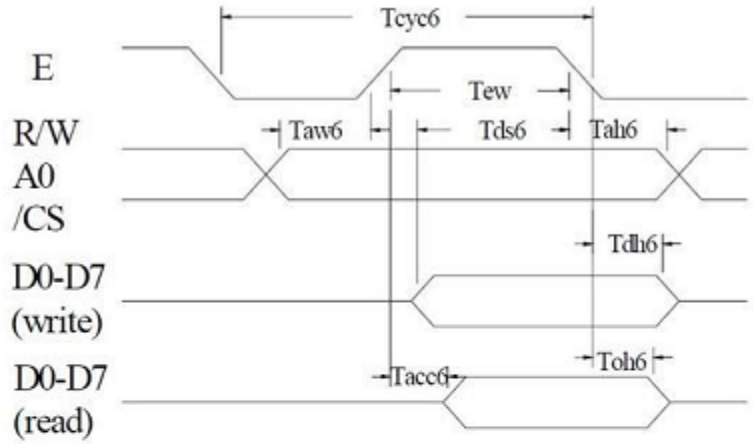
8080 时序

8080 时序，通常有下列接口信号

Vcc(工作主电源) Vss(公共端) Vee(偏置负电源，常用于调整显示对比度) /RES，复位线。DB0~DB7，双向数据线。D/I，数据/指令选择线 (1: 数据读写,0: 命令读写)。/CS，片选信号线 (如果有多片组合，可有多条片选信号线)。/WR，MPU 向 LCD 写入数据控制线。/RD，MPU 从 LCD 读入数据控制线。

- 6800 时序

2. M6800时序读写操作时序图



M6800时序表：

符号	参数说明	Vdd=4.5V～5.5V		Vdd=2.7V～4.5V		单位
		最小	最大	最小	最大	
Tcyc6	系统周期时间	550	—	550	—	ns
Taw6	地址建立时间	0	—	0	—	ns
Tah6	地址保持时间	0	—	10	—	ns
Tds6	数据建立时间	100	—	120	—	ns
Tdh6	数据保持时间	0	—	0	—	ns
Tacc6	输出建立时间	—	85	—	130	ns
Toh6	输出保持时间	10	50	10	75	ns
Tew	使能脉冲宽度	120	—	150	—	ns

M6800 接口信号的组合功能：

/CS	A0	R/W	E	功能
1	X	X	X	禁止操作
0	0	1	↓	读状态标志位
0	0	0	⎍	写指令参数和显示数据
0	1	1	↓	读显示数据和光标指针
0	1	0	⎍	写指令代码

6800 时序

上图即为摩托罗拉的 6800 时序

在这种模式下,Vcc、Vss、Vee、/RES、DB0~DB7、D/I 的功能同模式 (1), 其他信号线为: R/W, 读写控制 (1:MPU 读, 0:MPU 写)。E, 允许信号 (多片组合时, 可有多条允许信号线)。

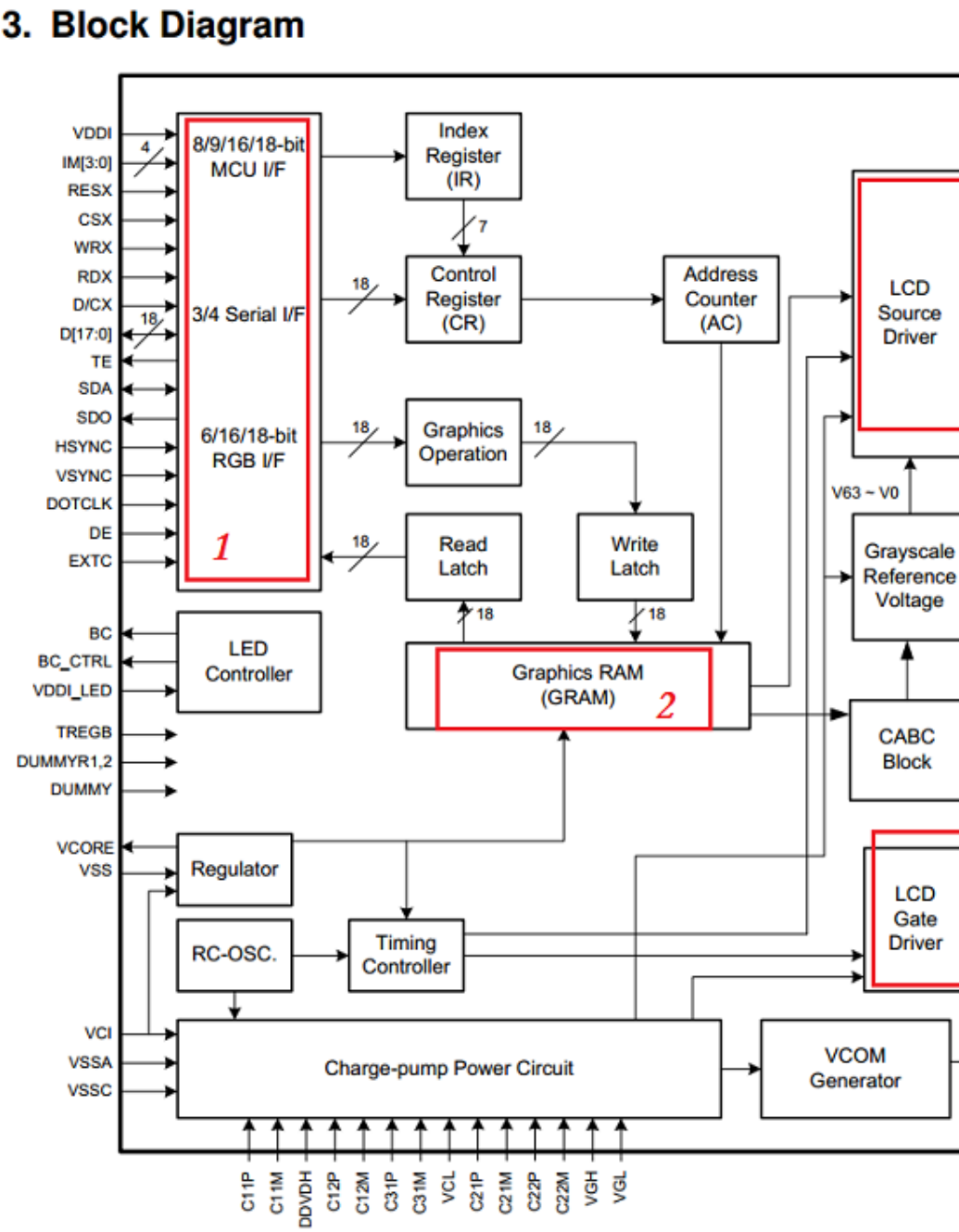
20.1.3 ILI9341

《ILI9341_DS_V1.09_20110315.pdf》打开这个文档即可看到文档标题

a-Si TFT LCD Single Chip Driver 240RGBx320 Resolution and 262K color

从标题就可以看出这个驱动 IC 的性能：

可以支持 240*320* 像素，只是表明驱动 *IC* 的性能，不代表 *LCD* 模组像素大小，*LCD* 像素可能会小于 *320*240。RGB 262K 色



在第 9 页有 9341 这颗芯片的框图
框图主要有以下说明：

1. 左上红框 1，说明这颗 IC 支持多种通信接口，RGB 接口、串口、MCU 接口。其中 MCU 接口就是我们常说的并口，通常是 6800 或 8800 时序。但是模组用什么接口，支持什么接口，不同厂家设计的不一样。所以用 9341 的 LCD 模组，接口可能会不同。
2. 中间红框 2，就是芯片的显存，使用 LCD 就是将显示数据写到这个地方。
3. 右上红框和右下红框，就是这颗芯片跟液晶面板的连线。

通常显示驱动 IC 就分两部分：控制，显示。

1. 对于显示，其实就是读写驱动 IC 里面的显示缓冲区，操作上跟读写外部 SRAM 基本一样。但是接口上有区别，在下面 FSMC 部分有说明。

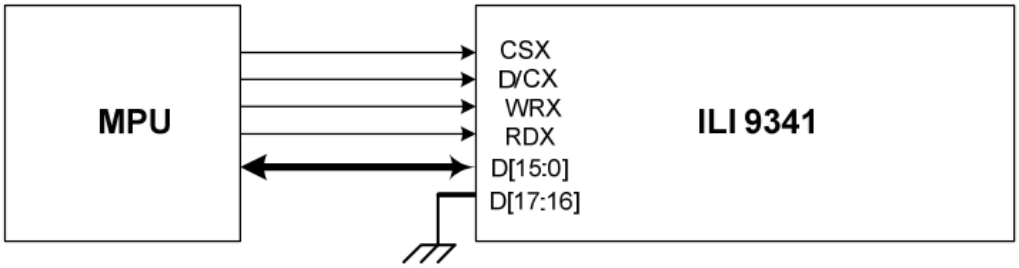
2. 控制部分就是读写驱动 IC 内的命令寄存器，在文档中对每一个命令的操作流程都有说明。过程跟 SPI FLASH 的控制过程类似。控制又可以分为两部分：初始化、显示过程可控制。

   初始化一般都是由厂家提供，我们作为用户，用就行了。有问题找原厂。显示过程的控制大概有：设置显示区域、设置扫描方向、设置屏幕方向、打开或关闭显示、开始填充数据等几个命令。

20.1.4 数据格式

9341 支持多种接口，不同接口的数据格式是不一样的。本次我们用的 LCD 模组是 16BIT 并口。格式说明在文档的 7.6.5 章节

The 8080-Ⅰ system 16-bit parallel bus interface of ILI9341 can be selected by setting hardware pin IM[3:0] to "0001".The following shown figure is the example of interface with 8080-Ⅰ MCU system interface.



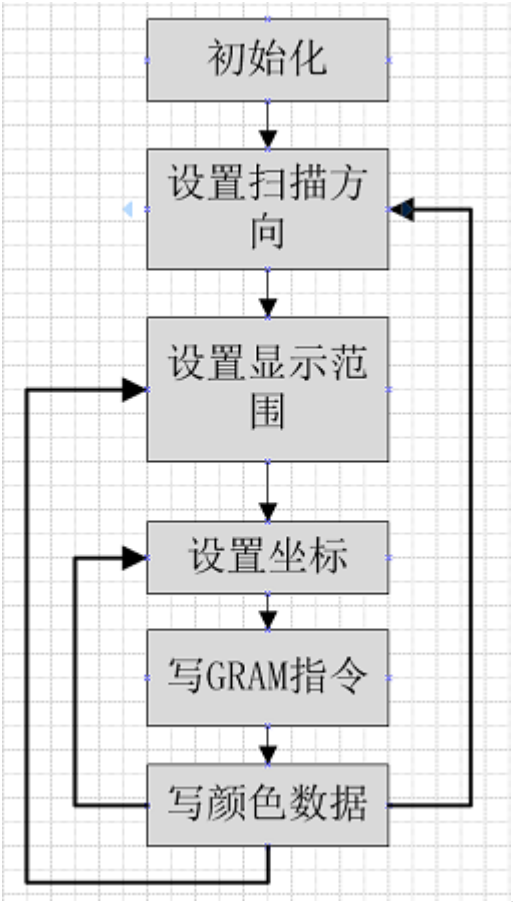
8I 在这种接口模式下，数据有两种格式：65K 色、256K 色。我们只用 65K 色。那么一次发送的数据格式就是:RGB565 格式。

One pixel (3 sub-pixels) display data is sent by 1 transfer when DBI [2:0] bits of 3Ah register are set to "101".

Count	0	1	2	3	…	238	239	240
D/CX	0	1	1	1	…	1	1	1
D15		0R4	1R4	2R4	…	237R4	238R4	239R4
D14		0R3	1R3	2R3	…	237R3	238R3	239R3
D13		0R2	1R2	2R2	…	237R2	238R2	239R2
D12		0R1	1R1	2R1	…	237R1	238R1	239R1
D11		0R0	1R0	2R0	…	237R0	238R0	239R0
D10		0G5	1G5	2G5	…	237G5	238G5	239G5
D9		0G4	1G4	2G4	…	237G4	238G4	239G4
D8		0G3	1G3	2G3	…	237G3	238G3	239G3
D7	C7	0G2	1G2	2G2	…	237G2	238G2	239G2
D6	C6	0G1	1G1	2G1	…	237G1	238G1	239G1
D5	C5	0G0	1G0	2G0	…	237G0	238G0	239G0
D4	C4	0B4	1B4	2B4	…	237B4	238B4	239B4
D3	C3	0B3	1B3	2B3	…	237B3	238B3	239B3
D2	C2	0B2	1B2	2B2	…	237B2	238B2	239B2
D1	C1	0B1	1B1	2B1	…	237B1	238B1	239B1
D0	C0	0B0	1B0	2B0	…	237B0	238B0	239B0

看上图，第 0 个字节只有低 8 位有效，这个字节是命令。从第 1 个字节开始，就是显示数据，每个数据 16 位，从高位到低位分 3 部分: 红色数据 5bit、绿色数据 6bit、蓝色数据 5bit。

20.1.5 显示流程



通常显示流程如下图。 显示流程

1. 初始化包括硬件复位和 LCD 控制初始化，初始化流程比较复杂，通常由模组厂家提供。
2. 设置扫描方向，当持续刷新显示数据时，控制器自动移动 GRAM 指针的方向。有两个：page 和 colum.
3. 设置显示范围，也就是显示窗口的意思，假如一个 320*240 的屏幕，要显示一张 100X100 的图片，我们就可以设置一个 100X100 的显示窗口。
4. 坐标，也就是开始写 GRAM 的位置，必须在显示窗口范围内。例如我们设置了一个 100*100 的窗口，左上角坐标是 (1,1)。显示坐标设置为 (1,1)。然后开始写 GRAM 指令，后面传输的显示数据就会自动在这个窗口内顺序刷新。

上面说的几点，不同的控制器有一点差别

20.1.6 命令

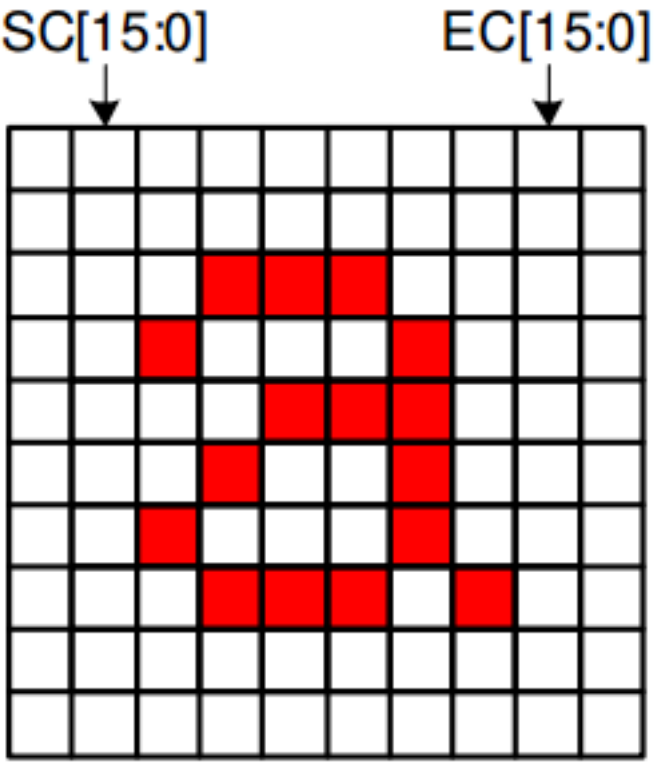
1. 设置扫描方向 9341 使用 36H 命令控制扫描方向。简单说就是连续读写 GRAM 时，GRAM 指针增长方向。在 8.2.29 有命令说明，其中关键的是 MY/MX/MV：这 3 个 bit 控制 memory 读写方向。

Bit	Name	Description
MY	Row Address Order	These 3 bits control MCU to memory write/read direction.
MX	Column Address Order	
MV	Row / Column Exchange	
ML	Vertical Refresh Order	LCD vertical refresh direction control.
BGR	RGB-BGR Order	Color selector switch control (0=RGB color filter panel, 1=BGR color filter panel)
MH	Horizontal Refresh ORDER	LCD horizontal refreshing direction control.

36

命令

2. 设置 column 范围 2AH 命令设置 column 范围，SC 是 column 起始，EC 就是结束。



2A 命令

那可设范围是多大呢？请看下面。根据 MADCTL 第 5 个 BIT，column 不能超过 0xEF/0x13f。也就是不能超过 239/319。啥意思？也就是**意味着，可以通过 MADCTL 第 5 个 BIT 调换 column 和 page**。

Restriction	SC [15:0] always must be equal to or less than EC [15:0]. Note 1: When SC [15:0] or EC [15:0] is greater than 00EFh (When MADCTL's B5 = 0) or 013Fh (When MADCTL's B5 = 1), data of out of range will be ignored

2A

命令

1. 设置 page 范围 2BH 命令设置 page 范围，SP 是 page 起始地址，EP 是 page 结束地址。2B 命令 2B 命令和 2A 命令类似。

Restriction	SP [15:0] always must be equal to or less than EP [15:0] Note 1: When SP [15:0] or EP [15:0] is greater than 013Fh (When MADCTL's B5 = 0) or 00EFh (When MADCTL's B5 = 1), data of out of range will be ignored.

2B 命令

很多人说这两个命令时用于设置扫描起始地址，这是不对的。这两个命令时用于设置扫描窗口范围。当设置窗口时，起始地址默认为 (sc,sp)

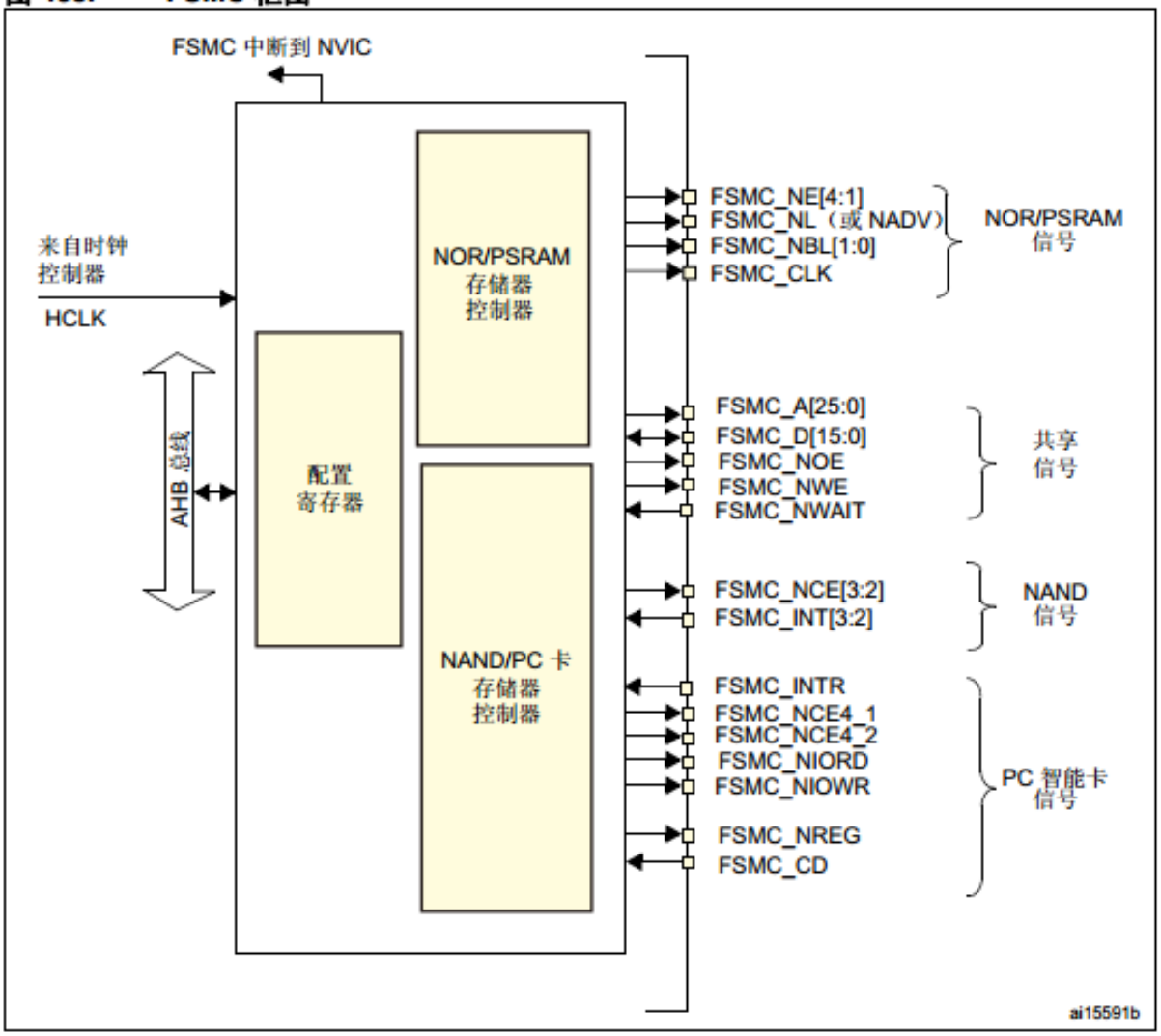
1. 写 GRAM 指令命令 2CH 用于开始写 GRAM。
2. 写数据紧跟 2C 命令后面的数据都是显示数据，直接写到 GRAM 中，GRAM 指针移动根据前面几个指令的设置决定。

20.2 FSMC

要了解 FSMC 接口，我们打开《STM32F4xx 中文参考手册.pdf》，翻到第 32 章：灵活的静态存储控制器。

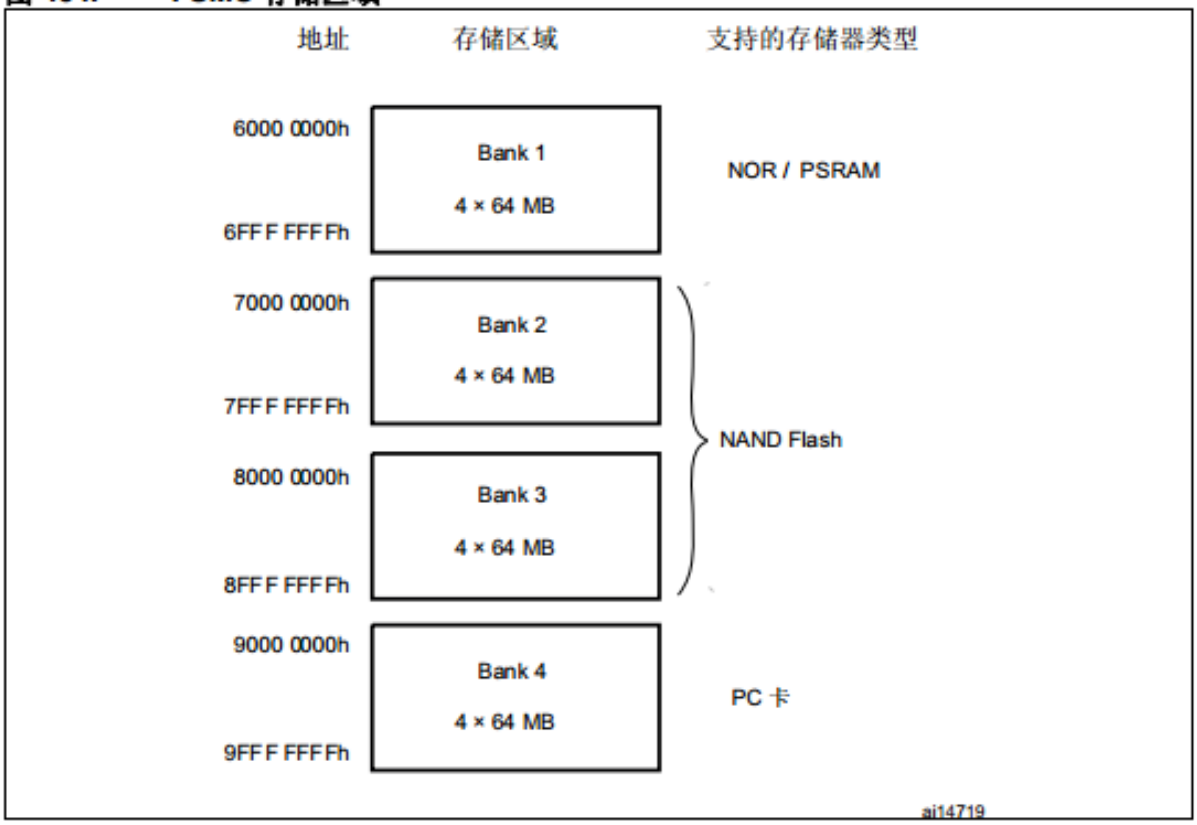
FSMC 能够连接同步、异步存储器和 16 位 PC 存储卡。所有外部存储器共享地址、数据和控制信号，但有各自的片选信号。FSMC 一次只能访问一个外部器件。

图 403. FSMC 框图



FSMC 框图上面是 FSMC 框图，从中可以看出，FSMC 将设备分为两部分，**NOR/PSRAM** 和 **NAND/PC**。两者有各自的控制器，共用地址线信号线等信号（见框图右边第二个大括号）。其中，控制信号还细分为三种：NOR/PSRAM 信号、NAND 信号、PC 智能卡信号。前面说到，操作 TFT LCD 就是操作驱动 IC 里面的显示缓冲区，跟操作 SRAM 一样。**因此我们将 TFT LCD 接在 NOR/PSRAM 存储器控制器上，使用 FSMC_NE4 作为片选信号**。但是 LCD 与 SRAM 还是有区别的，LCD 没有地址线，多一根 RS 线用于区分数据线上的是显示数据还是命令。如果我们将 RS 接到 FSMC 的某根地址线上，当我们操作一个地址，让这根线为高电平，就相当于向 LCD 写数据；当操作另外一个地址，是低电平时，则是命令了。那么这个地址如何定呢？继续往下看

图 404. FSMC 存储区域



FSMC

32.4.1 NOR/PSRAM 地址映射

HADDR[27:26] 位用于从*表 185* 中所示的四个存

表 185. NOR/PSRAM 存储区域选择

HADDR[27:26](1)	
00	存储区域
01	存储区域
10	存储区域
11	存储区域

1. HADDR 是 AHB 内部地址线，但也会参与对外部存储器

储存区域块划分上图是 FSMC 地址的划分，PSRAM 在 Bank1

HADDR[25:0] 包含外部存储器地址。由于 HADDR 为字节地址，而存储器按字寻址，所以根据存储器数据宽度不同，实际向存储器发送的地址也将有所不同，如下表所示。

表 186. 外部存储器地址

存储器宽度(1)	向存储器发出的数据地址	最大存储器容量（位）
8 位	HADDR[25:0]	64 MB x 8 = 512 Mb
16 位	HADDR[25:1] >> 1	64 MB/2 x 16 = 512 Mb

1. 如果外部存储器的宽度为 16 位，FSMC 将使用内部的 HADDR[25:1] 地址来作为对外部存储器的寻址地址 FSMC_A[24:00]。
无论外部存储器的宽度为 16 位还是 8 位，FSMC_A[0] 都应连接到外部存储器地址 A[0]。

地址映射 1 FSMC

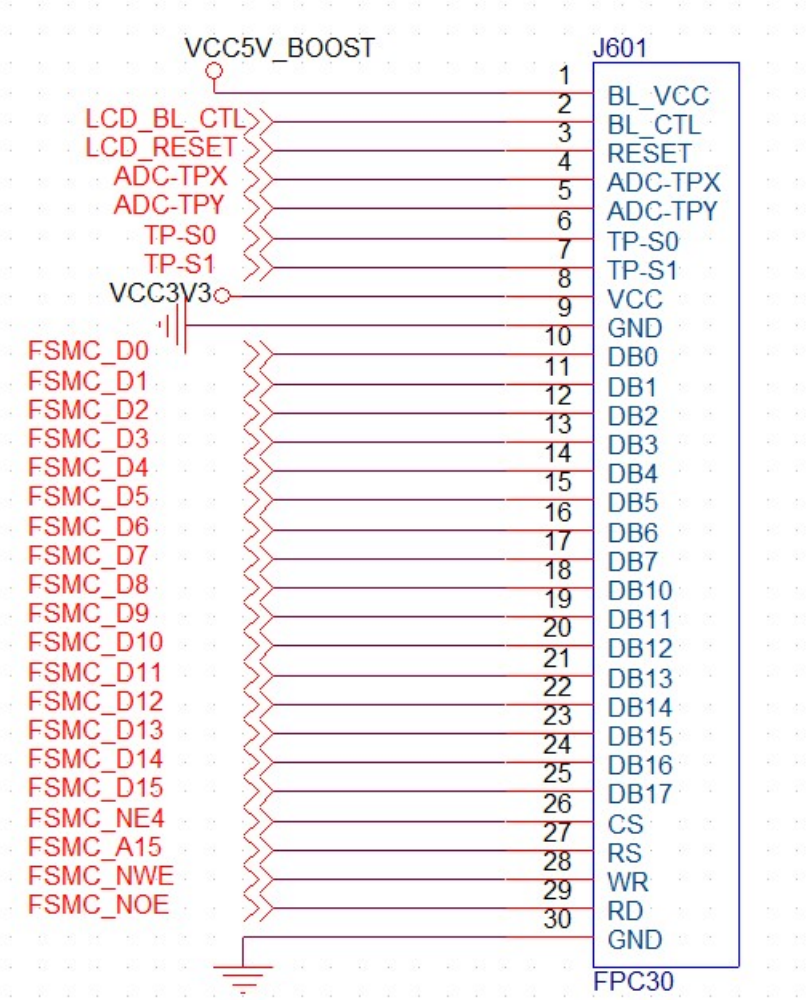
地址映射 2 在框图中，FSMC_NE 信号有 4 根，就分别对应其中的四个存储区域。每个区域 64M，偏移地址分别是 0x000 0000 0X400 0000 0X800 0000 0XC00 0000

对于 8 位跟 16 位，HADDR 内部连接不一样，但是 FSMC 与外部器件依然是 A0 连接 A0，也就是说：**位宽对总线地址线连接无影响，在内部控制器已经处理**。

> 现在我们将 RS 接在 FSMC_A15 上。使用的 FSMC_NE4，空间就是 BANK1 的存储区域 4。基地址是 0x6000 0000+0xc00 0000，RS 接在 FSMC_A15 上，那么在地址线上，我们希望的信号是 A15 出现 0 和 1 的变化，以便群命令和数据。但是，犹豫我们用的是 16 位位宽，地址会右移 1 位。也就是说，程序中操作 A15 地址的变化，才会在硬件信号 A15 上出现变化。那么区分命令和数据的地址位，就是 (1 0000 0000 0000 0000H). 从而数据地址是 0x6c01 xxxx，命令地址是 0x6c00 xxxx. 这只是其中的一组地址，只要地址的 A16 位是 0 和 1 的区别，并且地址在 0x6c00 0000~0x7000 0000 之间，都是符合的。

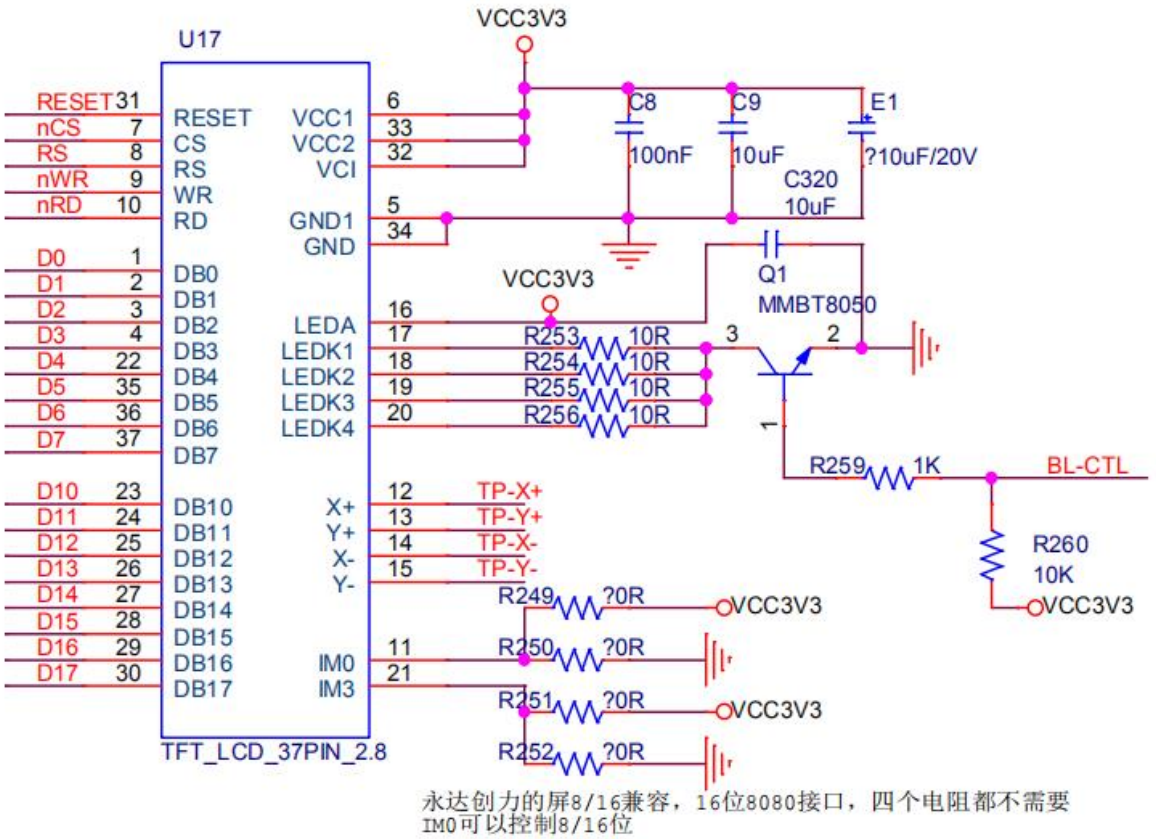
更多 FSMC 信息，例如寄存器使用、总线时序等，请参考芯片参考手册，同时分析源码。

20.3 原理图



- 接口 原理图

  1 脚背光电源（5V），2 脚背光控制 3 脚 LCD 复位信号 4、5、6、7 触摸屏控制信号 8，LCD 电源 9、30，地线 10~25 是双向数据线 26 CS，TFTLCD 片选信号。27 RS, 命令/数据标志（0，命令；1，数据）。28 WR，控制向 TFTLCD 写入数据。29 RD，从 TFTLCD 读取数据。



- Lcd 原理图

  原理图上图只是 LCD 的原理图，不包含触摸屏控制。左边信号是控制信号。右边是电源和背光。我们用的 LCD 默认就是 16BIT 并口了，不需要 IM 选择。

20.4 编码调试

知识点已经介绍完，下一步就是开始驱动编写，编写驱动前，先进行驱动设计。

20.4.1 驱动设计

通常的显示流程：

> APP 要在 LCD 上显示一个字符，调用**显示接口**，显示接口根据传入的**字符内码**查找**点阵字库**，组成显示数据，将显示数据**刷新到指定的 LCD**。

根据这个流程，显示要分两层，

1. 中间层是处理数据，类似 GUI 层，主要负责处理显示内容，包含查找字符点阵，画圆画线。
2. 底层就是 LCD 驱动，仅仅负责将显示数据写到 LCD，不关心显示内容。

这样理想的 LCD 驱动框架到后面我们在实现。当前主要是验证硬件，只做 LCD 驱动，并做几个简单的显示接口，可以显示英文，暂时不做中文显示。

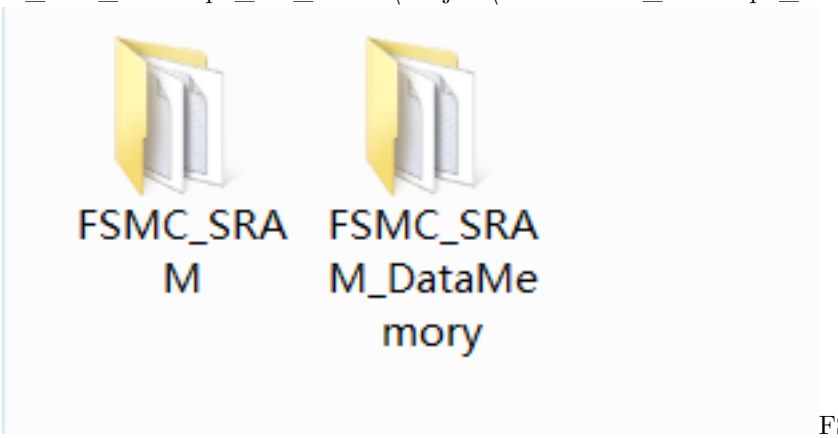
在进行 LCD 驱动设计时，请考虑以下问题：

1. lcd 驱动对下使用 FSMC 接口操作驱动 IC，对上应该提供什么接口？提供什么功能？对上，对谁？
2. 2.8 寸的 LCD，很可能有很多不同的驱动 IC 驱动，如何兼容不同的 IC？
3. 如果一个设备有两个 2.8 寸的 LCD，驱动如何设计？
4. 后续会做 0.96 OLED 跟 12864 COG LCD 驱动，跟当前的 TFT LCD 驱动什么关系？

   我认为 LCD 只是提供显示缓冲显示功能，或者说，提供一个个点显示功能。如果是黑白屏，就是一个点显示黑还是白，如果是彩屏，就是一个点显示什么颜色。LCD 驱动，不应该提供显示字符，划线等功能，因为这些功能应该是 GUI 负责。这是需要认识的一个很关键的地方。假设系统做各种字体的显示，中文，英文，阿拉伯文。首先要有字库，还需要有对字库的处理，将点阵转换为 LCD 显示缓冲。这些都不是 LCD 驱动负责。

20.4.2 FSMC 调试

我们在官方的例程中找 FSMC 的例程作为参考。在标准库目录下 STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Project\STM32F4xx_StdPeriph_Examples\FSMC 下有



两个例程 FSMC st 例程文件夹通过读文件夹内的 readme 文件可知道，第一个例程是如何通过读写将一个外部 SRAM 作为数据存储，第二个例程是如何将外部 SRAM 做为一个外部的 memory，包括作为堆和栈。很明显，我们应该参考第一个例程。

@par How to use it ? In order to make the program work, you must do the following:

- Copy all source files from this example folder to the template folder under Project\STM32F4xx_StdPeriph_Templates
- Open your preferred toolchain
- Select the project workspace related to the STM32F40_41xxx device and add the following files in the project source list:
  - Utilities\STM32_EVAL\STM3240_41_G_EVAL\stm324xg_eval.c
  - Utilities\STM32_EVAL\STM3240_41_G_EVAL\stm324xg_eval_fsmc_sram.c
- Rebuild all files and load your image into target memory

- Run the example

根据 SDIO 例程移植的经验，很容易看出，stm324xg_eval_fsmc_sram.c 才是我们要的源码。当然，例程里面的其他文件我们也需要跟我们工程的文件对比一下。

```
/* Initialize the SRAM memory */
 SRAM_Init();

 /* Fill the buffer to send */
 Fill_Buffer(aTxBuffer, BUFFER_SIZE, 0x250F);

 /* Write data to the SRAM memory */
 SRAM_WriteBuffer(aTxBuffer, WRITE_READ_ADDR, BUFFER_SIZE);

 /* Read back data from the SRAM memory */
 SRAM_ReadBuffer(aRxBuffer, WRITE_READ_ADDR, BUFFER_SIZE);

 /* Check the SRAM memory content correctness */
 for (uwIndex = 0; (uwIndex < BUFFER_SIZE) && (uwWriteReadStatus_SRAM == 0); uwIndex++)
 {
   if (aRxBuffer[uwIndex] != aTxBuffer[uwIndex])
   {
     uwWriteReadStatus_SRAM++;
   }
 }
```

main.c，很简单，初始化 SRAM，写，读，校验。首先将 stm324xg_eval_fsmc_sram.h 跟 stm324xg_eval_fsmc_sram.c 拷贝到 mcu_dev 目录下，并添加到 MDK 跟 SI 的工程。源代码就只有三个函数，就是在 main 函数中调用的函数。在.c 文件中包含库文件，.h 文件中对 stm324xg_eval.h 的包含去掉。编译通过。

```
/* Includes -----*/
#include "stm32f4xx.h"
#include "stm324xg_eval_fsmc_sram.h"
```

20.4.3 显示 IC 驱动调试

虽然说我们有驱动 IC 规格书，通常上还是需要 LCD 厂家提供驱动例程，特别是初始化代码。创建驱动文件 dev_ILI9341.c 跟 dev_ILI9341.h，保存在 board_dev 文件夹内。

- 按照经验，首先调试到成功读取驱动 IC 的 ID。

先根据硬件，修改 FSMC 驱动里面的配置。我们参考 SRAM_Init 函数，重写一个 LCD FSMC 初始化函数

```
/**
 *@brief:      mcu_fsmc_lcd_Init
 *@details:    LCD 使用的 FSMC 初始化
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
void mcu_fsmc_lcd_Init(void)
{
  FSMC_NORSRAMInitTypeDef  FSMC_NORSRAMInitStructure;
  FSMC_NORSRAMTimingInitTypeDef  w,r;
  GPIO_InitTypeDef GPIO_InitStructure;

  /* Enable GPIOs clock */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD | RCC_AHB1Periph_GPIOE |
                                                RCC_AHB1Periph_GPIOG, ENABLE);

  /* Enable FSMC clock */
  RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC, ENABLE);

/*-- GPIOs Configuration -----------------------------------------------------*/
/*
 +-------------------+--------------------+------------------+------------------+
PD0        <-> FSMC_D2
PD1        <-> FSMC_D3
PD4        <-> FSMC_NOE
PD5        <-> FSMC_NWE
PD8        <-> FSMC_D13
PD9        <-> FSMC_D14
PD10 <-> FSMC_D15
PD14 <-> FSMC_D0
PD15 <-> FSMC_D1

PE7        <-> FSMC_D4
PE8        <-> FSMC_D5
PE9        <-> FSMC_D6
PE10 <-> FSMC_D7
PE11 <-> FSMC_D8
PE12 <-> FSMC_D9
PE13 <-> FSMC_D10
```

(continues on next page)

(continued from previous page)

```
PE14 <-> FSMC_D11
PE15 <-> FSMC_D12

PG5 <-> FSMC_A15 |
PG12 <-> FSMC_NE4 |
*/

  /* GPIOD configuration */
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource0, GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource1, GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource4, GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource5, GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource8, GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource9, GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource10, GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource14, GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource15, GPIO_AF_FSMC);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0| GPIO_Pin_1|GPIO_Pin_4|GPIO_Pin_5 |
                                GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_14|GPIO_Pin_
↪15;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd        = GPIO_PuPd_NOPULL;

  GPIO_Init(GPIOD, &GPIO_InitStructure);

  /* GPIOE configuration */
  GPIO_PinAFConfig(GPIOE, GPIO_PinSource7 , GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOE, GPIO_PinSource8 , GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOE, GPIO_PinSource9 , GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOE, GPIO_PinSource10 , GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOE, GPIO_PinSource11 , GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOE, GPIO_PinSource12 , GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOE, GPIO_PinSource13 , GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOE, GPIO_PinSource14 , GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOE, GPIO_PinSource15 , GPIO_AF_FSMC);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7 |
```

(continues on next page)

(continued from previous page)

```
                                GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_11|
                                GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15;

  GPIO_Init(GPIOE, &GPIO_InitStructure);

  /* GPIOG configuration */
  GPIO_PinAFConfig(GPIOG, GPIO_PinSource5 , GPIO_AF_FSMC);
  GPIO_PinAFConfig(GPIOG, GPIO_PinSource12 , GPIO_AF_FSMC);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5  |GPIO_Pin_12;

  GPIO_Init(GPIOG, &GPIO_InitStructure);

/*-- FSMC Configuration ------------------------------------------------------*/
  w.FSMC_AddressSetupTime = 15;
  w.FSMC_AddressHoldTime = 0;
  w.FSMC_DataSetupTime = 15;
  w.FSMC_BusTurnAroundDuration = 0;
  w.FSMC_CLKDivision = 0;
  w.FSMC_DataLatency = 0;
  w.FSMC_AccessMode = FSMC_AccessMode_A;

  r.FSMC_AddressSetupTime = 16;
  r.FSMC_AddressHoldTime = 0;
  r.FSMC_DataSetupTime = 24;
  r.FSMC_BusTurnAroundDuration = 0;
  r.FSMC_CLKDivision = 0;
  r.FSMC_DataLatency = 0;
  r.FSMC_AccessMode = FSMC_AccessMode_A;

  FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM4;
  FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable;
  FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_SRAM;
  FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b;
  FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable;
  FSMC_NORSRAMInitStructure.FSMC_AsynchronousWait = FSMC_AsynchronousWait_Disable;
  FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;
  FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
  FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalActive_
↪BeforeWaitState;
```

(continues on next page)

(continued from previous page)

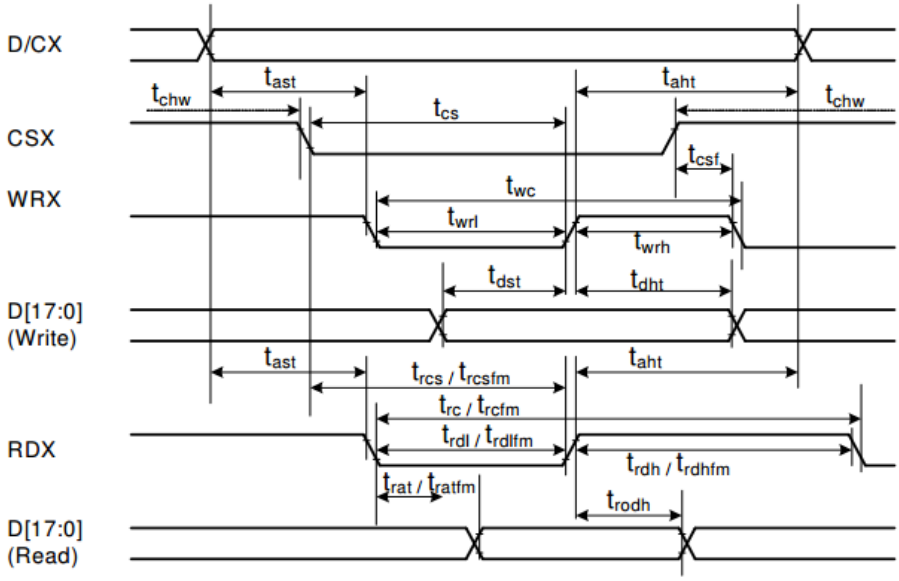
```
  FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
  FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
  FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Enable;
  FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
  FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &r;
  FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &w;

  FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);

  /*!< Enable FSMC Bank1_SRAM4 Bank */
  FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM4, ENABLE);

}
```

15/19 行，打开设备时钟。48~92，配置对应的 IO 口为 FSMC 功能 94~109，设置 FSMC 时序 111，选择 BANK 113，设置类型，LCD 可以设置为 SRAM。114，设置数据宽度。

在《ILI9341_DS_V1.09_20110315.pdf》第 18 章，有对时序说明，例如下图就是 8080-I 时序图。

18.3 AC Characteristics

18.3.1 Display Parallel 18/16/9/8-bit Interface Timing Characteristics (8080- I system)



9341

lcd 时序

配置好之后，编写 ILI6341 驱动初始化，读取设备 ID。读取 ID 的命令为 0XD3，具体见下图。

8.3.23. Read ID4 (D3h)

D3h	RDID4 (Read ID4)												
	D/CX	RDX	WRX	D17-8	D7	D6	D5	D4	D3	D2	D1	D0	HEX
Command	0	1	↑	XX	1	1	0	1	0	0	1	1	D3h
1st Parameter	1	↑	1	XX	X	X	X	X	X	X	X	X	X
2ndParameter	1	↑	1	XX	0	0	0	0	0	0	0	0	00h
3rdParameter	1	↑	1	XX	1	0	0	1	0	0	1	1	93h
4th Parameter	1	↑	1	XX	0	1	0	0	0	0	0	1	41h
Description	Read IC device code. The 1st parameter is dummy read period. The 2nd parameter means the IC version. The 3rd and 4th parameter mean the IC model name.												
Restriction	EXTC should be high to enable this command												
Register Availability	Status: Normal Mode ON, Idle Mode OFF, Sleep OUT — Availability: Yes Normal Mode ON, Idle Mode ON, Sleep OUT — Yes Partial Mode ON, Idle Mode OFF, Sleep OUT — Yes Partial Mode ON, Idle Mode ON, Sleep OUT — Yes Sleep IN — Yes												
Default	Status: Power ON Sequence — Default Value: 24'h009341h SW Reset — 24'h009341h HW Reset — 24'h009341h												

9341

读 ID 命令

初始化代码如下（只是读 ID，不是完整初始化）

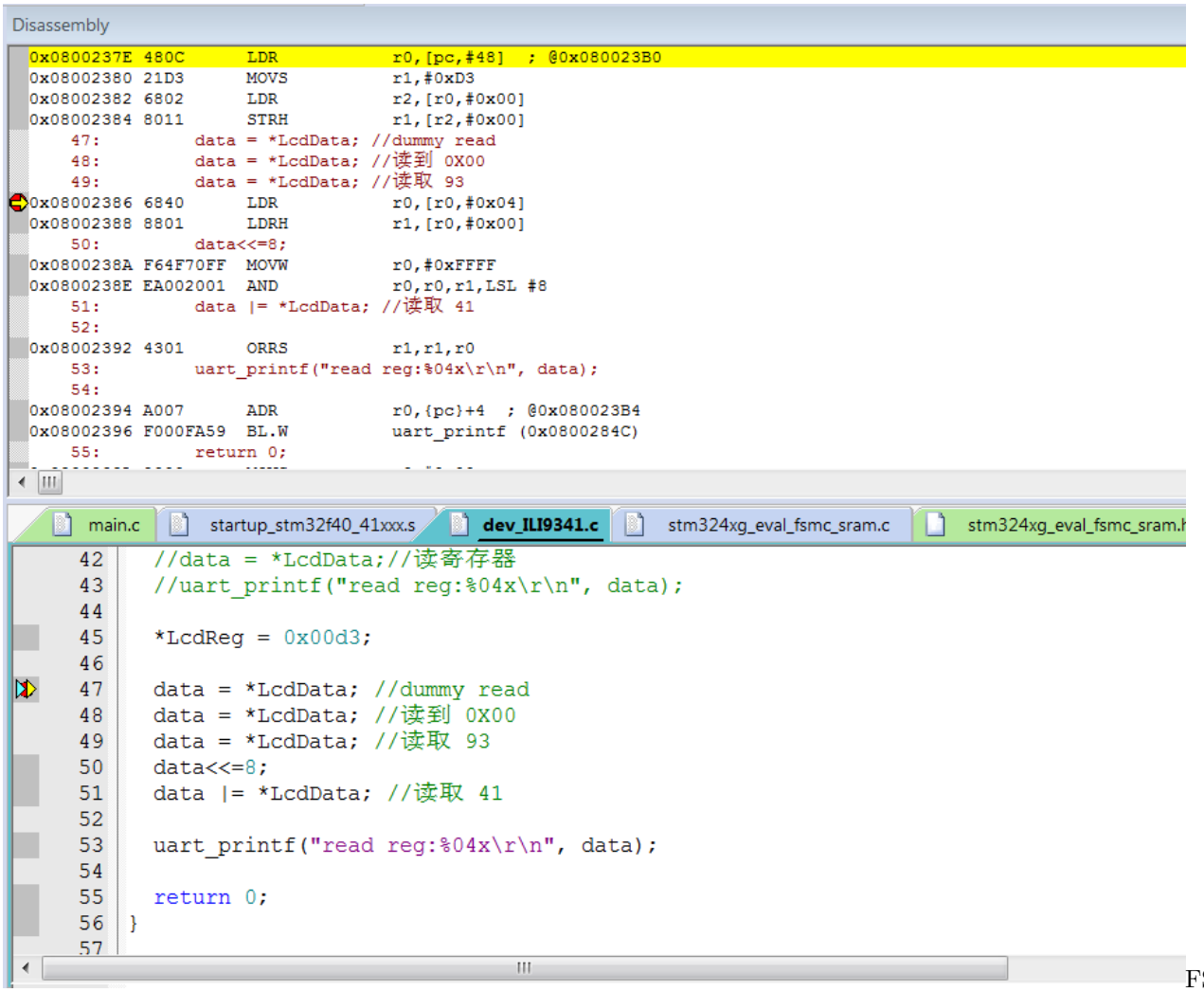
```
u16 *LcdReg = (u16*)0x6C000000;
u16 *LcdData = (u16*)0x6C010000;

s32 dev_ILI9341_init(void)
{
        u16 data;

        //初始化背光控制管脚

        //初始 FSMC
        mcu_fsmc_lcd_Init();
        uart_printf("init finish!\r\n");
        //延时 50 毫秒
        Delay(5);
```

(continues on next page)

(continued from previous page)

```
  /*
  LcdReg = 0x0000;//首先使用命令写入要读的寄存器的地址
        data = *LcdData;//读寄存器
        uart_printf("read reg:%04x\r\n", data);
  */
        *LcdReg = 0x00d3;

        data = *LcdData; //dummy read
        data = *LcdData; //读到 0X00
        data = *LcdData; //读取 93
        data<<=8;
        data |= *LcdData; //读取 41

        uart_printf("read reg:%04x\r\n", data);

        return 0;
}
```

为了方便操作, 将命令跟数据地址定义为一个 u16 指针, 初始化为对应的地址。程序首先初始化 FSMC, 然后按照 D3 命令流程进行操作。将 dev_ILI9341_init 添加到 main 函数中, 对 LCD 进行初始化。下载到硬件后, 发现读到的 ID 是 0X00。调试很久, 都没能读取 ID。在网上查询, 发现一个信息, 就是 MDK 会对程序进行优化, 需要增加一些额外的代码。我们先看下是不是程序被优化掉了。用 CMSIS DAP 进行调试, 在函数加断点。



FSMC 读数据被优化如图，47/48/49 三行读数据的代码，编译后被优化成读一次了。编译器估计是想，连续读三次没有意义。如何防止优化呢？**不需要添加额外代码**，编译器功能是很强大的，只需要在定义指针处，添加 **volatile** 关键字即可。如下

```
volatile u16 *LcdReg = (u16*)0x6C000000;
volatile u16 *LcdData = (u16*)0x6C010000;
```

编译后使用 CMSIS DAP 进行调试，查看汇编代码，区别很明显，每次读操作都执行了。

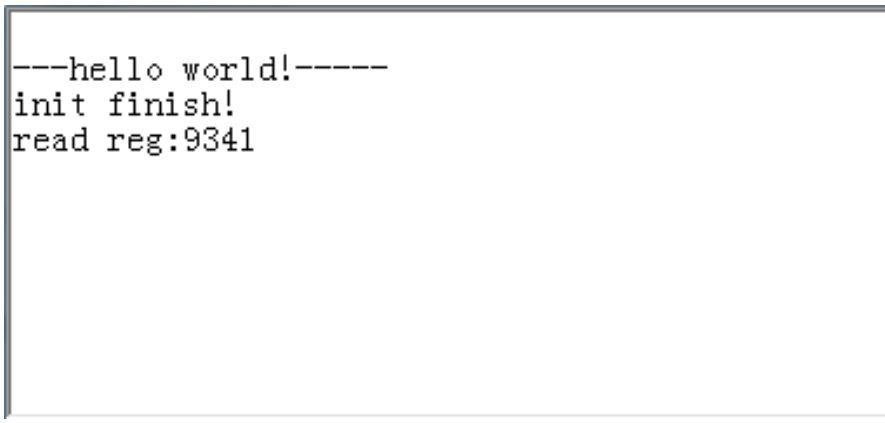
Disassembly

```
0x08002380 21D3      MOVS           r1,#0xD3
0x08002382 6802      LDR            r2,[r0,#0x00]
0x08002384 8011      STRH           r1,[r2,#0x00]
    45:          data = *LcdData; //dummy read
0x08002386 6840      LDR            r0,[r0,#0x04]
0x08002388 8801      LDRH           r1,[r0,#0x00]
    46:          data = *LcdData; //读到 0X00
0x0800238A 8801      LDRH           r1,[r0,#0x00]
    47:          data = *LcdData; //读取 93
0x0800238C 8801      LDRH           r1,[r0,#0x00]
    48:          data<<=8;
0x0800238E F64F72FF  MOVW           r2,#0xFFFF
0x08002392 EA022201  AND            r2,r2,r1,LSL #8
    49:          data |= *LcdData; //读取 41
    50:
0x08002396 8801      LDRH           r1,[r0,#0x00]
    51:          uart_printf("read reg:%04x\r\n", data);
    52:
0x08002398 A007      ADR            r0,{pc}+4   ; @0x080023B8
0x0800239A 4311      ORRS           r1,r1,r2
```

dev_ILI9341.c | stm324xg_eval_sdio_sd.c | stm32f4xx_rcc.c

```
44
45    data = *LcdData; //dummy read
46    data = *LcdData; //读到 0X00
47    data = *LcdData; //读取 93
48    data<<=8;
49    data |= *LcdData; //读取 41
50
51    uart_printf("read reg:%04x\r\n", data);
52
53    return 0;
54  }
55
56
```

修



改后不优化调试信息也可以看到，成功读取到 ID ID 成功对于 TFT LCD 与 FSMC 的调试，在硬件上已经完成。下一步就是如何实现 TFTLCD 提供给上一层的功能与接口了。

20.4.4 驱动接口定义

在参考屏厂提供的代码时发现，为了兼容多种驱动 IC，他们的代码变得如此混乱。

我们决定重新设计。

通过使用**函数指针**的方式，调用不同的驱动 IC 函数，而不是在一个函数内通过 if-else 选择不同驱动 IC 代码。每个 LCD 驱动对外接口和功能设计，经讨论，主要提供以下功能:

```
    s32 (*init)(void);
    s32 (*draw_point)(u16 x, u16 y, u16 color);
    s32 (*color_fill)(u16 sx,u16 ex,u16 sy,u16 ey, u16 color);
    s32 (*fill)(u16 sx,u16 ex,u16 sy,u16 ey,u16 *color);
    s32 (*onoff)(u8 sta);
    s32 (*prepare_display)(u16 sx, u16 ex, u16 sy, u16 ey);
    void (*set_dir)(u8 scan_dir);
    void (*backlight)(u8 sta);
```

1. 初始化
2. 画点
3. 将一个矩形区域显示为同一种颜色。
4. 将一个区域根据输入的颜色显示，点颜色不一样。
5. 开关显示
6. 准备显示区域（摄像头显示需要，其实就是接口 4 的准备过程）
7. 设置扫描方向
8. 背光控制

20.4.5 坐标定义

在开始写显示程序是，我们先要对坐标进行定义。并且将用户跟驱动 IC 进行统一。

- 用户角度

有横屏竖屏，无论横竖，人眼角度左上角是原点。坐标使用 XY 轴概念。例如：

> 一个 320*240 的彩屏，竖屏的时候，X 轴就是 240 短边，Y 轴就是 320 长边。如果是一个 COG12864 的黑白屏，默认就是横屏，128 长边就是 X 轴，64 短边就是 Y 轴，如果设置为竖屏，则短边是 X 轴。

- 驱动 IC

没有横屏竖屏的概念，只有扫描方向，也不使用 XY 轴，而是使用 **COLUM** 跟 **PAGE**（我们简称 CP 坐标概念）。不同的扫描方向，不会改变原点位置，也不改变 XY 方向，只是改变了显示数据的组织顺序。不同的驱动 IC，能设置的扫描方向不一样，同样的扫描方向设置，真正出来的效果也不一样。**因此每个驱动 IC 都需要将 XY 轴转换为自己的 CP 坐标**。

所有对外接口提供的是 XY 轴坐标，驱动内部自行转换。

- 参数

所有函数都是给人用的，所以坐标使用 XY 坐标。按照下面参数顺序：sx：X 轴起始，ex：X 轴结束坐标；sy：Y 轴起始，ey：Y 轴结束坐标；例如：

```
s32 drv_ILI9325_color_fill(u16 sx,u16 ex,u16 sy,u16 ey,u16 color)
```

我们定义了一个函数设置屏幕方向：

```
void dev_lcd_setdir(u8 dir, u8 scan_dir)
```

dir 就是横屏还是竖屏，scan_dir 就是扫描方向。这个函数名并没有使用驱动 IC 的关键字，就像前面说的，**横屏还是竖屏不是驱动 IC 关心的事**。在这个函数内，根据横屏竖屏对 scan_dir 进行映射，映射转换之后再设置驱动 IC。

20.4.6 显示字符

显示字符函数是从 TSLIB 移植过来的，只是为了显示基本英文字符。每个像素对应一个 GRAM 内存，只要按照字符排列将对应的 GRAM 设置为对应颜色，就是显示字符了。

20.4.7 液晶识别

很多开发板提供液晶驱动，都是在初始化函数内尝试读 ID，根据读到的 ID 进行 **if-else** 选择不同驱动 IC 的初始化。这样的代码组织非诚不好。我们的组织方式如下：

```
s32 dev_lcd_init(void)
{
        s32 ret = -1;

        /* 初始化 8080 接口，包括背光信号 */
        bus_8080interface_init();

        if(ret != 0)
        {
                /* 尝试初始化 9341*/
                ret = drv_ILI9341_init();
                if(ret == 0)
                {
                        LCD.drv = &TftLcdILI9341Drv;//将 9341 驱动赋值到 LCD
                        LCD.dir = H_LCD;//默认竖屏
                        LCD.height = 320;
                        LCD.width = 240;
                }
        }

        #ifdef TFT_LCD_DRIVER_9325
        if(ret != 0)
        {
                /* 尝试初始化 9325 */
                ret = drv_ILI9325_init();
                if(ret == 0)
                {
                        LCD.drv = &TftLcdILI9325Drv;
                        LCD.dir = H_LCD;
                        LCD.height = 320;
                        LCD.width = 240;
                }
        }
        #endif
        /* 设置屏幕方向，扫描方向 */
        dev_lcd_setdir(H_LCD, L2R_U2D);
        LCD.drv->onoff(1);//打开显示
        bus_8080_lcd_bl(1);//打开背光
        LCD.drv->color_fill(0, LCD.width, 0, LCD.height, YELLOW);
```

(continues on next page)

(continued from previous page)

```
        return 0;
}
```

在 **LCD 初始化函数**中调用**驱动 IC 初始化函数**，如果返回成功，则说明是这个驱动 IC 的 LCD。这样的组织方式有什么不一样呢?

1. 有层次，LCD_INIT 不知道各个驱动 IC 是怎么初始化的，只是调用函数。
2. 解耦合，每个驱动 IC 的初始化，只是识别自己的 ID，不与其它驱动 IC 的代码纠缠在一起。

20.4.8 测试

测试函数如下，LCD 的初始化不在测试函数初始化，在 main 函数中初始化。

```
void dev_lcd_test(void)
{

        while(1)
        {
                LCD.drv->color_fill(0,LCD.width,0,LCD.height,BLUE);
                Delay(100);
                LCD.drv->color_fill(0,LCD.width/2,0,LCD.height/2,RED);
                Delay(100);
                LCD.drv->color_fill(0,LCD.width/4,0,LCD.height/4,GREEN);
                Delay(100);

                put_string_center (LCD.width/2+50, LCD.height/2+50,
                           "ADCD WUJIQUE !", 0xF800);
                Delay(100);
        }

}
```

到此，LCD 调试结束，至于各种花哨的显示，属于应用范畴。各位可以尝试修改 dev_lcd_setdir(H_LCD, L2R_U2D)；看看不同的屏幕方向，不同的扫描方向，都是什么效果。**要看清扫描方向的区别，请在在 fill 函数内加延时**

20.4.9 总结

1. 本文档除了介绍 LCD 如何使用，更加希望大家能知道 LCD 驱动架构如何设计。
2. 本文档对应的代码并不是最完善的代码，只是为了测试硬件。

如果要用于实际项目，请到 github 或者官网下载最新代码。最新代码兼容了各种 LCD。并且有一套很好的代码架构。关于 LCD 驱动设计，同时可以参考我们编写的《LCD 驱动应该怎么写？.pdf》

20.5 end

CHAPTER 21

ADC-TSLIB-电阻式触摸屏调试

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

上一节已经将 TFT LCD 调通，这块 LCD 表面带了一块四线电阻式触摸屏。现在，就让我们来调试触摸功能。为了能深刻领会触摸屏检测方法，我们这次用内部 ADC 进行触摸屏检测。下一节我们再使用 XPT2046 控制触摸屏。

21.1 ADC

前面我们已经调试了 DAC，ADC 就是反过来：**将 IO 口上的电压（模拟信号）转换为数字值**。同样有参考电压，ADC 也有位数（精度）。百度百科：

> ADC，Analog-to-Digital Converter 的缩写，指模/数转换器或者模数转换器。是指将连续变化的模拟信号转换为离散的数字信号的器件。真实世界的模拟信号，例如温度、压力、声音或者图像等，需要转换成更容易储存、处理和发射的数字形式。模/数转换器可以实现这个功能，在各种不同的产品中都可以找到它的身影。与之相对应的 DAC，Digital-to-Analog Converter，它是 ADC 模数转换的逆向过程

21.2 STM32 ADC

21.2.1 简介

12 位 ADC 是逐次趋近型模数转换器。它具有多达 19 个复用通道，可测量来自 16 个外部源、两个内部源和 VBAT 通道的信号。这些通道的 A/D 转换可在单次、连续、扫描或不连续采样模式下进行。ADC 的结果存储在一个左对齐或右对齐的 16 位寄存器中。

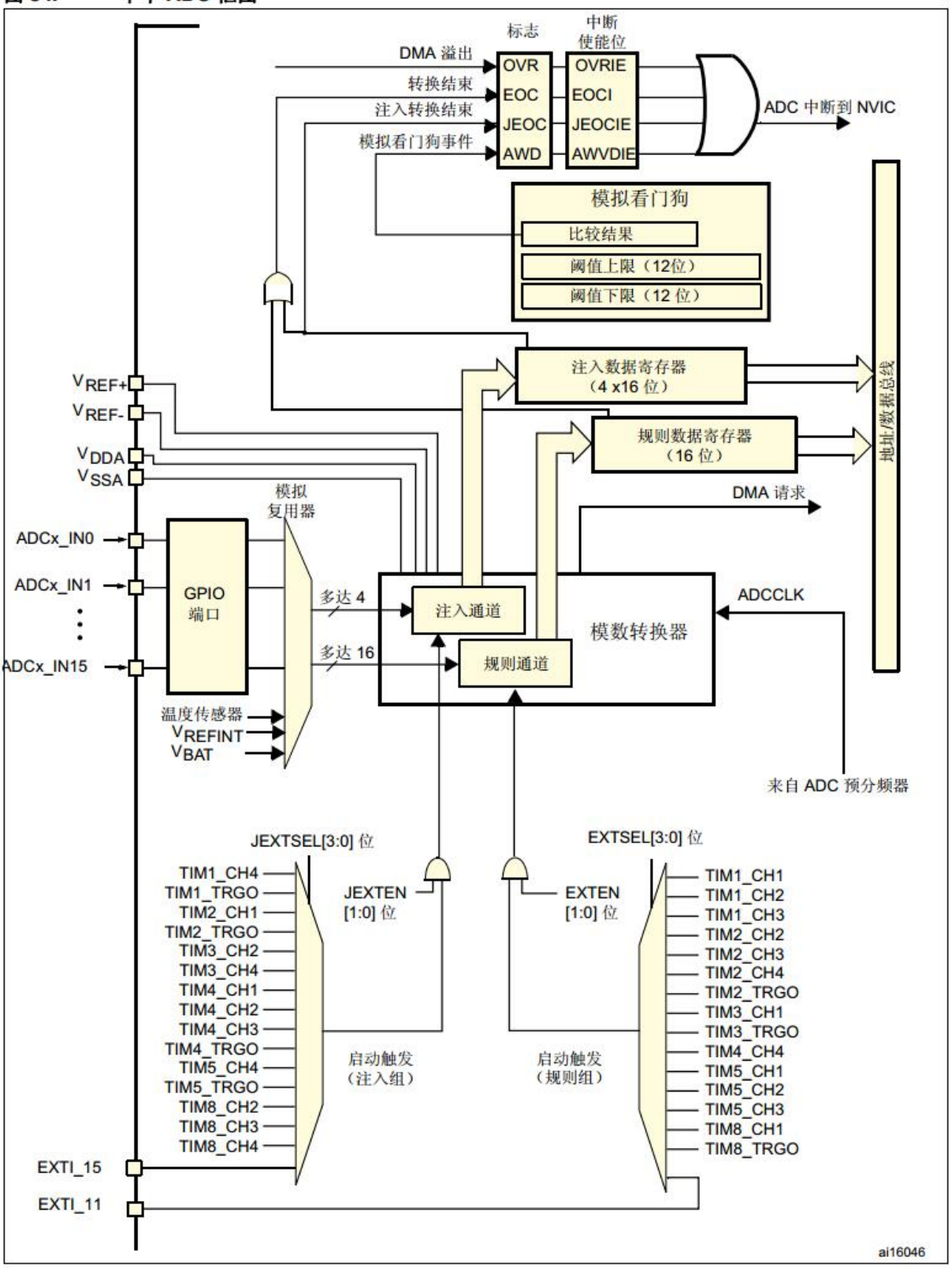
21.2.2 特性

- 可配置 12 位、10 位、8 位或 6 位分辨率
- 在转换结束、注入转换结束以及发生模拟看门狗或溢出事件
- 单次和连续转换模式
- 用于自动将通道 0 转换为通道“n”的扫描模式
- 数据对齐以保持内置数据一致性
- 可独立设置各通道采样时间
- 外部触发器选项，可为规则转换和注入转换配置极性
- 不连续采样模式
- 双重/三重模式（具有 2 个或更多 ADC 的器件提供）
- 双重/三重 ADC 模式下可配置的 DMA 数据存储
- 双重/三重交替模式下可配置的转换间延迟
- ADC 转换类型（参见数据手册）
- ADC 电源要求：全速运行时为 2.4 V 到 3.6 V，慢速运行时
- ADC 输入范围：$V_{REF-} \leq V_{IN} \leq V_{REF+}$
- 规则通道转换期间可产生 DMA 请求

STM32 的 ADC 功能强大，主要特性如下：
特性本次我们只会用到最简单的单次转换功能。

21.2.3 框图

图 34. 单个 ADC 框图



ADC 框图从框图可以看出：

1. 有两种触发：注入组、规则组。所谓的注入组，就像中断，可以打断规则组的转换。
2. 支持 DMA 和中断。
3. 左边输入多达 16 路。

在屋脊雀 STM32F407 硬件，使用 PB0 跟 PB1 作为 ADC 转换输入。**查数据手册可知 PB0 是 ADC12_IN8，PB1 是 ADC12_IN9**。

PB0	I/O	FT	(4)	TIM3_CH3 / TIM8_CH2N/ OTG_HS_ULPI_D1/ ETH_MII_RXD2 / TIM1_CH2N/ EVENTOUT	ADC12_IN8
PB1	I/O	FT	(4)	TIM3_CH4 / TIM8_CH3N/ OTG_HS_ULPI_D2/ ETH_MII_RXD3 / OTG_HS_INTN / TIM1_CH3N/ EVENTOUT	ADC12_IN9

PB 口

21.3 电阻触摸屏

四线电阻屏部分内容参考网络文章电阻式触摸屏的基本结构和驱动原理，网址 http://article.cechina.cn/2009-03/200937110119.htm

> 四线电阻式触摸屏的结构如图 1，在玻璃或丙烯酸基板上覆盖有两层透平，均匀导电的 ITO 层，分别做为 X 电极和 Y 电极，它们之间由均匀排列的透明格点分开绝缘。其中下层的 ITO 与玻璃基板附着，上层的 ITO 附着在 PET 薄膜上。X 电极和 Y 电极的正负端由“导电条”（图中黑色条形部分）分别从两端引出，且 X 电极和 Y 电极导电条的位置相互垂直。引出端 X-，X+，Y-，Y+ 一共四条线，这就是四线电阻式触摸屏名称的由来。

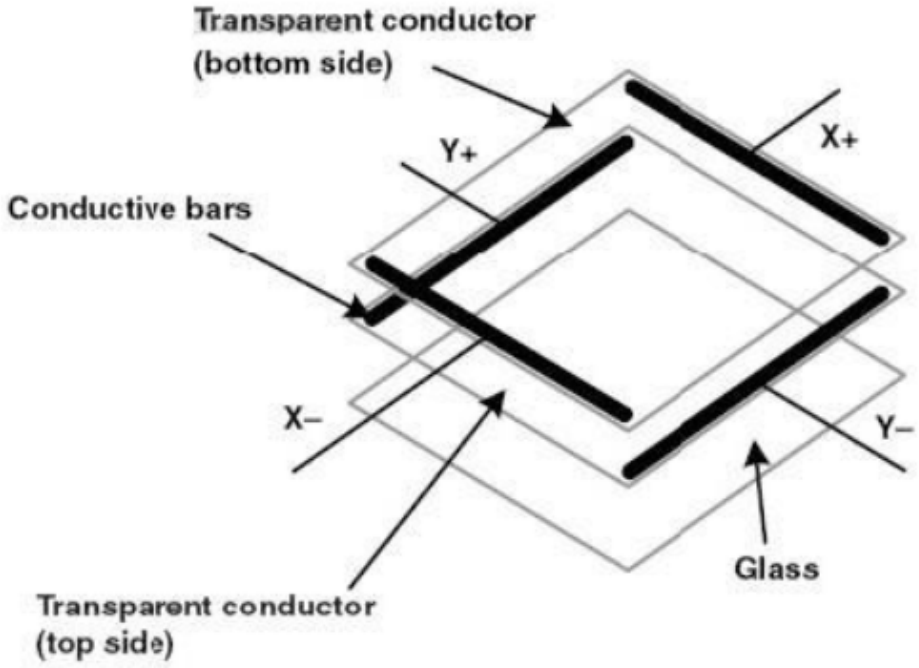


图1

触摸屏原理 1 当有物体接触触摸屏表面并施以一定的压力时，上层的 ITO 导电层发生形变与下层 ITO 发生接触，如下图 2:

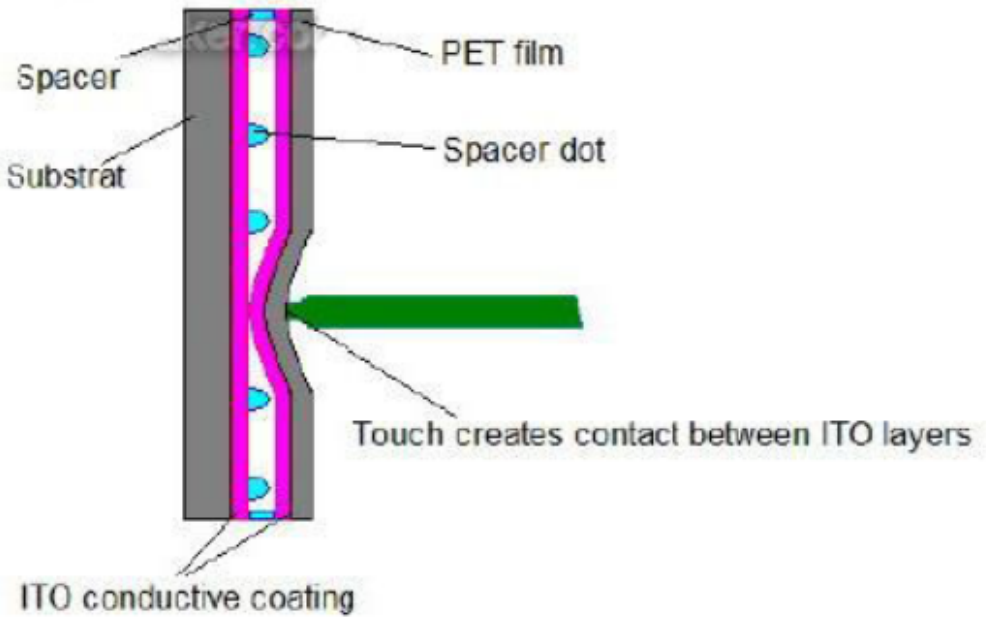


图2

触

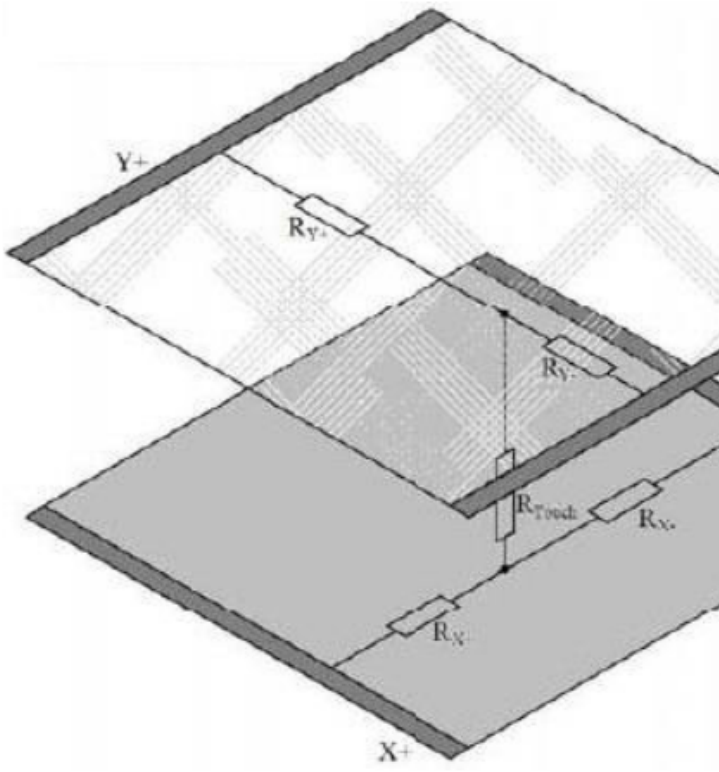
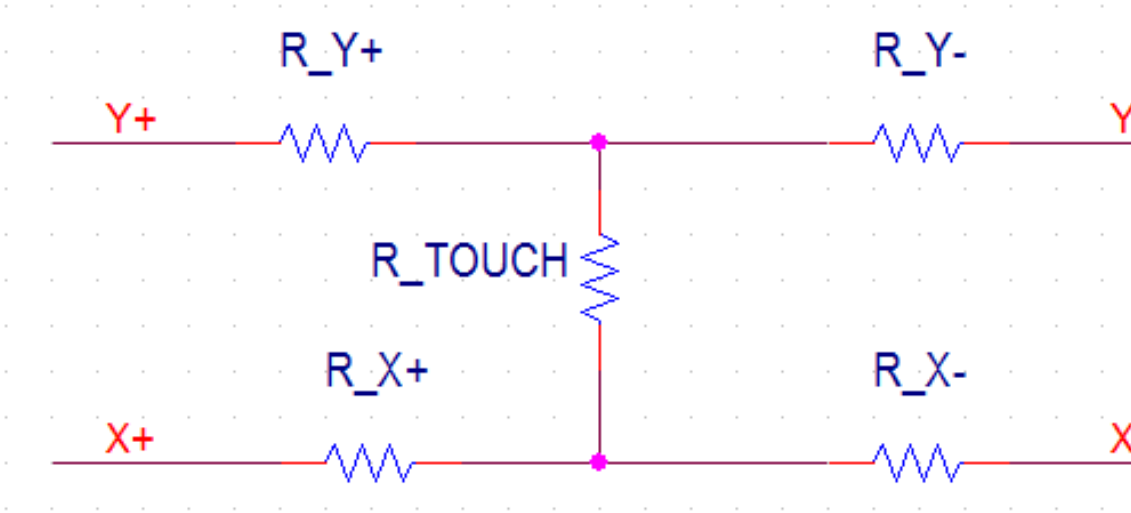


图3

摸屏压原理该结构可以等效为相应的电路，如图 3：
摸屏等效电路



我们将图三转化为原理图
摸屏等效电路原理图

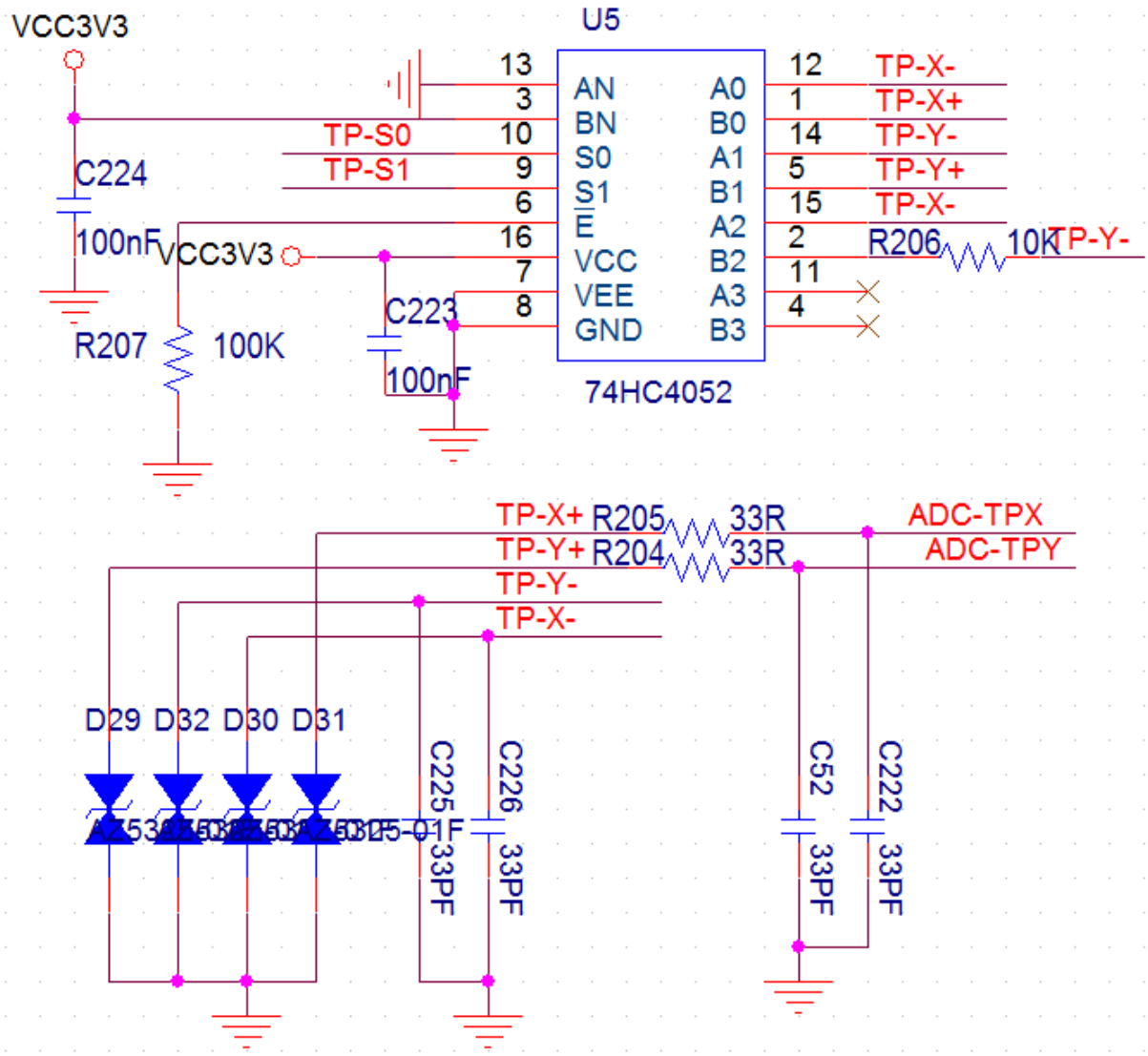
1. 在 Y+ 加上 VCC，Y-接地，不同的触摸点，R_Y+ 与 R_Y-的分压就不一样，通过 X+ 或者 X-，就可以检测出 Y 轴上的分压值。
2. 在 X+ 加上 VCC，X-接地，不同的触摸点，R_X+ 与 R_X-的分压就不一样，通过 Y+ 或者 Y-，就

可以检测出 X 轴上的分压值。

3. 在 X+ 加上 VCC，Y+ 接地，不同的触摸压力，R_touch 阻值不一样，压力越大，阻值越小，通过 X-和 Y-，就可以检测出 R_TOUCH 两端的分压值。

通过上面列出三步，我们就可以计算出一次触摸的触摸力度、X 坐标、Y 坐标。这就是四线触摸屏的基本原理。

21.4 方案原理说明



屋脊雀触摸屏电路上图为我们使用的 LCD 上的触摸屏处理电路路。使用了一片 74HC4052 “两路四选一模拟开关”作为电子开关，用于切换触摸屏四根线连接到哪里：电压、地、ADC。

TP-X-、TP-X+、TP-Y-、TP-Y+ 就是电阻屏的四根线。ADC-TPX、ADC-TPY 则为两个 ADC 的输入信号。TP-S0、TP-S1 为电子开关选择信号。

74HC4052 的逻辑

> 在第 6 脚使能脚为 0 时，S1=0，S0=0：AN 连到 A0，BN 连到 B0；S1=0，S0=1：AN 连到 A1，BN 连到 B1；S1=1，S0=0：AN 连到 A2，BN 连到 B2；S1=1，S0=1：AN 连到 A3，BN 连到 B3；使能脚为 1，则全部断开。

根据上一节分析的触摸屏原理，要获取触摸信号，我们程序的流程如下：

1. s1=0，s0=0，这时 TP-X-接地，TP-X+ 接 VCC3V3，通过 ADC-TPY 对 TP-Y+ 上的电压转换，就可以得到 X 轴上的坐标。
2. s1=0，s0=1，这时 TP-Y-接地，TP-Y+ 接 VCC3V3，通过 ADC-TPX 对 TP-X+ 上的电压转换，就可以得到 Y 轴上的坐标。
3. s1=1，s0=0，这时 TP-X-接地，TP-Y-接 VCC3V3，通过 ADC-TPY 对 TP-Y+ 上的电压转换，ADC-TPX 对 TP-X+ 上的电压转换，就可以算出 R_TOUCH 上的压降。因为 R_TOUCH 较小，为了防止当 R_Y-跟 R_X-也叫小的时候产生大电流，在 VCC3V3 跟 TP-Y-之间串接了一个 10K 电阻。

21.5 TsLib

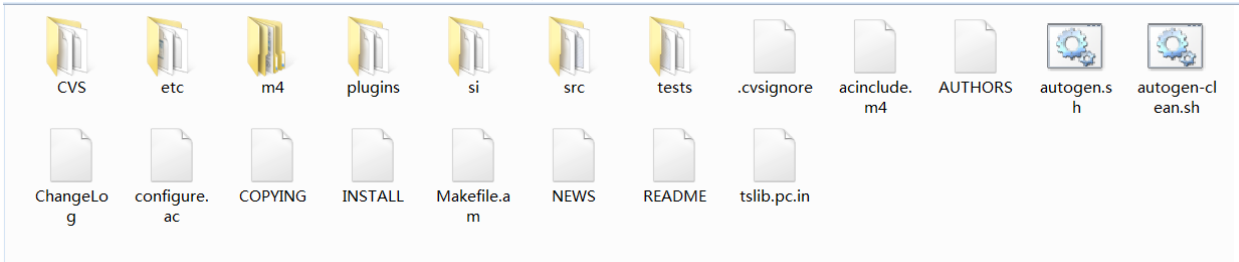
根据上一节的分析，通过 ADC 我们已经能获取到触摸点的坐标与压力值。但是是否就可以使用触摸屏了呢？可以非常肯定的说，不行。主要有下面两个问题：

1. 由于不同的触摸压力，加上触摸屏本身的离散性，ADC 得到的坐标会飘，也就是常说的飞点。
2. 触摸屏与 LCD 虽然物理上是对应的，触摸坐标与显示坐标之间却不是对应的，**并且不是普通的线性比例关系**。

由于有上面两个原因，**对采样得到的数据进行普通的滤波加权，再按线性比例转换为显示屏坐标，效果会很差**。很庆幸的是，在开源界有一个叫做 tslib 的程序

> Tslib 是一个开源的程序，能够为触摸屏驱动获得的采样提供诸如滤波、去抖、校准等功能，通常作为触摸屏驱动的适配层，为上层的应用提供了一个统一的接口。

最新的 tslib 库在 github 上 https://github.com/kergoth/tslib 由于 TSLIB 主要是 LINUX 平台的库，包含了很多 LINUX 框架代码。对源码我们就不做太多分析。同时，我们并没有基于最新的库做移植，而是使用更早的 1.4 版本。1.4 源码文件如下。



TSLIB

文件结构在网络有一篇文章对 TSLIB 的工作流程做了分析 http://blog.csdn.net/evilcode/article/details/7493704 一些关键信息如下：

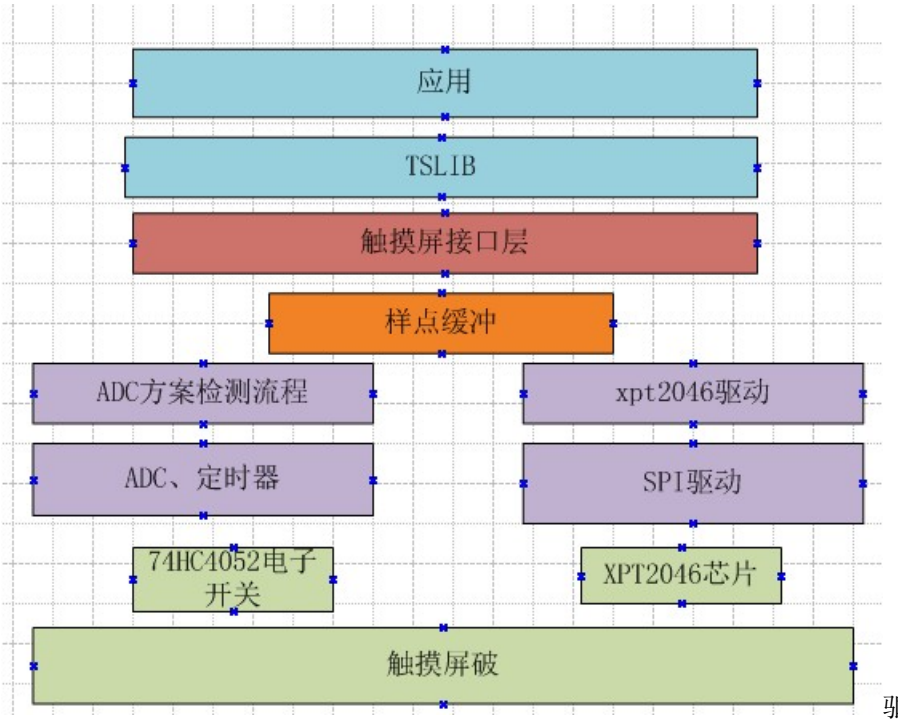
pthres 为 Tslib 提供的触摸屏灵敏度门槛插件；variance 为 Tslib 提供的触摸屏滤波算法插件；dejitter 为 Tslib 提供的触摸屏去噪算法插件；linear 为 Tslib 提供的触摸屏坐标变换插件。

在 tslib 中为应用层提供了 2 个主要的接口 ts_read() 和 ts_read_raw()，其中 ts_read() 为正常情况下的接口，ts_read_raw() 为校准情况下的接口。

正常情况下，tslib 对驱动采样到的设备坐标进行处理的一般过程如下：raw device –> pthres –> variance –> dejitter –> linear –> application

21.6 驱动程序设计

触摸屏程序架构如下：



驱动架构

1. 蓝色是应用层。主要就是 TSLIB 库。
2. 褐色是接口封装层，主要功能是将两种不同的驱动方案封装统一接口。触摸屏接口设计如下：

```
s32 dev_touchscreen_init(void)
s32 dev_touchscreen_open(void)
s32 dev_touchscreen_close(void)
s32 dev_touchscreen_read(struct ts_sample *samp, int nr)
s32 dev_touchscreen_write(struct ts_sample *samp, int nr)
s32 dev_touchscreen_ioctrl(void)
```

其中 dev_touchscreen_read 提供给上层使用。TSLIB 使用这个接口从缓冲读样点。dev_touchscreen_write 提供给下层使用。触摸检测程序（ADC 方案或者 XPT2046 驱动），检测到样点后用这个函数写到缓冲区。样点结构体如下，这是 tslib 中定义的，包含 x 轴坐标、y 轴坐标、压力、时间戳，时间戳我们不用，屏蔽掉。

```
struct ts_sample //触摸屏一个样点
{
        int                x;
        int                y;
        unsigned int        pressure;
        //struct timeval        tv;//时间，移植到 STM32 平台，应该不需要
};
```

1. 橙色是样点缓冲，是上层与底层联系。通过使用样点华冲，应用层和驱动层能做到解耦合。
2. 浅紫色就是两种不同方案的驱动代码，他们获取到样点后都填入缓冲。
3. 绿色是硬件。

21.7 编码调试记录

调试 ADC 触摸屏检测方案分三步：

1. ADC 转换功能。
2. 触摸检测流程。
3. TSLIB（校准和测试）。

21.7.1 调试 ADC 基本功能

在 mcu_dev 目录下添加 mcu_adc.c 和 mcu_adc.h。首先调试 ADC，跑通。

只要 ADC 转换可以完成，读到数据就可了，数据是否准确，暂时不用处理。

- 初始化

```
void mcu_adc_init(void)
{
    ADC_InitTypeDef ADC_InitStructure;
    ADC_CommonInitTypeDef ADC_CommonInitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);//使用 GPIOB 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;//---模拟模式
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;//---不上下拉
    GPIO_Init(GPIOB, &GPIO_InitStructure);//---初始化 GPIO
```

(continues on next page)

(continued from previous page)

```
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC2, ENABLE);//使能 ADC 时钟

    ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;//独立模式
    //两个采样阶段之间的延迟 5 个时钟
    ADC_CommonInitStructure.ADC_TwoSamplingDelay =         ADC_TwoSamplingDelay_20Cycles;
    ADC_CommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled; //DMA 失能
    ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div8;//预分频 6 分频。
    ADC_CommonInit(&ADC_CommonInitStructure);//初始化

    ADC_StructInit(&ADC_InitStructure);
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;//12 位模式
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;//非扫描模式
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;//非连续转换
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;//禁止触发
检测，使用软件触发
    //本值是触发源，我们已经禁止触发，因此本值无意义
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;//右对齐
    ADC_InitStructure.ADC_NbrOfConversion = 1;//1 个转换在规则序列中，也就是说一次转换一个
通道
    ADC_Init(ADC2, &ADC_InitStructure);//ADC 初始化

    #ifdef MCU_ADC_IRQ
    NVIC_InitTypeDef NVIC_InitStructure;

    NVIC_InitStructure.NVIC_IRQChannel = ADC_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;//抢占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;      //响应优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

   ADC_ITConfig(ADC2,  ADC_IT_EOC,  ENABLE);//打开 ADC EOC 中断
   ADC_ClearFlag(ADC2, ADC_FLAG_EOC);
   #endif

  ADC_Cmd(ADC2, ENABLE);
}
/**
```

初始化分 4 部分：

1. 7 到 12 行，初始化 IO，将对应的 IO 配置为 ADC 功能。
2. 14 到 21 行，ADC 通用配置
3. 23 到 32 行，ADC 转换配置。
4. 34 到 45 行，如果使用中断，就配置中断。

- 转换（查询方式）

第 10 行设置转换通道组。第 11 行启动转换，在 while 内查询标志，ADC_FLAG_EOC 标志置位就说明转换结束了。54 行调用 ADC_GetConversionValue 函数读取转换结果。

```
u16 mcu_adc_get_conv(u8 ch)
{
        u16 adcvalue;
        FlagStatus ret;

        //设置指定 ADC 的规则组通道，一个序列，采样时间
        MCU_ADC_DEBUG(LOG_DEBUG, "str--");
        ADC_ClearFlag(ADC2, ADC_FLAG_OVR);

        ADC_RegularChannelConfig(ADC2, ch, 1, ADC_SampleTime_480Cycles );
        ADC_SoftwareStartConv(ADC2);          //使能指定的 ADC 的软件转换启动功能

        while(1)//等待转换结束
        {
                ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_EOC);
                if(ret == SET)
                {
                        MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_EOC\r\n");
                        break;
                }
                ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_AWD);
                if(ret == SET)
                {
                        MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_AWD\r\n");

                }
                ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_JEOC);
                if(ret == SET)
                {
                        MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_JEOC\r\n");
```

(continues on next page)

(continued from previous page)

```
                }
                ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_JSTRT);
                if(ret == SET)
                {
                        MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_JSTRT\r\n");

                }

                ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_STRT);
                if(ret == SET)
                {
                        MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_STRT\r\n");

                }

                ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_OVR);
                if(ret == SET)
                {
                        MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_OVR\r\n");

                }
        }
        adcvalue = ADC_GetConversionValue(ADC2);
        return adcvalue;
}
```

- 转换（中断模式）

分两部分：启动转换、中断服务。启动如下，其实也就是查询模式一样。

```
s32 mcu_adc_start_conv(u8 ch)
{
        ADC_RegularChannelConfig(ADC2, ch, 1, ADC_SampleTime_480Cycles );
        ADC_SoftwareStartConv(ADC2);          //使能指定的 ADC 的软件转换启动功能
        return 0;
}
```

中断服务，其实跟查询模式 while 循环一样。

```
void mcu_adc_IRQhandler(void)
{
```

(continues on next page)

(continued from previous page)

```
    volatile u16 adc_value;
    FlagStatus ret;
    ITStatus itret;

    itret = ADC_GetITStatus(ADC2, ADC_IT_EOC);
    if( itret == SET)
{

            ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_EOC);
            if(ret == SET)
            {
                    //uart_printf("ADC_FLAG_EOC t\r\n");
                    adc_value = ADC_GetConversionValue(ADC2);
                    MCU_ADC_DEBUG(LOG_DEBUG, "-%d ", adc_value);
                    //dev_ts_adc_task(adc_value);

            }

            ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_AWD);
            if(ret == SET)
            {
                    MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_AWD t\r\n");

            }
            ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_JEOC);
            if(ret == SET)
            {
                    MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_JEOC t\r\n");

            }
            ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_JSTRT);
            if(ret == SET)
            {
                    MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_JSTRT t\r\n");

            }

            ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_STRT);
            if(ret == SET)
            {
```

(continues on next page)

(continued from previous page)

```
                        //uart_printf("ADC_FLAG_STRT t\r\n");

                }

                ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_OVR);
                if(ret == SET)
                {
                        MCU_ADC_DEBUG(LOG_DEBUG, "ADC_FLAG_OVR t\r\n");
                        ADC_ClearFlag(ADC2, ADC_FLAG_OVR);
                }

                ADC_ClearITPendingBit(ADC2, ADC_IT_EOC);
    }
}
```

从上可以看出，其实中断模式就是将查询模式拆成两部分。中断服务要在 stm32f4xx_it.c 中调用。

```
void ADC_IRQHandler(void)
{
        mcu_adc_IRQhandler();
}
```

- 测试程序

测试程序，支持查询和中断模式，通过宏 MCU_ADC_IRQ 控制。

```
s32 mcu_adc_test(void)
{
        mcu_adc_init();

#ifndef MCU_ADC_IRQ/* 查询模式 */
        u16 adc_value;

        wjq_log(LOG_FUN, "mcu_adc_test check\r\n");

        while(1)
        {
                adc_value = mcu_adc_get_conv(ADC_Channel_8);

                wjq_log(LOG_FUN, "ADC_Channel_8:%d\r\n", adc_value);
                Delay(1000);
```

(continues on next page)

(continued from previous page)

```
                adc_value = mcu_adc_get_conv(ADC_Channel_9);
                wjq_log(LOG_FUN, "ADC_Channel_9:%d\r\n", adc_value);
                Delay(1000);
        }
#else/* 中断模式 */
        wjq_log(LOG_FUN, "mcu_adc_test int\r\n");

        while(1)
        {
                wjq_log(LOG_FUN, "r ");
                mcu_adc_start_conv(ADC_Channel_8);
                Delay(1000);
                wjq_log(LOG_FUN, "d ");
                mcu_adc_start_conv(ADC_Channel_9);
                Delay(1000);
        }

#endif
}
```

在 main 函数中调用就可以测试了。

- 测试结果

ADC 是 12 位的，只要转换结果在 0-4096 范围内，ADC 工作就是正常的。

> mcu_adc_test int r 3403 d 1482 r 3401 d 3755 r 3399 d 3754 r 1794 d 3752 r 1814 d 3760 r 2752 d 3758 r 3407 d 2440 r 3398 d 2389 r 3405 d 1719 r 2462 d 2696 r 3398 d 1614 r 3402 d 2227 r 3399 d 1593 r 3395

21.7.2 调试触摸屏检测流程

在 board_dev 目录增加触摸屏驱动文件 dev_touchscreen.c 和 dev_touchscreen.h。代码根据前面分析的触摸屏检测流程实现。

- 74HC4052 硬件测试

首先使用 ADC 查询模式测试，验证电路切换功能是否正常。先检测触摸屏压力

> 用笔压触摸屏，检测到电压变化。理论上，在 Y+ 上检测到的电压应该大于等于 X+ 上的电压。当压触摸屏时，Y+ 将变小，但是不会小于 X+。压力越大，越靠近 X+。

再调试 X 轴的检测

如果没有触摸，检测到一个不准确的值。当有触摸时，电压值与触摸位置基本成线性关系。因此算法上必须要先判断压力，再检测坐标。

再调试 Y 轴的检测

与 X 轴类似

以上三步调通后，开始调试整个流程，并且要加快采样速度。三步都调通的测试代码如下：

```
s32 dev_touchscreen_test(void)
{
        u16 adc_x_value;
        u16 adc_y_value;
        u16 pre;//压力差

        dev_touchscreen_init();
        dev_touchscreen_open();

        while(1)
        {
                Delay(1000);
                /*  检测压力 */
                DEV_TP_PRESS_SCAN;
                adc_y_value = mcu_adc_get_conv(ADC_Channel_9);
                adc_x_value = mcu_adc_get_conv(ADC_Channel_8);
                pre = adc_y_value-adc_x_value;
                TS_DEBUG(LOG_FUN,"touch pre:%d, %d, %d\r\n",
                                adc_x_value, adc_y_value, pre);

                if(adc_x_value + 200 > adc_y_value)//200 为测试阀门，实际中要调试
                {
                        /* 检测 X 轴 */
                        DEV_TP_SCAN_X;
                        adc_x_value = mcu_adc_get_conv(ADC_Channel_9);
                        //uart_printf("ADC_Channel_8:%d\r\n", adc_x_value);

                        /* 检测 Y 轴 */
                        DEV_TP_SCAN_Y;
                        adc_y_value = mcu_adc_get_conv(ADC_Channel_8);
                        //uart_printf("ADC_Channel_8:%d\r\n", adc_y_value);

                        TS_DEBUG(LOG_FUN,"----------get a touch:%d, %d, %d\r\n",
                                        adc_x_value, adc_y_value, pre);
```

(continues on next page)

(continued from previous page)

```
                }
}
```

调试信息如下，可以看出 X+ 的电压基本接近 0，在没触摸时，Y+ 跟 X+ 的差距非常大。

> touch pre:69, 221, 152 ———-get a touch:1738, 2193, 152 touch pre:59, 223, 164 ———-get a touch:1741, 2191, 164 touch pre:61, 229, 168 ———-get a touch:1741, 2188, 168 touch pre:2, 4064, 4062 touch pre:9, 4065, 4056 touch pre:2, 4071, 4069 touch pre:9, 4068, 4059

21.7.3 使用中断方式检测触摸屏

使用 ADC 终端的触屏检测流程如下：

> 开一个定时器，10-50 毫秒启动一次触摸屏压力检测。ADC 结果在 ADC 中断中处理，如果检测到压下，则启动坐标检测。采样得到的样点写入样点缓冲。并且连续检测。因为用最快速度画过触摸屏，大约只需要 50 毫秒，50 毫秒/320 个点，一个点才 150us。因此有触摸时，连续转换。为了节省系统开销，只好用 ADC 中断。

修改 Touchscreen 驱动，编写使用 ADC 中断的转换流程。创建了一个 TASK 函数，在函数中处理 ADC 转换的各个步骤。这个函数在 ADC 中断中调用，每当采样到一个样点，就传入触摸屏处理流程。

```
void mcu_adc_IRQhandler(void)
{
        volatile u16 adc_value;
        FlagStatus ret;
        ITStatus itret;

        itret = ADC_GetITStatus(ADC2, ADC_IT_EOC);
        if( itret == SET)
        {

                ret = ADC_GetFlagStatus(ADC2, ADC_FLAG_EOC);
                if(ret == SET)
                {
                        //uart_printf("ADC_FLAG_EOC t\r\n");
                        adc_value = ADC_GetConversionValue(ADC2);
                        MCU_ADC_DEBUG(LOG_DEBUG, "%d ", adc_value);

                        dev_ts_adc_task(adc_value);

                }
```

在这个流程中，有一个地方比较关键，也就是压力的判断，压力差小于 250，算下笔，压力差大于 2000，算起笔，200 到 2000，是一个过渡，会抖动，丢弃。当然，250 跟 2000 是假设，具体根据硬件调试情况配置。

然后修改 dev_touchscreen_test 函数如下，在 main 中调用。

```
        struct ts_sample s;
        s32 ret;
        u8 pensta = 1;//没接触

        dev_touchscreen_init();
        dev_touchscreen_open();

        while(1)
        {
                ret = dev_touchscreen_read(&s,1);
                if(ret == 1)
                {
                        if(s.pressure != 0 && pensta == 1)
                        {
                                pensta = 0;
                                wjq_log(LOG_FUN, "pen down\r\n");
                                wjq_log(LOG_FUN, ">%d %d %d-\r\n", s.pressure, s.x, s.y);
                        }
                        else if(s.pressure == 0 && pensta == 0)
                        {
                                pensta = 1;
                                wjq_log(LOG_FUN, "\r\n--------pen up--------\r\n");

                        }
                }
        }
```

运行测试程序后，触摸四个角的样点分别如下：左上角

pen down 34 870 679- ———pen up———

左下角

pen down 117 857 3375- ———pen up———

右下角

pen down 146 3234 3369- ———pen up———

右上角

pen down 50 2991 709- ———pen up———

从数据可以看出，坐标基本符合方向。

- TASK 函数说明整个代码最重要的函数就是 s32 dev_ts_adc_task(u16 dac_value)，这个函数由 ADC 中断调用，实现完整的触摸屏检测。

```
s32 dev_ts_adc_task(u16 dac_value)
{
        static u16 pre_y, pre_x;
        static u16 sample_x;
        static u8 pendownup = 1;
        struct ts_sample tss;

        if(TsAdcGd != 0)
                return -1;

        if(TouchScreenStep == 0)//压力检测第一步 ADC 转换结束
        {
                pre_y = dac_value;
                TouchScreenStep        = 1;
                mcu_adc_start_conv(ADC_Channel_8);
        }
        else if(TouchScreenStep == 1)
        {
                pre_x = dac_value;
                //TS_DEBUG(LOG_DEBUG, "--press :%d %d\r\n", pre_y, pre_x);

                if(pre_x + DEV_TP_PENDOWN_GATE > pre_y)
                {
                        TouchScreenStep        = 2;
                        DEV_TP_SCAN_X;
                        mcu_tim7_start(2);
                }
                else if(pre_x + DEV_TP_PENUP_GATE < pre_y)
                {//没压力，不进行 XY 轴检测
                        /* 起笔只上送一点缓冲 */
                        if(pendownup == 0)
                        {
                                pendownup = 1;
                                tss.pressure = 0;//压力要搞清楚怎么计算
                                tss.x = 0xffff;
                                tss.y = 0xffff;
                                dev_touchscreen_write(&tss,1);
```

(continues on next page)

(continued from previous page)

```
                }
                TouchScreenStep        = 0;

                DEV_TP_PRESS_SCAN;
                //打开一个定时器，定时时间到了才进行压力检测
                mcu_tim7_start(100);
        }
        else
        {
                /* 上下笔的过渡，丢弃 */
                TouchScreenStep        = 0;

                DEV_TP_PRESS_SCAN;

                mcu_tim7_start(20);
        }
}
else if(TouchScreenStep == 2)
{
        sample_x =  dac_value;

        TouchScreenStep        = 3;
        DEV_TP_SCAN_Y;
        mcu_tim7_start(2);
}
else if(TouchScreenStep == 3)//一轮结束，重启启动压力检测
{
        //压力要搞清楚怎么计算
        tss.pressure = DEV_TP_PENDOWN_GATE-(pre_y - pre_x);
        tss.x = sample_x;
        tss.y = dac_value;
        dev_touchscreen_write(&tss,1);
        //TS_DEBUG(LOG_DEBUG, "tp :%d, %d, %d\r\n",
        //     tss.pressure, tss.x, tss.y);
        pendownup = 0;

        TouchScreenStep        = 0;
        DEV_TP_PRESS_SCAN;
        mcu_tim7_start(2);
```

(continues on next page)

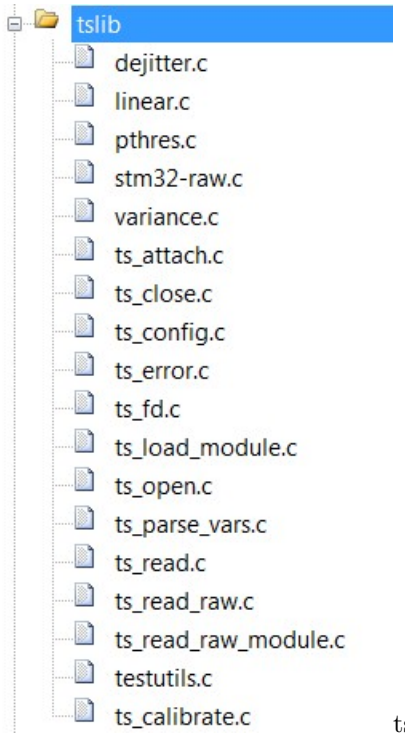
(continued from previous page)

```
        }
        else//异常，启动压力检测
        {
                TouchScreenStep        = 0;
                DEV_TP_PRESS_SCAN;
                mcu_tim7_start(100);
        }

        return 0;
}
```

1. 函数流程根据触摸屏检测流程分步骤执行，整个大流程是循环模式。
2. 11 到 16 行为第 0 步，读取 Y 轴电压，然后启动 X 轴 ADC 检测。这一步其实属于第 *1* 步，前面有一步启动压力检测，并开始检测 *Y* 轴电压。
3. 17 到 55 行，第 1 步，这一步比较复杂。读到 X 轴电压后，进行压力判断。22 行到 27 行，压下，启动 X 轴坐标检测。28 到 44 行，松开，上送一点起笔，重新配置到压力检测，也就是第 0 步。46 到 54 行，压力过渡区，重新配置到压力检测，也就是第 0 步。
4. 56 到 63 行，读 X 轴坐标转换结果，配置为 Y 轴坐标检测。
5. 64 到 75，读取 Y 轴坐标转换结果，写入缓冲，重新配置到压力检测，也就是第 0 步。
6. 77 到 82，处理异常。

这段代码跟前面原理分析一致，大家好好体会整个处理过程。

21.7.4 与 TSLIB 联合



tslib 参与编译的文件 stm32-raw.c 是 tslib 跟 dev_touchscreen.c 的接口文件。ts_input_read 函数调用 dev_touchscreen_read 函数读取样点。

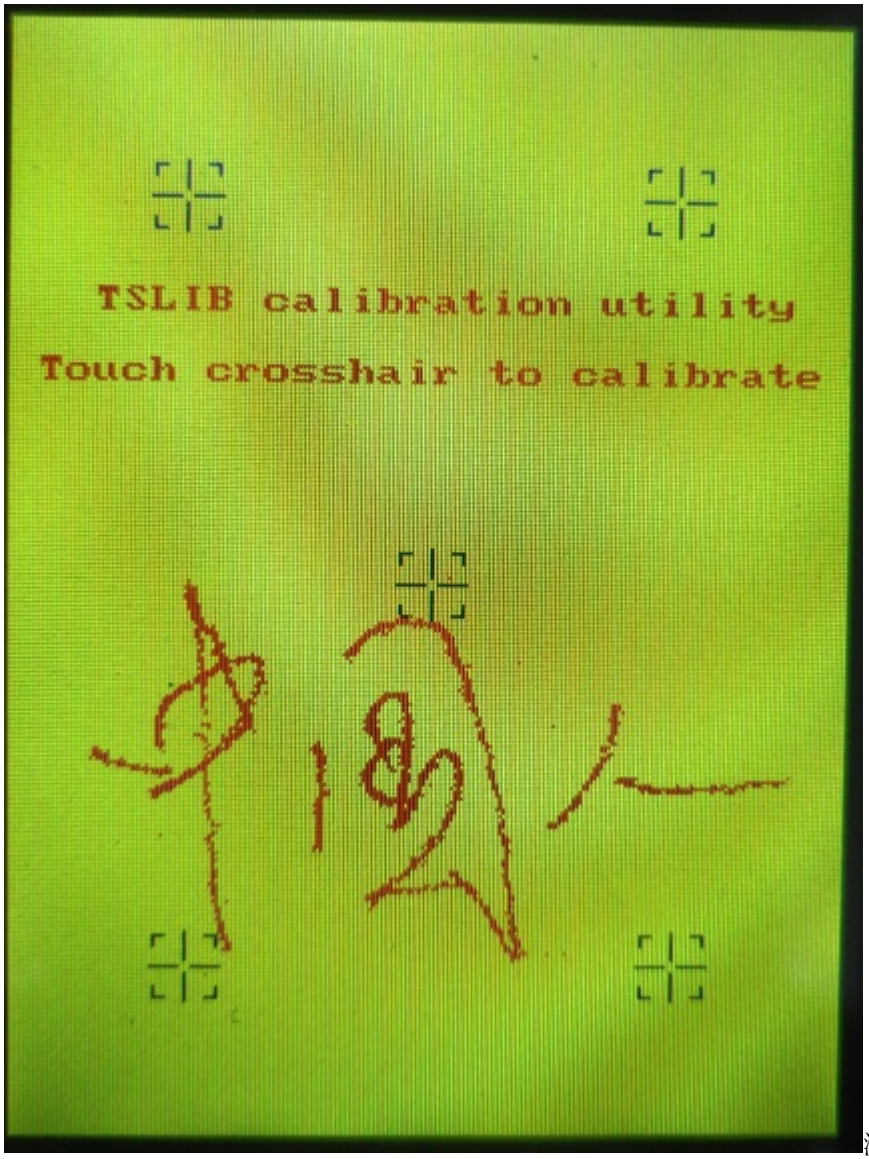
```
static int ts_input_read(struct tslib_module_info *inf,
                         struct ts_sample *samp, int nr)
{
        struct tslib_input *i = (struct tslib_input *)inf;
        struct tsdev *ts = inf->dev;
        int ret = nr;

        //uart_printf(" ts input read\r\n");
        ret = dev_touchscreen_read(samp, nr);
        #if 0
        if((samp->pressure != 0) && (ret == nr))
        {
                uart_printf("stm32 raw:%d, %d, %d\r\n", samp->pressure, samp->x, samp->
↪y);
        }
        #endif

        return ret;
}
```

要分析 tslib，可以从 ts_config.c 入手。由于 TSIB 比较复杂，后续有一个文档专门解释 tslib。在此，直接提供源码，不做解释。测试函数如下：

```
/* Infinite loop */
        mcu_uart_open(3);
        wjq_log(LOG_INFO, "hello word!\r\n");
        mcu_i2c_init();
        mcu_spi_init();
        dev_key_init();
        //mcu_timer_init();
        dev_buzzer_init();
        dev_tea5767_init();
        dev_dacsound_init();
        dev_spiflash_init();
        dev_wm8978_init();
        dev_lcd_init();

        //dev_dacsound_open();
        dev_key_open();
        //dev_wm8978_open();
        //dev_tea5767_open();
        //dev_tea5767_setfre(105700);

        #if 0
        mcu_adc_test();
        #endif

        #if 0
        dev_touchscreen_test();
        #endif

        #if 1
        dev_touchscreen_init();
        dev_touchscreen_open();
        ts_calibrate();
        ts_calibrate_test();
        #endif
        while (1)
        {
```

1. dev_touchscreen_init 初始化。
2. dev_touchscreen_open 打开设备

3. ts_calibrate 校准
4. ts_calibrate_test 测试。

测试效果如图： 

测试效果线条还算流畅。

21.8 总结

1. 测试中触摸屏还是出现飞点，请分析解决。
2. 触摸屏驱动如何提供接口给 APP 使用？也就是 TSLIB 要提供什么接口？

21.9 end

CHAPTER 22

模拟 SPI-XPT2046-电阻式触摸屏调试

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

上一节我们调试了触摸屏，用的是 STM32 内置 ADC 方案。现在就让我们调试常见的 XPT2046 方案。

22.1 XPT2046

XPT2046 是什么？很多朋友可能认为 XPT2046 是一个触摸屏检测 IC，这不是很准确。XPT2046 是一个 ADC 转换芯片，支持 4 线电阻屏的 AD 转换。**它并没有实现电阻屏检测流程**。

XPT2046 是一款 4 导线制触摸屏控制器，内含 12 位分辨率 125KHz 转换速率逐步逼近型 A/D 转换器。XPT2046 支持从 1.5V 到 5.25V 的低电压 I/O 接口。XPT2046 能通过执行两次 A/D 转换查出被按的屏幕位置，除此之外，还可以测量加在触摸屏上的压力。内部自带 2.5V 参考电压可以作为辅助输入、温度测量和电池监测模式之用，电池监测的电压范围可以从 0V 到 6V。XPT2046 片内集成有一个温度传感器。

- 工作原理

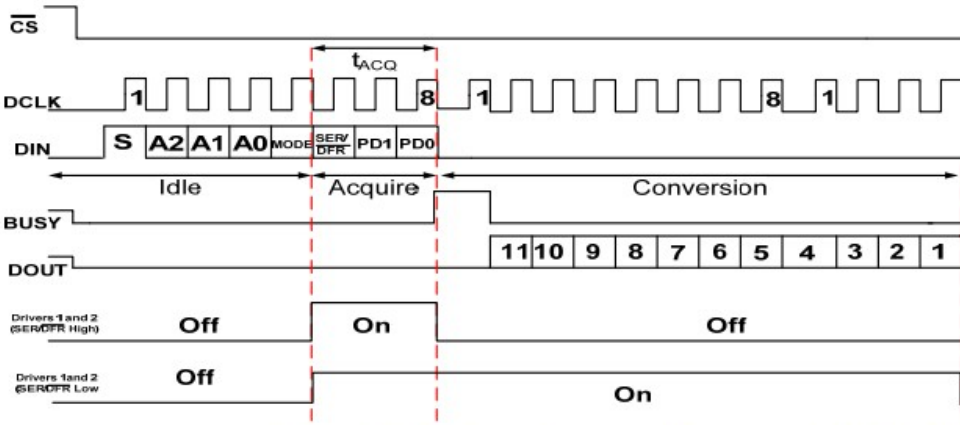


图 10 8 位总线接口，无 DCLK 时钟延迟，24 时钟周期转换

我们直接从数字接口时序看其工作原理：
序 1 DIN 是输入数据，首先发送一个字节到 XPT2046，这个字节叫命令字：

bit7：S，启动标志，固定为 1 bit6-bit4：A2/A1/A0, 通道选择，差分模式跟单端不一样，差分模式如下：

- 001，测 Y 轴 011，测 Z1 100，测 Z2 101，测 X 轴

bit3：MODE,1 =8 位 ADC，0 =12 位 ADC bit2：SEF/DFR，单端模式还是差分模式，XY 轴跟压力可以用差分，最好用差分；其他功能只能用单端模式。bit1-bit0：PD1/PD0, 工作模式：11，总是处于供电状态；00，在变换之间进入低功耗

发送完命令字后，XPT2046 开始转换，BUSY 管脚变高。当 BUSY 管脚变低时，XPT2046 转换结束。通过 DOUT 读数据，转换数据是 12 位（位数可设置，我们用 12 位）。分两字节读取，第一字节返回 BIT11-BIT5，也就是说，第 1 字节低 7 位有效，最高位无效。第二字节返回 BIT4-BIT0，高 5 位有效，低 3 位无效。下图是 XPT2046 压力测试原理，可见 Z1，Z2 测试位置跟我们用内置 ADC 的原理一

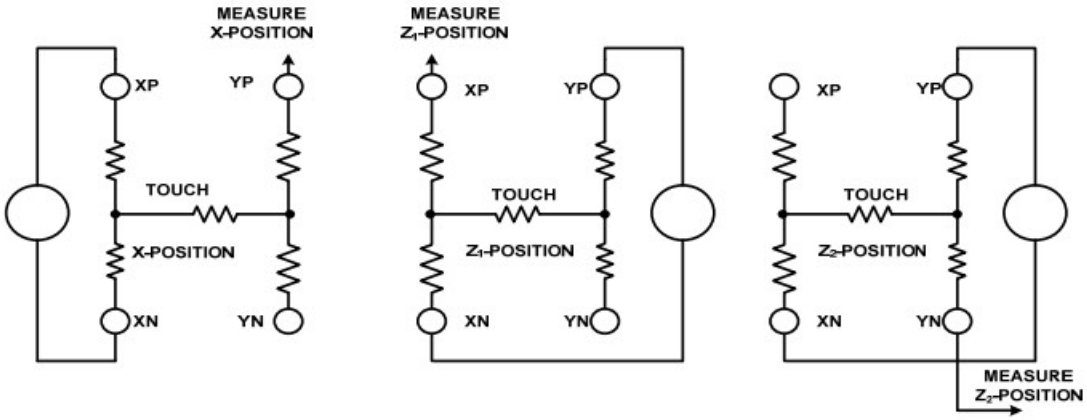


图 9 压力测量模块图

样。 测试原理按照上节说的触摸屏转换原理。使用 XPT2046 的转换流程就是：

> 测 Z2 通道，测试压力 Y 轴电压。测 Z1 通道，测试压力 X 轴电压。测 X 通道，测试位置 X 轴电压。测 Y 通道，测试位置 Y 轴电压。

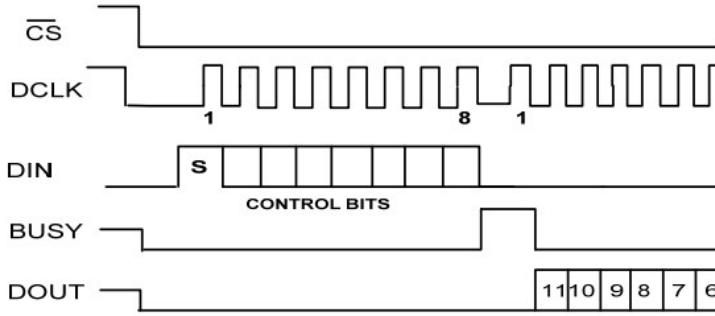


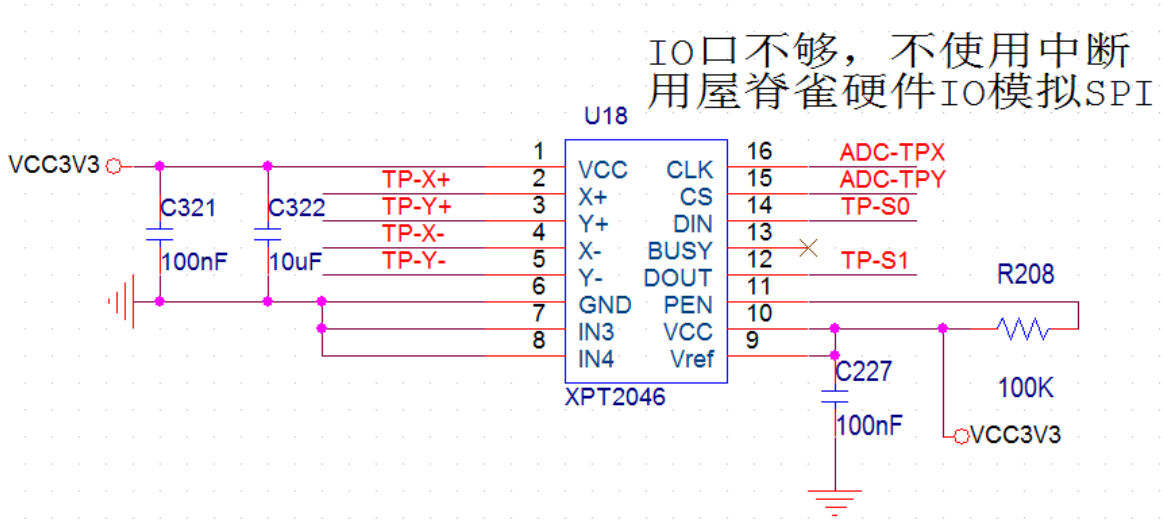
图 12 8 位总线接口，无 DCLK 时钟延

为了加快转换数据读取速度，XPT2046 提供了其他通信时序。

时序与前面时序的区别就是：在读最后一个字节数据时，启动下一次转换。需要注意的是，**读第一个字节时必须发送 0X00，不能发送其他数据**。

22.2 硬件原理

用 XPT2046 方案的 LCD，接口跟用 ADC 方案的一样。下图是 XPT2046 原理图，电阻屏四根线接到 2、3、4、5 管脚。SPI 时钟 CLD 接到原来的 ADC-TPX 脚，SPI 片选接到 ADC-TPY，SPI DIN 接到 TP-SO，SPI DOUT 接到 TP-SI。这四个管脚不是硬件 SPI 控制器的管脚，需要用 IO 模拟实现 SPI 功能。



原理图

22.3 驱动架构设计

整体架构上一节已经说明。驱动的大概流程就是：

> 启动定时器，定时时间为 1ms。在定时中断内用 VSPI 接口控制 XPT2406 转换，并读取数据。对数据按照 ADC 同样的方法处理。识别到样点就填入缓冲。

22.3.1 模拟 SPI（VSPI）设计

用 STM32 内置 SPI 控制器的 SPI 驱动我们已经完成。模拟 SPI 就按照硬件控制器的接口实现。

- 接口（与硬件 SPI 统一）VSPI 也是 SPI，接口当然跟 SPI 一样

```
static s32 mcu_vspi_init(void)
static s32 mcu_vspi_open(SPI_DEV dev, SPI_MODE mode, u16 pre)
static s32 mcu_vspi_close(SPI_DEV dev)
static s32 mcu_vspi_transfer(SPI_DEV dev, u8 *snd, u8 *rsv, s32 len)
static s32 mcu_vspi_cs(SPI_DEV dev, u8 sta)
```

同时将原来的硬件 SPI 控制器接口函数加上 h 标志，例如：

```
static s32 mcu_hspi_init(void);
```

对于上层来说，模拟 SPI 还是硬件 SPI，都是 SPI，因此原来的接口就是统一对外接口，在接口内通过判断，决定调用 VSPI 接口还是 HSPI 接口。例如：

```
s32 mcu_spi_open(SPI_DEV dev, SPI_MODE mode, u16 pre)
```

- 多个 VSPI 的统一

后续我们的外扩 IO 口可能会接 SPI 设备，也是需要用 IO 口模拟 SPI 的。前面我们说过驱动和设备的关系。**多个 VSPI 设备肯定只用一套 VSPI 代码**。为了实现这个目的，我们定义了 VSPI 设备对象，如下：

```
typedef struct
{
        char *name;
        SPI_DEV dev;
        s32 gd;

        u32 clkrcc;
        GPIO_TypeDef *clkport;
        u16 clkpin;

        u32 mosircc;
        GPIO_TypeDef *mosiport;
        u16 mosipin;

        u32 misorcc;
        GPIO_TypeDef *misoport;
        u16 misopin;

        u32 csrcc;
        GPIO_TypeDef *csport;
        u16 cspin;
}DevVspiIO;
```

当要使用一个 VSPI 设备时，只需要定义一个设备实体，并添加到 VPSI 列表即可：

```
DevVspiIO DevVspi1IO={
                "VSPI1",
                DEV_VSPI_1,
                -2,//未初始化

                VSPI1_RCC,
                VSPI1_CLK_PORT,
                VSPI1_CLK_PIN,

                VSPI1_RCC2,
```

(continues on next page)

(continued from previous page)

```
                VSPI1_MOSI_PORT,
                VSPI1_MOSI_PIN,

                VSPI1_RCC2,
                VSPI1_MISO_PORT,
                VSPI1_MISO_PIN,

                VSPI1_RCC,
                VSPI1_CS_PORT,
                VSPI1_CS_PIN,
        };

/* 无用的虚拟 SPI 设备，占位用 */
DevVspiIO DevVspiNULL={
                "VSPI0",
                DEV_VSPI_0,
                -2,//未初始化;
                };

DevVspiIO *DevVspiIOList[]={
        &DevVspiNULL,

        #ifdef SYS_USE_VSPI1
        &DevVspi1IO,
        #endif

        };
```

如上，DevVspi1IO 就是我们定义的 VSPI1 设备，然后添加到 DevVspiIOList 数组。以后就可以通过 mcu_spi_open 等 spi 接口操作这个 VSPI 设备了。

- VSPI 实现

VSPI 的代码就请看源码，不累述。

22.3.2 定时器改造

XPT2046 需要定时转换，跟 ADC 使用定时器一样，也使用定时器 7。但是在 ADC 中，定时器执行一次就停止了，下一次由 TASK 启动。XPT 需要连续启动。我们改造定时器 7 的代码，让它能用于这两种情况。

```
s32 mcu_tim7_start(u32 Delay_10us, void (*callback)(void), u8 type)
。。。
/**
 *@brief:      mcu_tim6_IRQhandler
 *@details:    定时器中断处理函数
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
void mcu_tim7_IRQhandler(void)
{
    if(TIM_GetITStatus(TpTim, TIM_FLAG_Update) == SET)
    {
        TIM_ClearFlag(TpTim, TIM_FLAG_Update);
                if(Tim7Type == 1)
                        TIM_Cmd(TpTim, DISABLE);//停止定时器

                Tim7Callback();

    }
}
```

在 mcu_tim7_start 函数增加两个参数：

> callback：中断服务程序回调，当定时器中断发生时，就执行这个 callback 函数。type：类型，是一次还是重复。

mcu_tim7_IRQhandler 同步改造。原来 ADC 方案用到 mcu_tim7_start 的地方也同步改造。

22.3.3 XTP2046 驱动说明

只有一个关键函数 dev_xpt2046_task，

```
void dev_xpt2046_task(void)
{

        static u16 pre_y, pre_x;
        static u16 sample_y, sample_x;

        static u8 pendownup = 1;
        struct ts_sample tss;

```

(continues on next page)

(continued from previous page)

```
        u8 stmp[4];
        u8 rtmp[4];

        if(DevXpt2046Gd != 0)
                return;
        /*

        整个流程分四步
        读 Z1,Z2, 用于计算压力
        读 X,Y 轴, 用于计算坐标

        1 使用了快速 16CLK 操作法, 过程 100us 左右。
        经测试, 中间不需要延时。
        2 没有使用下笔中断, 通过压力判断是否下笔。但是有点疑惑, 理论上接触电阻应该很小的,
         用 ADC 方案, 正常, 用 XPT2046 方案, 感觉接触电阻比较大, 不知道是哪里没有理解对。

        3 快速 CLK 操作, 也就是在读最后一个字节的时候同时发送下一个转换命令。
        一定要第一个字节发送 0X00, 第二个字节发送命令。如果第一个字节不是 00, 而且正好 BIT7
是 1,
        芯片会重新启动转换, 读回来的电压值就都不对了。

        4 实测, 不需要延时, 如果你的 SPI 时钟较快, 请注意延时等待转换结束。
        5 理论上还可以节省一个字节的发送时间, 请自行优化。
        */

        /*-------------------------*/

        stmp[0] = XPT2046_CMD_Z2;
        mcu_spi_transfer(XPT2046_SPI, stmp, NULL, 1);
        //vspi_delay(100);
        stmp[0] = 0x00;
        stmp[1] = XPT2046_CMD_Z1;
        mcu_spi_transfer(XPT2046_SPI, stmp, rtmp, 2);
        pre_y = ((u16)(rtmp[0]&0x7f)<<5) + (rtmp[1]>>3);
        /*-------------------------*/
        //vspi_delay(100);
        stmp[0] = 0x00;
        stmp[1] = XPT2046_CMD_X;
        mcu_spi_transfer(XPT2046_SPI, stmp, rtmp, 2);
        pre_x = ((u16)(rtmp[0]&0x7f)<<5) + (rtmp[1]>>3);
```

(continues on next page)

(continued from previous page)

```
        /*------------------------*/
        //vspi_delay(100);
        stmp[0] = 0x00;
        stmp[1] = XPT2046_CMD_Y;
        mcu_spi_transfer(XPT2046_SPI, stmp, rtmp, 2);
        sample_x = ((u16)(rtmp[0]&0x7f)<<5) + (rtmp[1]>>3);
        /*------------------------*/
        //vspi_delay(100);
        stmp[0] = 0x00;
        stmp[1] = 0X00;
        mcu_spi_transfer(XPT2046_SPI, stmp, rtmp, 2);
        sample_y = ((u16)(rtmp[0]&0x7f)<<5) + (rtmp[1]>>3);

        /*
                算压力
                简化算法
                实际:
                R 触摸电阻 =Rx 面板 * (X 位置/4096) *(Z2/Z1-1)
        */
        if(pre_x +  DEV_XPT2046_PENDOWN_GATE > pre_y)
        {
                /* 有压力 */
                tss.pressure = 200;//DEV_XPT2046_PENDOWN_GATE - rpress;
                tss.x = sample_x;
                tss.y = sample_y;
                dev_touchscreen_write(&tss,1);
                //uart_printf("%d,%d,%d\r\n", tss.pressure, tss.x, tss.y);
                pendownup = 0;
        }
        else if(pre_x + DEV_XPT2046_PENUP_GATE < pre_y)//没压力, 不进行 XY 轴检测
        {
                /* 起笔只上送一点缓冲 */
                if(pendownup == 0)
                {
                        pendownup = 1;
                        tss.pressure = 0;//压力要搞清楚怎么计算
                        tss.x = 0xffff;
                        tss.y = 0xffff;
                        dev_touchscreen_write(&tss,1);
                }
```

(continues on next page)

(continued from previous page)

```
        }
        else
        {
                //uart_printf("--press :%d %d\r\n", pre_y, pre_x);
                /* 上下笔的过渡，丢弃 */

        }

}
```

1. 36 到 60 行，读取压力跟坐标值。在 ADC 方案中，这些电压值通过多步读取，XPT2046 就直接一次性读取了。**如果想优化性能，减少没有触摸时读数据时间，可以按照 ADC 方案，判断到压力后再读取坐标**。
2. 68 行之后的就跟 ADC 方案类似了，判断压力的三个状态，进行分别数据处理。

22.3.4 XPT2046 跟 ADC 方案的兼容

修改下面三个接口函数，通过宏控制使用 ADC 方案还是 XPT2046 方案。具体修改见代码。

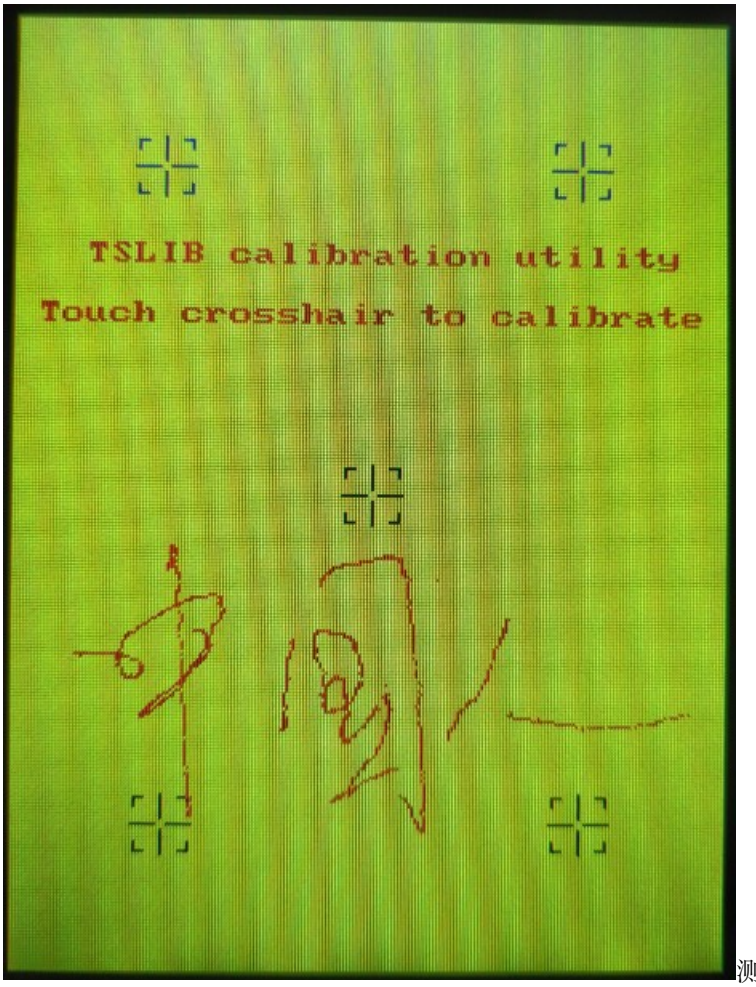
```
/*
        触摸屏方案选择
*/
//#define SYS_USE_TS_ADC_CASE
#define SYS_USE_TS_IC_CASE

extern s32 dev_touchscreen_init(void);
extern s32 dev_touchscreen_open(void);
extern s32 dev_touchscreen_close(void);
```

22.3.5 测试

测试程序跟上一节一样，测试效果如下图，跟 ADC 测试效果是有差别的：**线条细，飞点少**。原因有两个：

> 一是 ADC 采样率更高，同样时间得到的样点比 XPT2046 多，毕竟 XPT2046 是 1 毫秒采样一个样点。二是内置的 ADC 稳定性我们还没有处理，或者是我们的电子开关电路没调试好。

但是并不是说 XPT2046 效果就比 ADC 好，当快速划线时，由于 XPT2046 采样率只有 1K，会出现断线。可以通过加快采样率，或者是在应用层软件连线解决。



测试效果

22.4 总结

了解触摸屏原理后，使用 XTP2046 做触摸检测并不复杂。通过本章节，我们初步了解了同一个功能使用两种方案应该如何设计驱动程序。

22.5 end

CHAPTER 23

DCMI-摄像头功能调试

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

STM32F407 芯片带有 DCMI 接口，在我们的核心板上已经将接口用 18PIN 的 FPC 座子引出。这个接口可以接我们的 OV2640 接口。本节我们开始调试摄像头。

23.1 DCMI

DCMI 接口是 ST 自己定义的接口。

Digital camera interface (DCMI)，是意法半导体公司产品 STM32F4xx 系列芯片的快速摄像头接口。通过 HSVNC VSVNC PIXCLX 和 8 到 14 位的数据接口 D[0:13] 完成控制。

23.1.1 简介

DCMI 数字摄像头接口是一个同步并行接口，能接收外部 8 位、10 位、12 位或 14 位 CMOS 摄像头模块发出的高速数据流。可支持不同的数据格式:YCbCr4:2:2/RGB565 逐行视频和压缩数据（JPEG）。此接口适用于黑白摄像头、X24 和 X5 摄像头，并假定所有预处理（如调整大小）都在摄像头中执行。

23.1.2 特性

DCMI 主要特性

- 8 位、10 位、12 位或 14 位并行接口
- 内嵌码/外部行同步和帧同步
- 连续模式或快照模式
- 裁剪功能
- 支持以下数据格式：
  - 8/10/12/14 位逐行视频：单色或原始拜尔格式
  - YCbCr 4:2:2 逐行视频
  - RGB 565 逐行视频
  - 压缩数据：JPEG

DCMI 特性

23.1.3 引脚

表 60. DCMI 引脚

名称	信号类型
D[0:13]	数据输入
HSYNC	水平同步（行同步）输入
VSYNC	垂直同步（场同步）输入
PIXCLX	像素时钟输入

数据输入一共有 14 根引脚。

引脚

23.1.4 DCMI 框图

图 62. DCMI 框图

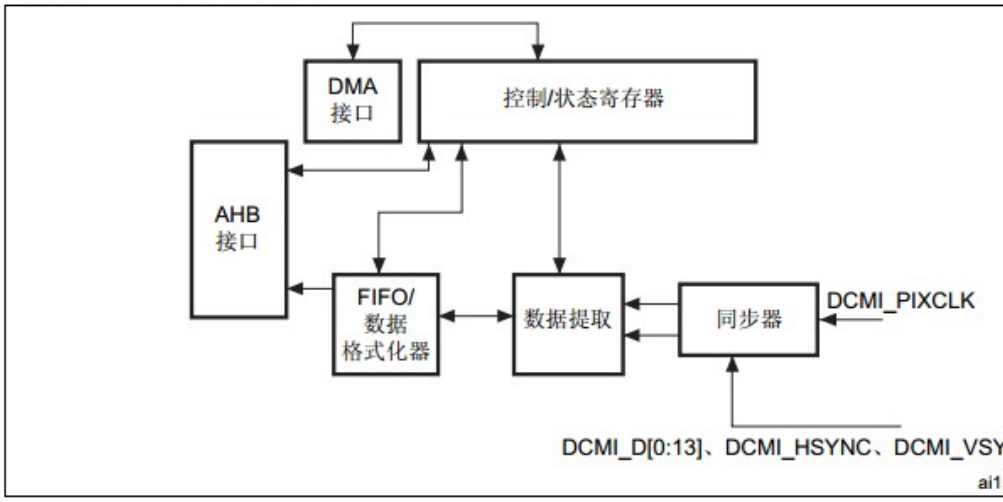
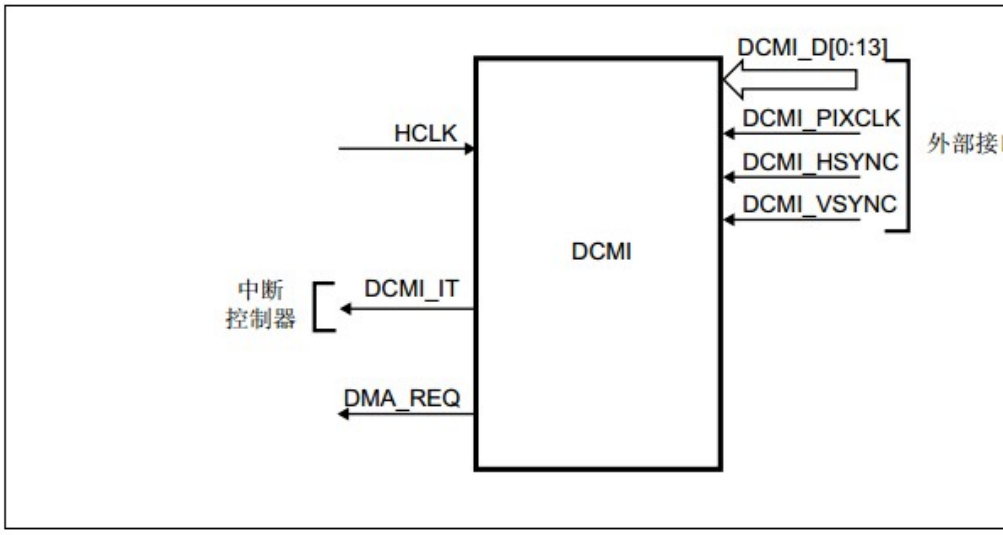


图 63. 顶级框图



从框图可见,DCMI 支持 DMA 传输。

框图

23.2 OV2640

OV2640 是 OV（OmniVision）公司生产的一颗 1/4 寸的 CMOS UXGA（1632*1232）图像传感器。该传感器体积小、工作电压低，提供单片 UXGA 摄像头和影像处理器的所有功能。通过 SCCB 总线控制，可以输出整帧、子采样、缩放和取窗口等方式的各种分辨率 8/10 位影像数据。该产品 UXGA 图像最高达到 15 帧/秒（SVGA 可达 30 帧，CIF 可达 60 帧）。用户可以完全控制图像质量、数据格式和传输方式。所有图像处理功能过程包括伽玛曲线、白平衡、对比度、色度等都可以通过 SCCB 接口编程。OmmiVision 图像传感器应用独有的传感器技术，通过减少或消除光学或电子缺陷如固定图案噪声、拖尾、浮散等，提高图像质量，得到

清晰的稳定的彩色图像。

23.2.1 主要特性

1. 总共 1632*1232* 像素，最大输出尺寸 *UXGA(1600*1200)，即 200W 像素。
2. 模拟功能：模拟放大、增益控制、通道平衡、平衡控制。
3. 10 位 ID 转换。
4. 自带数字信号处理器，主要功能：边沿锐化、颜色空间转换、色相和饱和度控制、降噪等。
5. 自带压缩引擎。

23.2.2 接口

- SCCB 接口

SCCB 接口控制图像传感器芯片的运行。SCCB 接口相当于 I2C。

- 数字视频接口

OV2640 拥有一个 10 位数字视频接口，接口支持 8 位接法。

23.2.3 图像格式

OV2640 支持多种图像格式：UXGA 1600*1200 像素。SXGA 1280*1024 WXGA+ 1440*900 XVGA 1280*960 WXGA 1280*800 XGA 1024*768 SVGA 800*600 VGA 640*480 CIF 352*288 WQVGA 400*240 QCIF 176*144 QQVGA 160*120

23.2.4 主要操作

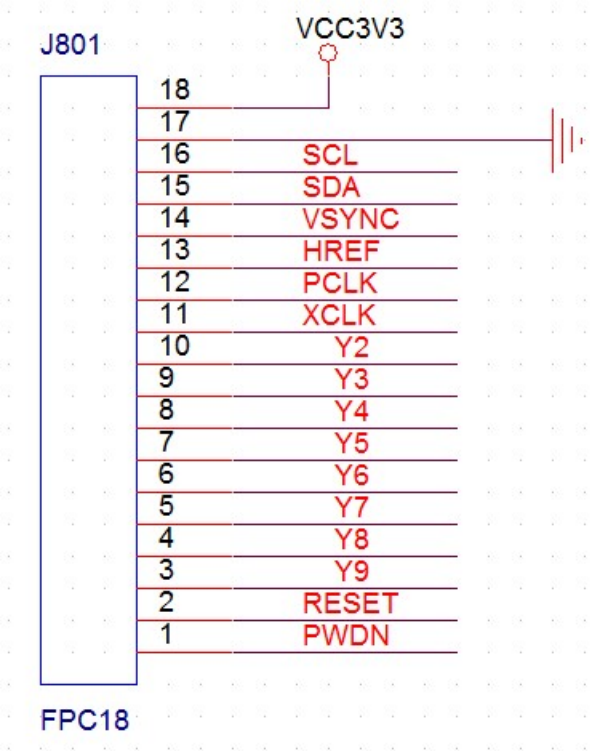
OV2640 支持传感器窗口设置、图像尺寸设置、图像窗口设置和图像输出大小设置。

- 传感器窗口设置设置整个传感器区域感光部分。开窗范围从 2*2~1632*1220。开窗必须大于图像尺寸设置。
- 图像尺寸设置 DSP 输出图像的尺寸。
- 图像窗口设置在 DSP 输出的图像上开窗，窗口必须小于等于 DSP 输出图像的尺寸。
- 图像输出大小设置最终输出图像的尺寸。这个图像是 DSP 将图像窗口中的图像**缩放**而得。

23.2.5 使用

OV2640 设置不简单，文档也复杂，我们使用 OV260，其实是将 ST 官方的例程移植到我们的硬件上而已。

23.3 接口原理图



DCMI 接口

1. SCL 和 SDA 是摄像头控制信号，接到 CPU 硬件 I2C 上，不使用模拟 I2C
2. RESET 和 PWDN 信号是摄像头复位和上电信号，可以不接。
3. XCLK 是时钟，屋脊雀 OV2640 摄像头模块不带晶振，因此需要 STM32 提供时钟。
4. 17/18 脚供电和地。
5. 余下的就是 DCMI 信号。**其中数据线只使用 8 根**。

23.4 编码与调试

标准库提供了 DCMI 例程。在 STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Project\STM32F4xx_StdPeriph_Examples\DCM
目录的 readme.txt 文件中有详细说明。

> This example shows how to use the DCMI to control the OV9655 or OV2640 Camera module mounted on STM324xG-EVAL or STM32437I-EVAL evaluation boards. ···

从 readme 中可知：

1. 使用 SCCB 接口配置 OV9655，SCCB 是一种类似 I2C 的协议。
2. 将图像显示到 LCD，使用了 DMA。是为了释放 CPU，以便执行其他任务。
3. OV9655 可以做到 15 帧。

4. 相机亮度可以微调（分析后可知，是通过一个 ADC 采样，然后去设置摄像头）
5. QQVGA(160x120) or QVGA(320x240) 两种格式。

23.4.1 分析例程代码

从 main 入手分析。

- 主流程分析

第 5 行，调用 OV2640_HW_Init 初始化硬件。第 8/9 行，读 ID(**基本上，调试外设，都是先获取 ID**)。第 11 行，Camera_Config。这个函数在 camera_api.c，没有实际功能，只是根据摄像头型号以及 ImageFormat（图像格式）调用对应摄像头的驱动函数配置摄像头。第 14 行，启动 DMA。第 15 行，开启 DCMI 第 18 行，设置 LCD 显示区域第 19/20 行，准备写数据到 LCD。**在这里直接操作 LCD 寄存器，是一个很粗暴的做法**。然后就进入 while(1) 了，在 while 中，根据 ADC 采样，调节摄像头亮度。

```
/* ADC configuration */
ADC_Config();

/* Initializes the DCMI interface (I2C and GPIO) used to configure the camera */
OV2640_HW_Init();

/* Read the OV9655/OV2640 Manufacturer identifier */
OV9655_ReadID(&OV9655_Camera_ID);
OV2640_ReadID(&OV2640_Camera_ID);
。。。 一些显示 LCD 提示语
Camera_Config();
。。。
/* Enable DMA2 stream 1 and DCMI interface then start image capture */
DMA_Cmd(DMA2_Stream1, ENABLE);
DCMI_Cmd(ENABLE);

/* LCD Display window */
LCD_SetDisplayWindow(179, 239, 120, 160);
LCD_WriteReg(LCD_REG_3, 0x1038);
LCD_WriteRAM_Prepare();

while(1)
{
  /* Get the last ADC3 conversion result data */
  uhADCVal = ADC_GetConversionValue(ADC3);

  /* Change the Brightness of camera using "Brightness Adjustment" register:
```

(continues on next page)

(continued from previous page)

```
      For OV9655 camera Brightness can be positively (0x01 ~ 0x7F)
            and negatively (0x80 ~ 0xFF) adjusted
      For OV2640 camera Brightness can be positively (0x20 ~ 0x40)
            and negatively (0 ~ 0x20) adjusted */
   if(Camera == OV9655_CAMERA)
   {
     OV9655_BrightnessConfig(uhADCVal);
   }
   if(Camera == OV2640_CAMERA)
   {
     OV2640_BrightnessConfig(uhADCVal/2);
   }
 }
```

- OV2640 驱动

例程中的 dcmi_ov2640.c 就是 OV2640 驱动。开头就是几个长数组，这些数组是配置到 OV2640 的，不同数组配置不同的图像格式。话说，这么长的数组，如果没有原厂提供，根据资料你自己能写好吗？所以我认为这个跟 LCD 初始化一样，我们不用去分析，只要用就行了。

```
const char OV2640_QQVGA[][2]
const unsigned char OV2640_QVGA[][2]
const unsigned char OV2640_JPEG_INIT[][2]
const unsigned char OV2640_YUV422[][2]
const unsigned char OV2640_JPEG[][2]
const unsigned char OV2640_160x120_JPEG[][2]
const unsigned char OV2640_176x144_JPEG[][2]
const unsigned char OV2640_320x240_JPEG[][2]
const unsigned char OV2640_352x288_JPEG[][2]
```

接下来是 HW 初始化函数，将对应的 IO 初始化为 DCMI 和 I2C 功能。

```
void OV2640_HW_Init(void)
```

接下来的一个重要函数就是 OV2640 初始化，完成 DCMI 和 DMA 的初始化。

```
/**
  * @brief  Configures DCMI/DMA to capture image from the OV2640 camera.
  * @param  ImageFormat: Image format BMP or JPEG
  * @param  BMPImageSize: BMP Image size
  * @retval None
  */
```

(continues on next page)

(continued from previous page)

```
void OV2640_Init(ImageFormat_TypeDef ImageFormat)
{
  DCMI_InitTypeDef DCMI_InitStructure;
  DMA_InitTypeDef  DMA_InitStructure;

  /*** Configures the DCMI to interface with the OV2640 camera module ***/
  /* Enable DCMI clock */
  RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_DCMI, ENABLE);

  /* DCMI configuration */
  DCMI_InitStructure.DCMI_CaptureMode = DCMI_CaptureMode_Continuous;
  DCMI_InitStructure.DCMI_SynchroMode = DCMI_SynchroMode_Hardware;
  DCMI_InitStructure.DCMI_PCKPolarity = DCMI_PCKPolarity_Rising;
  DCMI_InitStructure.DCMI_VSPolarity = DCMI_VSPolarity_Low;
  DCMI_InitStructure.DCMI_HSPolarity = DCMI_HSPolarity_Low;
  DCMI_InitStructure.DCMI_CaptureRate = DCMI_CaptureRate_All_Frame;
  DCMI_InitStructure.DCMI_ExtendedDataMode = DCMI_ExtendedDataMode_8b;

  /* Configures the DMA2 to transfer Data from DCMI */
  /* Enable DMA2 clock */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE);

  /* DMA2 Stream1 Configuration */
  DMA_DeInit(DMA2_Stream1);

  DMA_InitStructure.DMA_Channel = DMA_Channel_1;
  DMA_InitStructure.DMA_PeripheralBaseAddr = DCMI_DR_ADDRESS;
  DMA_InitStructure.DMA_Memory0BaseAddr = FSMC_LCD_ADDRESS;
  DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;
  DMA_InitStructure.DMA_BufferSize = 1;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Enable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
```

(continues on next page)

(continued from previous page)

```
  switch(ImageFormat)
  {
    case BMP_QQVGA:
    {
      /* DCMI configuration */
      DCMI_InitStructure.DCMI_VSPolarity = DCMI_VSPolarity_High;
      DCMI_Init(&DCMI_InitStructure);

      /* DMA2 IRQ channel Configuration */
      DMA_Init(DMA2_Stream1, &DMA_InitStructure);
      break;
    }
    case BMP_QVGA:
    {
      /* DCMI configuration */
      DCMI_Init(&DCMI_InitStructure);

      /* DMA2 IRQ channel Configuration */
      DMA_Init(DMA2_Stream1, &DMA_InitStructure);
      break;
    }
     default:
    {
      /* DCMI configuration */
      DCMI_InitStructure.DCMI_VSPolarity = DCMI_VSPolarity_High;
      DCMI_Init(&DCMI_InitStructure);

      /* DMA2 IRQ channel Configuration */
      DMA_Init(DMA2_Stream1, &DMA_InitStructure);
      break;
    }
  }
}
```

32~46 行是 DMA 的初始化，前面已经说过 DMA，本处就不累赘了。17~23 是 DCMI 的初始化。

> 17 行，连续模式 18 行，硬件同步 19 行，PCK 上升沿有效 20 行，VSYNC 低电平有效 21 行，HSYNC 低电平有效 22 行，全帧捕获 23 行，8 位数据模式。

48 行 ~77 行，这么多代码完成了什么功能？**这么多代码只做了一件事，就是当 QQVGA 模式时，VSYNC 用高电平有效。QVGA 才用低电平有效。**

剩下的其他函数就是将前面说的数组配置到 OV2640，没什么其他重要功能了。

这样看来，OV2640 用起来也不算难。

23.4.2 移植

将 dcmi_ov9655.c、dcmi_ov9655.h、dcmi_ov2640.c、dcmi_ov2640.h、camera_api.c、camera_api.h 拷贝到我们工程的 board_dev 目录，添加到 SI 跟 MDK 工程，开始移植。程序结构上做以下修改：

1. 将 DCMI 相关函数放到 mcu_dcmi.c 内。
2. 将 SCCB 相关函数放到 mcu_i2c.c 内。
3. 原来 main 函数中的测试代码放到 camera_api.c 中。

差异：

摄像头上没有晶振，需要使用 STM32 的 MCO1 输出时钟给摄像头使用。

1. 初始化

创建了一个初始化函数，硬件相关的初始化都放在这个函数内。官方例程放在 OV9655 和 OV2640 内，两套，不合理。这里所谓的初始化，只是初始化摄像头接口。跟你用什么摄像头，无关。MCO1_Init 就是初始化 MCO 管脚，输出时钟给摄像头。

```
s32 dev_camera_init(void)
{
        /* camera xclk use the MCO1 */
        MCO1_Init();
        DCMI_PWDN_RESET_Init();

        /* Initializes the DCMI interface (I2C and GPIO) used to configure the camera */
        BUS_DCMI_HW_Init();

        SCCB_GPIO_Config();
        return 0;
}
```

1. 修改 I2C 配置，调试到能读到摄像头 ID

也就是修改

```
void SCCB_GPIO_Config(void)
uint8_t bus_sccb_writereg(uint8_t DeviceAddr, uint16_t Addr, uint8_t Data)
uint8_t bus_sccb_readreg(uint8_t DeviceAddr, uint16_t Addr)
```

查参考手册，我们用的硬件 I2C 是 I2C2，把上面 3 个函数中的控制器全部改为 I2C2？—-这方法不好，应该全部改为宏定义。宏定义改起来方便。

调试信息，能正确读到 ID。

> —hello world!—— init finish! read reg:9341 lcd init ok! camera test…. OV9655 Camera ID 0x96 Camera_Config…test camera! test camera! test camera! test camera!

1. 移植 DCMI 跟 DMA 功能

   修改 DCMI 相关 IO。修改 LCD 显示 RAM 的地址，在 LCD 章节，我们说过这个地址为什么是 0x6C010000。

```
#define DCMI_DR_ADDRESS       0x50050028
#define FSMC_LCD_ADDRESS      0x6C010000
```

1. 摄像头数据采集方向和 LCD 扫描方向要一致。

LCD 驱动器 ILI9341 如果配置为横屏模式，则要先左右，后上下；先左还是先右，先上还是先下，图像会不一样。如果配置为竖屏，则需要使用后面四种扫描方向，也就是先上下，后左右。简单的说，就是先扫描 320 像素长边，再扫描 240 像素的短边。

```
#define L2R_U2D  (0) //从左到右，从上到下
#define L2R_D2U  (0 + UD_BIT_MASK)//从左到右，从下到上
#define R2L_U2D  (0 + LR_BIT_MASK) //从右到左，从上到下
#define R2L_D2U  (0 + UD_BIT_MASK + LR_BIT_MASK) //从右到左，从下到上

#define U2D_L2R  (LRUD_BIT_MASK)//从上到下，从左到右
#define U2D_R2L  (LRUD_BIT_MASK + LR_BIT_MASK) //从上到下，从右到左
#define D2U_L2R  (LRUD_BIT_MASK + UD_BIT_MASK) //从下到上，从左到右
#define D2U_R2L  (LRUD_BIT_MASK + UD_BIT_MASK+ LR_BIT_MASK) //从下到上，从右到左
```

摄像头调试过程，参考官方例程，很快就调通了。

1. 代码结构调整

最终的代码，我们对原厂的例程进行了部分的重新封装和调整。使得代码架构层次清晰，模块化更好。

23.5 总结

无

23.6 end

CHAPTER 24

USB 调试记录

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

前面有几节在讨论移植官方例程时都提到过 USB 例程。现在就让我们开始移植 USB 例程。

24.1 USB

USB 大家都不陌生，但是 USB 有很多概念，估计大家不一定都能分清楚。

USB，是英文 Universal Serial Bus（通用串行总线）的缩写。

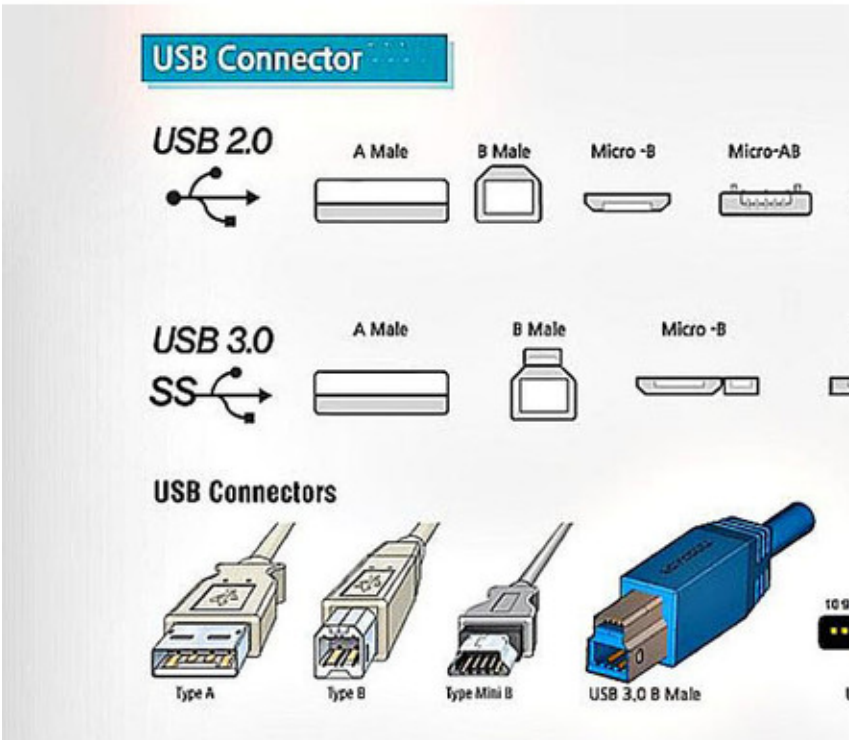
24.1.1 协议版本

USB1.1 普遍的 USB 规范，其高速方式的传输速率为 12Mbps，低速方式的传输速率为 1.5Mbps（b 是 Bit 的意思），1MB/s（兆字节/秒）=8MBPS（兆位/秒），12Mbps=1.5MB/s。**USB2.0** 由 USB1.1 规范演变而来的。它的传输速率达到了 480Mbps。USB 2.0 标准将 USB 接口速度划分为三类，从高到低分别为 480Mbps、12Mbps 和 1.5Mbps。High-speed 25Mbps ~ 400Mbps(最大 480Mbps) 视频、存储、照片和宽带 Full-speed 500Kbps ~ 10Mbps(最大 12Mbps) 宽带、音频和耳麦 Low-speed 10Kbps ~ 100Kbps(最大 1.5Mbps) 键盘、鼠标和游戏外设 **USB OTG** 基本兼容 USB2.0 协议。通过一根 ID 线，确认 HOST 还是 SLAVE。**USB3.0** USB3.0 ——也被认为是 SuperSpeedUSB。实际传输速率大约是 3.2Gbps（即 320MB/S）。理论上的最高速率是 5.0Gbps（即 500MB/S）。USB3.0 引入全双工数据传输。5 根线路中 2 根用来发送数据，另 2 根用来接收数据，还有 1 根是地线。

24.1.2 设备分类

HOST 主机，通常的电脑就是 HOST 设备。**SLAVE** 从机，U 盘，鼠标键盘等外设就是 SLAVE 设备。**OTG** OTG 就是 On The Go，正在进行中的意思。既能充当 HOST，亦能充当 SLAVE。现在的智能手机，大部分都是 OTG 设备了。

24.1.3 硬件接口



先来一个图(图片出自网络),图中是各种 USB 接口：

接口种类 **Type-A** 电脑上的基本都是 Type-A 接口。**Type-B** 打印机，或者以前一些老的开发板会使用 B 口。**Type-c** 新智能手机流行接口。支持 3.0 协议。**Mini-USB** 2.0 协议定义的接口类型，以前 MP3 常用，

5PIN，支持 OTG **Micro-USB** Micro USB 是 USB 2.0 标准的一个便携版本，在 TYPE-C 之前，智能手机使用 Micro 接口。5PIN，支持 OTG。我们的硬件，3 个 USB 接头都是 Micro 接口。**以上接口类型，全部有公头母座之分**

24.1.4 设备类型

一个 USB 设备，使用 USB 线，根据 USB 规范通信。为了规范，还对这些设备的行为进行了分类，定义了子协议，也就区分了不同的设备。USB 定义了种类代码信息，它被用来识别设备的功能，根据这些功能，以加载设备驱动。这种信息包含在名为基类，子类和协议的 3 个字节里。这些类，也就是我们常常听说的：HID、CDC 等。

24.2 STM32F407 USB

STM32F407 有两个 USB 接口：FS（全速）、HS 高速。而且两个 USB 都是 OTG 接口。高速 USB 需要外接 USB 芯片。本次我们调试的是全速 USB。

24.2.1 USB 简介

本节介绍了 OTG_FS 控制器的架构和编程模型。

使用了以下首字母缩略词：

FS	全速
LS	低速
MAC	介质访问控制器
OTG	On-the-go
PFC	数据包 FIFO 控制器
PHY	物理层
USB	通用串行总线
UTMI	USB 2.0 收发器宏单元接口 (UTMI)

参考文档如下：

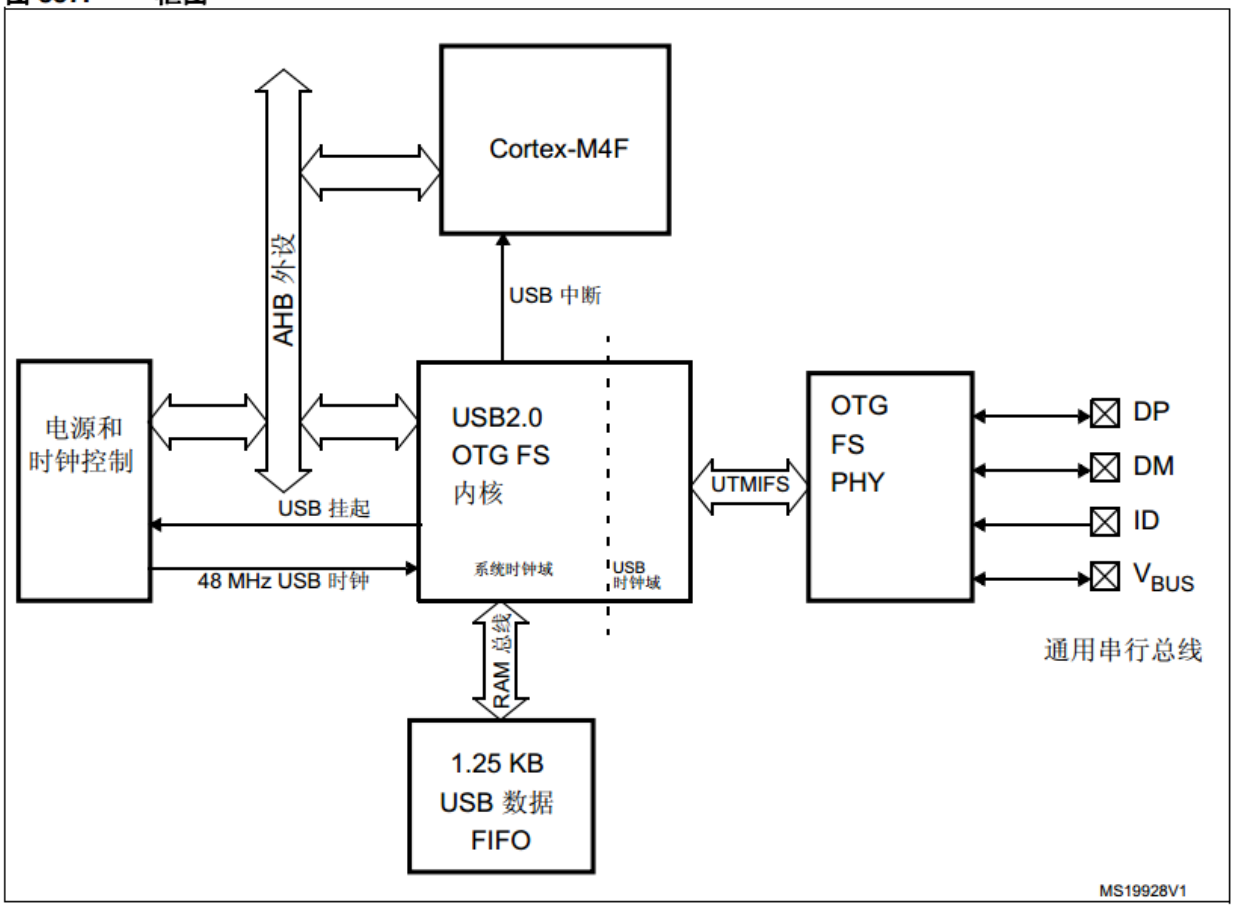
- USB On-The-Go 补充标准，第 1.3 版
- 通用串行总线规范第 2.0 版

OTG_FS 是一款双角色设备 (DRD) 控制器，同时支持从机功能和主机功能，完全符合 *USB 2.0 规范的 On-The-Go 补充标准*。此外，该控制器也可配置为“仅主机”模式或“仅从机”模式，完全符合 *USB 2.0 规范*。在主机模式下，OTG_FS 支持全速（FS，12 Mb/s）和低速（LS，1.5 Mb/s）收发器，而从机模式下则仅支持全速（FS，12 Mb/s）收发器。OTG_FS 同时支持 HNP 和 SRP。主机模式下需要的唯一外部设备是提供 V_{BUS} 的电荷泵。

USB 简介

24.2.2 USB 框图

图 357. 框图



USB 框图

24.3 ST USB 协议栈

要熟悉一个软件，最快的速度还是**阅读官方文件**。下面路径可以下载 STM32 USB 例程，文档名称 UM1021。包含一个 PDF 跟一个 75M 的例程。http://www.stmcu.org/document/detail/index/id-213598

> CD00289278.pdf stm32_f105-07_f2_f4_usb-host-device_lib.zip

ST 官方推广时，会针对某些外设进行介绍，我们可以通过这些文档大概了解 STM32 的外设使用情况。在下面路径，左边资料栏里，有每年新 IC 发布时的培训资料。http://www.stmcu.org/document/list/index/category-466 STM MCU 资料我们可以找到 2012 年 USB 的培训资料

> USB 培训 _Part1_ 协议.pdf USB 培训 _Part2_USB_IP 及其库的使用.pdf USB 培训 _Part3_USB_OTG_IP 及其库的使用.pdf

这些资料请自行消化。建议先看培训资料。

24.3.1 USB 库说明

从 CD00289278.pdf 看起。

24.3.2 例程文件结构

官方例程文件结构 USB 例程文件结构其中

> Libraries 中，有 3 个 USB 库，分别是 Device、HOST、OTG。Project 目录下有 3 种例程，分别对应 Device、HOST、Host_Device。

我们移植 Host_Device 里面的 OTG 工程 DRD。

24.3.3 DRD 例程大概流程

- 主流程 Demo_Init 初始化，然后就进入 while，Demo_Process 函数。

```
int main(void)
{
  __IO uint32_t i = 0;

  /*!< At this stage the microcontroller clock setting is already configured,
  this is done through SystemInit() function which is called from startup
  file (startup_stm32fxxx_xx.s) before to branch to application main.
  To reconfigure the default setting of SystemInit() function, refer to
  system_stm32fxxx.c file
  */

  Demo_Init();

  while (1)
  {

    Demo_Process();

    if (i++ == 0x10000)
    {
      STM_EVAL_LEDToggle(LED1);
      STM_EVAL_LEDToggle(LED2);
      STM_EVAL_LEDToggle(LED3);
      STM_EVAL_LEDToggle(LED4);
      i = 0;
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

Demo_Init 函数中 USB 相关的只有下面几句代码，调用 USBH_Init 初始化为 HOST 模式。

```
USBH_Init(&USB_OTG_Core,
#ifdef USE_USB_OTG_FS
            USB_OTG_FS_CORE_ID,
#elif defined USE_USB_OTG_HS
            USB_OTG_HS_CORE_ID,
#endif
            &USB_Host,
            &USBH_MSC_cb,
            &USR_USBH_MSC_cb);

  USB_OTG_BSP_mDelay(500);
  DEMO_UNLOCK();
```

Demo_Process 函数，如果是 HOST 模式，调用 USBH_Process 处理。Demo_Application 是 USB 应用层流程，也是主要流程。

```
void Demo_Process (void)
{
  if(demo.state == DEMO_HOST)
  {
    if(HCD_IsDeviceConnected(&USB_OTG_Core))
    {
      USBH_Process(&USB_OTG_Core, &USB_Host);
    }
  }
  Demo_Application();

}
```

Demo_Application 函数中，根据流程步骤进行处理

```
switch (demo.state)
{
case  DEMO_IDLE:
break;
```

(continues on next page)

(continued from previous page)

```
case  DEMO_WAIT:
break;
case  DEMO_HOST:
break;
case  DEMO_DEVICE:
break;
```

在 DEMO_WAIT 步骤中，等待用户选择 HOST 模式还是 DEVICE 模式。DEMO_HOST 步骤跟 DEMO_DEVICE 步骤又分别处理 USB 状态。

这整个就是一个状态机。

整个主流程只是应用层的。

24.3.4 USB 协议栈大概

哪么 USB 是怎么处理的呢? USB 协议大部分使用回调函数。在选择模式后，就会初始化 USB。例如 HOST 模式:

```
USBH_Init(&USB_OTG_Core,
#ifdef USE_USB_OTG_FS
          USB_OTG_FS_CORE_ID,
#elif defined USE_USB_OTG_HS
          USB_OTG_HS_CORE_ID,
#endif
          &USB_Host,
          &USBH_MSC_cb,
          &USR_USBH_MSC_cb);
```

USBH_MSC_cb 和 USR_USBH_MSC_cb 就是回调函数列表，cb 就是 call back 的意思。这两个结构体分别是:

```
USBH_Class_cb_TypeDef  USBH_MSC_cb =
{
  USBH_MSC_InterfaceInit,
  USBH_MSC_InterfaceDeInit,
  USBH_MSC_ClassRequest,
  USBH_MSC_Handle,
};

USBH_Usr_cb_TypeDef USR_USBH_MSC_cb =
{
```

(continues on next page)

(continued from previous page)

```
  USBH_USR_Init,
  USBH_USR_DeInit,
  USBH_USR_DeviceAttached,
  USBH_USR_ResetDevice,
  USBH_USR_DeviceDisconnected,
  USBH_USR_OverCurrentDetected,
  USBH_USR_DeviceSpeedDetected,
  USBH_USR_Device_DescAvailable,
  USBH_USR_DeviceAddressAssigned,
  USBH_USR_Configuration_DescAvailable,
  USBH_USR_Manufacturer_String,
  USBH_USR_Product_String,
  USBH_USR_SerialNum_String,
  USBH_USR_EnumerationDone,
  USBH_USR_UserInput,
  USBH_USR_MSC_Application,
  USBH_USR_DeviceNotSupported,
  USBH_USR_UnrecoveredError

};
```

我们看 USR 里面的函数，有初始化，复位设备，断开设备，电流溢出等处理函数。当 USB 协议栈发生这些消息是，就会回调这些函数处理。

24.3.5 bsp

与硬件相关的操作放在 usb_bsp.c 文件内。

```
void USB_OTG_BSP_Init(USB_OTG_CORE_HANDLE *pdev)
void USB_OTG_BSP_EnableInterrupt(USB_OTG_CORE_HANDLE *pdev)
void USB_OTG_BSP_DriveVBUS(USB_OTG_CORE_HANDLE *pdev, uint8_t state)
void  USB_OTG_BSP_ConfigVBUS(USB_OTG_CORE_HANDLE *pdev)
static void USB_OTG_BSP_TimeInit ( void )
。。。。
```

大概有两部分：硬件初始化和 VBUS 控制。VBUS 就是控制是否输出电压到 USB 口，当 HOST 模式时，就需要提供电源。

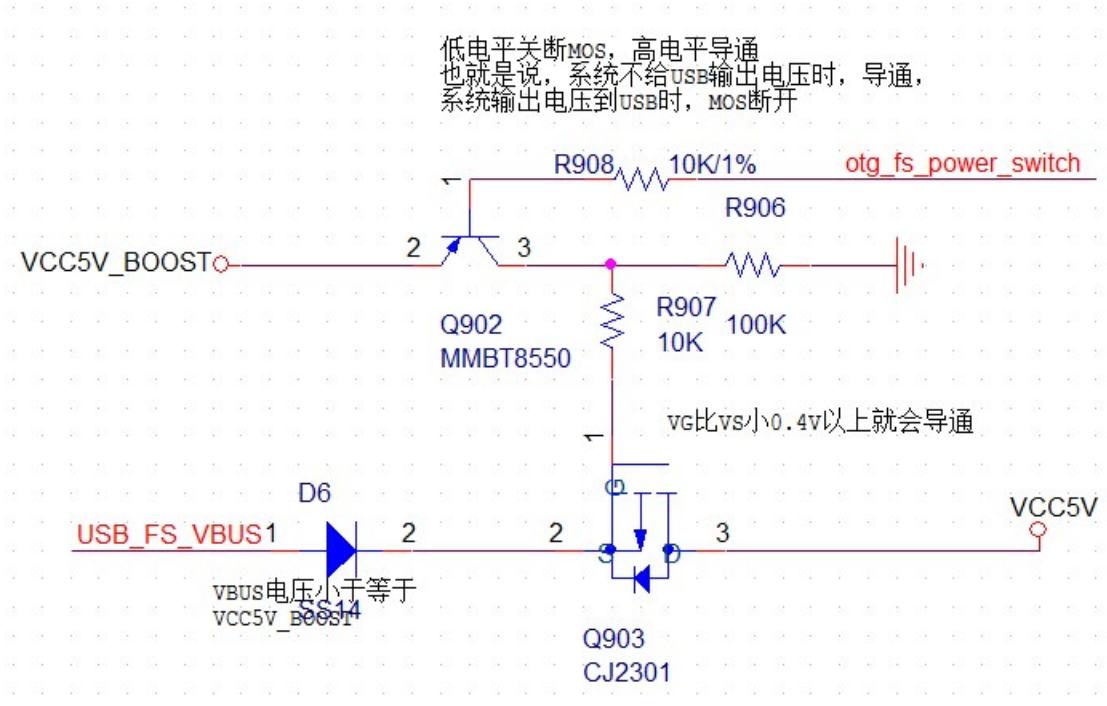
24.3.6 三节

从上面分析可以看出，USB 大概分 3 节：**应用、协议栈、底层 BSP**。我们要管的是两头：应用层和底层。应用层就是 Demo_Application。底层就是 BSP 硬件相关的，底层只要根据硬件修改一次就可以了。

24.4 硬件说明

24.4.1 接口

接口使用 Micro 接口，电气信号如下图 USB 电气信号 5 根信号线



DEVICE 供电接 U 盘，STM32 作为 HOST，otg_fs_power_switch 输出低电平，升压得到的 5V 通过 U901 输出，供电给 U 盘，同时 Q903 MOS 管关断，防止 USB_FS_VBUS 反向倒灌到系统 5V 电源。接电脑，STM32 作为 DEVICE，otg_fs_power_switch 转输入或者高阻态，U901 关断，Q903 导通，USB_FS_VBUS 供电给核心板使用。电路没加保险丝，当 LCD，摄像头等都接上时，电流会大于 500ma，直接接电脑可能会过流，建议通过带电源的 HUB 接到电脑。

24.5 移植调试过程

1 首先将 STM32_USB-Host-Device_Lib_V2.2.0\Libraries 下的 USB 库拷贝到我们工程的 StLib 目录下 拷贝 USB 库

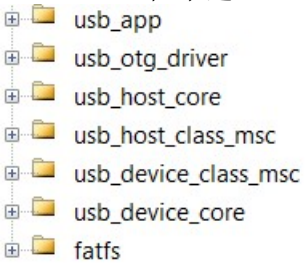
2 例程工程跟 readme 文件 STM32_USB-Host-Device_Lib_V2.2.0\Project\USB_Host_Device_Examples\DRD

2.2.0 例程里面没有 MDK 工程，只有 IAR 工程，如果没安装 IAR，可以看 2.1.0 库里面的例程。

工程主要有 3 部分：

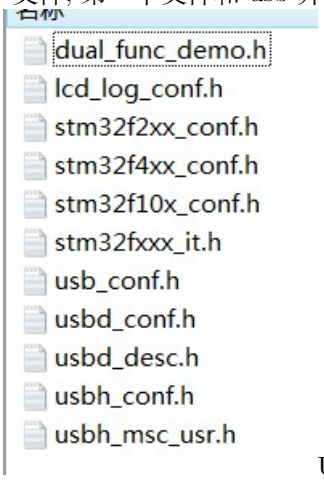
- USB 应用 DRD 例程 IAR 工程 APP
- 文件系统 DRD 例程 IAR 工程文件系统
- usb 库 DRD 例程 IAR 工程 USB 库

3 在 MDK 工程中建立 USB 文件组织结构并添加对应文件在 MDK 中我们将文件如下组织



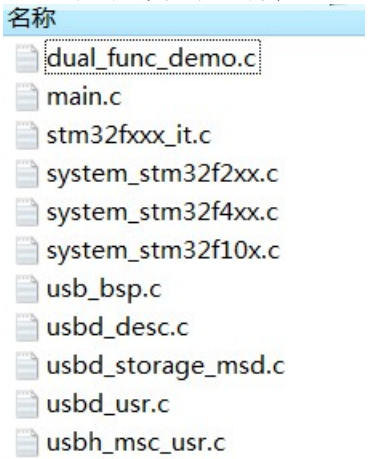
MDK 工程文件组织方式

4 在我们 MDK 工程 app 目录下，建立 usb 目录，并将例程目录 Project\USB_Host_Device_Examples\DRD 下的 inc 跟 src 文件夹拷贝过来。inc 目录有以下文件，第一个文件和 usb 开头的文件是我们需要的，其他文件在前面例程已经添加，没有什么差异就删除。



USB DRD INC

SRC 目录有以下文件，main.c 跟 stm32fxxx_it.c 的代码需要移植。其他文件拷贝到 usb_app 目录



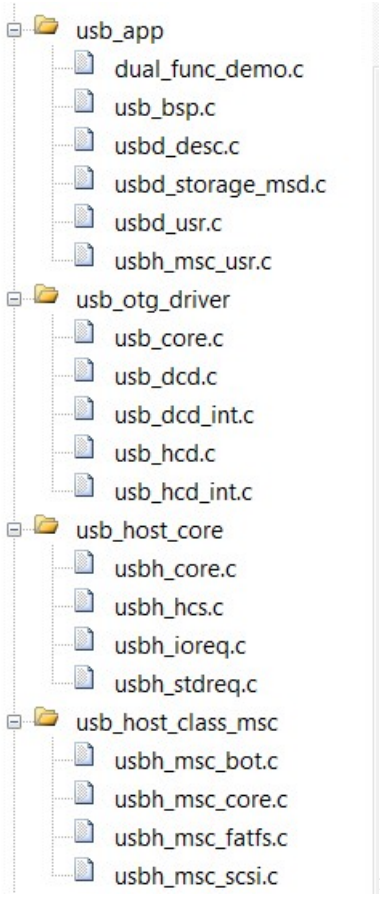
USB DRD SRC

main.c 很简单，主要代码都在 dual_func_demo.c stm32fxxx_it.c 有三个中断函数需要处理，移植到我们自

已的 stm32fxxx_it.c

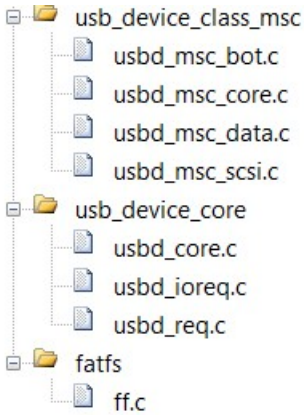
EXTI1 是电流检测 IO 口中断 TIM2_IRQHandler 不知道用来做什么，估计是超时管理，细节回头再研究。OTG_FS_IRQHandler 是 USB 的主要中断。

5 将相关文件添加到 MDK 工程



添加文件到工程

core 和 fatfs



添加文件到工程记得添加头文件路径。

6 查看 app\usb\inc 下的文件，根据实际情况配置。也可以先编译一次，根据错误提示修改。主要修改点：

1 usb_conf.h 中根据不同的官方硬件，包含了不同的头文件，把这些包含全部屏蔽。2 在 usb_conf.h 中打开 USE_USB_OTG_FS 宏定义 3 经常搞不懂的 VBUS 应用，在现在的工程内已

经没有定义。在 usb_conf_template.h 范例里面还有//#define VBUS_SENSING_ENABLED。

7 修改 usb_bsp.c，根据硬件配置修改。过流检测我们用 PE2。电源开关控制，用的是 PC0。过流检测中断函数也需要修改到 IO 中断 2（EXTI_Line2）。

```
/**
  * @brief  EXTI2_IRQHandler
  *         This function handles External line 1 interrupt request.
  * @param  None
  * @retval None
  */
void EXTI2_IRQHandler(void)
{
  if(EXTI_GetITStatus(EXTI_Line2) != RESET)
  {
      USB_Host.usr_cb->OverCurrentDetected();
      EXTI_ClearITPendingBit(EXTI_Line2);
  }
}
```

其中 ID/DP/DN 也要修改。在 main 函数中调用 dual_func_demo.c 文件内的 usb_main 函数。

更多修改细节，可以通过对比官方例程，有些小地方修改过没记录

24.5.1 U 盘测试

通过一根 OTG 转接线，将 U 盘接到核心板上的 micro 接口。

修改 Demo_Application 函数内的 case DEMO_WAIT，强制设置为 HOST 模式。

```
        /* 选择 HOST 还是 DEVICE*/
  case  DEMO_WAIT:
        demo.state = DEMO_HOST;
    demo.Host_state = DEMO_HOST_IDLE;
        //demo.state = DEMO_DEVICE;
    //demo.Device_state = DEMO_DEVICE_IDLE;
    break;
```

通过一个 OTG 转接头，插上 U 盘。识别金士顿 U 盘，U 盘内有一个 wav 文件。

```
Board : wujique stm32f407.
Device: STM32F407.
USB Host Library v2.2.0.
USB Device Library v1.2.0.
```

(continues on next page)

(continued from previous page)

```
USB OTG Driver v2.2.0
STM32 Std Library v1.5.0.
> Full speed device detected
> Mass storage device connected
> Manufacturer : Kingston
> Product : DataTraveler 2.0
> Serial Number : 5B811D00168F
> Enumeration completed
> USB Host Full speed initialized.
> File System initialized.
> Disk capacity : 1998585344 Bytes
   |__STEREO~1.WAV
```

24.5.2 读卡器测试

使用一根手机数据线将核心板连到电脑。

1 实现 sd 卡的 diso 接口。参考 usbh_msc_fatfs.c 文件，在 stm324xg_eval_sdio_sd.c 实现要 SD 卡的操作函数，并添加到 diskio.c 文件内。

```
extern DSTATUS SD_disk_initialize (
                         BYTE drv                /* Physical drive number (0) */
                           );
extern DSTATUS SD_disk_status (
                       BYTE drv                /* Physical drive number (0) */
                         );
extern DRESULT SD_disk_read (
                    BYTE pdrv,                      /* Physical drive number (0) */
                    BYTE *buff,                      /* Pointer to the data buffer to␣
↪store read data */
                    DWORD sector,                /* Start sector number (LBA) */
                    UINT count                       /* Sector count (1..255) */
                      );
extern DRESULT SD_disk_write (
                     BYTE pdrv,                      /* Physical drive number (0) */
                     const BYTE *buff,        /* Pointer to the data to be written */
                     DWORD sector,                /* Start sector number (LBA) */
                     UINT count                       /* Sector count (1..255) */
                       );
```

(continues on next page)

(continued from previous page)

```
extern DRESULT SD_disk_ioctl (
                    BYTE drv,                /* Physical drive number (0) */
                    BYTE ctrl,                /* Control code */
                    void *buff                /* Buffer to send/receive control data */
                      );
```

2 修改 Demo_Application 函数内的 case DEMO_WAIT，强制设置为 DEVIC 模式。

```
        /* 选择 HOST 还是 DEVICE*/
  case  DEMO_WAIT:
        //demo.state = DEMO_HOST;
    //demo.Host_state = DEMO_HOST_IDLE;
        demo.state = DEMO_DEVICE;
    demo.Device_state = DEMO_DEVICE_IDLE;
    break;
```

3 编译下载，用 MICRO 线将开发板连到电脑，电脑能识别到 SD 卡内的文件。LOG 如下：

```
Board : wujique stm32f407.
Device: STM32F407.
USB Host Library v2.2.0.
USB Device Library v1.2.0.
USB OTG Driver v2.2.0
STM32 Std Library v1.5.0.
> Single Lun configuration.
> microSD is used.
> Device In suspend mode.

-------SD_PowerON ok----------
-------SD_InitializeCards ok----------
-------SD_csd.DeviceSize:15701----------
-------SD_GetCardInfo ok----------
-------CardCapacity:00000001----------
-------CardCapacity:EAB00000----------
-------CardBlockSize:512 ----------
-------RCA:8597 ----------
-------CardType:2 ----------
-------SD_SelectDeselect ok----------
SDIO_GetResponse ok
FindSCR:0
SDEnWideBus:0
```

(continues on next page)

(continued from previous page)

```
-------SD_EnableWideBusOperation:0----------
> MSC Interface started.
```

24.6 总结

当前代码有以下问题需要解决：1 能不能自动识别 HOST/DEVICE? 2 如果系统在使用 SD 卡，能同时运行读卡器程序吗?

24.7 end

CHAPTER 25

ETH LAN8720 调试记录

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

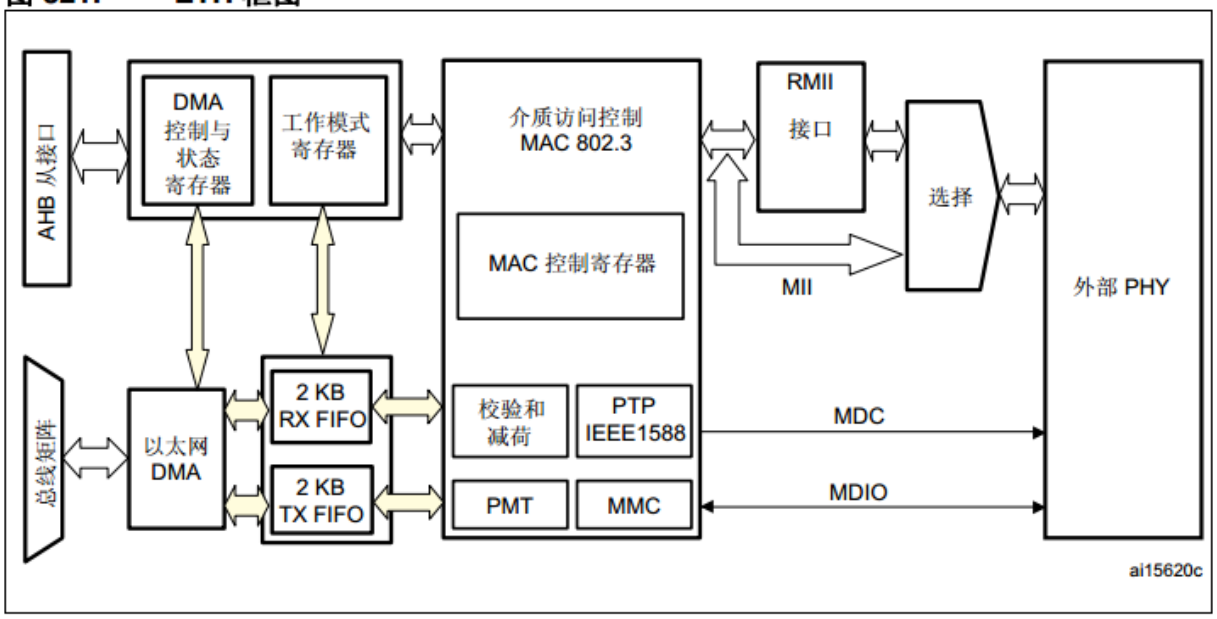
QQ 群：767214262

本节我们开始调试网络功能。主要分三部分:STM32 以太网控制器、PHY 芯片 LAN8720、LWIP 协议栈。

25.1 STM32 MAC 控制器

25.1.1 框图

图 321. ETH 框图



1. STM32 的以太网是基于 DMA 控制器的。
2. 介质访问控制，也就是我们通常说的 MAC 控制器。这个是以太网功能的核心部分。
3. 以太网提供 3 种接口：SMI、MII、RMII。
4. SMI 叫做站管理接口。用来访问 PHY 的寄存器。
5. MII&RMII 功能一样，都是 MAC 控制器跟 PHY 进行数据传输的接口。RMII 是精简的 MII，用更少的 IO 口。
6. 框图里面的外部 PHY 并不包含在 STM32 芯片内，我们外部的 LAN8720 就是这个外部 PHY。
7. 在 PHY 外，应该还有一个带变压器的网口。

25.1.2 特性

在《STM32F4xx 中文参考手册.pdf》中列出了以太网的 3 种特性

1. MAC 内核特性
2. DMA 特性
3. PTP 特性

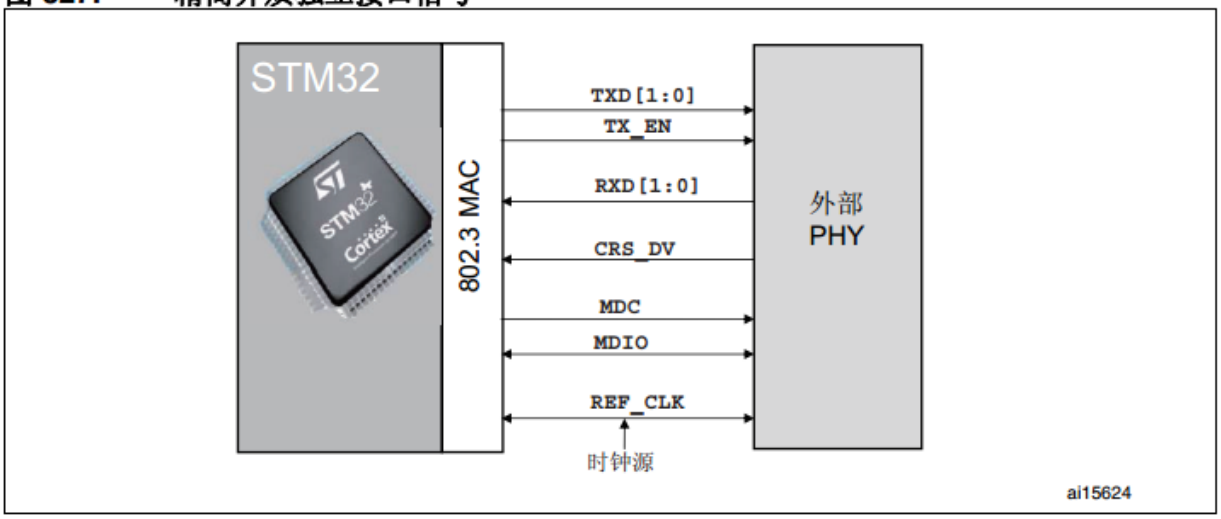
细节请看文档。

25.1.3 RMII 接口

精简介质独立接口 (RMII) 只要用 7 个引脚 (MII 需要 16 个)，因此我们选用这个接口控制 PHY 芯片。RMII 有以下特性：

支持 10/100M 运行速率参考时钟必须是 50MHz 相同的参考时钟必须从外部提供给 MAC 和外部 PHY 提供独立的 2 位宽的发送和接收数据路径

图 327. 精简介质独立接口信号



图中的 REF_CLK 是共用的参考时钟，50MHz。我们用 LAN8720 方案，这个时钟由 LAN8720 提供给 STM32 MAC 控制器。

25.2 LAN8720A 芯片

LAN8720A 是低功耗的 10/100M 以太网 PHY 芯片，支持通过 RMII 接口和 MAC 层通信。

25.2.1 特性

- 10/100M
- 支持 RMII 接口
- 支持全双工和半双工
- 使用外部 25M 晶振，生成 50MHz 参考时钟给 MAC 层使用
- 支持自协商模式
- 支持 HP Auto-MDIX 自动翻转
- 支持 SMI 串行管理接口

25.2.2 框图

- 内部框图

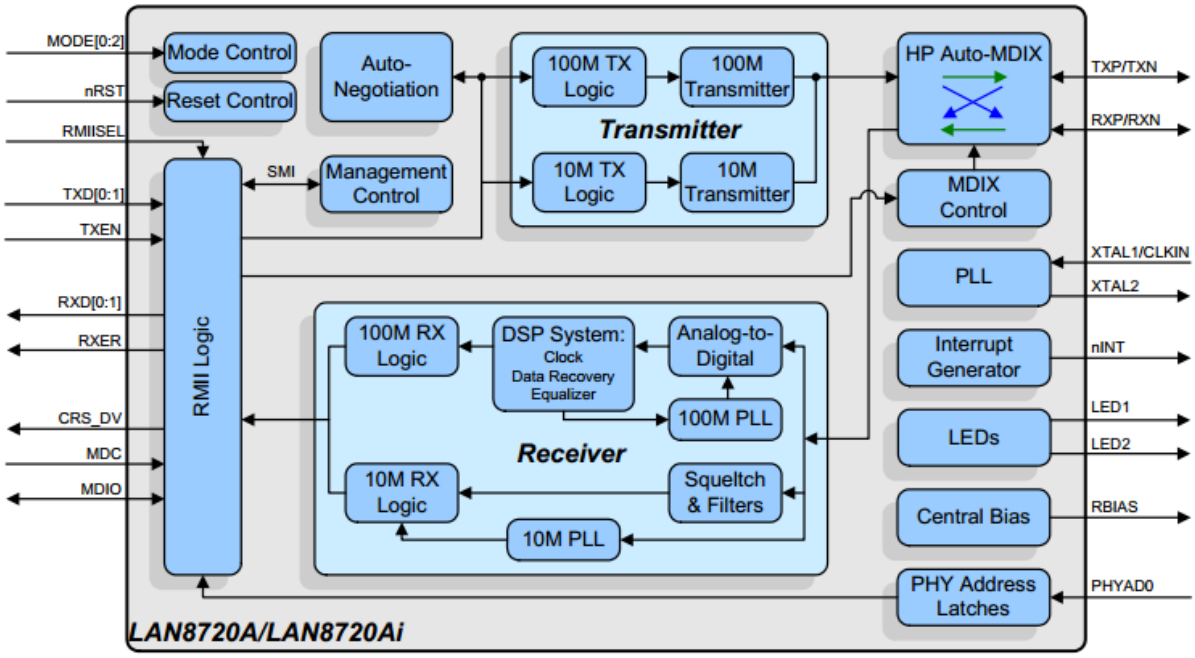


Figure 1.2 Architectural Overview

- 应用图

左边 10/100M 网络控制器就是 STM32 内部的 MAC 控制器。右边的 RJ45 就是网口。下边框图说明 LAN8720 需要一个外部晶振。

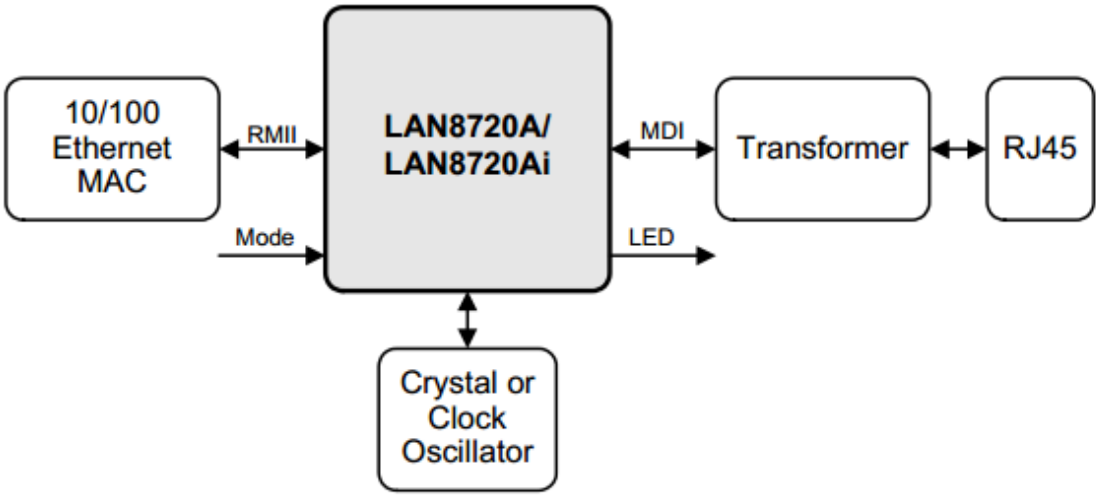


Figure 1.1 System Block Diagram

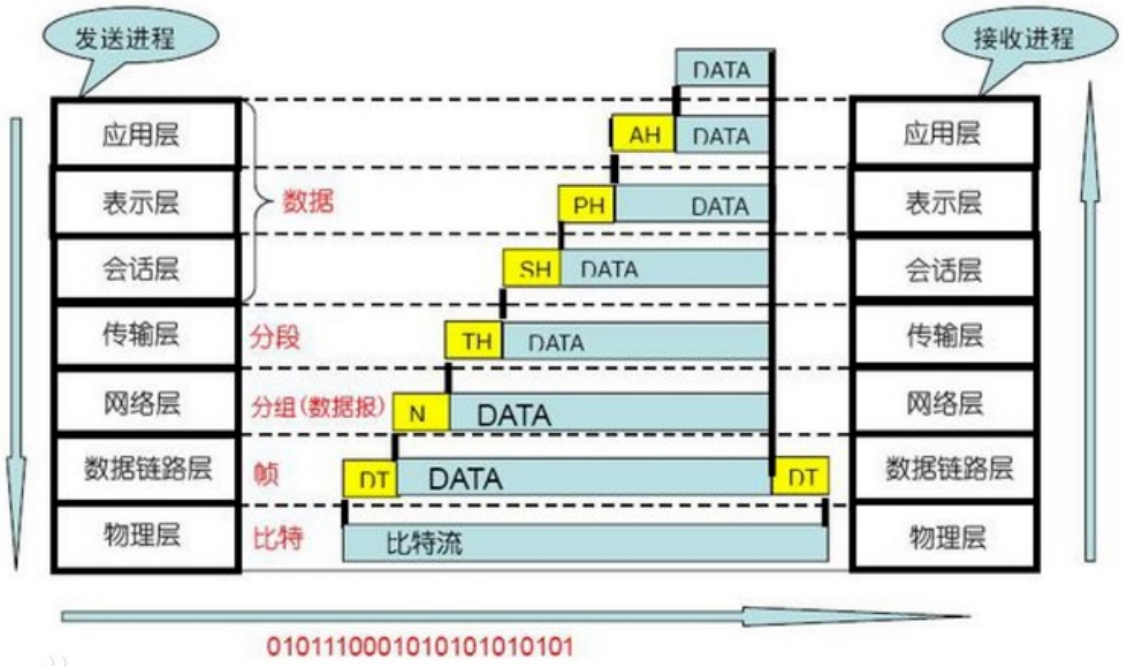
更多请参考《LAN8720A.pdf》文档。

25.3 LWIP

现在我们天天上网，基本上都知道，有一种 TCP/IP 协议。协议是什么？最底层的协议就是数据传输的格式。可以相当于网络上的一种语言。高级的协议，就是一种行为规范。以太网世界设备几十亿，如果没有行为规范，机器之间就无法正常进行通信。不理解的话可以想象在一间房子里面有 10 个不同国家的人，各自都在胡言乱语。

25.3.1 网络七层协议

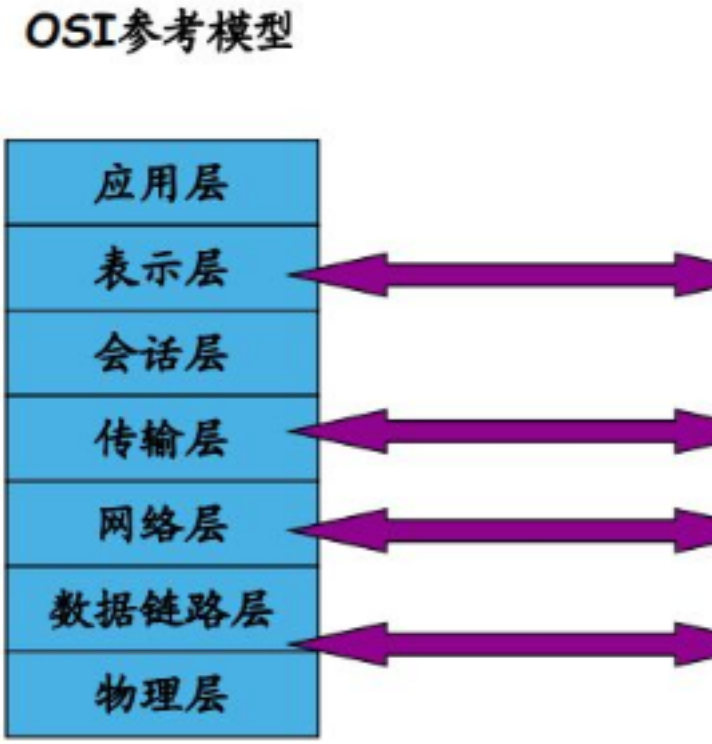
国际标准化组织 ISO 于 1981 年正式推荐了一个网络系统结构—-七层参考模型，叫做开放系统互连模型（Open System Interconnection，OSI)。由于这个标准模型的建立，使得各种计算机网络向它靠拢，大大推动了网络通信的发展。OSI 参考模型将整个网络通信的功能划分为七个层次，见图。它们由低到高分别是物理层（PH)、数据链路层（DL)、网络层（N)、传输层（T)、会话层（S)、表示层（P)、应用层（A)。每层完成一定的功能，每层都直接为其上层提供服务，并且所有层次都互相支持。第四层到第七层主要负责互操作性，而一层到三层则用于创造两个网络设备间的物理连接。



25.3.2 TCP/IP 协议

Transmission Control Protocol/Internet Protocol 的简写，中译名为传输控制协议/因特网互联协议，又名网络通讯协议，是 Internet 最基本的协议、Internet 国际互联网络的基础，由网络层的 IP 协议和传输层的 TCP 协议组成。TCP/IP 定义了电子设备如何连入因特网，以及数据如何在

它们之间传输的标准。协议采用了 4 层的层级结构，每一层都呼叫它的下一层所提供的协议来完成自己的需求。通俗而言：TCP 负责发现传输的问题，一有问题就发出信号，要求重新传输，直到所有数据安全正确地传输到目的地。而 IP 是给因特网的每一台联网设备规定一个地址。



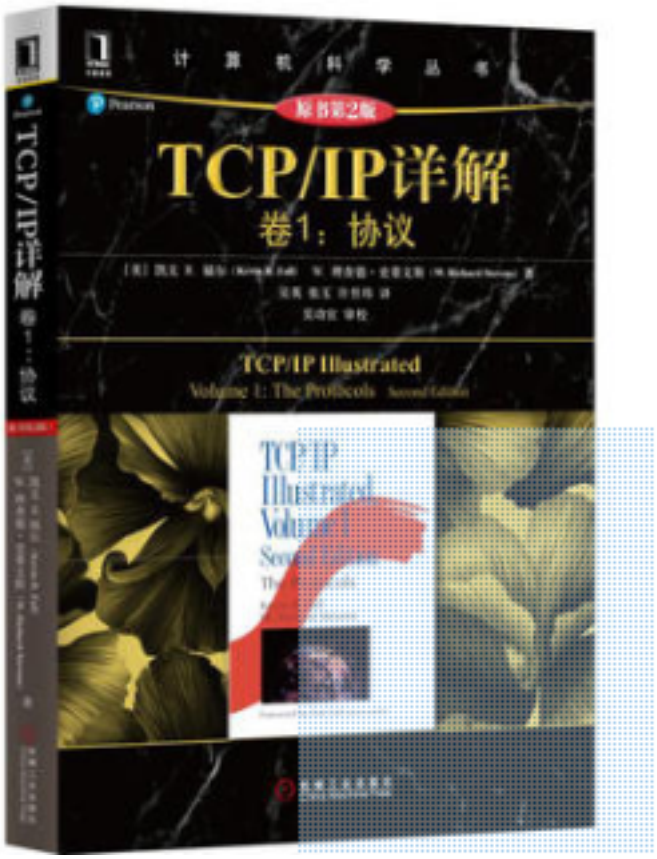
TCP/IP 协议只使用了 4 层结构,跟 OSI 的对应关系如下图。

25.3.3 LWIP

LwIP 是 Light Weight (轻型)IP 协议，有无操作系统的支持都可以运行。LwIP 实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用，它只需十几 KB 的 RAM 和 40K 左右的 ROM 就可以运行，这使 LwIP 协议栈适合在低端的嵌入式系统中使用。lwIP 协议栈主要关注的是怎么样减少内存的使用和代码的大小，这样就可以让 lwIP 适用于资源有限的小型平台例如嵌入式系统。为了简化处理过程和内存要求，lwIP 对 API 进行了裁减，可以不需要复制一些数据。

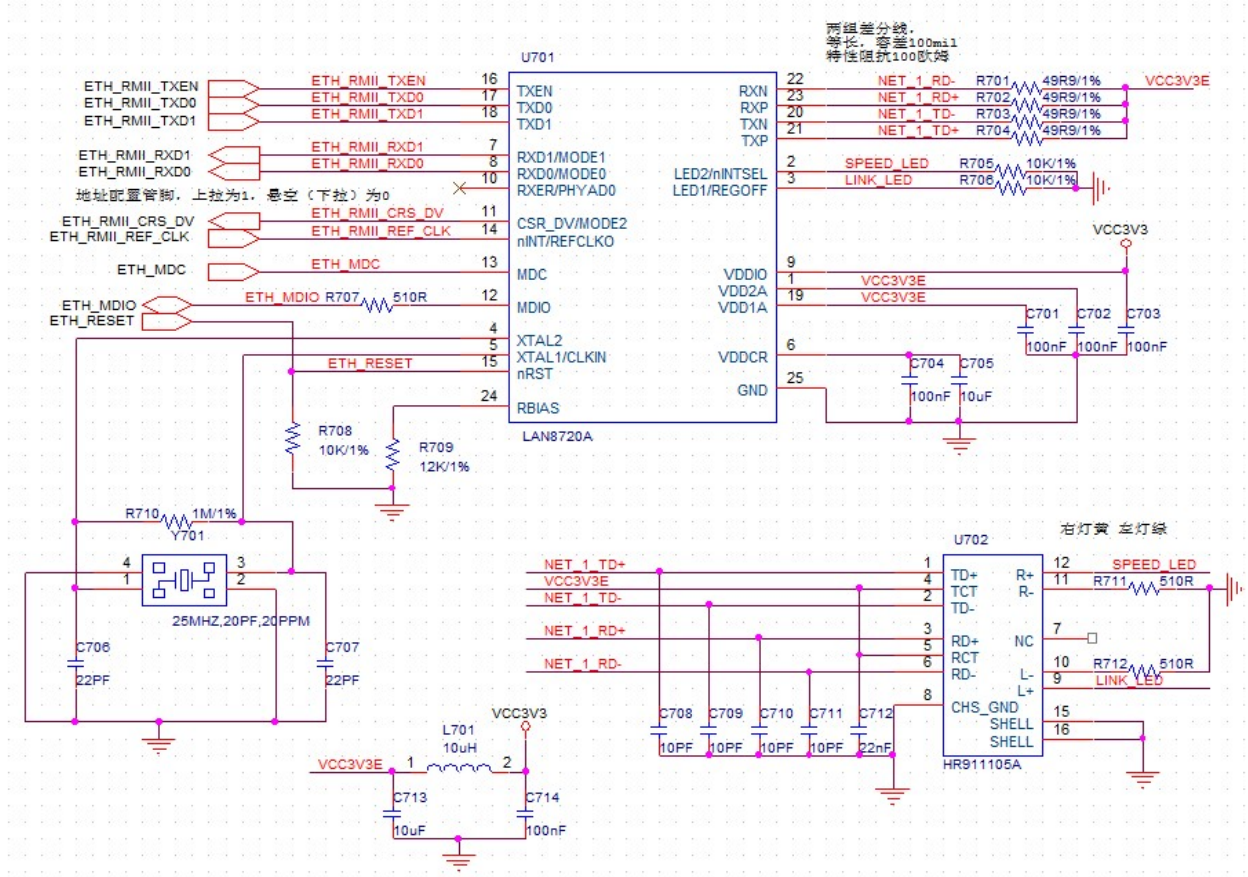
25.3.4 学习

我们本次只是移植官方以太网的例子。不会对 LWIP 做深入学习。因为 TCP/IP 协议太复杂了。以前公司做无线通信的同事，每天都抗一本书看，对，就是下面这本，他说这书是一套，总共更有 3 本。



如果大家想学习网络协议，可以买这个书看看。如果只是想了解 LWIP 的使用，那就先从例程上学习学习，再看看 LWIP 的结构跟接口就可以了。**以后我们会单独出一个对于 *LWIP* 的使用说明**

25.4 原理说明

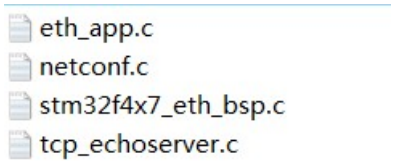


原理图

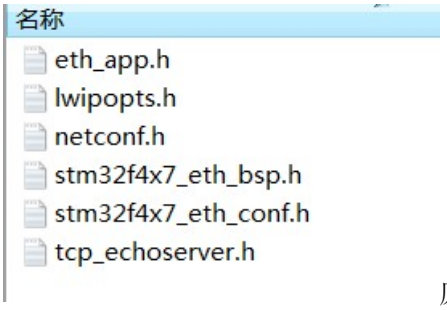
1. 原理图分两部分：PHY 芯片 LAN8720A、HR911105A（带变压器 RJ45 网口）。
2. PHY 芯片通过 RMII 接口与 STM32 内部 MAC 层通信。
3. LAN8720 需要一个 25M 的晶振。
4. STM32 通过一个 SMI 接口控制 LAN8720。
5. 第 10 脚可以配置 LAN8720 地址。

25.5 移植调试

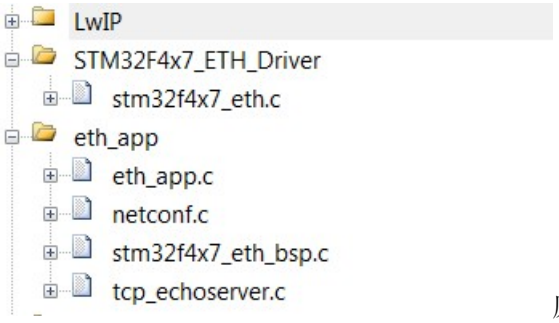
ST 提供了 ETH 例程《STM32F4x7_ETH_LwIP_V1.1.1》。在 Libraries 文件夹内有 STM32F4x7_ETH_Driver 库文件。Project 文件夹内有两个文件夹，FreeRTOS 是带操作系统的例程，Standalone 则是不带操作系统的例程。目前我们还没有移植操作系统，先用不带操作系统的例程测试硬件。我们选择里面的 **tcp_echo_server** 例程。在 app 文件夹建立一个 eth 文件，用于存放网络应用。把例程中 src 和 inc 文件夹内的相关文件拷贝到 eth。例程的 main.c 跟 main.h 改名 eth_app。



- C 文件 原理图



- 头文件 原理图



- 将文件添加到工程 原理图 lwip 文件较多，一共 34 个。lwip-1.4.1\src\api 目录下 8 个。lwip-1.4.1\src\core 目录下 16 个。lwip-1.4.1\src\core\ipv4 目录下 8 个。lwip-1.4.1\src\netif 目录下的 etharp.c lwip-1.4.1\port\STM32F4x7\Standalone 目录下的 ethernetif.c

25.5.1 修改

1. 修改 ETH_GPIO_Config 函数，根据我们的硬件配置 GPIO。
2. 修改所有 DP83848_PHY_ADDRESS，改为 ETH_PHY_ADRESS，在 stm32f4x7_eth_bsp.h 宏定义

```
//#define DP83848_PHY_ADDRESS       0x01 /* Relative to STM324xG-EVAL Board */
#define LAN8720A_PHY_ADDRESS       0x00 /* Relative to WUJIQUE F407 Board */

#define ETH_PHY_ADRESS                                    LAN8720A_PHY_ADDRESS
```

1. 打开宏，使用 DHCP，DHCP 就是自动获取 IP 的意思。

```
#define USE_DHCP       /* enable DHCP, if disabled static address is used */
```

1. 打开接口定义，我们用的是 RMII 模式，原来例程用的是 MII 模式

```
/* wujique F407 硬件使用 RMII 接口*/
#define RMII_MODE  // User have to provide the 50 MHz clock by soldering a 50 MHz
                   // oscillator (ref SM7745HEV-50.0M or equivalent) on the U3
                   // footprint located under CN3 and also removing jumper on JP5.
                   // This oscillator is not provided with the board.
                   // For more details, please refer to STM3240G-EVAL evaluation
                   // board User manual (UM1461).

//#define MII_MODE
```

对源码移植过程大概修改上面这些，具体修改了什么，可以跟原来例程对比。

25.5.2 修改系统滴答

网络需要一个 Time_Get_LocalTime 函数，其实是一个系统滴答。我们代码中一直使用一个 Delay 函数，我们修改这个 Delay，改为**滴答形式**。

main 函数开始初始化系统滴答，改为 1MS

```
  /* SysTick end of count event each 10ms */
  RCC_GetClocksFreq(&RCC_Clocks);
  SysTick_Config(RCC_Clocks.HCLK_Frequency / 1000);
```

将原来的延时函数改为下面三个函数：Delay 还是延时；Time_Get_LocalTime 获取系统滴答；Time_Update 放到 SysTick_Handler 函数内，替换原来的函数。

```
/* this variable is used to create a time reference incremented by 10ms */
__IO uint32_t LocalTime = 0;
uint32_t timingdelay;

/**
  * @brief  Inserts a delay time.
  * @param  nCount: number of 10ms periods to wait for.
  * @retval None
  */
void Delay(uint32_t nCount)
{
  /* Capture the current local time */
  timingdelay = LocalTime + nCount;

  /* wait until the desired delay finish */
  while(timingdelay > LocalTime)
```

(continues on next page)

(continued from previous page)

```
  {
  }
}

uint32_t Time_Get_LocalTime(void)
{
        return LocalTime;
}

/**
  * @brief  Updates the system local time
  * @param  None
  * @retval None
  */
void Time_Update(void)
{
  LocalTime += SYSTEMTICK_PERIOD_MS;
}
```

25.5.3 读芯片 ID

到现在，大家应该都熟悉外设调试流程了：**可以读 ID 的芯片，肯定是先调试能读取 ID，再调试其他功能**。在函数 ETH_BSP_Config 内增加读 ID 功能，初始化 ETH 后就读。

```
  /* Configure the GPIO ports for ethernet pins */
  ETH_GPIO_Config();

  /* Configure the Ethernet MAC/DMA */
  ETH_MACDMA_Config();

    uart_printf("read phy id\r\n");
        ID1 = ETH_ReadPHYRegister(ETH_PHY_ADRESS, 0X02);
        ID2 = ETH_ReadPHYRegister(ETH_PHY_ADRESS, 0X03);
        uart_printf("PHY ID:%02x %02x\r\n", ID1, ID2);
```

修改完之后成功读取 ID。

25.5.4 获取 IP

接上网线，通过路由器分配 IP 地址。

hello word! ETH_BSP_Config read phy id PHY ID:07 c0f1 PHY_BSR: 782d phy ETH_LINK_FLAG Looking forDHCP serverplease wait···IP address assigned by a DHCP server192.168.2.169

成功获取 IP 地址

25.5.5 通信测试

我们移植的是 tcp_echo_server，也就是一个 TCP 协议自动回显的 server。在函数 tcp_echoserver_init(); 中有以下初始化代码

```
void tcp_echoserver_init(void)
{
  /* create new tcp pcb */
  tcp_echoserver_pcb = tcp_new();

  if (tcp_echoserver_pcb != NULL)
  {
    err_t err;

    /* bind echo_pcb to port 7 (ECHO protocol) */
    err = tcp_bind(tcp_echoserver_pcb, IP_ADDR_ANY, 7);

    if (err == ERR_OK)
    {
      /* start tcp listening for echo_pcb */
      tcp_echoserver_pcb = tcp_listen(tcp_echoserver_pcb);

      /* initialize LwIP tcp_accept callback function */
      tcp_accept(tcp_echoserver_pcb, tcp_echoserver_accept);
    }
    else
    {
      /* deallocate the pcb */
      memp_free(MEMP_TCP_PCB, tcp_echoserver_pcb);
      printf("Can not bind pcb\n");
    }
  }
  else
  {
    printf("Can not create new pcb\n");
  }
```

(continues on next page)

(continued from previous page)

```
}
```

其中第 11 行代码，将 tcp 绑定到**端口 7**。

我们运行网络调试助手，设置如下图，理图

协议选择 Tcp Client IP 地址选择开发板获取到的地址使用端口 7

点击**链接**连接成功后点击发送，开发板会显数据给电脑。测试成功。勾上左下角数据流循环发送，点击发送，就可以重复发送，测试是否会出现丢包。

25.6 总结

ST 提供的例程有多种，大家可以尝试其他例程。

25.7 end

CHAPTER 26

can 总线调试记录

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

本节将向大家介绍如何使用 STM32F4 自带的 CAN 控制器实现两个开发板之间的 CAN 通信。

26.1 CAN

CAN 是控制器局域网络 (Controller Area Network, CAN) 的简称，是由以研发和生产汽车电子产品著称的德国 BOSCH 公司开发的，并最终成为国际标准（ISO 11898），是国际上应用最广泛的现场总线之一。在北

美和西欧，CAN 总线协议已经成为汽车计算机控制系统和嵌入式工业控制局域网的标准总线，并且拥有以 CAN 为底层协议专为大型货车和重工机械车辆设计的 J1939 协议。

百度百科

> CAN 是 Controller Area Network 的缩写（以下称为 CAN），是 ISO 国际标准化的串行通信协议。在汽车产业中，出于对安全性、舒适性、方便性、低公害、低成本的要求，各种各样的电子控制系统被开发了出来。由于这些系统之间通信所用的数据类型及对可靠性的要求不尽相同，由多条总线构成的情况很多，线束的数量也随之增加。为适应“减少线束的数量”、“通过多个 LAN，进行大量数据的高速通信”的需要，1986 年德国电气商博世公司开发出面向汽车的 CAN 通信协议。此后，CAN 通过 ISO11898 及 ISO11519 进行了标准化，在欧洲已是汽车网络的标准协议。CAN 的高性能和可靠性已被认同，并被广泛地应用于工业自动化、船舶、医疗设备、工业设备等方面。现场总线是当今自动化领域技术发展的热点之一，被誉为自动化领域的计算机局域网。它的出现为分布式控制系统实现各节点之间实时、可靠的数据通信提供了强有力的技术支持。

26.1.1 特点

- 完成对通信数据的成帧处理集成了 CAN 协议的物理层和数据链路层功能，可完成对通信数据的成帧处理，包括位填充、数据块编码、循环冗余检验、优先级判别等项工作。
- 使网络内的节点个数在理论上不受限制 CAN 协议的一个最大特点是废除了传统的站地址编码，而代之以对通信数据块进行编码。采用这种方法的优点可使网络内的节点个数在理论上不受限制，数据块的标识符可由 11 位或 29 位二进制数组成，因此可以定义 2 或 2 个以上不同的数据块，这种按数据块编码的方式，还可使不同的节点同时接收到相同的数据，这一点在分布式控制系统中非常有用。数据段长度最多为 8 个字节，可满足通常工业领域中控制命令、工作状态及测试数据的一般要求。同时，8 个字节不会占用总线时间过长，从而保证了通信的实时性。CAN 协议采用 CRC 检验并可提供相应的错误处理功能，保证了数据通信的可靠性。CAN 卓越的特性、极高的可靠性和独特的设计，特别适合工业过程监控设备的互连，因此，越来越受到工业界的重视，并已公认为最有前途的现场总线之一。
- 可在各节点之间实现自由通信 CAN 总线采用了多主竞争式总线结构，具有多主站运行和分散仲裁的串行总线以及广播通信的特点。CAN 总线上任意节点可在任意时刻主动地向网络上其它节点发送信息而不分主次，因此可在各节点之间实现自由通信。CAN 总线协议已被国际标准化组织认证，技术比较成熟，控制的芯片已经商品化，性价比高，特别适用于分布式测控系统之间的数据通讯。CAN 总线插卡可以任意插在 PC AT XT 兼容机上，方便地构成分布式监控系统。
- 结构简单只有 2 根线与外部相连，并且内部集成了错误探测和管理模块。
- 传输距离和速率 CAN 总线特点：(1) 数据通信没有主从之分，任意一个节点可以向任何其他（一个或多个）节点发起数据通信，靠各个节点信息优先级先后顺序来决定通信次序，高优先级节点信息在 134μs 通信; (2) 多个节点同时发起通信时，优先级低的避让优先级高的，不会对通信线路造成拥塞; (3) 通信距离最远可达 10KM(速率低于 5Kbps) 速率可达到 1Mbps(通信距离小于 40M)；(4) CAN 总线传输介质可以是双绞线，同轴电缆。CAN 总线适用于大数据量短距离通信或者长距离小数据量，实时性要求比较高，多主多从或者各个节点平等的现场中使用。

26.1.2 通信

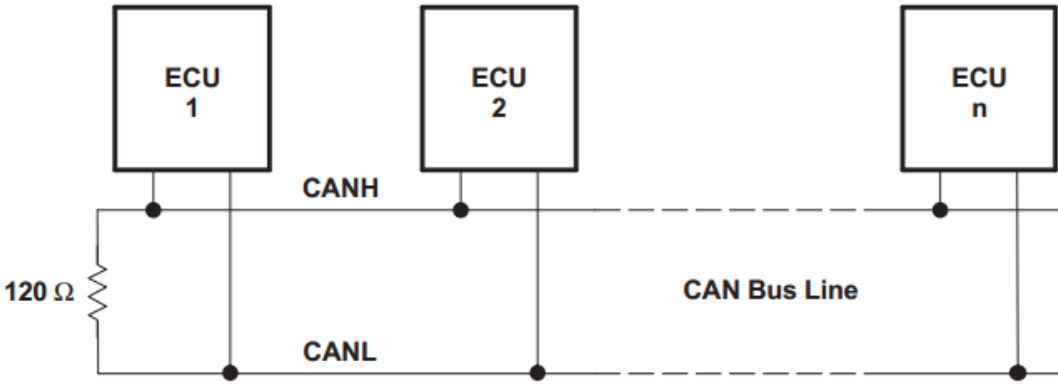


Figure 29. Typical CAN Network

典型的 CAN 通信网络如下图。

片

> 所有 CAN 节点通过 CANH 和 CANL 连接到 CAN 网络上。前面我们学习串口的时候知道，串口是发送和接收交叉相连。CAN 节点并没有所谓的发送和接收，所有的 CAN 节点，都是 CANH 与 CANH 相连，CANL 与 CANL 相连。

那么 CAN 是如何通信的呢?

- 物理层

请看下图：当 CANH 等于 CANL，电平都是 2.3V，叫做隐性电平。当 CANH 为高电平，CANL 为低电平时（两者之差大于 0.9V），叫做显性电平。按照规定，隐性代表逻辑 1，显性代表逻辑 0。

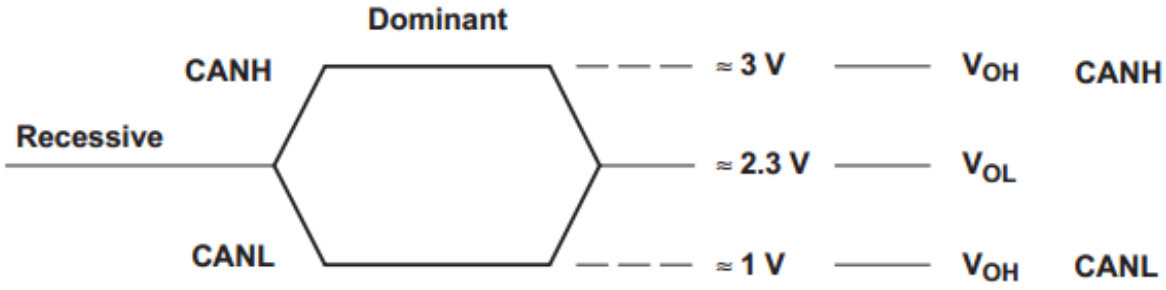


Figure 3. Driver Output Voltage Definitions

图

片

电平说清楚了，但是这个没有接收发送管脚，如何通信呢?

- 数据链路层

前面说到 CAN 特点的时候，我们提到过：CAN 协议不仅仅实现了物理层连接，还实现了数据链路层的功能。什么叫数据链路层？在调试网络功能的时候我们提到过 OSI 的数据链路层。

> 数据链路层是 OSI 参考模型中的第二层，介乎于物理层和网络层之间。数据链路层在物理层提供的服务的基础上向网络层提供服务，其最基本的服务是将源自网络层来的数据可靠地传输到相

> 邻节点的目标机网络层。为达到这一目的，数据链路必须具备一系列相应的功能，主要有：如何将数据组合成数据块，在数据链路层中称这种数据块为帧（frame），帧是数据链路层的传送单位；如何控制帧在物理信道上的传输，包括如何处理传输差错，如何调节发送速率以使与接收方相匹配；以及在两个网络实体之间提供数据链路通路的建立、维持和释放的管理。

通俗的说，数据链路层的功能就是保证数据传输的可靠性，并且管理数据传输。例如数据丢包了，要不要重发？重发几次？为了达到这个功能，通常需要对原始数据进行封装处理。也就是所谓的帧。不过大家要搞清楚，数据链路层不是应用层，数据链路只负责数据传输，不关心数据内容和数据功能。

例如前面的串口，它只是一个物理层的。如果用于两个设备之间的通信，为了保证通信的可靠性，我们需要加上一定的机制，例如握手，重发，校验，数据头数据尾等。这些，就是数据链路层功能。

26.1.3 协议

CAN 总线是一个**广播类型**的总线，所以任何在总线上的节点都可以监听总线上传输的数据。也就是说总线上的传输不是点到点的，而是一点对多点的传输，这里多点的意思是总线上所有的节点。但是总线上的节点如何知道那些数据是传送给自己的呢？CAN 总线的硬件芯片提供了一种叫做**本地过滤**的功能，通过这种本地过滤的功能可以过滤掉一些和自己无关的数据，而保留一些和自己有关的信息。

协议就是数据链路层的实现。

- 帧类型

报文传输由以下 4 个不同的帧类型所表示和控制：

- 数据帧：数据帧携带数据从发送器至接收器。
- 远程帧：总线单元发出远程帧，请求发送具有同一识别符的数据帧。
- 错误帧：任何单元检测到一总线错误就发出错误帧。
- 过载帧：过载帧用以在先行的和后续的数据帧（或远程帧）之间提供一附加的

数据帧（或远程帧）通过帧间空间与前述的各帧分开。

CAN 协议有以下 4 中帧:
片

更信息规范请查阅资料中的 <CAN BUS 规范 v2.0+ 中文版.pdf>

26.2 STM32 CAN

STM32 带的 CAN 控制器叫**基本扩展 CAN 外设**，又称 bxCAN。支持 2.0A 和 B 版本协议。

26.2.1 特性

- 支持 2.0 A 及 2.0 B Active 版本 CAN 协议
- 比特率高达 1 Mb/s
- 支持时间触发通信方案

发送

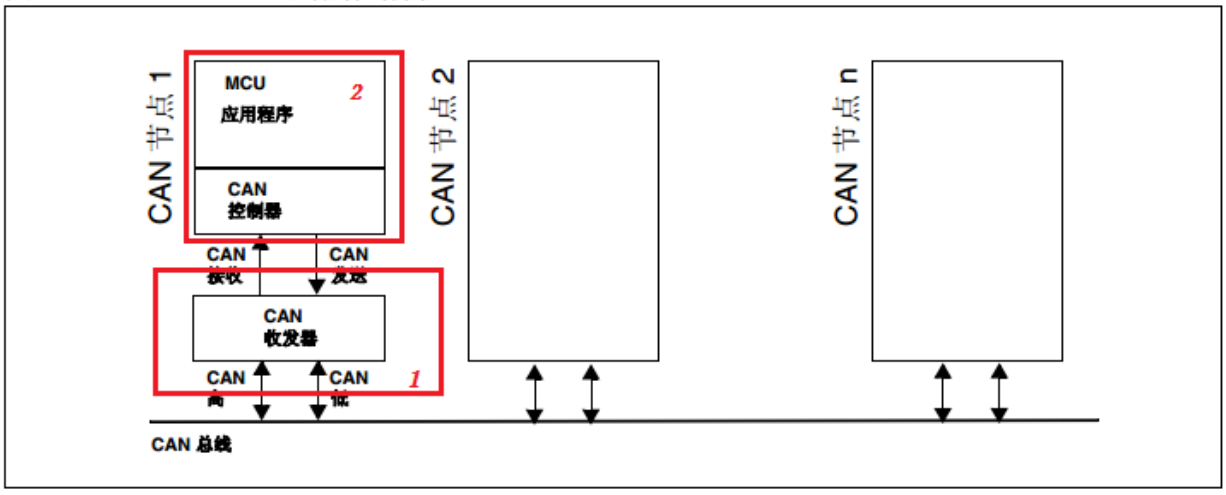
- 三个发送邮箱
- 可配置的发送优先级
- SOF 发送时间戳

接收

- 两个具有三级深度的接收 FIFO
- 可调整的筛选器组：
  - CAN1 和 CAN2 之间共享 28 个筛选器组
- 标识符列表功能
- 可配置的 FIFO 上溢
- SOF 接收时间戳

图片

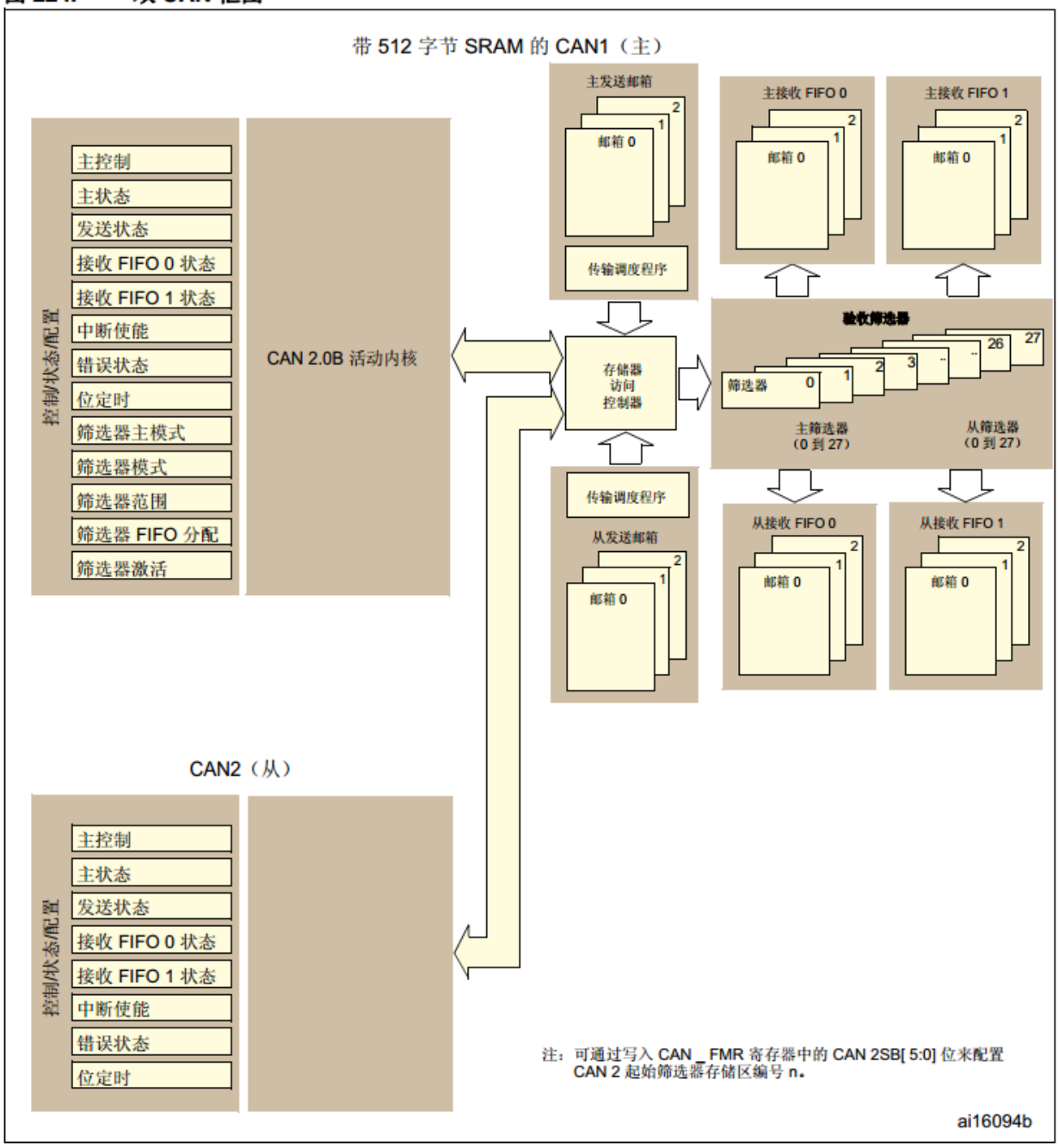
图 223. CAN 网络拓扑结构



图片上图是 CAN 的应用拓扑结构。红框 1 里面是 CAN 芯片，也就是我们的 VP230 芯片。红框 2 就是 STM32 芯片的 CAN 控制器。

26.2.2 框图

图 224. 双 CAN 框图



图片

STM32 有两个 CAN，CAN1 做主 bxCAN，can2 做从 bxCAN。

26.2.3 过滤器

关于过滤器的说明在文档 <7.4，标识符筛选 >。

STM32 CAN 外设一个重要的功能就是硬件过滤功能。前面提到，CAN 使用的是广播方式通信，一个节点会收到总线上的所有数据（所有帧）。但是并不是所有消息都是我需要的。有了过滤器，就可以在软件不干预的情况下丢弃那些我不要的数据。

每组过滤器包括了 2 个可配置的 32 位寄存器：CAN_FxR0 和 CAN_FxR1。这些过滤器相当于关卡，每当收到一条报文时，CAN 要先将收到的报文从这些过滤器上”过”一下，能通过的报文是有效报文，收进相关联 FIFO（FIFO1 或 FIFO2），不能通过的是无效报文 (不是发给”我”的报文)，直接丢弃。

我们从配置看看过滤器功能。

```
/**
  * @brief  CAN filter init structure definition
  */
typedef struct
{
  uint16_t CAN_FilterIdHigh; /*!< Specifies the filter identification number (MSBs for a␣
↪32-bit
                                configuration, first one for a 16-bit configuration).
                                This parameter can be a value between 0x0000 and 0xFFFF␣
↪*/

  uint16_t CAN_FilterIdLow; /*!< Specifies the filter identification number (LSBs for a␣
↪32-bit
                                configuration, second one for a 16-bit configuration).
                                This parameter can be a value between 0x0000 and 0xFFFF␣
↪*/

  uint16_t CAN_FilterMaskIdHigh; /*!< Specifies the filter mask number or identification␣
↪number,
                                according to the mode (MSBs for a 32-bit configuration,
                                first one for a 16-bit configuration).
                                This parameter can be a value between 0x0000 and 0xFFFF␣
↪*/

  uint16_t CAN_FilterMaskIdLow; /*!< Specifies the filter mask number or identification␣
↪number,
                                according to the mode (LSBs for a 32-bit configuration,
                                second one for a 16-bit configuration).
```

(continues on next page)

(continued from previous page)

```
                                This parameter can be a value between 0x0000 and 0xFFFF␣
↪*/

  uint16_t CAN_FilterFIFOAssignment; /*!< Specifies the FIFO (0 or 1) which will be␣
↪assigned
                                      to the filter.
                                     This parameter can be a value of @ref CAN_filter_
↪FIFO */

  uint8_t CAN_FilterNumber; /*!< Specifies the filter which will be initialized.
                               It ranges from 0 to 13. */

  uint8_t CAN_FilterMode;    /*!< Specifies the filter mode to be initialized.
                              This parameter can be a value of @ref CAN_filter_mode */

  uint8_t CAN_FilterScale;  /*!< Specifies the filter scale.
                              This parameter can be a value of @ref CAN_filter_scale */

  FunctionalState CAN_FilterActivation; /*!< Enable or disable the filter.
                              This parameter can be set either to ENABLE or DISABLE. */
} CAN_FilterInitTypeDef;
```

上面就是一个 CAN 过滤器的配置。

- CAN_FilterMode 过滤模式。有两种掩码模式和列表模式。

```
#define CAN_FilterMode_IdMask       ((uint8_t)0x00)  /*!< identifier/mask mode */
#define CAN_FilterMode_IdList       ((uint8_t)0x01)  /*!< identifier list mode */
```

- CAN_FilterNumber 过滤器编号，STM32 有 14 个过滤器。
- CAN_FilterScale 设置过滤器刻度，16 位或 32 位。
- CAN_FilterFIFOAssignment 设置使用过滤器的接收 FIFO，有两个：0 和 1。请看框图右上角。读数据时需要指定 FIFO 的。
- CAN_FilterIdHigh、CAN_FilterIdLow 叫做 FilterId
- CAN_FilterMaskIdHigh、CAN_FilterMaskIdLow 叫它 FilterMaskId

  这四个值，对应过滤器的 2 个 32 位寄存器：CAN_FxR1、CAN_FxR2

  1. 如果 CAN_FilterScale 是 16 位：那么 CAN_FilterMaskIdLow<<16+CAN_FilterIdLow 设置到 CAN_FR1。CAN_FilterMaskIdHigh<<16+CAN_FilterIdHigh 设置到 CAN_FR2。

2. 如果 CAN_FilterScale 是 32 位：CAN_FilterMaskIdHigh<<16 + CAN_FilterMaskIdLow 设置到 CAN_FR2。

CAN_FilterIdHigh<<16 + CAN_FilterIdLow 设置到 CAN_FR1

查看设置函数可见上述细节

```
void CAN_FilterInit(CAN_FilterInitTypeDef* CAN_FilterInitStruct)
```

按工作模式和宽度，一个过滤器组可以变成以下几中形式之一：

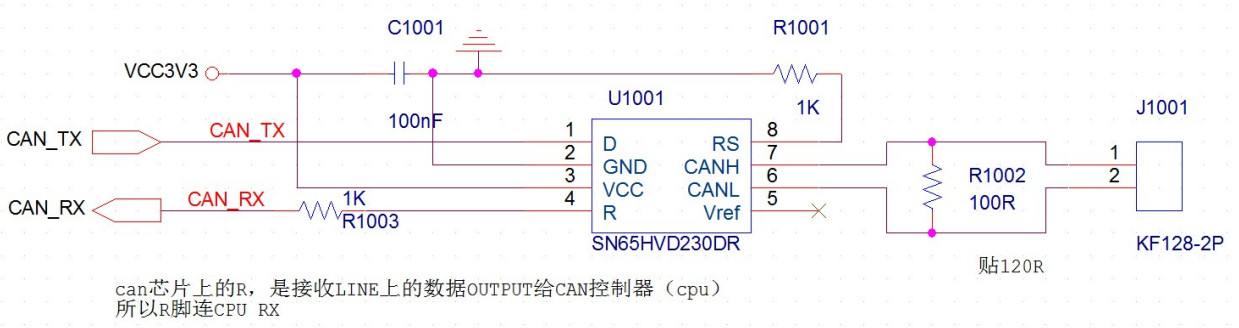
- 每组过滤器组必须关联且只能关联一个 FIFO。复位默认都关联到 FIFO_0。
- 每个 FIFO 的所有过滤器都是并联的，只要通过了其中任何一个过滤器，该报文就有效。
- 如果一个报文既符合 FIFO_0，又符合 FIFO_1，显然，根据操作顺序，它只会放到 FIFO_0 中。

更多的 CANX 相关请查看文档

26.3 VP230 芯片

VP230 是 SN65HVD230DR 的俗称，是一款 TI 出的 CAN 芯片。我们选用这块芯片的原因是：**工作电压 3.3V，且可以和常用的 5V CAN 芯片 TJA1050 通信**。

26.4 原理图

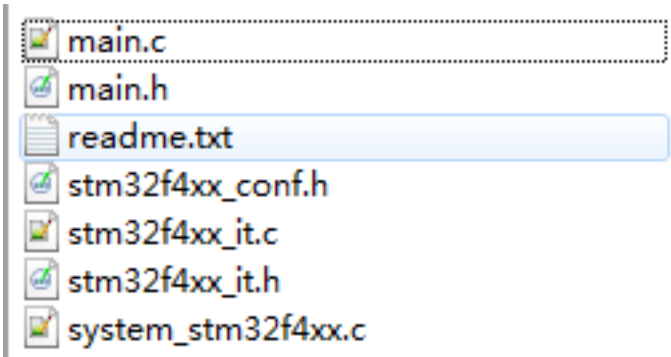


原理图

26.5 移植官方例程

官方 CAN 例程有两个:CAN_LoopBack(回环测试);CAN_Networking(组网模式)。所谓的回环模式就是将自身发送的消息作为接收的消息来处理并存储。我们移植组网模式。

26.5.1 例程分析



在 CAN 例程内，并没有任何 CAN 相关的文件。图片

查看 main.c，CAN_Config 配置 CAN 后，进入 while 循环。循环中如果判断到按键按下，在 else 分支就会将键值赋值到发送消息。然后调用 CAN_Transmit 发送消息。

```
/* CAN configuration */
  CAN_Config();

  while(1)
  {
    while(STM_EVAL_PBGetState(BUTTON_KEY) == KEY_PRESSED)
    {
      if(ubKeyNumber == 0x4)
      {
        ubKeyNumber = 0x00;
      }
      else
      {
        LED_Display(++ubKeyNumber);
        TxMessage.Data[0] = ubKeyNumber;
        CAN_Transmit(CANx, &TxMessage);
        /* Wait until one of the mailboxes is empty */
        while((CAN_GetFlagStatus(CANx, CAN_FLAG_RQCP0) !=RESET) || \
              (CAN_GetFlagStatus(CANx, CAN_FLAG_RQCP1) !=RESET) || \
              (CAN_GetFlagStatus(CANx, CAN_FLAG_RQCP2) !=RESET));

        while(STM_EVAL_PBGetState(BUTTON_KEY) != KEY_NOT_PRESSED)
        {
        }
      }
```

(continues on next page)

(continued from previous page)

```
    }
  }
```

分析 CAN_Config 函数。13~25 行，配置对应 IO 为 CAN 功能。32~48，can 基本配置，具体意思看注释 50~60，滤波器设置，R1 和 R2 都设置为 0，也就是所有报文都能通过滤波器。62~67，初始化一组发送消息 70 行，使能 CAN 中断。

```
/**
  * @brief  Configures the CAN.
  * @param  None
  * @retval None
  */
static void CAN_Config(void)
{
  GPIO_InitTypeDef  GPIO_InitStructure;

  /* CAN GPIOs configuration **************************************************/

  /* Enable GPIO clock */
  RCC_AHB1PeriphClockCmd(CAN_GPIO_CLK, ENABLE);

  /* Connect CAN pins to AF9 */
  GPIO_PinAFConfig(CAN_GPIO_PORT, CAN_RX_SOURCE, CAN_AF_PORT);
  GPIO_PinAFConfig(CAN_GPIO_PORT, CAN_TX_SOURCE, CAN_AF_PORT);

  /* Configure CAN RX and TX pins */
  GPIO_InitStructure.GPIO_Pin = CAN_RX_PIN | CAN_TX_PIN;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;
  GPIO_Init(CAN_GPIO_PORT, &GPIO_InitStructure);

  /* CAN configuration ********************************************************/
  /* Enable CAN clock */
  RCC_APB1PeriphClockCmd(CAN_CLK, ENABLE);

  /* CAN register init */
  CAN_DeInit(CANx);
```

(continues on next page)

(continued from previous page)

```
  /* CAN cell init */
  CAN_InitStructure.CAN_TTCM = DISABLE; //非时间出发通信模式
  CAN_InitStructure.CAN_ABOM = DISABLE; //软件自动离线管理
  CAN_InitStructure.CAN_AWUM = DISABLE; //睡眠模式通过软件唤醒
  CAN_InitStructure.CAN_NART = DISABLE; //禁止报文自动传送
  CAN_InitStructure.CAN_RFLM = DISABLE; //报文不锁定，新的覆盖旧的
  CAN_InitStructure.CAN_TXFP = DISABLE; //优先级由报文标识符决定
  CAN_InitStructure.CAN_Mode = CAN_Mode_Normal; //模式设置为普通模式
  CAN_InitStructure.CAN_SJW = CAN_SJW_1tq; //重新同步跳跃宽度为 1 个时间单位

  /* CAN Baudrate = 1 MBps (CAN clocked at 30 MHz) 这里是设置 CAN 波特率 */
  CAN_InitStructure.CAN_BS1 = CAN_BS1_6tq; //时间段 1 占用 6 个时间单位
  CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq; //时间段 2 占用 8 个时间单位
  CAN_InitStructure.CAN_Prescaler = 2;//分频系数
  CAN_Init(CANx, &CAN_InitStructure);

  /* CAN filter init  设置滤波器 */
  CAN_FilterInitStructure.CAN_FilterNumber = 0; //使用滤波器组 0
  CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdMask; //
  CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_32bit;
  CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
  CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
  CAN_FilterInitStructure.CAN_FilterMaskIdHigh = 0x0000;
  CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
  CAN_FilterInitStructure.CAN_FilterFIFOAssignment = 0;
  CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;
  CAN_FilterInit(&CAN_FilterInitStructure);

  /* Transmit Structure preparation 消息初始化 */
  TxMessage.StdId = 0x321;
  TxMessage.ExtId = 0x01;
  TxMessage.RTR = CAN_RTR_DATA;
  TxMessage.IDE = CAN_ID_STD;
  TxMessage.DLC = 1;

  /* Enable FIFO 0 message pending Interrupt 初始化中断 */
  CAN_ITConfig(CANx, CAN_IT_FMP0, ENABLE);
}
```

其中的发送数据结构是库定义的

```
/**
  * @brief  CAN Tx message structure definition
  */
typedef struct
{
  uint32_t StdId;  /*!< Specifies the standard identifier.
                        This parameter can be a value between 0 to 0x7FF. */

  uint32_t ExtId;  /*!< Specifies the extended identifier.
                    This parameter can be a value between 0 to 0x1FFFFFFF. */

  uint8_t IDE;     /*!< Specifies the type of identifier for the message that
                        will be transmitted. This parameter can be a value
                        of @ref CAN_identifier_type */

  uint8_t RTR;     /*!< Specifies the type of frame for the message that will
                        be transmitted. This parameter can be a value of
                        @ref CAN_remote_transmission_request */

  uint8_t DLC;     /*!< Specifies the length of the frame that will be
                        transmitted. This parameter can be a value between
                        0 to 8 */

  uint8_t Data[8]; /*!< Contains the data to be transmitted. It ranges from 0
                        to 0xFF. */
} CanTxMsg;
```

CAN 的配置到底什么意思，看 stm32f4xx_can.h 就可以知道大概了。

在 CONFIG 之前就调用了 NVIC 配置初始化 CAN 的中断优先级

```
/**
  * @brief  Configures the NVIC for CAN.
  * @param  None
  * @retval None
  */
static void NVIC_Config(void)
{
  NVIC_InitTypeDef  NVIC_InitStructure;

  NVIC_InitStructure.NVIC_IRQChannel = CAN1_RX0_IRQn;
```

(continues on next page)

(continued from previous page)

```
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

其中在中断处理如下: 收到数据后，判断 STDID 和 IDE DLC 标志。

```
/**
  * @brief  This function handles CAN1 RX0 request.
  * @param  None
  * @retval None
  */
void CAN1_RX0_IRQHandler(void)
{
  CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);

  if ((RxMessage.StdId == 0x321)&&(RxMessage.IDE == CAN_ID_STD)
      && (RxMessage.DLC == 1))
  {
    LED_Display(RxMessage.Data[0]);
    ubKeyNumber = RxMessage.Data[0];
  }
}
```

接收数据的格式也是在 stm32f4xx_can.h 中有定义。

26.5.2 移植

1. 将 main.c 里面的 CAN 控制器配置代码拷贝到我们工程 mcu_can.c.
2. 将 stm32f4xx_it.c 中的 CAN1_RX0_IRQHandler 接收中断拷贝到我们的 stm32f4xx_it.c。
3. 修改 IO 口配置。使用 PB8/PB9，通过查数据手册，可知是 CAN1。

- 另外，官方例程代码层次不是很好，我们重新划分。

具体见代码。

26.6 测试

本次测试需要两块开发板进行测试。接线同向接线，也即是 CANH 连接 CANH，CANL 连接 CANL。测试程序如下：

```
int mcu_can_test(void)
{
        uint8_t data = 0;
  /* NVIC configuration */
  NVIC_CAN_Config();

  /* CAN configuration */
  mcu_can_config();

  while(1)
  {
        #if 1/* 测试发送端，发送后等待接收端响应 */
        Delay(1000);
        data++;
        /* Transmit Structure preparation */
        TxMessage.StdId = 0x321;
        TxMessage.ExtId = 0x01;
        TxMessage.RTR = CAN_RTR_DATA;
        TxMessage.IDE = CAN_ID_STD;
        TxMessage.DLC = 1;

    TxMessage.Data[0] = data;
    CAN_Transmit(CANx, &TxMessage);

    /* Wait until one of the mailboxes is empty */
    while((CAN_GetFlagStatus(CANx, CAN_FLAG_RQCP0) !=RESET) || \
          (CAN_GetFlagStatus(CANx, CAN_FLAG_RQCP1) !=RESET) || \
          (CAN_GetFlagStatus(CANx, CAN_FLAG_RQCP2) !=RESET));

        uart_printf("can transmit :%02x\r\n", data);

        while(1)
        {
                if(CanRxFlag == 1)
                {
                        CanRxFlag = 0;
```

(continues on next page)

(continued from previous page)

```
                if ((RxMessage.StdId == 0x321)&&(RxMessage.IDE == CAN_ID_STD)
                   && (RxMessage.DLC == 1))
                {

                        uart_printf("can rep :%02x\r\n", RxMessage.Data[0]);

                }

                break;
        }
}

#else
/*  测试接收端，接收到数据后返回给发送端 */
if(CanRxFlag == 1)
{
        CanRxFlag = 0;

        if ((RxMessage.StdId == 0x321)&&(RxMessage.IDE == CAN_ID_STD)
            && (RxMessage.DLC == 1))
        {
                uart_printf("can receive :%02x\r\n", RxMessage.Data[0]);
                /* Transmit Structure preparation */
                TxMessage.StdId = 0x321;
                TxMessage.ExtId = 0x01;
                TxMessage.RTR = CAN_RTR_DATA;
                TxMessage.IDE = CAN_ID_STD;
                TxMessage.DLC = 1;

                TxMessage.Data[0] = RxMessage.Data[0];
                CAN_Transmit(CANx, &TxMessage);

                /* Wait until one of the mailboxes is empty */
                while((CAN_GetFlagStatus(CANx, CAN_FLAG_RQCP0) !=RESET) || \
                      (CAN_GetFlagStatus(CANx, CAN_FLAG_RQCP1) !=RESET) || \
                      (CAN_GetFlagStatus(CANx, CAN_FLAG_RQCP2) !=RESET));

                  }
            uart_printf("can send rep ok\r\n");
```

(continues on next page)

(continued from previous page)

```
        }
        #endif
  }
}
```

while 循环内的代码使用条件编译，分别是接收与发送。测试时使用两块开发板，一个下载发送的代码，一块下载接收代码。接收端先启动，发送端后启动。发送端调试信息如下：

> hello word! can transmit :01 can rx message can rep :01 can transmit :02 can rx message can rep :02

接收端调试信息如下：

> hello word! can rx message can receive :01 can send rep ok can rx message can receive :02 can send rep ok

发送端发送 01，接收端收到 01 后返回 rep，回显 01，接收端收到 rep:01。流程重复，发送的数据不断加 1 。

26.7 总结

1. CAN 的基本应用不难，如果真正应用在项目中，需要考虑更多的协议处理，例如仲裁。
2. 通过 CAN 的学习，希望大家有数据链路层的概念。以后做项目，每一种通信，最好加上数据链路层，这样可以保证数据传输的可靠性。

26.8 end

CHAPTER 27

RS485-串口驱动改造

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

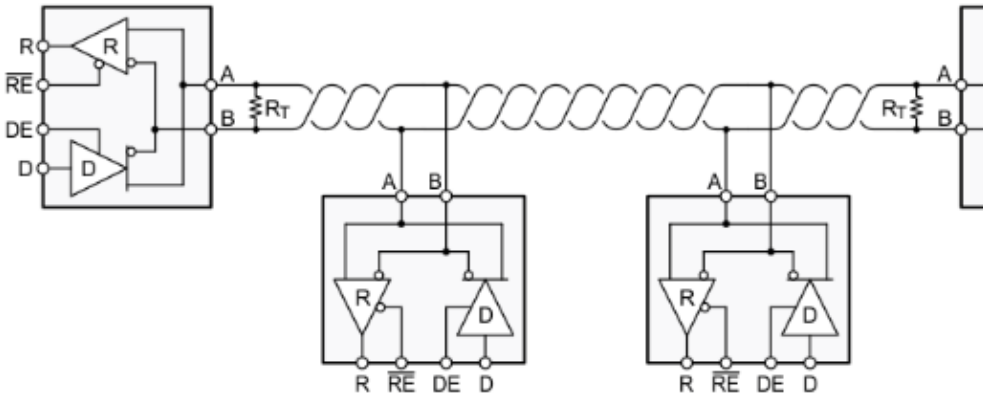
上一节我们调试了 CAN 总线，常见的总线还有 RS-485 总线。本节我们一起调试 407 的 485 总线。

27.1 RS485

惯例，百度百科：

RS-485 又名 TIA-485-A, ANSI/TIA/EIA-485 或 TIA/EIA-485。RS485 是一个定义平衡数字多点系统中的驱动器和接收器的电气特性的标准，该标准由电信行业协会和电子工业联盟定义。使用该标准的数字通信网络能在远距离条件下以及电子噪声大的环境下有效传输信号。RS-485 使得廉价本地网络以及多支路通信链路的配置成为可能。RS485 有两线制和四线制两种接线，四线制只能实现点对点的通信方式，现很少采用，现在多采用的是两线制接线方式，这种接线方式为总线式拓扑结构，在同一总线上最多可以挂接 32 个节点。在 RS485 通信网络中一般采用的是主从通信方式，即一个主机带多个从机。

27.1.1 总线网络



一个典型的 RS485 网络通常如下图：
络

多个端点用**双绞线**连接在一起。两端有两个匹配电阻，通常是 120 欧姆。这些端点中只有一个做主机。剩下的都是从机。

27.1.2 电平

RS485 与 CAN 有点类似，都是用两根线之间的压差传输数据。两线之间压差为 +(2~6)V，逻辑 1。两线之间压差为-(2~6)V，逻辑 0。

27.1.3 协议

RS485 仅仅规定了接收端和发送端的电气特性。它并没有规定或推荐任何数据协议。和 CAN 对比一下就知道，RS485 只是定义了物理层，没有定义数据链路层。

RS485 可以认为是一个半双工的串口。以前调试的串口，用的是 RS232 电平（连接方式/物理层）。现在调试的就是一个用 RS485 电平（连接方式/物理层）的串口。并且是一个半双工的串口。

27.1.4 特点

由于使用了 RS485 连接方式，相对于 RS232 的优点就是：

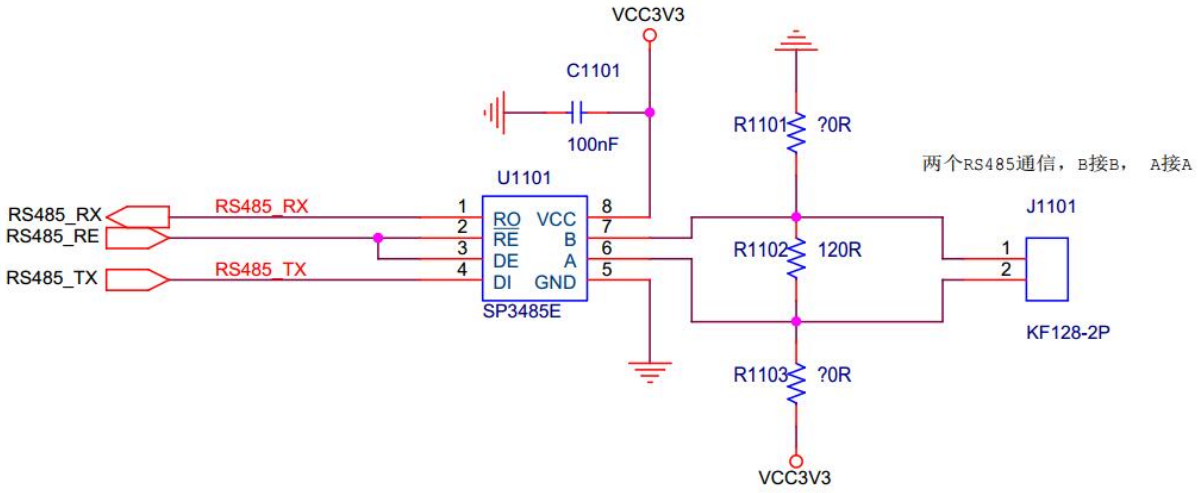
1. 传输距离远，在低速率时，最远可以传输 1200 米。
2. 可以通在网络上连接多个节点。
3. 相对 RS232，电压更低，更安全。
4. 使用双绞线，差分传输，抗干扰能力强。

27.2 SP3485E

SP3485E 是 EXAR 公司的 485 芯片，有以下特点：

1. 3.3V 工作电压。
2. 10Mbsps。
3. 最多支持 32 个节点。
4. 输出短路保护。

27.3 原理图



原理图

- TX 接在 PA9 上，RX 接在 PA10 上，与**外扩串口**、**USB**、**摄像头**共用。使用 RS485 时，这三个外设不要使用。

- RX&TX 连接到串口。
- RE 是接收发送控制管脚（485 是半双工）

27.4 驱动设计与调试

在 dev_board 建立 dev_rs485 驱动文件。RS485 驱动基于串口驱动。在前面做串口驱动程序时，并没有考虑多个串口的情况。因此在设计 RS485 驱动前，要先改造串口驱动。

27.4.1 串口驱动改造

怎么改造？思考下面情景

- 3 个串口：

  uart1 用来输出调试信息，那么它的上一层就是 LOG 功能。uart2 用于 485，上一层是 485 外设驱动。uart3 用于 8266 wifi 模块，上一层则是 8266 驱动程序。

- 串口需要什么？或者说一个设备基于什么设备？

  1 串口控制器。2 硬件 IO 口。3 硬件 RAM（接收发送缓冲）4 中断 5 不需要时间片轮询。

- 串口驱动要提供什么接口？

1. 初始化（**重要，不能跟打开混为一谈**）
2. 打开
3. 关闭
4. 读数据（接收）
5. 写数据（发送）

- 串口工作情景在时间角度上，在数据流转角度上是怎么样的？

  1 中断接收到数据，保存在缓冲。上一层调用读数据接口，读的其实是接收缓冲内的数据。2 上一层调用写数据时，将数据通过串口发送出去。阻塞发送？还是通过中断发送？这个是可以考虑的，一般情况阻塞发送问题也不大。但是接收肯定不能轮询等待接收。理由就是：**接收，是别人发给你的，你不知道什么时候有数据来。发送，是我要发的，我知道什么时候发。**

综上考虑我们抽象设计一个串口设备（一个结构体）。

```
/*
@bref: 串口设备
*/
typedef struct
{
        /* 硬件相关 */
```

(continues on next page)

(continued from previous page)

```
        /*
                STM IO 配置需要太多数据，可以直接在代码中定义， 或者用宏定义
                如果是更上一层设备驱动，例如 FLASH ，就可以在设备抽象中定义一个用哪个 SPI 的
定义。
        */

        USART_TypeDef* USARTx;

        /*RAM 相关 */
        s32 gd;         //设备句柄 小于等于 0 则为未打开设备

        u16 size;// buf 大小
        u8 *Buf;//缓冲指针
        u16 Head;//头
        u16 End;//尾
        u8  OverFg;//溢出标志
}_strMcuUart;
```

1. 理论上，硬件相关的应该定义在设备结构体内。串口，基本上写了就不改了，因此，直接定义到代码内也可以。
2. USARTx 串口号。
3. gd 设备控制符，设备句柄。
4. size 缓冲大小
5. Buf 缓冲指针，指向接收缓冲。
6. Head、End 环形缓冲头尾指针。
7. OverFg 溢出标志

同时用枚举定义串口设备

```
typedef enum {
  MCU_UART_1 =0,
  MCU_UART_2,
  MCU_UART_3,
  MCU_UART_MAX,
}
McuUartNum;
```

串口设备这样定义后，接口就改成如下：

```
extern s32 mcu_uart_open (McuUartNum comport);
extern s32 mcu_uart_close (McuUartNum comport);
extern s32 mcu_uart_write (McuUartNum comport, u8 *buf, s32 len);
extern s32 mcu_uart_read (McuUartNum comport, u8 *buf, s32 len);
extern s32 mcu_uart_set_baud (McuUartNum comport, s32 baud);
extern s32 mcu_uart_tcflush (McuUartNum comport);
```

每个接口有一个 comport 参数，在操作串口时，通过这个参数传入串口设备编号，就能操作串口了。

经过这样改造，串口驱动就可以支持多个串口了。如果需要支持多平台，就还需要将初始化中相关的硬件参数也抽象到串口设备结构体内。

RS485 驱动

485 驱动就很简单了。功能跟串口类似（本来就是基于串口驱动），因此接口定义如下：

```
extern s32 dev_rs485_init(void);
extern s32 dev_rs485_open(void);
extern s32 dev_rs485_close(void);
extern s32 dev_rs485_read(u8 *buf, s32 len);
extern s32 dev_rs485_write(u8 *buf, s32 len);
extern s32 dev_rs485_ioctl(void);
```

标准设备接口。

设备初始化，就初始化 IO，并且配置为接收模式。

```
/**
 *@brief:      dev_rs485_init
 *@details:    初始化 485 设备
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 dev_rs485_init(void)
{
        GPIO_InitTypeDef GPIO_InitStructure;

        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG,ENABLE); //使能 PG 时钟
        //PG8 推挽输出， 485 模式控制
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8; //GPIOG8
```

(continues on next page)

(continued from previous page)

```
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //速度 100MHz
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
        GPIO_Init(GPIOG, &GPIO_InitStructure); //初始化 PG8

        //初始化设置为接收模式
        GPIO_ResetBits(GPIOG, GPIO_Pin_8);

        return 0;
}
```

打开设备的时候才打开串口

```
/**
 *@brief:      dev_rs485_open
 *@details:    打开 RS485 设备
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 dev_rs485_open(void)
{
        mcu_uart_open(DEV_RS485_UART);
        mcu_uart_set_baud(DEV_RS485_UART, 9600);
        return 0;
}
```

跟串口不一样的就是发送，要将 485 设置为发送模式，并且是阻塞发送。发送结束后切回接收模式。

```
/**
 *@brief:      dev_rs485_write
 *@details:    rs485 发送数据
 *@param[in]   u8 *buf
               s32 len
 *@param[out]  无
 *@retval:
 */
s32 dev_rs485_write(u8 *buf, s32 len)
{
        s32 res;
```

(continues on next page)

(continued from previous page)

```
        GPIO_SetBits(GPIOG, GPIO_Pin_8);//设置为发送模式
        res = mcu_uart_write(DEV_RS485_UART, buf, len);
        GPIO_ResetBits(GPIOG, GPIO_Pin_8);//发送结束后设置为接收模式

        return res;
}
```

更多请查看代码

调试

1 下程序，接收端没有接收到 2 看原理图，串口跟 485 芯片的地方不知道为什么加了 3 个电阻，先去掉。3 去掉电阻后还是不行。4 详细对原理图，发现硬件 RX 跟 TX 接反。

测试代码与 CAN 类似，接收端跟发送端使用不同程序。发送端循环发送字符串”rs485 test”，接收端收到字符串后通过调试串口输出。

```
s32 dev_rs485_test(void)
{
        u8 buf[20];
        u8 len;
        s32 res;

        dev_rs485_init();
        dev_rs485_open();

        #if 0// 发送端测试
        while(1)
        {
                Delay(1000);
                res = dev_rs485_write("rs485 test\r\n", 13);
                uart_printf("dev rs485 write:%d\r\n", res);
        }
        #else//接收端测试
        while(1)
        {
                Delay(20);
                len = dev_rs485_read(buf, sizeof(buf));
                if(len > 0)
                {
```

(continues on next page)

(continued from previous page)

```
                        buf[len] = 0;
                        uart_printf("%s", buf);
                        memset(buf, 0, sizeof(buf));
                }
        }
        #endif
}
```

27.5 总结

RS485 是一个简单的设备。本节更希望大家了解串口驱动的改造。

27.6 end

CHAPTER 28

RTC

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

本章我们调试 STM32F407 的内置实时时钟 (RTC)，同时介绍与 RTC 相关的 BKP 功能。

28.1 RTC

RTC = Real-Time Clock，实时时钟。实时时钟有两种方式：芯片内置、CPU 外置。外置的 RTC 通常会选用一款芯片，例如 PCF8563、DS1302。当前大部分单片机芯片都有内置的 RTC。

RTC 通常有以下特点：

1. 低功耗。
2. 使用纽扣电池供电。
3. 主进入睡眠后，RTC 依然运行。
4. 时钟 32.768K

28.2 STM32 RTC

实时时钟 (RTC) 是一个独立的 BCD 定时器/计数器。RTC 提供一个日历时钟、两个可编程闹钟中断，以及一个具有中断功能的周期性可编程唤醒标志。RTC 还包含用于管理低功耗模式的自动唤醒单元。

两个 32 位寄存器包含二进码十进数格式 (BCD) 的秒、分钟、小时（12 或 24 小时制）、星期几、日期、月份和年份。此外，还可提供二进制格式的亚秒值。

系统可以自动将月份的天数补偿为 28、29（闰年）、30 和 31 天。并且还可以进行夏令时补偿。

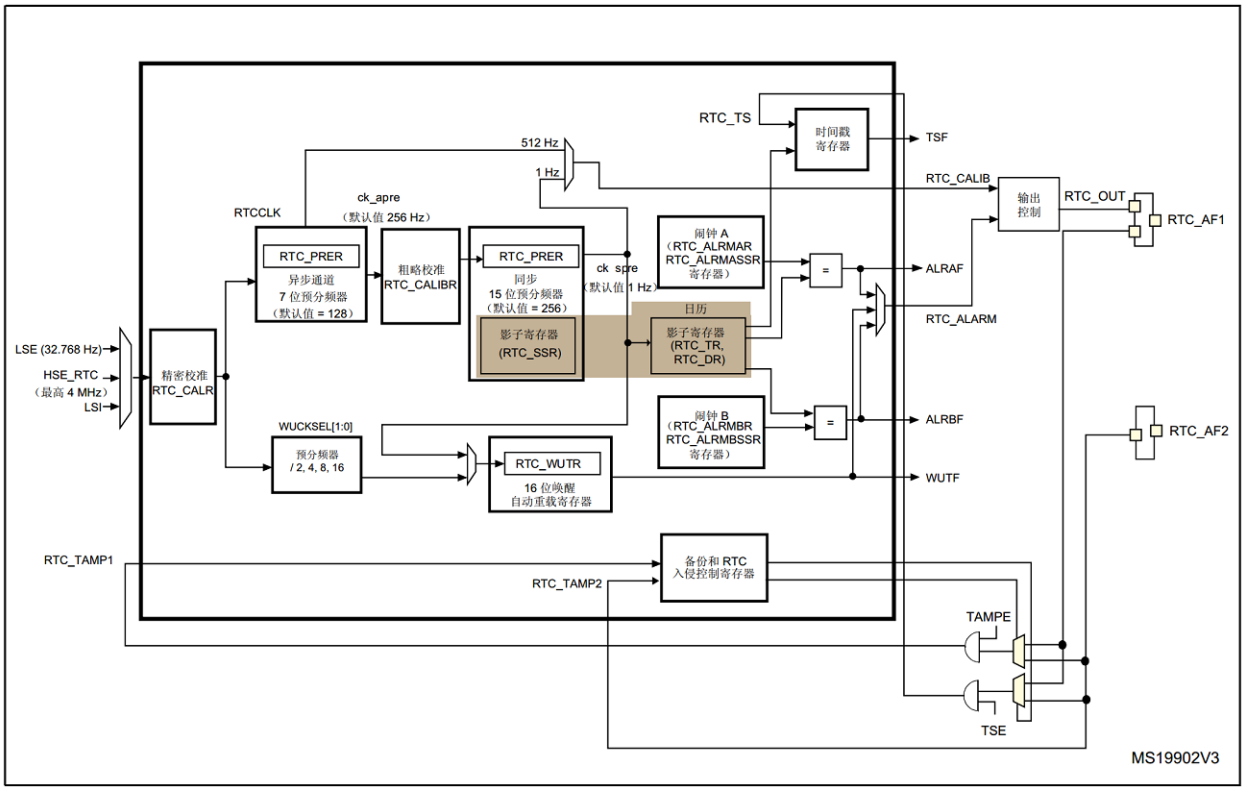
其它 32 位寄存器还包含可编程的闹钟亚秒、秒、分钟、小时、星期几和日期。

此外，还可以使用数字校准功能对晶振精度的偏差进行补偿。

上电复位后，所有 RTC 寄存器都会受到保护，以防止可能的非正常写访问。

无论器件状态如何（运行模式、低功耗模式或处于复位状态），只要电源电压保持在工作范围内，RTC 便不会停止工作。

28.2.1 框图


RTC_TS
时间戳
寄存器
TSF
512 Hz
1 Hz
RTC_CALIB
输出
控制
RTC_OUT
RTC_AF1
RTCCLK
ck_apre
（默认值 256 Hz）
RTC_PRER
异步通道
7 位预分频器
（默认值 = 128）
粗略校准
RTC_CALIBR
RTC_PRER
同步
15 位预分频器
（默认值 = 256）
ck_spre
（默认值 1 Hz）
闹钟 A
（RTC_ALRMAR
RTC_ALRMASSR
寄存器）
=
ALRAF
RTC_ALARM
日历
影子寄存器
(RTC_SSR)
影子寄存器
(RTC_TR,
RTC_DR)
LSE (32.768 Hz)
HSE_RTC
（最高 4 MHz）
LSI
精密校准
RTC_CALR
闹钟 B
（RTC_ALRMBR
RTC_ALRMBSSR
寄存器）
=
ALRBF
RTC_AF2
WUCKSEL[1:0]
预分频器
/ 2, 4, 8, 16
RTC_WUTR
16 位唤醒
自动重载寄存器
WUTF
RTC_TAMP1
RTC_TAMP2
备份和 RTC
入侵控制寄存器
TAMPE
TSE
MS19902V3

28.2.2 特性

- 包含亚秒、秒、分钟、小时（12/24 小时制）、星期几、日期、月份和年份的日历。
- 软件可编程的夏令时补偿。
- 两个具有中断功能的可编程闹钟。可通过任意日历字段的组合驱动闹钟。
- 自动唤醒单元，可周期性地生成标志以触发自动唤醒中断。
- 参考时钟检测：可使用更加精确的第二时钟源（50 Hz 或 60 Hz）来提高日历的精确度。
- 利用亚秒级移位特性与外部时钟实现精确同步。
- 可屏蔽中断/事件：
  - 闹钟 A
  - 闹钟 B
  - 唤醒中断
  - 时间戳
  - 入侵检测
- 数字校准电路（周期性计数器调整）
  - 精度为 5 ppm
  - 精度为 0.95 ppm，在数秒钟的校准窗口中获得
- 用于事件保存的时间戳功能（1 个事件）
- 入侵检测：
  - 2 个带可配置过滤器和内部上拉的入侵事件
- 20 个备份寄存器（80 字节）。发生入侵检测事件时，将复位备份寄存器。

28.2.3 备份寄存器 BKP

细节在 <23.3.13 入侵检测 >

STM32F407 RTC 附带两个入侵检测输入。与入侵检测配套 RTC 系统包含 20 个备份寄存器（80 字节）。发生入侵检测事件时，将复位备份寄存器。只要 RTC 后备电池正常，RTC 和备份寄存器区就会一直工作。不会在系统复位或电源复位时复位，也不会从器件待机模式唤醒时复位。

28.2.4 写操作

RTC 寄存器和备份寄存器都属于后备区，每次写操作前，需要先解锁后备区，以防后备区数据被串改。

更多资料请查看中文参考手册 *23* 章

28.3 编码调试

本节我们仅验证 RTC 能正常工作。

28.3.1 编码

RTC 初始化函数

```
/**
 *@brief:      mcu_rtc_init
 *@details:    复位时初始化 RTC
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 mcu_rtc_init(void)
{

        RTC_InitTypeDef RTC_InitStructure;
        volatile u32 cnt = 0;

          RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR|RCC_AHB1Periph_BKPSRAM, ENABLE);
        /* 操作 RTC 寄存器，需要使能备份区 */
        PWR_BackupAccessCmd(ENABLE);

        if(RTC_ReadBackupRegister(RTC_BKP_DR0)!=0x55aa)
        {
                wjq_log(LOG_DEBUG, " init rtc\r\n");
                /* 开启 LSE 时钟 */
                RCC_LSEConfig(RCC_LSE_ON);
                /* 等待 RCC LSE 时钟就绪 */
                while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
                {
                        cnt++;
                        if(cnt>0x2000000)
                        {
                                wjq_log(LOG_ERR, "lse not rdy\r\n");
                                return -1;

                        }
                }

                RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);
                RCC_RTCCLKCmd(ENABLE);
                //RTC 异步分频系数 (1~0X7F)
```

(continues on next page)

(continued from previous page)

```
        RTC_InitStructure.RTC_AsynchPrediv = 0x7F;
        //RTC 同步分频系数 (0~7FFF)
        RTC_InitStructure.RTC_SynchPrediv  = 0xFF;
        //RTC 设置为,24 小时格式
        RTC_InitStructure.RTC_HourFormat   = RTC_HourFormat_24;
        RTC_Init(&RTC_InitStructure);

        RTC_TimeTypeDef RTC_TimeStructure;

        RTC_TimeStructure.RTC_H12 = RTC_H12_AM;
        RTC_TimeStructure.RTC_Hours = 0;
        RTC_TimeStructure.RTC_Minutes = 0;
        RTC_TimeStructure.RTC_Seconds = 0;
        RTC_SetTime(RTC_Format_BIN, &RTC_TimeStructure);

        RTC_DateTypeDef RTC_DateStructure;
        RTC_DateStructure.RTC_Date = 1;
        RTC_DateStructure.RTC_Month = RTC_Month_January;
        RTC_DateStructure.RTC_WeekDay = RTC_Weekday_Thursday;
        RTC_DateStructure.RTC_Year = 0;//已 1970 年为起点,
        RTC_SetDate(RTC_Format_BIN, &RTC_DateStructure);

        RTC_WriteBackupRegister(RTC_BKP_DR0,0x55aa);        //标记已经初始化过了
    }

    wjq_log(LOG_INFO, " init rtc finish\r\n");
    return 0;
}
```

14 行，打开 RTC 时钟 16 行，解锁 BKP 区 19~59 行，配置 RTC。并不是每次都配置 RTC，因此我们用一个 BKP 寄存器记录是否已经配置。22 行，开启 LSE 时钟，LSE 时钟就是外部 RTC 晶振，32.768K。STM32 也支持使用内部时钟。24~33，等待 LSE 时钟稳定。38~41，配置 RTC 时钟。45~56，配置时间，分两部分，日期和时间。58 行，写 BKP 标志，只要 RTC 不断电，下次初始化就不会再初始化 RTC 了。

设置获取时间函数，这四个函数都是调用库函数而已。

```
s32 mcu_rtc_set_time(u8 hours, u8 minutes, u8 seconds)
s32 mcu_rtc_set_date(u8 year, u8 weekday, u8 month, u8 date)
s32 mcu_rtc_get_date(void)
s32 mcu_rtc_get_time(void)
```

28.3.2 测试

测试函数大概如下，在 main 函数中初始化 RTC，在 while（1）循环中按下按键后打印当前 RTC 时钟。

```
int main(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;

        /* Configure the NVIC Preemption Priority Bits */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

  /* SysTick end of count event */
  RCC_GetClocksFreq(&RCC_Clocks);
  SysTick_Config(RCC_Clocks.HCLK_Frequency / (1000/SYSTEMTICK_PERIOD_MS));

  /* Add your application code here */
  /* Insert 5 ms delay */
  Delay(5);

        /* Infinite loop */
        mcu_uart_init();
        mcu_uart_open(PC_PORT);
        wjq_log(LOG_INFO, "hello word!\r\n");
        mcu_rtc_init();
        dev_key_init();

        dev_key_open();
        while (1)
        {
                /* 驱动轮询 */
                dev_key_scan();
                eth_loop_task();

                /* 应用 */
                u8 key;
                s32 res;
                res = dev_key_read(&key, 1);
                if(res == 1)
                {
                        if(key == DEV_KEY_PRESS)
                        {
```

(continues on next page)

(continued from previous page)

```
                                /* 读时间 */
                                mcu_rtc_get_date();
                                mcu_rtc_get_time();
                                /* 设置时间 */
                                //mcu_rtc_set_date(2018, 2, 4, 17);
                                //mcu_rtc_set_time(2, 47, 0);
                        }
                        else if(key == DEV_KEY_REL)
                        {

                        }
                }
                Delay(1);

        }
}
```

- 测试 1

没有初始化过的 RTC：新硬件，或者是旧硬件扣下纽扣电池，并等待电容放完电（可以短路 RTC 电源快速放电）上电后调试信息会看到，程序进入 RTC 初始化。

> hello word! init rtc init rtc finish bus_vspi_init VSPI1 vspi init finish! board_spiflash jid:0xc22016 board_spiflash mid:0xc215 core_spiflash jid:0xef4017 core_spiflash mid:0xef16

- 测试 2

经第一步后，直接按复位键复位系统，可见 RTC 没有进入初始化。

> hello word! init rtc finish bus_vspi_init VSPI1 vspi init finish! board_spiflash jid:0xc22016 board_spiflash mid:0xc215 core_spiflash jid:0xef4017 core_spiflash mid:0xef16

- 测试 3

按下按键，查看系统时间，可见时间走走动。

> 1970, 1, 1, 4 0, 1, 34 1970, 1, 1, 4 0, 1, 40 1970, 1, 1, 4 0, 1, 42 1970, 1, 1, 4 0, 1, 47

- 测试 4

拔掉外电，只保留纽扣电池，放一段时间，再重新上电，可见 RTC 正常运行，并没有复位。

> hello word! init rtc finish bus_vspi_init VSPI1 vspi init finish! board_spiflash jid:0xc22016 board_spiflash mid:0xc215 core_spiflash jid:0xef4017 core_spiflash mid:0xef16 1970, 1, 1, 4 0, 8, 41 1970, 1, 1, 4 0, 8, 42 1970, 1, 1, 4 0, 8, 43 1970, 1, 1, 4 0, 8, 44

28.4 总结

本次我们验证了 RTC 的硬件和基本功能。其实 RTC 还有很丰富的功能，例如闹钟，大家可以自行实验。

28.5 end

CHAPTER 29

内存管理

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

前面已经将所有的硬件驱动实现，验证了硬件功能。但是每一个硬件都是单独测试的，而且并不完善。下一步，我们需要对各个驱动进行整合完善。在整合之前，需要做一些基础工作。其中之一就是实现**内存管理**。什么叫内存管理呢？为什么要做内存管理？前面我们已经大概了解了程序中的变量现在我们复习一下：**局部变量**、**全局变量**。

局部变量在进入函数时从栈空间分配，退出函数前释放。全局变量则在整个程序运行其中一直使用。在程序编译时就已经分配了 RAM 空间。

那还有没有第三种变量呢？可以说没有。但是如果从**生存周期**上看，是有的：**一个变量，在多个函数内使用，但是又不是整个程序运行期间都使用**。或：**一个变量，在一段时间内使用，不是整个程序运行生命周期都要用，但是用这个变量的函数会退出，然后重复进入**（用 static 定义的局部变量相当于全局变量）

如果不使用动态内存管理，这样的变量就只能定义为全局变量。如果将这些变量定义为指针，当要使用时，通过内存管理分配，使用完后就释放，这就叫做**动态分配**。举个实际的例子：

> 一个设备，有三种通信方式：串口，USB，网络，在通信过程每个通信方式需要 1K RAM。经过分析，3 种通信方式不会同时使用。那么，如果不使用动态内存，则需要 3K 变量。如果使用内存管理动态分配，则只需要 1K 内存就可以了。(这个只是举例，如果简单的系统，确定三种方式不同时使用，可以直接复用内存)

通信方式只是举例，其实一个系统中，并不是所有设备都一直使用，如果使用动态内存管理，RAM 的**峰值**用量将会大大减少。

29.1 内存管理方案

不发明车轮，只优化轮胎。

内存管理是编程界的一个大话题，有很多经典的方案。很多人也在尝试写新的方案。内存分配模块我们使用 K&R C examples 作为基础，然后进行优化。K&R 是谁？就是写《C 程序设计语言》的两个家伙。如果你没有这本书，真遗憾。这本书的 8.7 章节，< 实例–存储分配程序 >，介绍了一种基本的存储分配方法。代码见 alloc.c，整个代码只有 120 行，而且**结构很美**。

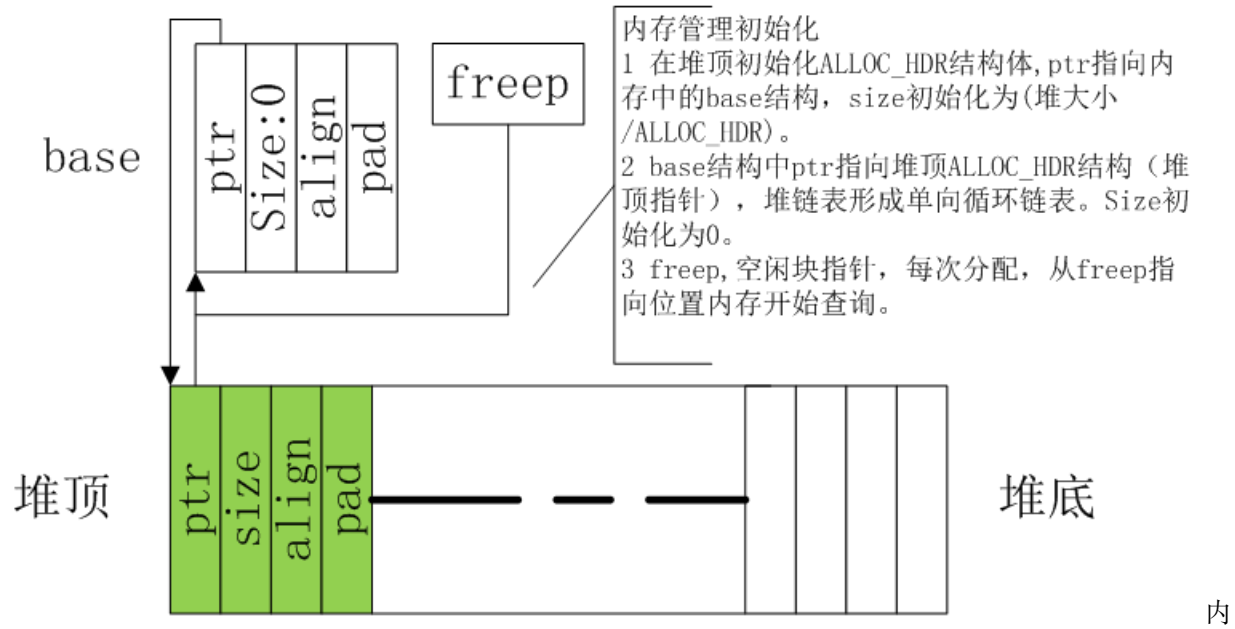
29.2 K&R 内存管理方案分析

下面我们结合代码分析这种内存分配方案。代码在 wujique\Utilities\alloc 文件夹。

29.2.1 内存分析

- 初始化

在 malloc 函数中，如果是第一次调用就会初始化内存链表。代码原来是通过获取堆地址，在堆上建立内存池。我们把他改为更直观的数组定义方式。内存建立后的内存视图如下：



内存初始化

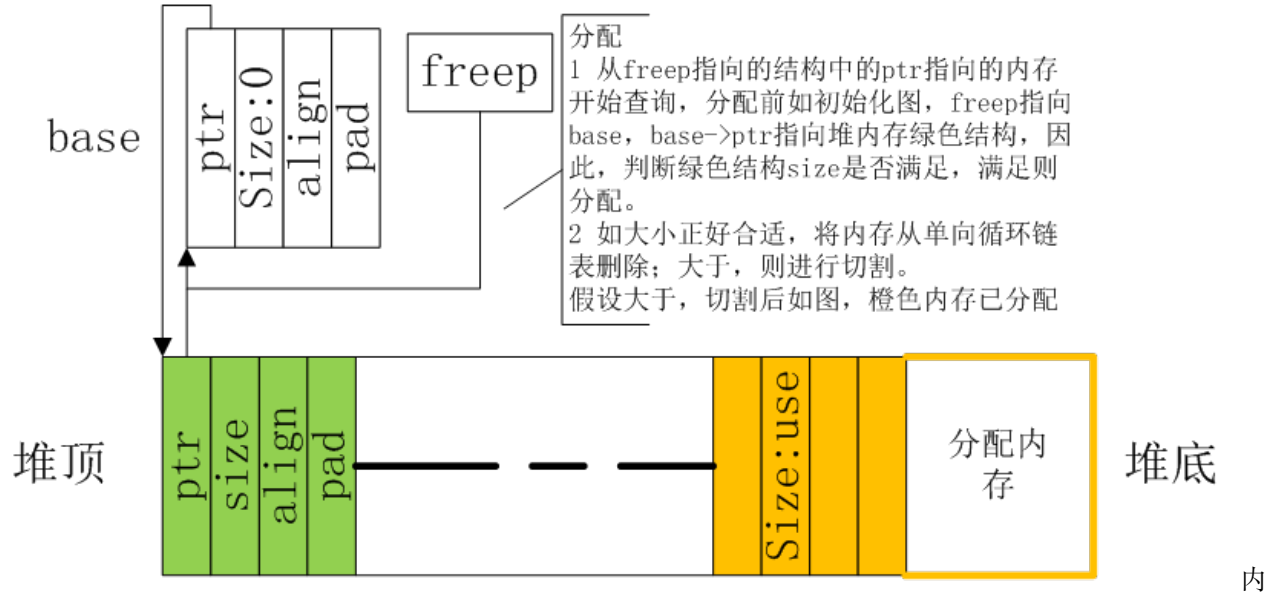
内存分配的最小单元是：

```
typedef struct ALLOC_HDR
{
  struct
  {
    struct ALLOC_HDR *ptr;
    unsigned int size;/* 本快内存容量 */
  } s;
  unsigned int align;
  unsigned int pad;
} ALLOC_HDR;
```

这也就是内存管理结构体。在 32 位 ARM 系统上，这个结构体是 16 字节。

- 第一次分配

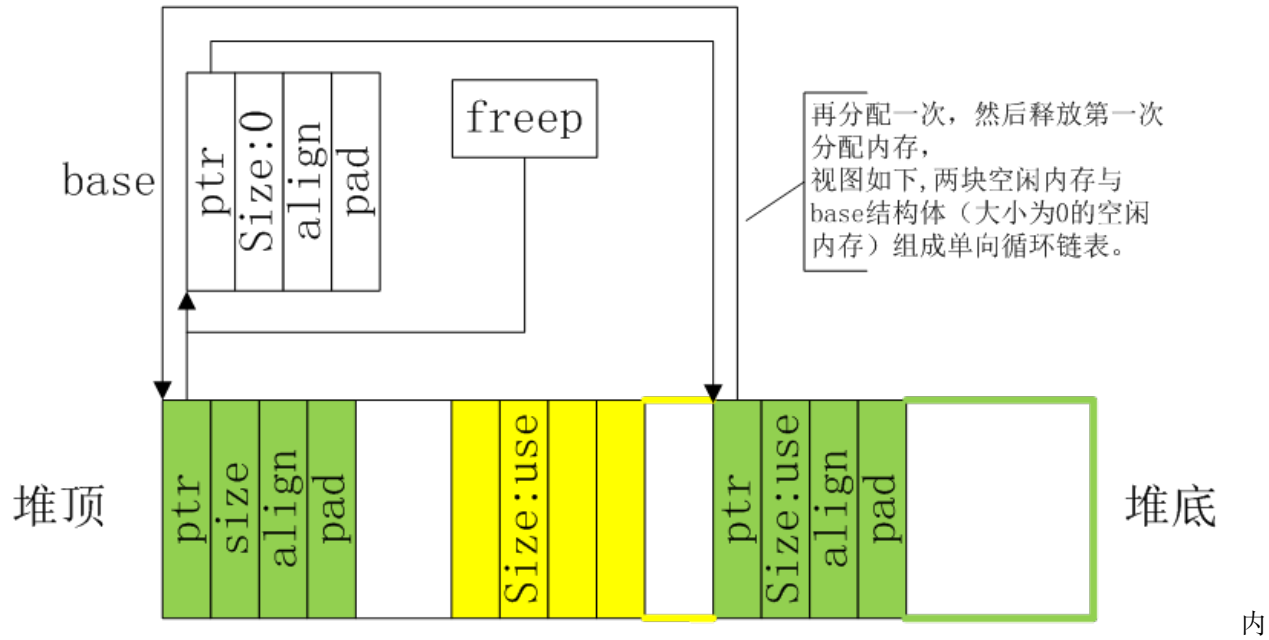
每次分配，就是在一块可以分配的空间尾部切割一块出来，切割的大小是 16 字节的倍数，而且会比需要的内存多一块头。这块头在内存释放时需要使用。这一块，也就是内存管理的开销。



内存第一次分配

- 分配释放后

经过多次分配释放后，内存可能如下图，绿色是两块不连续的空闲块，黄色是分配出去的块。分配出去的块，已经不在内存链表里面。



内存分配释放

29.2.2 缺点

一般情况上面的代码已经能满足需求。但是，有以下缺陷：

- 缺点 1：容易碎片化

分配使用首次适应法，也即是找到一块大于等于要分配内存的空闲块，立刻进行分配。这种方法的优点是速度较快，缺点是容易内存碎片化，分配时将很多大块内存切割成小内存了。经过多次分配后，很可能出现以下情况：

> 空闲内存总量还有 10K，但是却被分散在 10 个块内，而且没有大容量的内存块，再申请 2K 内存就出现失败。如果对时间并不是那么敏感，我们可以使用最适合法，也即是遍历空闲链表，查找一个最合适的内存（**大于要分配内存且容量最小的空闲块**），减少大内存被切碎的概率。需要注意的是，最适合法，除了会增加分配时间，**不会减少内存碎片数量，只是增加了空闲内存的集中度**。假设经过多次分配后，空闲总量还是 10K，也是分散在 10 个空闲块，但是在这 10 个空闲块中，会有 5K 的大块，再申请 2K 的时候，就可以申请到 2K 内存了。

- 缺点 2：内存消耗

内存分配方案使用了一个结构体，每次分配的最小单位就是这个结构体的大小 16 字节。

```
typedef struct ALLOC_HDR
{
  struct
  {
    struct ALLOC_HDR *ptr;
    unsigned int size;/* 本快内存容量 */
  } s;
  unsigned int align;
  unsigned int pad;
} ALLOC_HDR;
```

一次分配，最少就是 2 个结构体（一个结构体用于管理分配出去的内存，其余结构体做为申请内存），也就是 32 字节。如果代码有大量小内存申请，例如申请 100 次 8 个字节

> 需求内存：100X8=800 字节实际消耗内存 100X32 = 3200 字节利用率只有 800/3200 =25%

如果内存分配只有 25% 的使用率，对于小内存嵌入式设备来说，是致命的方案缺陷。

如何解决呢？我们可以参考 LINUX 内存分配方案 SLAB。在 LINUX 中，有很多模块需要申请固定大小的内存（例如 node 结构体），为了加快分配速度，系统会使用 malloc 先从大内存池中申请一批 node 结构体大小的内存，作为一个 slab 内存池。当需要分配 node 结构体时，就直接从 slab 内存池申请。同理，可以将内存分配优化为：需要小内存时，从大块内存池分配一块大内存，例如 512，使用**新算法管理**，用于小内存分配。当 512 消耗尽，再从大内存池申请第二块 512 字节大内存。当小内存释放时，判断小块内存池是否为空，如为空，将小块内存池释放回大内存池。那如何管理这个小内存池呢？

- 缺点 3：没有管理已分配内存

内存分配没有将已分配内存管理起来。我们可以对已分配内存进行统一管理：

> 1 已分配内存在头部有原来的结构体，通过 ptr 指针，将所有已分配内存连接在已分配链表上。2 利用不使用的 align 跟 pad 成员，记录分配时间跟分配对象 (记录哪个驱动申请的内存)

通过上面优化后，就可以**统计已经分配了多少内存，还有多少空闲内存，哪个模块申请了最多内存**等数据。

29.3 使用

1 将代码中的所有 free 改为为 wjq_free，malloc 改为 wjq_malloc。

串口缓冲用了 free 跟 malloc. fatfs 的 syscall.c 用了 lwip 的 mem.h 用了。

2 修改启动代码, 栈跟堆改小。不用库的 malloc，堆可以完全不要。栈，还是要保留，但是不需要那么大，如果函数内用到比较大的局部变量，改为动态申请。

```
Stack_Size      EQU     0x00002000

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp

; <h> Heap Configuration
;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
; </h>

Heap_Size       EQU     0x00000010

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

3 内存池开了 80K，编译不过

linking…\Objects\wujique.axf: Error: L6406E: No space in execution regions with .ANY selector matching dev_touchscreen.o(.bss). .\Objects\wujique.axf: Error: L6406E: No space in execution regions with .ANY selector matching mcu_uart.o(.bss). .\Objects\wujique.axf: Error: L6406E: No space in execution regions with .ANY selector matching etharp.o(.bss). .\Objects\wujique.axf: Error: L6406E: No space in execution regions with .ANY selector matching mcu_can.o(.bss). .\Objects\wujique.axf: Error: L6406E: No space in execution regions with .ANY selector matching netconf.o(.bss).

先把内存池改小，编译通过之后，分析 map 文件，用了较多全局变量的统统改小或者改为动态申请。分析 map 文件，还可以检查还有没有使用库里面的 malloc。

Code (inc. data) RO Data RW Data ZI Data Debug Object Name

124	32	0	4	40976	1658	alloc.o
16	0	0	0	0	2474	def.o
96	34	8640	4	0	1377	dev_dacsound.o
300	36	0	0	0	2751	dev_esp8266.o
204	38	0	1	0	1446	dev_key.o
436	98	0	10	16	3648	dev_touchkey.o
310	18	0	14	3000	3444	dev_touchscreen.o
932	18	0	4	0	15981	dhcp.o
0	0	0	0	3964	5933	dual_func_demo.o
280	14	12	0	200	5963	etharp.o
0	0	0	0	0	35864	ethernetif.o
0	0	0	0	0	3820	inet.o
98	0	0	0	0	2022	inet_chksum.o
0	0	0	0	0	4163	init.o
168	4	0	20	0	4763	ip.o
0	0	4	0	0	6463	ip_addr.o
386	4	0	0	0	4118	ip_frag.o
264	38	0	8	16	383399	main.o
84	8	0	0	0	1410	mcu_adc.o
60	32	0	1	68	1511	mcu_can.o
12	0	0	0	0	521	mcu_dac.o
128	14	0	0	0	2352	mcu_i2c.o
28	8	0	1	0	630	mcu_i2s.o
336	92	0	0	0	2689	mcu_rtc.o
430	86	0	1	0	4396	mcu_timer.o
1564	82	0	0	328	9072	mcu_uart.o
504	20	0	12	0	4510	mem.o
56	10	0	0	9463	3250	memp.o
120	14	0	0	0	1651	misc.o
0	0	0	0	56	1066	netconf.o
118	0	0	0	0	4267	netif.o
684	0	0	0	0	6971	pbuf.o
36	8	392	0	8192	824	startup_stm32f40_41xxx.o

从上面数据可以看出以下源文件用了较多 RAM。

> alloc.o 内存池 dev_touchscreen.o 触摸屏缓冲 dual_func_demo.o USB，应该能优化 memp.o 什么鬼？又一个内存池？应该是要优化掉 startup_stm32f40_41xxx.o 启动代码，是栈跟堆用的 RAM.

由于编译器的优化，项目没用到的代码没有编译进来，上面的 map 数据并不完整。等后面我们做完全部测试程序，所有用到的代码都会参与连接，到时还需要优化一次。

29.4 总结

内存管理暂时到此，等后面所有功能都完成后，再进行一次优化。如果对内存分配时间有更高要求，可使用伙伴内存分配法。大家可以参考《都江堰操作系统与嵌入式系统设计》，这个文档里面的一些软件设计策略非常好。http://www.djyos.com/?page_id=50

最新内存管理请查看屋脊雀在 github 上托管的代码

29.5 end

CHAPTER 30

COG LCD & OLED LCD 调试记录

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

前面我们调试过彩屏，彩屏显示效果很好，但是价格高。而且在一些低主频的单片机上带不动，如果换 SPI 的彩屏，240x320 像素太大，刷屏慢。在很多场合，人机交互界面并不需要那么复杂。简单的交互，只要一块 128*64 像素的单色屏就足够了。

30.1 LCD 种类

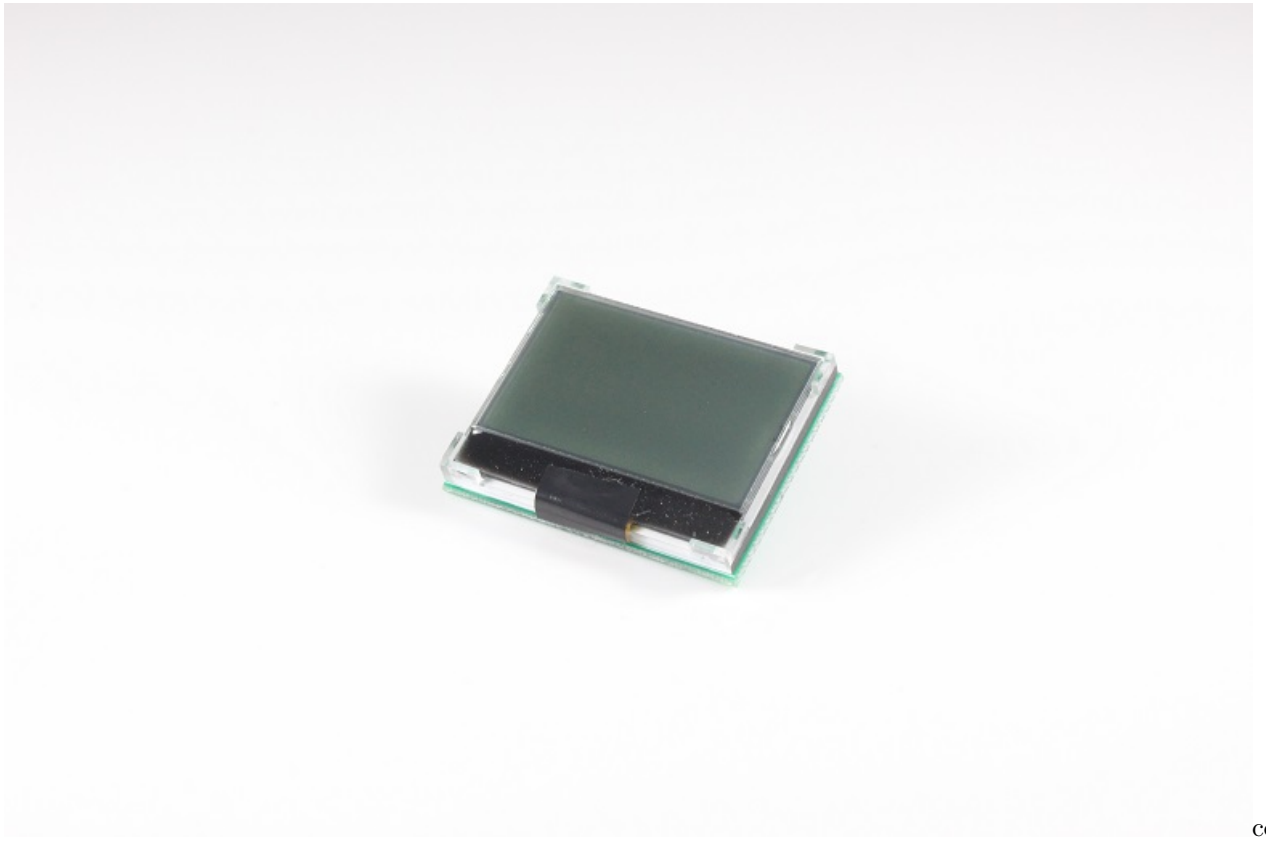
液晶 LCD 有很多种类，很早之前，**12864** 就是一种屏幕的通称。这种屏幕是绿色的，很大，非常笨重，通常在工业仪器上使用。现在常用的液晶除了 **TFT**，就是 **COG** 和 **OLED**。

30.1.1 COG lcd

很多人可能不知道 COG LCD 是什么，我觉得跟现在开发板销售有关：大家都出大屏，玩酷炫界面。离实际应用太远。单片机产品使用的 LCD 其实更常见的是 COG LCD。

所谓的 COG LCD：

> COG 是 Chip On Glass 的缩写，就是驱动芯片直接绑定在玻璃上，透明的。



实物如下图：

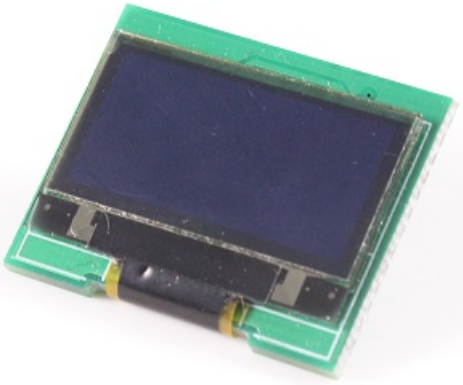
cog

显示效果 

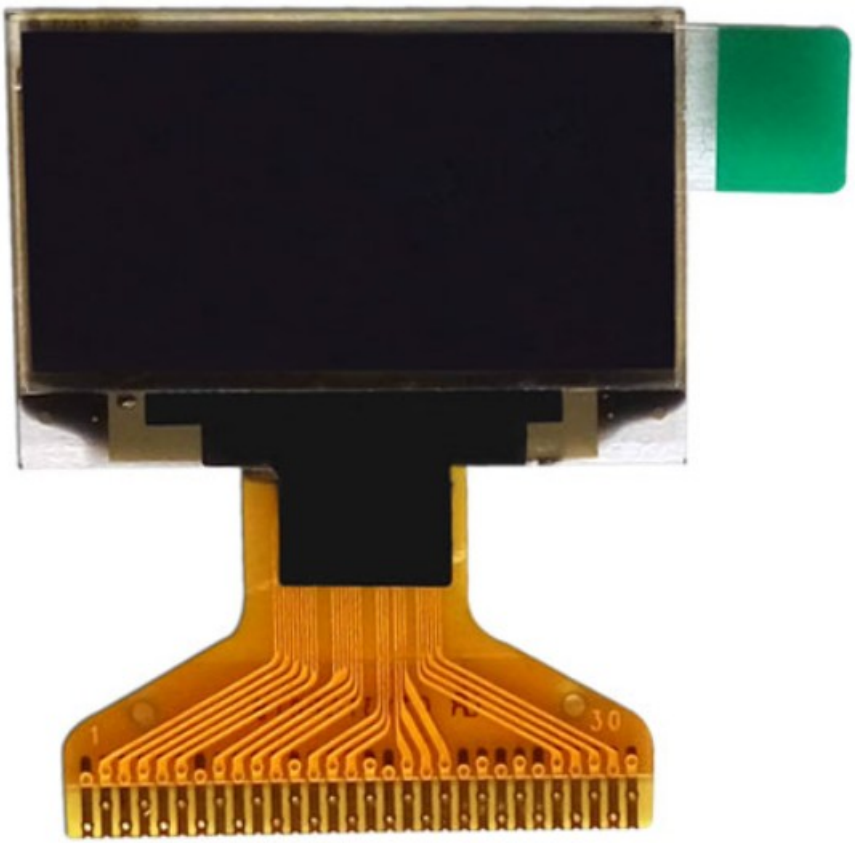
COG 显示效果这种 LCD 通常像素不高，常用的有 128X64，128X32。一般只支持黑白显示，也有灰度屏。接口通常是 SPI，I2C。也有号称支持 8 位并口的，不过基本不会用。3 根 IO 能解决的问题，没必要用 8 根吧？常用的驱动 IC：STR7565。

30.1.2 OLED lcd

买过开发板的应该基本用过。新技术，大家都感觉高档，手环智能手表等时尚产品多是用 OLED。OLED 目前屏幕较小，大一点的都很贵。在控制上跟 COG LCD 类似，区别是两者的显示方式不一样。从程序角度看，最大的差别就是：OLED LCD 不用控制背光。实物如下图，



oled



裸屏

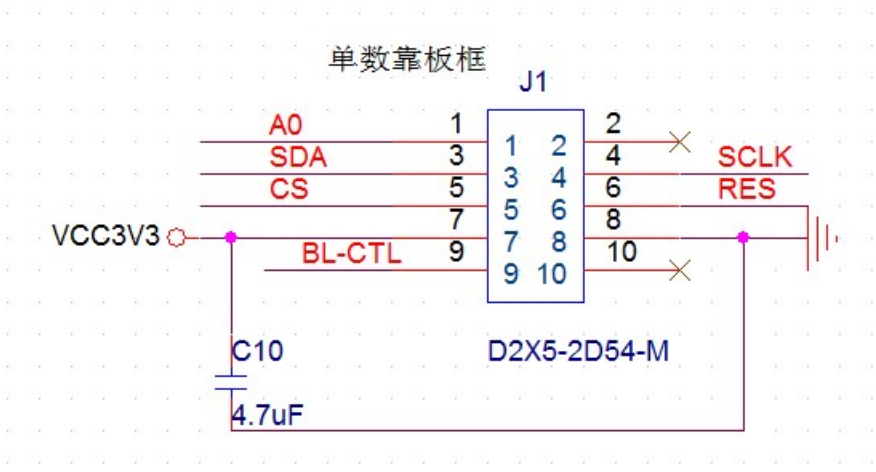
裸屏

常见的是 SPI 和 I2C 接口。常见驱动 IC：SSD1615。

30.2 接口

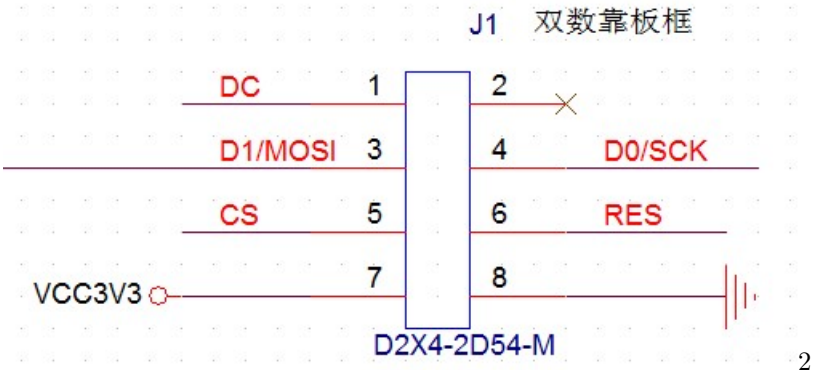
30.2.1 硬件接口

COG 屏跟 OLED 屏，内置控制器不同，但是基本上都支持 I2C，SPI 通信。这次调试的模块使用 SPI 通信。

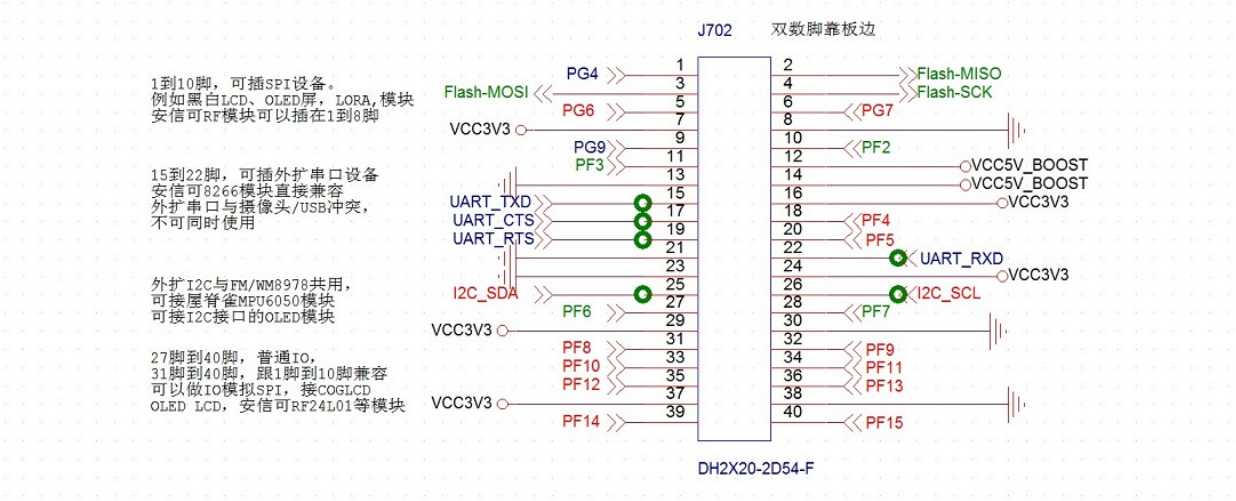


1 上图是 COG LCD 信号

1 脚 A0，选择命令或数据通信。3 脚 SDA，相当于 SPI 的 mosi 4 脚时钟 5 脚片选 6 脚复位信号 9 脚背光



2 上图是 OLED 模块接口，跟 COG 接口是兼容的。这两个 LCD，可以接到我们核心板的外扩 SPI 接口上，使用 SPI3 控制器。外扩接口信号如下图：



外扩接口 LCD 的 1 脚接到外扩接口的 1 脚即可。

30.2.2 控制

spi 接口的 LCD 控制其实跟 TFT LCD 的控制逻辑相似。SPI 上的数据也分命令和显示数据，由 A0 脚的电平决定。OLED 和 COG LCD 的初始化，通常也是由模组厂提供。

30.3 驱动设计

30.3.1 硬件接口

COG 跟 OLED，也是 LCD，那么功能应该跟 TFT lcd 是一致的。驱动提供的接口也应该一致，都是下面的接口：

```
typedef struct
{
        u16 id;

        s32 (*init)(void);
        s32 (*draw_point)(u16 x, u16 y, u16 color);
        s32 (*color_fill)(u16 sx,u16 ex,u16 sy,u16 ey, u16 color);
        s32 (*fill)(u16 sx,u16 ex,u16 sy,u16 ey,u16 *color);
        s32 (*onoff)(u8 sta);
        s32 (*prepare_display)(u16 sx, u16 ex, u16 sy, u16 ey);
        void (*set_dir)(u8 scan_dir);
        void (*backlight)(u8 sta);
}_lcd_drv;
```

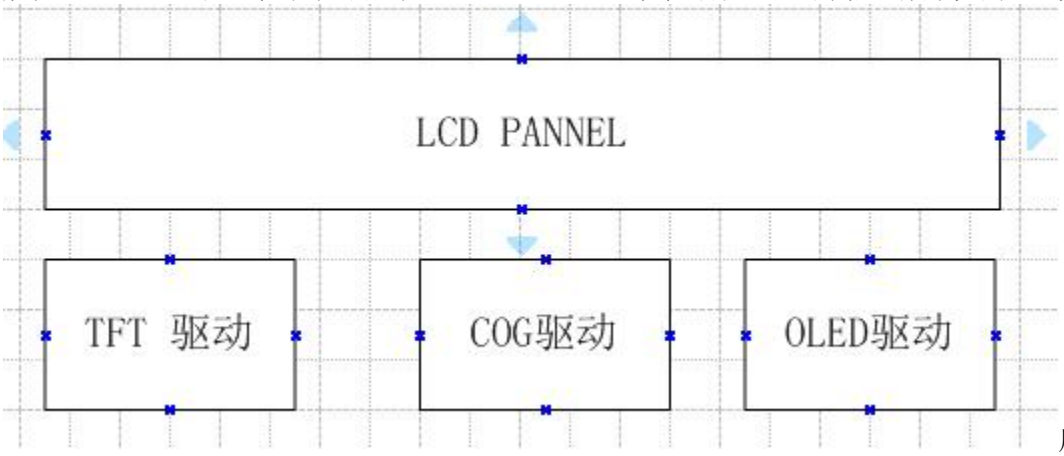
如果某种 LCD 不支持某些接口，执行空操作即可。例如 OLED 就没有背光控制。

30.3.2 层次

在调试 TFT LCD 的时候，在驱动中实现了一些显示函数，例如：

```
s32 dev_lcd_drawpoint(u16 x, u16 y, u16 color)
void put_string_center(int x, int y, char *s, unsigned colidx)
...
```

dev_lcd 开头的函数，其实应该归类为 GUI 层（或者 LCD panel 层），而不是 LCD 驱动层。当时为了方便，暂时放在 TFT LCD 源码内。现在添加了 COG 跟 OLED 驱动，要将 LCD 中间层的分离出来。函数层次应该如下：



层次因此我们将这些函数抽取出来，单独做一个 dev_lcd.c 的源码。

30.3.3 LCD PANNEL 接口跟 LCD 驱动连通

这个框架其实非常简单：1 在 dev_lcd_init 函数内**选择**要初始化的 LCD，

```
/* 初始化 OLED LCD */
                ret = drv_ssd1615_init();
                if(ret == 0)
                {
                        LCD.drv = &OledLcdSSD1615rv;
                        LCD.dir = W_LCD;
                        LCD.height = 64;
                        LCD.width = 128;
                }
```

2 初始化之后将对应 LCD 的驱动 _lcd_drv 指针赋值到 LCD 结构体，LCD 结构体如下，第一个结构体成员就是驱动，另外四个是相关参数。

```
struct _strlcd_obj
{
        _lcd_drv *drv;

        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

};
```

3 当应用程序调用函数时，例如 put_char，在函数内通过 LCD 结构体内的 drv 指针调用对应的 LCD 驱动函数。

30.3.4 COG/OLED 驱动

STR7565 跟 SSD1615 非常类似，两个驱动除了初始化外，其他驱动函数共用。

- 硬件接口

我们定义一个 LCD 接口，叫做 bus_seriallcd，主要基于 SPI3 控制器，还有命令数据选择脚、复位管脚，COG 还有背光控制管脚。这个接口，封装以下接口给 LCD 驱动使用：

```
void bus_seriallcd_IO_init(void)
s32 bus_seriallcd_bl(u8 sta)
s32 bus_seriallcd_init()
s32 bus_seriallcd_open()
s32 bus_seriallcd_close()
s32 bus_seriallcd_write_data(u8 *data, u16 len)
s32 bus_seriallcd_write_cmd(u8 cmd)
```

- 显存

Display Data RAM

The display data RAM stores the dot data for the LCD. It has a 65 (8 page x 8 bit +1) x 132 bit structure. As is shown in Figure 3, the D7 to D0 display data from the MPU corresponds to the LCD display common direction; there are few constraints at the time of display data transfer when multiple ST7565P are used, thus and display structures can be created easily and with a high degree of freedom.

D0	0	1	1	1		0
D1	1	0	0	0		0
D2	0	0	0	0		0
D3	0	1	1	1		0
D4	1	0	0	0		0
-						

Display data RAM

COG 跟 OLED 每个点的显示数据只用一个 BIT 表示。

如上图的说明，str7565 芯片内部显存 65*132bit，我们的 64*128 液晶，并没有全部用完。不同的液晶使用的显存不一样，要根据实际情况做**偏移**，我们的液晶从（0.0）开始，因此不需要做偏移。显示一个点用一个位，但是我们操作一次是写一个字节，为了操作方便，我们开辟一片**显示缓存**，用于记录当前 LCD 显示内容。

```
struct _cog_drv_data
{
        u8 gram[8][128];
};
```

修改显示内容时先将数据组织到显存，再通过刷屏函数更新到 LCD。**一个像素点占显存的一个 *BIT*，但是 *SPI* 一次传输的是 *1* 个字节 *8* 个 *BIT***。刷屏函数如下：

```
/**
 *@brief:       drv_ST7565_refresh_gram
 *@details:       刷新指定区域到屏幕上
                  坐标是横屏模式坐标
 *@param[in]   u16 sc
               u16 ec
               u16 sp
               u16 ep
 *@param[out]  无
```

(continues on next page)

(continued from previous page)

```
 *@retval:     static
 */
static s32 drv_ST7565_refresh_gram(u16 sc, u16 ec, u16 sp, u16 ep)
{
        struct _cog_drv_data *gram;
        u8 i;

        //uart_printf("drv_ST7565_refresh:%d,%d,%d,%d\r\n", sc,ec,sp,ep);
        gram = (struct _cog_drv_data *)&LcdGram;

        bus_seriallcd_open();
        for(i=sp/8; i <= ep/8; i++)
        {
                bus_seriallcd_write_cmd (0xb0+i);    //设置页地址（0~7)
                bus_seriallcd_write_cmd (((sc>>4)&0x0f)+0x10);      //设置显示位置—列高地
址
                bus_seriallcd_write_cmd (sc&0x0f);      //设置显示位置—列低地址

                bus_seriallcd_write_data(&(gram->gram[i][sc]), ec-sc+1);

        }
        bus_seriallcd_close();

        return 0;
}
```

- 测试

现在程序不能支持多个 lcd 同时工作，需要在初始化函数中选择使用哪个 LCD。

```
#if 1
        if(ret != 0)
        {
                /* 初始化 COG 12864 LCD */
                ret = drv_ST7565_init();
                if(ret == 0)
                {
                        LCD.drv = &CogLcdST7565Drv;
                        LCD.dir = W_LCD;
                        LCD.height = 64;
                        LCD.width = 128;
```

(continues on next page)

(continued from previous page)

```
                }
        }
        #else
        if(ret != 0)
        {
                /* 初始化 OLED LCD */
                ret = drv_ssd1615_init();
                if(ret == 0)
                {
                        LCD.drv = &OledLcdSSD1615rv;
                        LCD.dir = W_LCD;
                        LCD.height = 64;
                        LCD.width = 128;
                }
        }
        #endif
```

测试程序也和彩屏不一样，

```
put_string_center (20, 32,
                           "ADCD WUJIQUE !", BLACK);
                Delay(1000);
                LCD.drv->color_fill(0,LCD.width,0,LCD.height,WHITE);
                Delay(1000);
                LCD.drv->color_fill(0,LCD.width,0,LCD.height,BLACK);
                Delay(1000);
                LCD.drv->color_fill(0,LCD.width,0,LCD.height,WHITE);
                Delay(1000);
```

更多驱动细节请查看代码 dev_str7565.c

30.4 总结

请看下节，用 VSPI 控制 LCD。

实际项目最好不要使用本节例程的驱动，请从 github 上获取最新的驱动代码本节只是 LCD 驱动架构演进的中间过程。

30.5 end

CHAPTER 31

VSPI 控制 COG LCD & I2C 控制 OLED

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

上一节我们已经点亮了 COG LCD 跟 OLED LCD，用的是外扩 SPI。在核心板上的外扩接口中，除了硬件 SPI 外，还有多个 IO 口，可以用来模拟 SPI。还有一个 I2C，正好可以用来控制 I2C 接口的 OLED LCD。就让我们来完善我们的 LCD 驱动，让它支持更多方式吧。

31.1 驱动

驱动上一节已经实现，不需要修改。

31.2 底层接口

上一节我们已经定义了一些函数，用于 LCD 驱动操作硬件。如下：

```
void bus_seriallcd_IO_init(void)
s32 bus_seriallcd_bl(u8 sta)
s32 bus_seriallcd_init()
s32 bus_seriallcd_open()
s32 bus_seriallcd_close()
s32 bus_seriallcd_write_data(u8 *data, u16 len)
s32 bus_seriallcd_write_cmd(u8 cmd)
```

其中 bus_seriallcd_IO_init 内部接口。现在我们要添加模拟 SPI 跟 I2C。因此将这个串行 LCD 接口抽象一个结构体，

```
typedef struct
{
        char * name;

        s32 (*init)(void);
        s32 (*open)(void);
        s32 (*close)(void);
        s32 (*writedata)(u8 *data, u16 len);
        s32 (*writecmd)(u8 cmd);
        s32 (*bl)(u8 sta);
}_lcd_bus;
```

也即是说，一个 LCD 接口，只需要提供这些功能即可。然后定义三个 LCD 接口，并且实现他们的功能函数，以下分别是 I2C、VSPI、SPI。函数具体实现请看代码。

```
_lcd_bus BusSerialLcdVI2C={
                .name = "BusSerivaLcdVI2C",
                .init =bus_seriallcd_vi2c_init,
                .open =bus_seriallcd_vi2c_open,
                .close =bus_seriallcd_vi2c_close,
                .writedata =bus_seriallcd_vi2c_write_data,
                .writecmd =bus_seriallcd_vi2c_write_cmd,
```

(continues on next page)

(continued from previous page)

```
                .bl =bus_seriallcd_vi2c_bl,
};

_lcd_bus BusSerialLcdVSpi={
                .name = "BusSerivaLcdVSpi",
                .init =bus_seriallcd_vspi_init,
                .open =bus_seriallcd_vspi_open,
                .close =bus_seriallcd_vspi_close,
                .writedata =bus_seriallcd_vspi_write_data,
                .writecmd =bus_seriallcd_vspi_write_cmd,
                .bl =bus_seriallcd_vspi_bl,
};

_lcd_bus BusSerialLcdSpi={
                .name = "BusSerivaLcdSpi",
                .init =bus_seriallcd_spi_init,
                .open =bus_seriallcd_spi_open,
                .close =bus_seriallcd_spi_close,
                .writedata =bus_seriallcd_spi_write_data,
                .writecmd =bus_seriallcd_spi_write_cmd,
                .bl =bus_seriallcd_spi_bl,
};
```

那我们用哪个接口呢？定义一个接口指针，要用哪个接口就赋值对应结构体。

```
_lcd_bus *LcdBusDrv = &BusSerialLcdVI2C;
```

并且修改驱动，原来直接调用函数的地方改为变量指针，例如：

```
static s32 drv_ST7565_refresh_gram(u16 sc, u16 ec, u16 sp, u16 ep)
{
        struct _cog_drv_data *gram;
        u8 i;

        //uart_printf("drv_ST7565_refresh:%d,%d,%d,%d\r\n", sc,ec,sp,ep);
        gram = (struct _cog_drv_data *)&LcdGram;

        LcdBusDrv->open();
    for(i=sp/8; i <= ep/8; i++)
    {
        LcdBusDrv->writecmd (0xb0+i);     //设置页地址（0~7）
```

(continues on next page)

(continued from previous page)

```
        LcdBusDrv->writecmd (((sc>>4)&0x0f)+0x10);      //设置显示位置—列高地址
        LcdBusDrv->writecmd (sc&0x0f);      //设置显示位置—列低地址

         LcdBusDrv->writedata(&(gram->gram[i][sc]), ec-sc+1);

        }
        LcdBusDrv->close();

        return 0;
}
```

31.2.1 模拟 SPI

在触摸芯片章节我们已经实现了一个模拟 SPI。现在多增加一个。

```
#define VSPI2_CS_PORT GPIOF
#define VSPI2_CS_PIN GPIO_Pin_12

#define VSPI2_CLK_PORT GPIOF
#define VSPI2_CLK_PIN GPIO_Pin_11

#define VSPI2_MOSI_PORT GPIOF
#define VSPI2_MOSI_PIN GPIO_Pin_10

#define VSPI2_MISO_PORT GPIOF
#define VSPI2_MISO_PIN GPIO_Pin_9

#define VSPI2_RCC RCC_AHB1Periph_GPIOF

DevVspiIO DevVspi2IO={
                "VSPI2",
                DEV_VSPI_2,
                -2,//未初始化

                VSPI2_RCC,
                VSPI2_CLK_PORT,
                VSPI2_CLK_PIN,
```

(continues on next page)

(continued from previous page)

```
            VSPI2_RCC,
            VSPI2_MOSI_PORT,
            VSPI2_MOSI_PIN,

            VSPI2_RCC,
            VSPI2_MISO_PORT,
            VSPI2_MISO_PIN,

            VSPI2_RCC,
            VSPI2_CS_PORT,
            VSPI2_CS_PIN,
    };
```

31.2.2 I2C

I2C 就用前面调好的，不需要修改。

31.2.3 层次拆分

到此，我们实现了功能，但是，你有没有感觉，**LCD 硬件接口**跟 **LCD 驱动**，是两层意思？也就是说驱动分四层：**LCD 中间层**——**LCD 驱动层**——**LCD 硬件接口**——**对应的通信接口**。对应：LCD 显示——SSD1565 驱动——SPI LCD 接口——SPI 驱动

31.3 总结

由于我们程序良好的架构设计，仅仅修改了 LCD 硬件接口层的处理方式，就将原来使用 SPI 接口的 LCD 驱动，改成 VSPI 接口和 I2C 接口。而且，LCD 驱动可以说基本上没修改。

1. LCD 驱动已经能支持我们的外扩 3 种接口。请问：TFT 8080 接口是否能用这种封装？
2. 现在的代码，同时只能支持一个硬件设备。具体用什么设备，在 dev_lcd_init 中选，用什么总线，通过 LcdBusDrv 指针选。如果需要同时使用，怎么办？比如 8080，VSPI,I2C,SPI，四个接口全部接上 LCD。

请看下一节。

本节例程代码，只是给大家参考，如果设计项目使用，建议用屋脊雀在 github 上的最新代码架构

31.4 end

CHAPTER 32

LCD 驱动应该怎么写？

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

网络上配套 STM32 开发板有很多 LCD 例程，主要是 TFT LCD 跟 OLED 的。从这些例程，大家都能学会如何点亮一个 LCD。但是不知道有多少人会直接使用这些代码，至少我不用，不是不用，而是用不了。因为这些代码都有下面这些问题：

1 分层不清晰，通俗讲就是模块化太差。2 接口乱。其实只要接口不乱，分层就会好很多了。3 可移植性差。4 通用性差。

为什么这样说呢？如果你已经了解了 LCD 的操作，请思考如下情景：

1 代码空间不够，只能保留 9341 的驱动，其他 LCD 驱动全部删除。能一键（一个宏定义）删除吗？删除后要改多少地方才能编译通过？2 有一个新产品，收银设备。系统有两个 LCD，一个叫做主显示，收银员用；一个叫顾显，顾客看金额。怎么办？这些例程代码要怎么改才能支持两个屏幕？复制一套然后改函数名称？这样确实能完成任务，只不过程序从此就进入**恶性循环**了。文艺点说，就是程序变得不美了。3 一个 OLED，原来接在 SPI，后来改到 I2C，容易改吗？4 原来只是支持中文，现在要卖到南美，要支持多米尼加语言，好改吗？

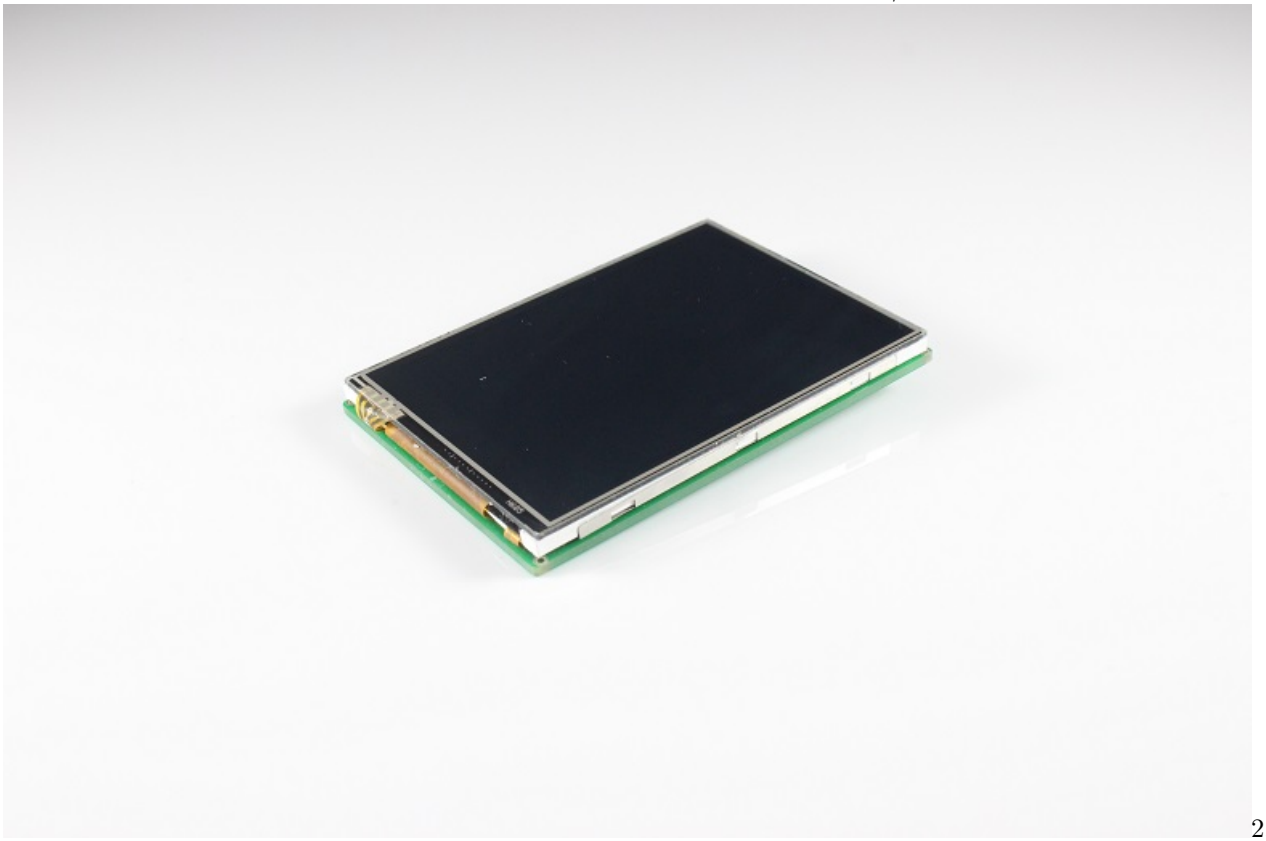
大家慢慢想。

32.1 LCD 种类概述

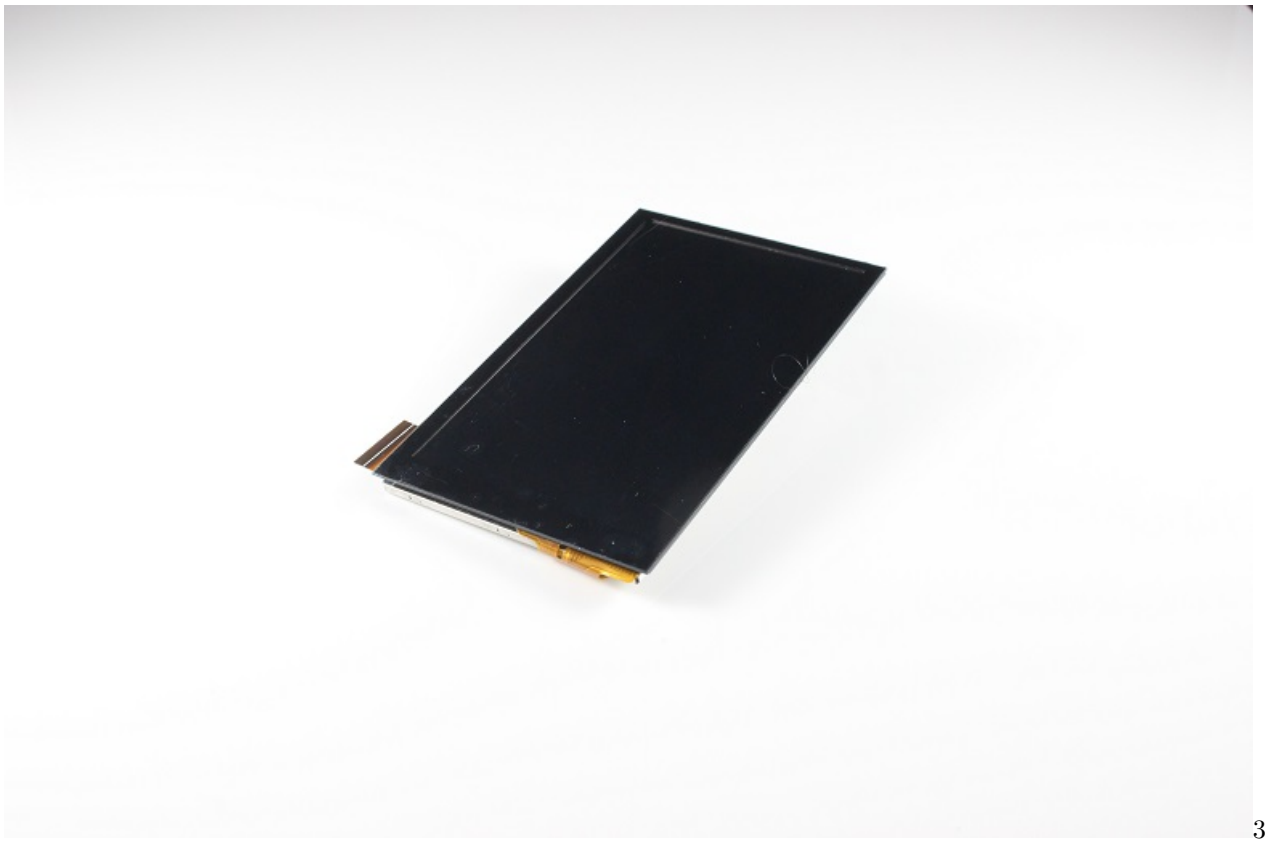
在讨论怎么写 LCD 驱动之前，我们先大概了解一下嵌入式常用 LCD。只是概述一些跟驱动架构设计有关的概念。至于原理跟细节，在此不做深入讨论，会有专门文章介绍，或者参考网络文档。

32.1.1 TFT lcd

TFT LCD，也就是我们常说的彩屏。通常像素较高，例如常见的 2.8 寸，320X240 像素。4.0 寸的，像素 800X400。这些屏通常使用并口，也就是 8080 或 6800 接口（STM32 的 FSMC 接口）；或者是 RGB 接口，STM32F429 等部分较贵的芯片支持。其他例如手机上使用的有 MIPI 接口。也有一些支持 SPI 接口的，不过除非是比较小的屏幕，否则不建议使用 SPI 接口，速度慢，刷屏闪屏。玩 STM32 常用的 TFT lcd 屏幕驱动 IC 通常有：ILI9341/ILI9325 等。2.8 寸 tft lcd



2.8

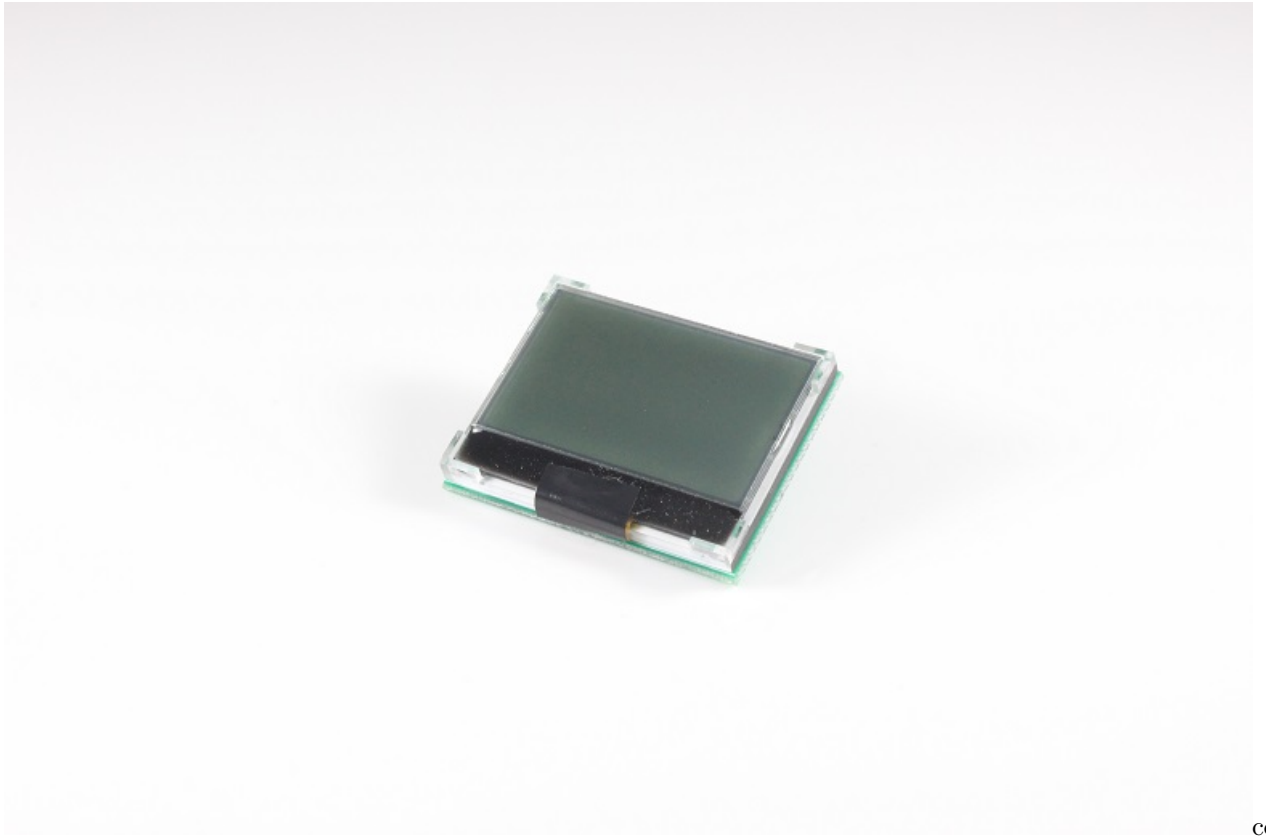


4 寸 IPS 3.0

32.1.2 COG lcd

很多人可能不知道 COG LCD 是什么，我觉得跟现在开发板销售方向有关系，大家都出大屏，玩酷炫界面。使用单片机的产品，COG LCD 其实占比非常大。所谓的 COG LCD，

> COG 是 Chip On Glass 的缩写，就是驱动芯片直接绑定在玻璃上，透明的。



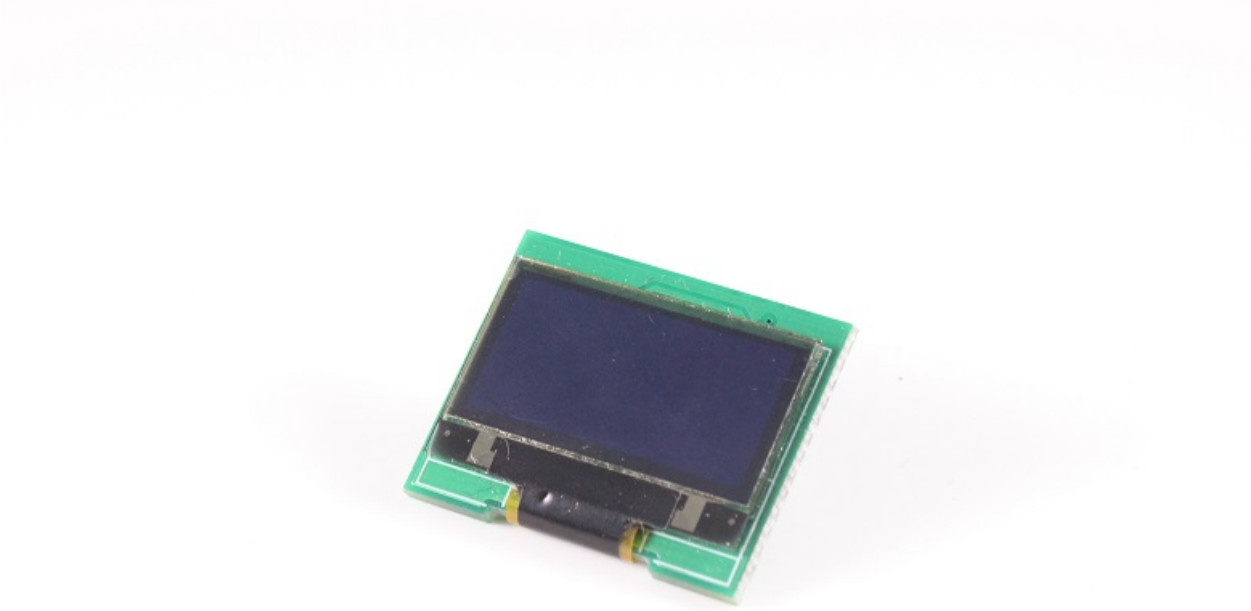
实物像下图：

cog

这种 LCD 通常像素不高，常用的有 128X64，128X32。一般只支持黑白显示，也有灰度屏，我没怎么用过。接口通常是 SPI，I2C。也有号称支持 8 位并口的，不过基本不会用，3 根 IO 能解决的问题，没必要用 8 根吧？常用的驱动 IC：STR7565。

32.1.3 OLED lcd

买过开发板的应该基本用过。新技术，大家都感觉高档，在手环等产品常用。OLED 目前屏幕较小，大一点的都很贵。在控制上跟 COG LCD 类似，区别是两者的显示方式不一样。从我们程序角度来看，最大的差别就是，OLED LCD，不用控制背光。 。 。 。 。实物如下图，

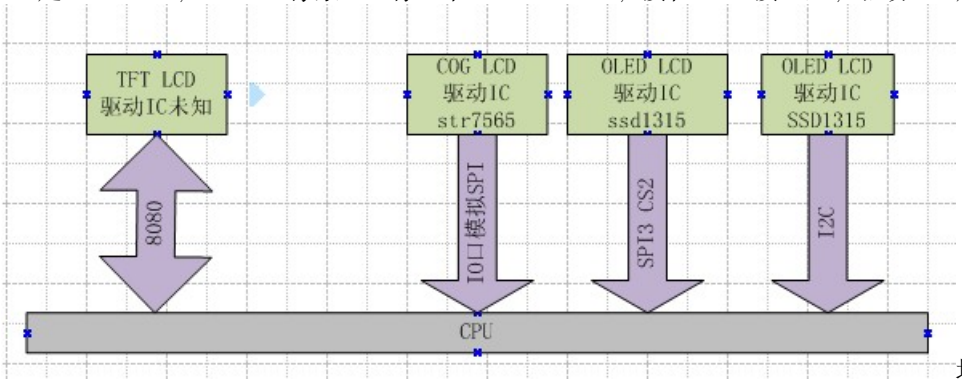


oled

常见的是 SPI 跟 I2C 接口。常见驱动 IC：SSD1615。

32.2 硬件场景

接下来的讨论，都基于以下硬件信息：1 有一个 TFT 屏幕，接在 FSMC 接口，什么型号屏幕？不知道。2 有一个 COG LCD，接在几根 IO 口上，驱动 IC 是 STR7565，128X32 像素。3 有一个 OLED LCD，接在硬件 SPI3 和几根 IO 口上，驱动 IC 是 SSD1315，128x64 像素。4 有一个 OLED LCD，接在 I2C 接口上，驱动 IC 是



SSD1315，128x64 像素场景

32.3 预备知识

在进入讨论之前，我们先大概说一下下面几个概念，对于这些概念，如果你想深入了解，请 GOOGLE。

32.3.1 面向对象

面向对象，是编程界的一个概念，常在 C++ 中出现。什么叫面向对象呢？编程有两种要素：程序（方法），数据（属性）。例如：一个 LED，我们可以点亮或者熄灭它，这叫方法。LED 什么状态？亮还是灭？这就是属性。我们通常这样编程：

```
u8 ledsta = 0;

void ledset(u8 sta)
{

}
```

这样的编程有一个问题，假如我们有 10 个这样的 LED，怎么写？最简单粗暴的方法就是用 10 个函数。不过只要是做过一点程序的人都会觉得这个方法太蠢。至少，大家都会在函数增加一个参数，然后在函数里面用 if-else 或者 switch 语句处理不同 LED。其实，更高级一点的是用面向对象，将方法和属性分开。我们可将每一个 LED 封装为一个对象。可以这样做：

```
/*
定义一个结构体，将 LED 这个对象的属性跟方法封装。
这个结构体就是一个对象。
但是这个不是一个真实的存在，而是一个对象的抽象。
*/
typedef struct
{
        u8 sta;
        void (*setsta)(u8 sta);
}LedObj;

/*
        声明一个 LED 对象，名称叫做 LED1，
        并且实现它的方法 drv_led1_setsta
*/
void drv_led1_setsta(u8 sta)
{
```

(continues on next page)

(continued from previous page)

```
}

LedObj LED1={
                .sta = 0,
                .setsta = drv_led1_setsta,
        };

/*
        声明一个 LED 对象，名称叫做 LED2,
        并且实现它的方法 drv_led2_setsta
*/
void drv_led2_setsta(u8 sta)
{

}

LedObj LED2={
                .sta = 0,
                .setsta = drv_led2_setsta,
        };

/*
        操作 LED 的函数，参数指定哪个 led
*/
void ledset(LedObj *led, u8 sta)
{
        led->setsta(sta);
}
```

抛砖引玉，很多地方不正确，但是不想展开，大家自己搜索资料学习。

是的，在 C 语言中，**实现面向对象的手段就是结构体的使用**。上面的代码，对于 API 来说，就很友好了。操作所有 LED，使用同一个接口，只需告诉接口哪个 LED。大家想想，前面说的 LCD 硬件场景。4 个 LCD，如果不面向对象，显示汉字的接口是不是要实现 4 个？每个屏幕一个？

32.3.2 驱动与设备分离

如果要深入了解驱动与设备分离，请看 LINUX 驱动的书籍。

什么是设备？设备就是属性，就是参数，就是驱动程序要用到的数据和硬件接口。那么驱动就是控制这些数据和接口的**代码过程**。通常来说，如果 LCD 的驱动 IC 相同，就用相同的驱动。有些不同的 IC 也可以用相同的，例如 SSD1315 跟 STR7565，除了初始化，其他都可以用相同的驱动。例如一个 COG lcd:

驱动 IC 是 STR7565 128*64 像素用 SPI3 背光用 PF5 命令线用 PF4 复位脚用 PF3

上面所有的信息综合，就是一个设备。驱动就是 STR7565 的驱动。

为什么要驱动跟设备分离，因为要解决下面问题：

有一个新产品，收银设备，系统有两个 LCD，一个叫做主显示，收银员用，一个叫顾显，顾客看金额使用。怎么办？这些例程代码要怎么改才能支持两个屏幕？复制一套然后改函数名称？这样确实能完成任务，只不过程序从此就进入恶性循环了。

这个问题，两个设备用同一套程序控制才是最好的解决办法。

驱动与设备分离的手段：

在驱动程序接口中增加设备参数，驱动用到的所有资源从设备参数传入。

驱动如何跟设备绑定呢？通过设备的驱动 IC 型号。

32.3.3 模块化

模块化就是将一段程序封装，提供一套相同接口，给不同的驱动使用。不模块化就是，在不同的驱动中都实现这段程序。例如字库处理，在显示汉字的时候，我们要找点阵，在打印机打印汉字的时候，我们也要找点阵，你觉得程序要怎么写？把点阵处理做成一个模块，就是模块化。

非模块化的典型特征就是一根线串到底，没有任何层次感。

32.4 LCD 到底是什么

前面我们说了面向对象，想要对 LCD 进行抽象，得出一个对象，就需要知道 LCD 到底是什么。我们问自己下面几个问题：1 LCD 能做什么？2 要 LCD 做什么？3 谁想要 LCD 做什么？

刚刚接触嵌入式的朋友可能不是很了解，可能会想不通。我们模拟一下 LCD 的功能操作数据流。APP 想要显示一个汉字。

1 首先，需要一个显示汉字的接口，APP 调用这个接口就可以显示汉字了。假设接口叫做 lcd_display_hz。2 汉字从哪来？从点阵字库来，所以在 lcd_display_hz 函数内就要调用一个叫做 find_font 的函数获取点阵。3 获取点阵后要将点阵显示到 LCD 上，那么我们调用一个 ILL9341_dis 的接口，将点阵刷新到驱动 IC 型号为 ILI9341 的 LCD 上。4 ILI9341_dis 怎么将点阵显示上去？调用一个 8080_WRITE 的接口。

好的，这个就是大概过程，我们从这个过程去抽象 LCD 功能接口。汉字跟 LCD 对象有关吗？无关。在 LCD 眼里，无论汉字还是图片，都是一个个点。那么前面问题的答案就是：

1 LCD 可以一个点一个点显示内容。2 要 LCD 显示汉字或图片——转化后就是显示一堆点 3 APP 想要 LCD 显示图片或文字。

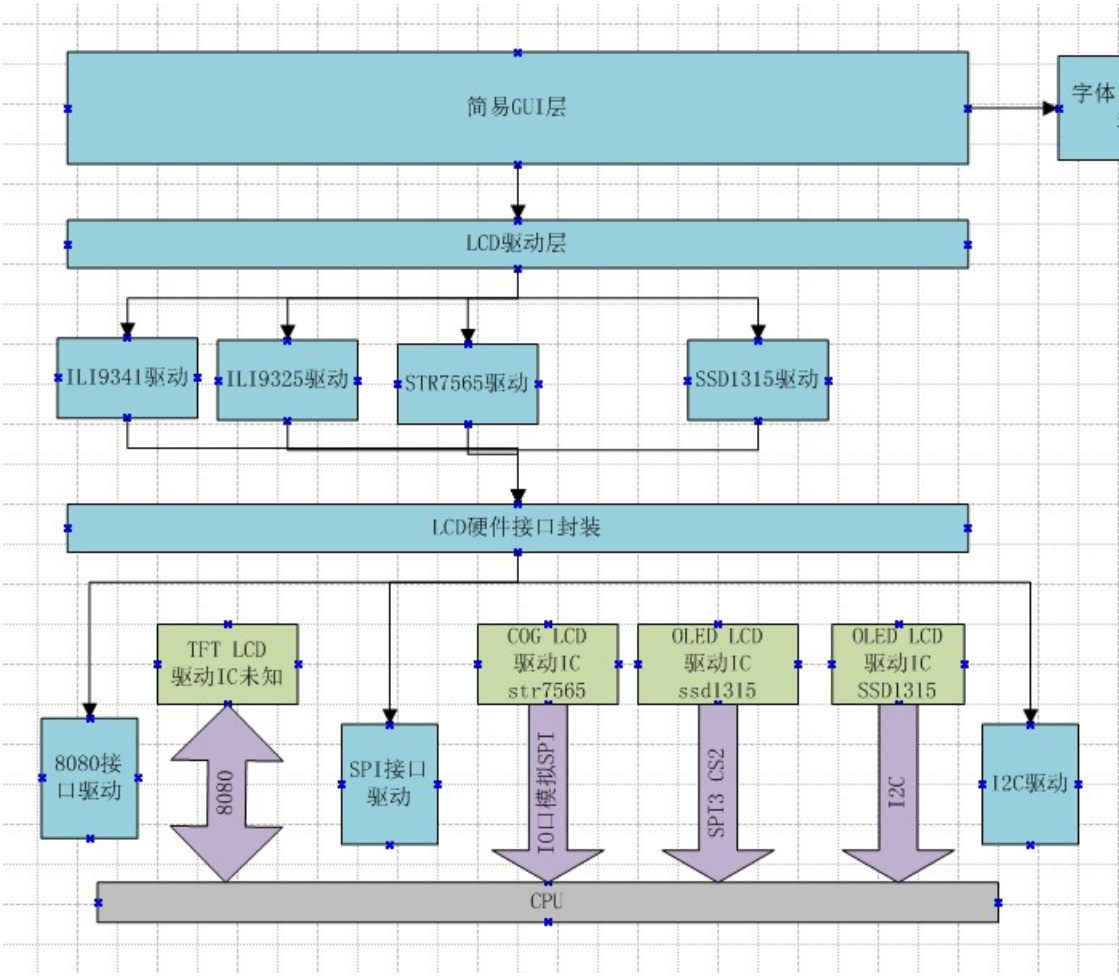
结论就是：所有 LCD 对象的功能就是显示点。那么驱动只要提供显示点的接口就可以了，显示一个点，显示一片点。抽象接口如下：

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

        s32 (*init)(DevLcd *lcd);
        s32 (*draw_point)(DevLcd *lcd, u16 x, u16 y, u16 color);
        s32 (*color_fill)(DevLcd *lcd, u16 sx,u16 ex,u16 sy,u16 ey, u16 color);
        s32 (*fill)(DevLcd *lcd, u16 sx,u16 ex,u16 sy,u16 ey,u16 *color);
        s32 (*onoff)(DevLcd *lcd, u8 sta);
        s32 (*prepare_display)(DevLcd *lcd, u16 sx, u16 ex, u16 sy, u16 ey);
        void (*set_dir)(DevLcd *lcd, u8 scan_dir);
        void (*backlight)(DevLcd *lcd, u8 sta);
}_lcd_drv;
```

上面的接口，也就是对应的驱动，包含了一个驱动 id 号。

1 id，驱动型号 2 初始化 3 画点 4 将一片区域的点显示某种颜色 5 将一片区域的点显示某些颜色 6 显示开关 7 准备刷新区域（主要彩屏直接 DMA 刷屏使用）8 设置扫描方向 9 背光控制

32.5 LCD 驱动框架



我们设计了如下的驱动框架
动框架从上到下分别是：

1. GUI 层：如果不使用 GUI，普通的划线，画圆等，也算 GUI。
2. LCD 驱动层：主要是封装下一层驱动 IC 层的接口，以便 GUI 层用一套接口操作多种 LCD。
3. 驱动 IC 驱动层，实现不同的 LCD 控制，对上提供同样的接口（前面说的 _lcd_drv 结构体）
4. 对不同的硬件接口封装，以便一种驱动使用多种接口，例如 SSD1315 驱动可以用 I2C，也可以用 SPI。
5. 接口层，例如 SPI 驱动，其实不算 LCD 功能范畴。

32.6 代码分析

代码分四层：

1. GUI 和 LCD 驱动层，代码在下面两个文件 dev_lcd.c dev_lcd.h
2. 显示驱动 IC 层 dev_str7565.c & dev_str7565.h dev_ILI9341.c & dev_ILI9341.h

3. LCD 接口封装层 dev_lcdbus.c&dev_lcdbus.h
4. 接口层 mcu_spi.c & mcu_spi.h mcu_i2c.c & mcu_i2c.h stm324xg_eval_fsmc_sram.c & stm324xg_eval_fsmc_sram.h

32.6.1 GUI 和 LCD 层

这层主要有 3 个功能 **1 设备管理**首先定义了一堆 LCD 参数结构体，结构体包含 ID，像素。并且把这些结构体组合到一个 list 数组内。

```
/*
        各种 LCD 的规格参数
*/
_lcd_pra LCD_IIL9341 ={
                .id          = 0x9341,
                .width = 240,        //LCD 宽度
                .height = 320,        //LCD 高度
};
...
/* 各种 LCD 列表 */
_lcd_pra *LcdPraList[5]=
                        {
                                &LCD_IIL9341,
                                &LCD_IIL9325,
                                &LCD_R61408,
                                &LCD_Cog12864,
                                &LCD_Oled12864,
                        };
```

然后定义了所有驱动 list 数组，数组内容就是驱动，在对应的驱动文件内实现。

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

定义了设备树，即是定义了系统有多少个 LCD，接在哪个接口，什么驱动 IC。如果是一个完整系统，可以做成一个类似 LINUX 的设备树。

```
/*
        设备树定义
        指明系统有多少个 LCD 设备，挂在哪个 LCD 总线上。
*/
#define DEV_LCD_C 4//系统存在 3 个 LCD 设备
LcdObj LcdObjList[DEV_LCD_C]=
{
        {"i2coledlcd",  LCD_BUS_I2C,  0X1315},
        {"vspioledlcd", LCD_BUS_VSPI, 0X1315},
        {"spicoglcd",   LCD_BUS_SPI,  0X7565},
        {"tftlcd",      LCD_BUS_8080, NULL},
};
```

2 接口封装

```
void dev_lcd_setdir(DevLcd *obj, u8 dir, u8 scan_dir)
s32 dev_lcd_init(void)
DevLcd *dev_lcd_open(char *name)
s32 dev_lcd_close(DevLcd *dev)
s32 dev_lcd_drawpoint(DevLcd *lcd, u16 x, u16 y, u16 color)
s32 dev_lcd_prepare_display(DevLcd *lcd, u16 sx, u16 ex, u16 sy, u16 ey)
s32 dev_lcd_display_onoff(DevLcd *lcd, u8 sta)
s32 dev_lcd_fill(DevLcd *lcd, u16 sx,u16 ex,u16 sy,u16 ey,u16 *color)
s32 dev_lcd_color_fill(DevLcd *lcd, u16 sx,u16 ex,u16 sy,u16 ey,u16 color)
s32 dev_lcd_backlight(DevLcd *lcd, u8 sta)
```

大部分接口都是对驱动 IC 接口的二次封装。有区别的是初始化和打开接口。初始化，就是根据前面定义的设备树，寻找对应驱动，找到对应设备参数，并完成设备初始化。打开函数，根据传入的设备名称，查找设备，找到后返回设备句柄，后续的操作全部需要这个设备句柄。

3 简易 GUI 层主要是一些简单的显示字符函数。

```
s32 dev_lcd_put_string(DevLcd *lcd, FontType font, int x, int y, char *s, unsigned␣
↪colidx)
```

其他划线画圆的函数目前只是测试，后续会完善。

32.6.2 驱动 IC 层

驱动 IC 层主要完成不同的显示 IC 驱动，也就是实现下面这个结构体内的所有函数，例如 STR7565 驱动：

```
_lcd_drv CogLcdST7565Drv = {
        .id = 0X7565,

        .init = drv_ST7565_init,
        .draw_point = drv_ST7565_drawpoint,
        .color_fill = drv_ST7565_color_fill,
        .fill = drv_ST7565_fill,
        .onoff = drv_ST7565_display_onoff,
        .prepare_display = drv_ST7565_prepare_display,
        .set_dir = drv_ST7565_scan_dir,
        .backlight = drv_ST7565_lcd_bl
        };
```

32.6.3 接口封装层

为了上一层驱动 IC 层能用于不同的接口，需要对接口进行统一抽象封装。抽象接口如下：

```
/*
        LCD 接口定义
*/
typedef struct
{
        char * name;

        s32 (*init)(void);
        s32 (*open)(void);
        s32 (*close)(void);
        s32 (*writedata)(u8 *data, u16 len);
        s32 (*writecmd)(u8 cmd);
        s32 (*bl)(u8 sta);
}_lcd_bus;
```

接口包含：初始化，打开，关闭，写数据，写命令，背光控制。**当前没有将 *8080* 封装，接口抽象没有读数据**

实现 3 个接口封装：SPI、VSPI、I2C：

```
_lcd_bus BusSerialLcdSpi={
                .name = "BusSerivaLcdSpi",
                .init =bus_seriallcd_spi_init,
                .open =bus_seriallcd_spi_open,
                .close =bus_seriallcd_spi_close,
```

(continues on next page)

(continued from previous page)

```
                .writedata =bus_seriallcd_spi_write_data,
                .writecmd =bus_seriallcd_spi_write_cmd,
                .bl =bus_seriallcd_spi_bl,
};

_lcd_bus BusSerialLcdVSpi={
                .name = "BusSerivaLcdVSpi",
                .init =bus_seriallcd_vspi_init,
                .open =bus_seriallcd_vspi_open,
                .close =bus_seriallcd_vspi_close,
                .writedata =bus_seriallcd_vspi_write_data,
                .writecmd =bus_seriallcd_vspi_write_cmd,
                .bl =bus_seriallcd_vspi_bl,
};

_lcd_bus BusSerialLcdVI2C={
                .name = "BusSerivaLcdVI2C",
                .init =bus_seriallcd_vi2c_init,
                .open =bus_seriallcd_vi2c_open,
                .close =bus_seriallcd_vi2c_close,
                .writedata =bus_seriallcd_vi2c_write_data,
                .writecmd =bus_seriallcd_vi2c_write_cmd,
                .bl =bus_seriallcd_vi2c_bl,
};
```

32.6.4 接口层

8080 层比较简单，用的是官方接口。SPI、VSPI、I2C 接口参考其他章节。

32.6.5 总体流程

前面说的几个模块时如何联系在一起的呢？请看下面结构体：

```
/*
        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 _strDevLcd
```

(continues on next page)

(continued from previous page)

```
{
        s32 gd;//句柄，控制是否可以打开

        LcdObj          *dev;
        /* LCD 参数，固定，不可变 */
        _lcd_pra *pra;

        /* 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
};
```

每一个设备都会有一个这样的机构体，这个结构体在初始化 LCD 时初始化。

- 成员 dev 指向设备树，从这个成员可以知道设备名称，挂在哪个 LCD 总线，设备 ID。

```
typedef struct
{
        char *name;//设备名字
        LcdBusType bus;//挂在那条 LCD 总线上
        u16 id;
}LcdObj;
```

其中 LcdBusType 定义如下：

```
/*
系统总共有三种 LCD 总线
*/
typedef enum{
        LCD_BUS_NULL = 0,
        LCD_BUS_SPI,
        LCD_BUS_VSPI,
        LCD_BUS_I2C,//OLED 使用，只要两根线，背光也不需要控制，复位也不需要
        LCD_BUS_8080,
```

(continues on next page)

(continued from previous page)

```
        LCD_BUS_MAX
}LcdBusType;
```

-成员 pra 指向 LCD 参数，可以知道 LCD 的规格。

```
typedef struct
{
        u16 id;
        u16 width;          //LCD 宽度   竖屏
        u16 height;          //LCD 高度     竖屏
}_lcd_pra;
```

-成员 drv 指向驱动，所有操作通过 drv 实现。

```
typedef struct
{
        u16 id;

        s32 (*init)(DevLcd *lcd);

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

        s32 (*prepare_display)(DevLcd *lcd, u16 sx, u16 ex, u16 sy, u16 ey);

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

- 成员 dir、scandir、width、height 是驱动要使用的通用变量。因为每个 LCD 都有一个结构体，一套驱动程序就能控制多个设备而互不干扰。
- 成员 pri 是一个私有指针，某些驱动可能需要有些比较特殊的变量，就全部用这个指针记录，通常这个指针指向一个结构体，结构体由驱动定义，并且在设备初始化时申请变量空间。目前主要用于 COG LCD 跟 OLED LCD 显示缓存。

整个 LCD 驱动，就通过这个结构体组合在一起。具体如下：1 初始化，根据设备树 LcdObjList，找到驱动跟参数，然后初始化结构体 DevLcdList。2 要使用 LCD 前，调用 dev_lcd_open 函数。打开成功就返回一个 DevLcd 结构体指针。3 调用 dev_lcd_drawpoint 函数显示一个点

```
s32 dev_lcd_drawpoint(DevLcd *lcd, u16 x, u16 y, u16 color)
{
        if(lcd == NULL)
                return -1;

        return lcd->drv->draw_point(lcd, x-1, y-1, color);
}
```

第一个参数 Lcd 就是打开 LCD 是得到的指针，函数通过指针调用对应驱动的 draw_point 函数。加入是 str7565 驱动，就是执行下面函数

```
static s32 drv_ST7565_drawpoint(DevLcd *lcd, u16 x, u16 y, u16 color)
```

在这个函数前面有两行代码，这两行代码的作用是根据传入的总线类型，找到对应 LCD 总线的操作函数。

```
        _lcd_bus *LcdBusDrv;
        LcdBusDrv = dev_lcdbus_find(lcd->dev->bus);
```

加入 lcd->dev->bus 是 LCD_BUS_VSPI，LcdBusDrv 将指向 BusSerialLcdSpi，那么最后的几句代码，就是执行 BusSerialLcdSpi 内的函数。

```
/* 效率不高 */
        LcdBusDrv->open();
    LcdBusDrv->writecmd (0xb0 + page );
    LcdBusDrv->writecmd (((colum>>4)&0x0f)+0x10);
    LcdBusDrv->writecmd (colum&0x0f);
    LcdBusDrv->writedata( &(gram->gram[page][colum]), 1);
        LcdBusDrv->close();
```

BusSerialLcdSpi 内的函数将直接操作对应 SPI 的接口操作硬件。

32.7 用法和好处

- 好处 1

请看测试程序

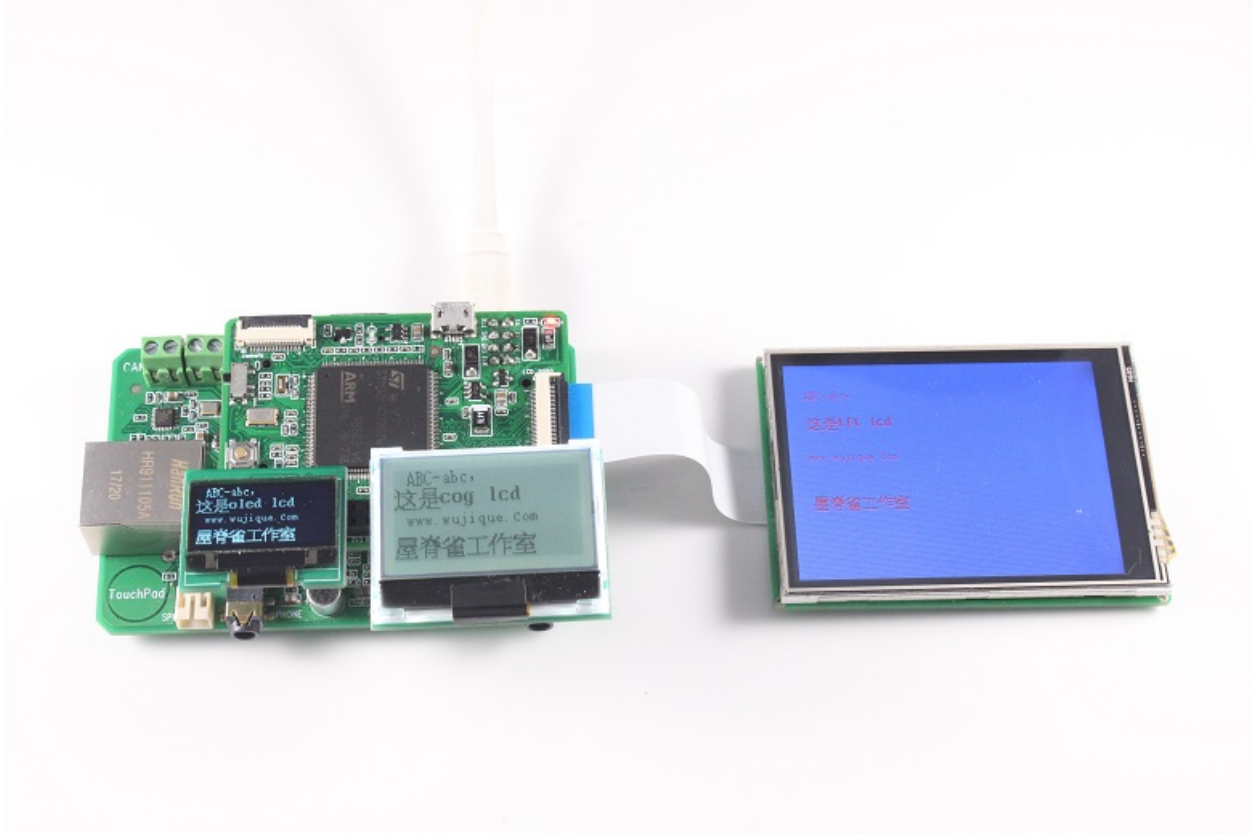
```
void dev_lcd_test(void)
{
        DevLcd *LcdCog;
        DevLcd *LcdOled;
        DevLcd *LcdTft;
```

(continues on next page)

(continued from previous page)

```
        /*  打开三个设备 */
        LcdCog = dev_lcd_open("coglcd");
        if(LcdCog==NULL)
                uart_printf("open cog lcd err\r\n");

        LcdOled = dev_lcd_open("oledlcd");
        if(LcdOled==NULL)
                uart_printf("open oled lcd err\r\n");

        LcdTft = dev_lcd_open("tftlcd");
        if(LcdTft==NULL)
                uart_printf("open tft lcd err\r\n");

        /* 打开背光 */
        dev_lcd_backlight(LcdCog, 1);
        dev_lcd_backlight(LcdOled, 1);
        dev_lcd_backlight(LcdTft, 1);

        dev_lcd_put_string(LcdOled, FONT_SONGTI_1212, 10,1, "ABC-abc, ", BLACK);
        dev_lcd_put_string(LcdOled, FONT_SIYUAN_1616, 1, 13, "这是 oled lcd", BLACK);
        dev_lcd_put_string(LcdOled, FONT_SONGTI_1212, 10,30, "www.wujique.com", BLACK);
        dev_lcd_put_string(LcdOled, FONT_SIYUAN_1616, 1, 47, "屋脊雀工作室", BLACK);

        dev_lcd_put_string(LcdCog, FONT_SONGTI_1212, 10,1, "ABC-abc, ", BLACK);
        dev_lcd_put_string(LcdCog, FONT_SIYUAN_1616, 1, 13, "这是 cog lcd", BLACK);
        dev_lcd_put_string(LcdCog, FONT_SONGTI_1212, 10,30, "www.wujique.com", BLACK);
        dev_lcd_put_string(LcdCog, FONT_SIYUAN_1616, 1, 47, "屋脊雀工作室", BLACK);

        dev_lcd_put_string(LcdTft, FONT_SONGTI_1212, 20,30, "ABC-abc, ", RED);
        dev_lcd_put_string(LcdTft, FONT_SIYUAN_1616, 20,60, "这是 tft lcd", RED);
        dev_lcd_put_string(LcdTft, FONT_SONGTI_1212, 20,100, "www.wujique.com", RED);
        dev_lcd_put_string(LcdTft, FONT_SIYUAN_1616, 20,150, "屋脊雀工作室", RED);

        while(1);
}
```

使用一个函数 dev_lcd_open，可以打开 3 个 LCD，获取 LCD 设备。然后调用 dev_lcd_put_string 就可以在不同的 LCD 上显示。其他所有的 gui 操作接口都只有一个。这样的设计对于 APP 层来说，就很友好。显示效果



显示效果

- 好处 2

现在的设备树是这样定义的

```
LcdObj LcdObjList[DEV_LCD_C]=
{
        {"oledlcd", LCD_BUS_VSPI, 0X1315},
        {"coglcd", LCD_BUS_SPI,  0X7565},
        {"tftlcd", LCD_BUS_8080, NULL},
};
```

某天，oled lcd 要接到 SPI 上，只需要将设备树数组里面的参数改一下，就可以了，当然，在一个接口上不能接两个设备。

```
LcdObj LcdObjList[DEV_LCD_C]=
{
        {"oledlcd", LCD_BUS_SPI, 0X1315},
        {"tftlcd", LCD_BUS_8080, NULL},
};
```

32.8 字库

暂时不做细说，例程的字库放在 SD 卡中，各位移植的时候根据需要修改。具体参考 font.c

32.9 总结

本章节之后，一个 LCD 驱动的雏形，基本完成了。也基本上将 LCD 驱动框架的实现思想说明了。但是还有很多需要改进优化的地方。最终的软件架构，请参考 github 仓库最新代码。或参考《产品手册（软件）.pdf》

32.10 end

CHAPTER 33

汉字点阵字库模块

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

前面几个小节，我们已经点亮了多种 LCD，实现英文显示功能。现在，我们开始添加汉字点阵功能。汉字显示原理和英文一样，都是在 LCD 上描点。区别是，英文只有 26 个字母，算上大小写和其他字符，也就 128 个字符，就是 ASC 编码范围。字符如何描点，只要在程序内保存字符点阵数据，也就是 128 个点阵数据。汉字就不一样了，汉字有几千几万个，但是也没有办法，只能一个汉字做一个点阵数据，所有的点阵数据组成一个点阵字库。

33.1 汉字编码标准

33.1.1 编码标准

在讨论点阵字库前，先要了解字符编码。英文的字符编码，就是 ASC 字符编码。

> ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）是基于拉丁字母的一套电脑编码系统，主要用于显示现代英语和其他西欧语言。它是现今最通用的单字节编码系统，并等同于国际标准 ISO/IEC 646。

ASCII表

（ American Standard Code for Information Interchange 美国标准信息交换代码 ）

高四位 / 低四位		ASCII控制字符 0000 0						ASCII控制字符 0001 1						ASCII打印字符 0010 2		ASCII打印字符 0011 3		ASCII打印字符 0100 4		ASCII打印字符 0101 5		ASCII打印字符 0110 6		ASCII打印字符 0111 7	
		十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符
0000	0	0		^@	NUL	\0	空字符	16	►	^P	DLE		数据链路转义	32		48	0	64	@	80	P	96	`	112	p
0001	1	1	☺	^A	SOH		标题开始	17	◄	^Q	DC1		设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q
0010	2	2	☻	^B	STX		正文开始	18	↕	^R	DC2		设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r
0011	3	3	♥	^C	ETX		正文结束	19	‼	^S	DC3		设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s
0100	4	4	♦	^D	EOT		传输结束	20	¶	^T	DC4		设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t
0101	5	5	♣	^E	ENQ		查询	21	§	^U	NAK		否定应答	37	%	53	5	69	E	85	U	101	e	117	u
0110	6	6	♠	^F	ACK		肯定应答	22	▬	^V	SYN		同步空闲	38	&	54	6	70	F	86	V	102	f	118	v
0111	7	7	•	^G	BEL	\a	响铃	23	↨	^W	ETB		传输块结束	39	'	55	7	71	G	87	W	103	g	119	w
1000	8	8	◘	^H	BS	\b	退格	24	↑	^X	CAN		取消	40	(	56	8	72	H	88	X	104	h	120	x
1001	9	9	○	^I	HT	\t	横向制表	25	↓	^Y	EM		介质结束	41	)	57	9	73	I	89	Y	105	i	121	y
1010	A	10	◙	^J	LF	\n	换行	26	→	^Z	SUB		替代	42	*	58	:	74	J	90	Z	106	j	122	z
1011	B	11	♂	^K	VT	\v	纵向制表	27	←	^[	ESC	\e	溢出	43	+	59	;	75	K	91	[	107	k	123	{
1100	C	12	♀	^L	FF	\f	换页	28	∟	^\	FS		文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|
1101	D	13	♪	^M	CR	\r	回车	29	↔	^]	GS		组分隔符	45	-	61	=	77	M	93	]	109	m	125	}
1110	E	14	♫	^N	SO		移出	30	▲	^^	RS		记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~
1111	F	15	☼	^O	SI		移入	31	▼	^-	US		单元分隔符	47	/	63	?	79	O	95	_	111	o	127	⌂

^Backs 代码：

注：表中的ASCII字符可以用“Alt + 小键盘上的数字键”方法输入。 2013/08/08

下图就是 ASC 码表码表英文字符少，只需要一个字节编码。汉字那么多字符，通常用 2 字节编码，部分字符还用了 4 字节编码（这些字一般人都不认识）。

常用的汉字内码标准有 GB2312、GBK、GB18030、BIG5、UNICODE。

- 大陆编码

  GB= 国标 GB2312 包含 7000 多字符，GBK 包含 2 万 1 左右字符，GB18030 包含 2 万 7 千左右字符。3 者向下兼容，也即是说，GBK 是 GB2312 的扩展，GB18030 则是 GBK 的扩展。

在标准中，还有一个区位码和内码的概念。区位码，表示这个汉字在编码标准中的位置，区位码从 1 开始编码。但是，由于国际通用标准 ASC 编码时 0 到 128，因此在保存文件时，汉字不能用，所以，保存时，需要偏移，偏移后的编码叫机内码，俗称内码。在编程中，我们都是用内码，因为保存文件都是用内码的。在

取点阵时，就需要根据内码算出区位码，根据区位码到点阵字库取点阵数据。

更多细节请参考《汉字编码字符 gb18030.pdf》。下图是国标编码标准

表 1 码位范围分配图

字节数	码位空间				码位数目
单字节	0x00~0x80				129 个码位
双字节	第一字节		第二字节		23940 个码位
	0x81 ~ 0xFE		0x40 ~ 0x7E， 0x80 ~ 0xFE		
四字节	第一字节	第二字节	第三字节	第四字节	1587600 个码位。
	0x81~ 0xFE	0x30~ 0x39	0x81~ 0xFE	0x30~0x39	

ASC

码表

- 台湾编码

BIG5 是台湾汉字编码标准，主要是繁体。

- 国际编码

UNICODE 编码。

33.2 字库概念

33.2.1 点阵字库

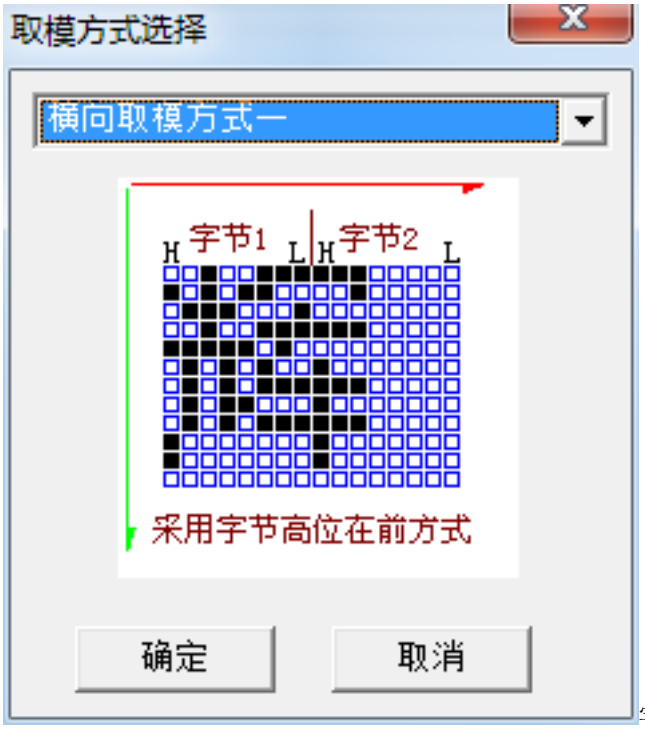
点阵字库，也就是包含很多点阵字体的一个文件。点阵字体也叫位图字体，其中每个字形都以一组二维像素信息表示。点阵字体优点是显示速度快，不像矢量字体需要计算；其最大的缺点是不能放大，一旦放大后就会发现文字边缘的锯齿。

- 大小

汉字最小一般是 12x12，还有 16x16，20x20，24x24，32x32，48x48。一个 16x16 的 GB18030 字库，大概 750K。

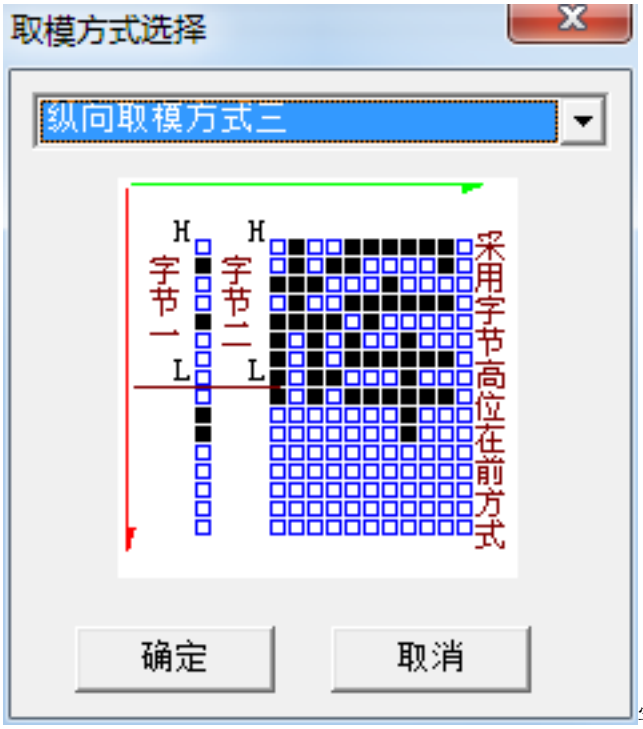
- 取模方式

大方向有两种：纵向取模、横向取模。字节顺序有多种。两者配合起来就有很多种取模方

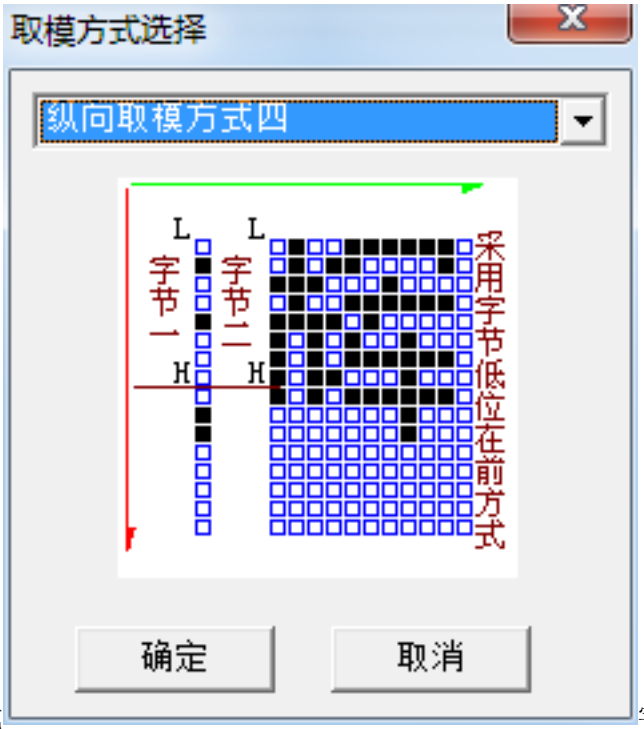


式了。下面是取模软件的 4 种方式:字体取模方式

字体取模方式



字体取模方式



字体取模方式

- 取点阵算法

不同的字库取点阵算法会有差异。请根据使用的点阵设计算法。我们自己生成的汉字点阵算法比较简单：

```
addr = (hcode-0x81)*190;
if(lcode<0x7f)
{
        addr = addr+lcode-0x40;
}
else
{
        addr = addr+lcode-0x41;
}
addr = addr*FontList[type]->size;
```

hcode 是汉字内码高字节，lcode 是汉字内码低字节，经过计算后，得到汉字在点阵中的偏移 addr。其中 size 是一个汉字点阵的字节数，一个 16*16 的汉字，通常是 32 字节。**本算法没有处理 4 字节内码的汉字**

33.2.2 矢量字库

矢量字库保存的是对每一个汉字的描述信息，比如一个笔划的起始、终止坐标，半径、弧度等等。在显示、打印这一类字库时，要经过一系列的数学运算才能输出结果，但是这一类字库保存的汉

字理论上可以被无限地放大，笔划轮廓仍然能保持圆滑，打印时使用的字库均为此类字库。

因为 LCD 是一个点一个点显示的，矢量字库通过处理后，最后还是一堆点阵数据，才能在 LCD 上显示。

33.3 制作点阵字库

33.3.1 字库版权

字体是有版权的，通过字模软件获取点阵，理论上是侵权的。大家可以看方正官网 http://ifont.foundertype.com/index/embedfont.html

> 5、从 Windwos 系统复制出的字体是否可以免费使用吗？从操作系统和各类网站上都可以找到字库。如果认为这些字体都可以任意的免费使用就错了。字库是知识产权产品，字库的著作权属于字库的设计开发者。根据相关法律，必须在获得版权人的授权后才可以使用。6、用工具从 TTF 字库中生成的点阵字库是否可以使用？网站上有许多“点阵字库生成工具”。这些工具的主要作用就是用 TTF 字库生成任意尺寸的点阵字库。点阵字库是 TTF 字库的一部分。许多 TTF 字库为了使小字是显示清晰，都内嵌了点阵字库。所以未经字库版权人授权从 TTF 字库中生成点阵字库在产品中使用的行为实际上是盗版行为。

当然，既然有开源软件，肯定就有开源字体为了长远着想，我们要找一款开源字体做点阵字库。关于字体授权，大家可以参考知乎的一个话题：https://www.zhihu.com/question/19727859

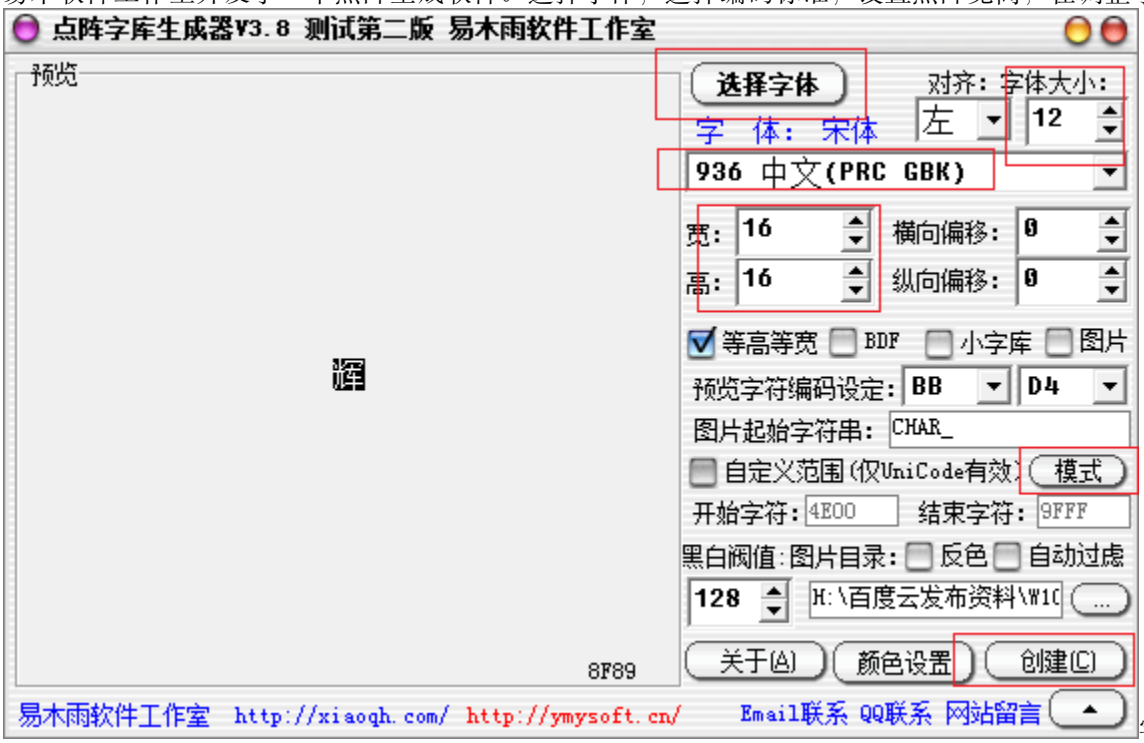
33.3.2 思源宋体

经过长达一点年半的研发，Adobe 联合 Google 于 2017 年 4 月 3 日发布了思源宋体（Source Han Serif，Google 称 Noto Serif CJK）。和思源黑体一样，思源宋体以“SIL 开放字体许可证”开源发行，且同样含简繁日韩四种汉字写法和七种粗细字重给出。

先在电脑上安装思源汉字字体，安装方法可以参考 https://baijiahao.baidu.com/s?id=1563997223669087&wfr=spider&for=pc

33.3.3 制作字库

易木软件工作室开发了一个点阵生成软件。选择字体，选择编码标准，设置点阵宽高，在调整字体大小。



字库软件

字体取模方式点击生成即可。我们用软件生成思源字体的点阵字库 shscn1212.DZK、shscn1616.DZK。

生成的字库可能不包含四字节内码汉字。

33.4 应用点阵字体

33.4.1 存储

字库较大，1212 的 561K，1616 的 748K。不可能保存在单片机内。只能保存在外部储存上。最好是保存在核心板上的 FLASH。但是目前我们还没有在 FLASH 上做好文件系统，就暂时放在 TF 卡上。通过 FATFS 文件系统读取字库，速度可能会慢一点，等 FLASH 管理处理好后，再搬移到核心板上的 FALSH 上。

前面调试 USB 时已经移植好文件系统，在 main 函数初始化硬件后挂载 SD 卡。

```
s32 fun_mount_sd(void)
```

SD 卡中包含以下字体文件：

songti1616.DZK songti1212.DZK shscn1616.DZK shscn1212.DZK

其中前面两个字体是宋体，有一定版权风险，仅供测试。后面两个是思源字体，开源。

ASC 字体内嵌在代码中，\Utilities\font 文件夹中的三个文件就是：

font_6x12.c、font_8x8.c、font_8x16.c

33.4.2 接口

提供两个接口，一个获取 asc 点阵，一个获取汉字点阵。

```
extern s32 font_get_asc(FontType type, char *ch, char *buf);
extern s32 font_get_hz(FontType type, char *ch, char *buf);
```

其中第一个参数 type 为枚举类型，后续增加新字体，可以增加。

```
/*
        字体类型定义
*/
typedef enum{
        FONT_SONGTI_1616 = 0,//1616 字体，对应的 ASC 则是 8*16
        FONT_SONGTI_1212,        //1212 字体，对应的 ASC 是 6*12
        FONT_SIYUAN_1616,
        FONT_SIYUAN_1212,
        FONT_LIST_MAX
}FontType;
```

在 lcd 中间层只增加了一个函数，最主要也是这个函数。

```
s32 dev_lcd_put_string(DevLcd *lcd, FontType font, int x, int y, char *s, unsigned␣
↪colidx)
```

汉字功能就添加完成了。

33.4.3 使用

在代码中直接嵌入中文，文件保存中文用的就是内码，编译后就是一个内码字符串。

```
dev_lcd_put_string(LcdOled, FONT_SONGTI_1212, 10,1, "ABC-abc, ", BLACK);
dev_lcd_put_string(LcdOled, FONT_SIYUAN_1616, 1, 13, "这是 oled lcd", BLACK);
dev_lcd_put_string(LcdOled, FONT_SONGTI_1212, 10,30, "www.wujique.com", BLACK);
dev_lcd_put_string(LcdOled, FONT_SIYUAN_1616, 1, 47, "屋脊雀工作室", BLACK);
```

33.5 总结

如何实现多国语言?

33.6 end

CHAPTER 34

wav 文件播放

够用的硬件 能用的代码 实用的教程屋脊雀工作室编撰 -20190101 愿景：做一套能用的开源嵌入式驱动（非 LINUX）官网：www.wujique.com github: https://github.com/wujique/stm32f407 淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf 技术支持邮箱：code@wujique.com、github@wujique.com 资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg QQ 群：767214262

前面硬件调试时，已经完成 WM8978 调试，由于当时没有文件系统支撑，没做音乐播放。如今，FATFS 已经使用，就让我们开始完成 WAV 播放功能。

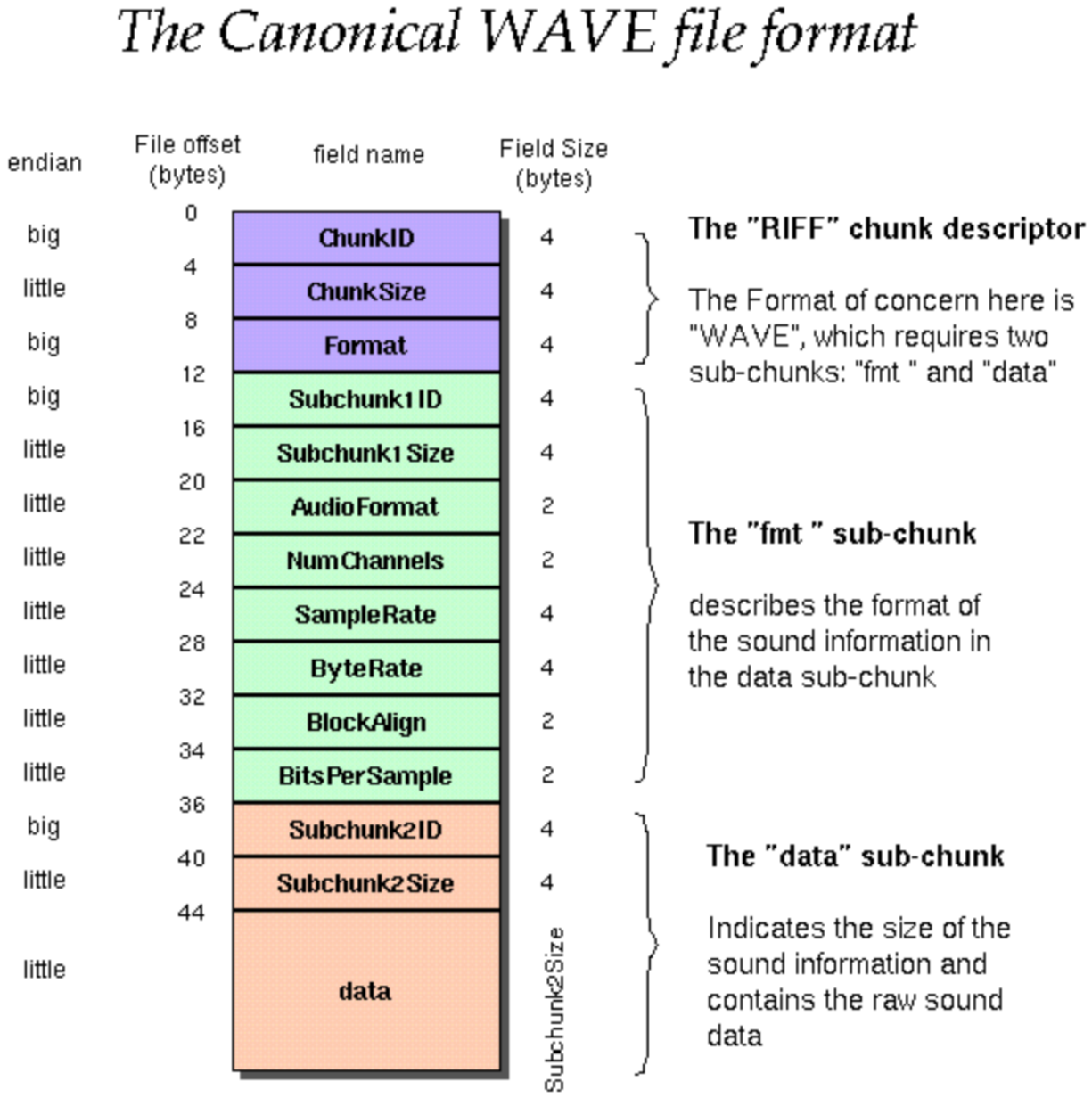
34.1 WAV 格式

前面调试 I2S 和 WM8978 时，其实已经提到过 WAV 文件，当时只是将 WAV 文件转换为数组直接播放，并没有深入了解 WAV 文件的格式。

> WAV 为微软公司 (Microsoft) 开发的一种声音文件格式，它符合 RIFF(Resource Interchange File Format) 文件规范，用于保存 Windows 平台的音频信息资源，被 Windows 平台及其应用程序所广泛支持，该格式也支持 MSADPCM，CCITT A LAW 等多种压缩运算法，支持多种音频数字，取样频率和声道，标准格式化的 WAV 文件和 CD 格式一样，也是 44.1K 的取样频率，16 位量化数字，因此在声音文件质量和 CD 相差无几!

在资料中包含了几个 *WAV* 格式说明文档

34.1.1 WAV 文件头



WAV

上图就是一个 WAV 文件的格式，在代码中我们如下面定义：

```
/*wav 文件结构 */
typedef struct _TWavHeader
{
        /*RIFF 块 */
    int rId;    //标志符（RIFF）   0x46464952
    int rLen;   //数据大小，包括数据头的大小和音频文件的大小   （文件总大小-8）
```

(continues on next page)

(continued from previous page)

```
    int wId;     //格式类型（"WAVE"）    0x45564157

    /*Format Chunk*/
    int fId;     //"fmt " 带一个空格 0X20746D66
    int fLen;    //Sizeof(WAVEFORMATEX)
    short wFormatTag;        //编码格式，包括 1 WAVE_FORMAT_PCM，WAVEFORMAT_ADPCM 等
↪
    short nChannels;         //声道数，单声道为 1，双声道为 2
    int nSamplesPerSec;    //采样频率
    int nAvgBytesPerSec;   //每秒的数据量
    short nBlockAlign;       //块对齐
    short wBitsPerSample;    //WAVE 文件的采样大小
    int dId;               //"data"      有可能是 FACT 块
    int wSampleLength;     //音频数据的大小
    /* 紧跟后面可能有一个 fact 块，跟压缩有关，如果没有，就是 data 块 */
}TWavHeader;
```

int 型变量占用 4 字节空间，short 型变量占用 2 字节空间。

我们可以通过 **winhex** 软件查看一个 wav 的格式，其中头部如下：

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
00000000	52	49	46	46	96	F8	48	00	57	41	56	45	66	6D	74	20	RIFF莱H.WAVEfmt
00000016	10	00	00	00	01	00	01	00	40	1F	00	00	80	3E	00	00	........@...€'..
00000032	02	00	10	00	64	61	74	61	00	B4	48	00	00	00	00	00	....data.?H.....
00000048	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000064	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000096	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000112	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000128	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000144	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000176	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000192	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

WAV

前面 4 个字节就是 RIFF 标志。**程序中写的是 0X46464952**，这是经常遇到的问题，读文件后保存到一个变量中，字节顺序是不一样的。

34.1.2 音频数据排布

WAV 文件有多种格式：单声道，双声道。8 位、16 位、24 位。文件中样点保存顺序是：左声道样点-右声道样点-左声道样点-右声道样点。单声道就只有一个声道的样点。每个样点的数据顺序是：低字节-高字节。如

果是 8 位，就只有一个字节，16 位则有 2 字节，24 位则有 3 字节。

34.2 中间层设计

音乐播放功能，不属于设备驱动，也不算应用程序。我们通常把它叫做中间件。所谓的中间件，就是将驱动设备接口进行二次封装，并完成一定的流程。向 APP 提供固定接口，屏蔽更多的底层细节。前面我们做 LCD 时，LCD 显示接口也可以算做中间件。

34.2.1 接口设计

我们在做程序，要时刻考虑上下游，特别是上游，也就是应用工程师开始应用调用底层。

- 驱动要提供什么给上层使用?
- 中间层要封装什么样的流程?

音乐播放就是一个中间层。需要提供什么接口呢？可以参考音乐播放器的功能：

播放停止暂停设置音量下上一首指定起始位置播放（快进快退）歌词

这些功能都是音乐播放的功能，但是那些是中间层实现？提供什么接口?

1. 播放，中间层肯定要实现，但是只是播放指定文件，也就是输入一个文件名作为参数。
2. 停止/暂停也是中间层要实现。
3. 音量，中间层也需要提供接口。
4. 上下一首呢？由于中间层不管理播放列表，这个功能就不该中间层实现，而是由 APP 层通过播放接口实现。
5. 指定起始位置播放？APP 层实现，但是中间层要提供接口：APP 通过接口获取文件播放时长，再提供一个接口设置播放位置。

本次测试我们只做了下面几个接口，其他请大家自行完善。

```
/**
 *@brief:       fun_sound_play
 *@details:     通过指定设备播放指定声音
 *@param[in]    char *name
                char *dev
 *@param[out]   无
 *@retval:
 */
int fun_sound_play(char *name, char *dev)
/**
 *@brief:       fun_sound_get_sta
```

(continues on next page)

(continued from previous page)

```
 *@details:    查询音乐播放状态
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
SOUND_State fun_sound_get_sta(void)
/**
 *@brief:      fun_sound_stop
 *@details:    停止音乐播放
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 fun_sound_stop(void)

/**
 *@brief:      fun_sound_pause
 *@details:    暂停播放
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 fun_sound_pause(void)

/**
 *@brief:      fun_sound_resume
 *@details:    恢复播放
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 fun_sound_resume(void)

/**
 *@brief:      fun_sound_setvol
 *@details:    设置音量
 *@param[in]   u8 vol
 *@param[out]  无
 *@retval:
 */
```

(continues on next page)

(continued from previous page)

```
s32 fun_sound_setvol(u8 vol)
```

34.2.2 流程设计

流程设计基本按照前面 WM8978 实验中做的测试程序。

双 DMA 缓冲。在 DMA 中断中设置标志缓冲切换。在主 TASK 中填充数据。

- 播放

```
/**
 *@brief:      fun_sound_play
 *@details:    通过指定设备播放指定声音
 *@param[in]   char *name
               char *dev
 *@param[out]  无
 *@retval:
 */
int fun_sound_play(char *name, char *dev)
{
        FRESULT res;
        unsigned int len;

        SoundSta = SOUND_BUSY;
        /*
                打开文件是否需要关闭？
                同时打开很多文件事发后会内存泄漏。
        */
        res = f_open(&SoundFile, name, FA_READ);
        if(res != FR_OK)
        {
                SOUND_DEBUG(LOG_DEBUG, "sound open file err:%d\r\n", res);
                SoundSta = SOUND_IDLE;
                return -1;
        }

        SOUND_DEBUG(LOG_DEBUG, "sound open file ok\r\n");

        wav_header = (TWavHeader *)wjq_malloc(sizeof(TWavHeader));
        if(wav_header == 0)
        {
```

(continues on next page)

(continued from previous page)

```
        SOUND_DEBUG(LOG_DEBUG, "sound malloc err!\r\n");
        SoundSta = SOUND_IDLE;
        return -1;
    }
    SOUND_DEBUG(LOG_DEBUG, "sound malloc ok\r\n");

    res = f_read(&SoundFile, (void *)wav_header, sizeof(TWavHeader), &len);
    if(res != FR_OK)
    {
        SOUND_DEBUG(LOG_DEBUG, "sound read err\r\n");
        SoundSta = SOUND_IDLE;
        return -1;
    }

    SOUND_DEBUG(LOG_DEBUG, "sound read ok\r\n");
    if(len != sizeof(TWavHeader))
    {
        SOUND_DEBUG(LOG_DEBUG, "read wav header err %d\r\n", len);
        SoundSta = SOUND_IDLE;
        return -1;
    }

    SOUND_DEBUG(LOG_DEBUG, "---%x\r\n", wav_header->rId);
    SOUND_DEBUG(LOG_DEBUG, "---%x\r\n", wav_header->rLen);
    SOUND_DEBUG(LOG_DEBUG, "---%x\r\n", wav_header->wId);
    SOUND_DEBUG(LOG_DEBUG, "---%x\r\n", wav_header->fId);
    SOUND_DEBUG(LOG_DEBUG, "---%x\r\n", wav_header->fLen);
    SOUND_DEBUG(LOG_DEBUG, "---wave 格式 %x\r\n", wav_header->wFormatTag);
    SOUND_DEBUG(LOG_DEBUG, "---声道      %x\r\n", wav_header->nChannels);
    SOUND_DEBUG(LOG_DEBUG, "---采样频率  %d\r\n", wav_header->nSamplesPerSec);
    SOUND_DEBUG(LOG_DEBUG, "---每秒数据量 %d\r\n", wav_header->nAvgBytesPerSec);
    SOUND_DEBUG(LOG_DEBUG, "---样点字节数 %d\r\n", wav_header->nBlockAlign);
    SOUND_DEBUG(LOG_DEBUG, "---位宽 :    %d bit\r\n", wav_header->wBitsPerSample);
    SOUND_DEBUG(LOG_DEBUG, "---data =    %x\r\n", wav_header->dId);
    SOUND_DEBUG(LOG_DEBUG, "---数据长度: %x\r\n", wav_header->wSampleLength);

    if(wav_header->nSamplesPerSec <= I2S_AudioFreq_16k)
    {
        SoundBufSize = DAC_SOUND_BUFF_SIZE2;
    }
```

(continues on next page)

(continued from previous page)

```
        else
        {
                SoundBufSize = I2S_DMA_BUFF_SIZE1;
        }
        /*

        */
        SoundBufP[0] = (u16 *)wjq_malloc(SoundBufSize*2);
        SoundBufP[1] = (u16 *)wjq_malloc(SoundBufSize*2);

        SOUND_DEBUG(LOG_DEBUG, "%08x, %08x\r\n", SoundBufP[0], SoundBufP[1]);
        if(SoundBufP[0] == NULL)
        {

                SOUND_DEBUG(LOG_DEBUG, "sound malloc err\r\n");
                SoundSta = SOUND_IDLE;
                return -1;
        }

        if(SoundBufP[1] == NULL )
        {
                wjq_free(SoundBufP[0]);
                SoundSta = SOUND_IDLE;
                return -1;
        }

        /* 根据文件内容设置采样频率跟样点格式 */
        u8 format;
        if(wav_header->wBitsPerSample == 16)
        {
                format =        WM8978_I2S_Data_16b;
        }
        else if(wav_header->wBitsPerSample == 24)
        {
                format =        WM8978_I2S_Data_24b;
        }
        else if(wav_header->wBitsPerSample == 32)
        {
                format =        WM8978_I2S_Data_32b;
        }
```

(continues on next page)

(continued from previous page)

```
    /* 打开指定设备 */
    if(0 == strcmp(dev, "wm8978"))
    {
            dev_wm8978_open();
            dev_wm8978_dataformat(wav_header->nSamplesPerSec,
                    WM8978_I2S_Phillips, format);
            mcu_i2s_dma_init(SoundBufP[0], SoundBufP[1], SoundBufSize);
            SoundDevType = SOUND_DEV_2CH;
    }

    playlen = 0;

    u32 rlen;

    /* 音源单声道, 设备双声道, 对数据复制一份到另外一个声道 */
    if((wav_header->nChannels == 1) && (SoundDevType == SOUND_DEV_2CH))
    {
            rlen = SoundBufSize;
            f_read(&SoundFile, (void *)SoundBufP[0], rlen, &len);
            fun_sound_deal_1ch_data((u8*)SoundBufP[0]);
            f_read(&SoundFile, (void *)SoundBufP[1], rlen, &len);
            fun_sound_deal_1ch_data((u8*)SoundBufP[1]);

    }
    else
    {
            rlen = SoundBufSize*2;
            f_read(&SoundFile, (void *)SoundBufP[0], rlen, &len);
            f_read(&SoundFile, (void *)SoundBufP[1], rlen, &len);
    }

    playlen += rlen*2;

    if(0 == strcmp(dev, "wm8978"))
    {
            dev_wm8978_transfer(1);//启动 I2S 传输
    }

    SoundSta = SOUND_PLAY;
```

(continues on next page)

(continued from previous page)

```
        return 0;
}
```

函数参数包含文件名和音频设备。dev 就是用来指定音频设备，用 WM8978 播放还是 DAC-SOUND 播放。19~66 行，读文件，并判断是不是 WAV 文件，如果是 WAV 文件，就将 WAV 文件头读出。68~96，申请两个缓冲区。99~112，设置格式 115~120，打开设备 127~144，准备数据，填满两个缓冲。151，启动设备，开始播放。

- 流程流程也是一个 TASK 函数，在 main 函数的 while(1) 中执行。

```
/**
 *@brief:       fun_sound_task
 *@details:     声音播放轮询任务，执行间隔不可以太久，-
                否则声音会有杂音，也就是断续
 *@param[in]    void
 *@param[out]   无
 *@retval:
 */
void fun_sound_task(void)
{
        FRESULT res;
        unsigned int len;
        volatile s32 buf_index = 0;
        int rlen;
        u16 i;
        u8 *p;

        if(SoundSta == SOUND_BUSY
                || SoundSta == SOUND_IDLE)
                return;

        buf_index = fun_sound_get_buff_index();
        if(0xff != buf_index)
        {
            if(SoundSta == SOUND_PAUSE)//暂停
            {
                for(i=0;i<SoundBufSize;i++)
                {
                        *(SoundBufP[buf_index]+i)= 0x0000;
                }
```

(continues on next page)

(continued from previous page)

```
            }
            else
            {

                if((wav_header->nChannels == 1) && (SoundDevType == SOUND_DEV_2CH))
                {
                    rlen = SoundBufSize;
                    res = f_read(&SoundFile, (void *)SoundBufP[buf_index], rlen, &len);
                    fun_sound_deal_1ch_data((u8*)SoundBufP[buf_index]);
                }
                else
                {
                    rlen = SoundBufSize*2;
                    res = f_read(&SoundFile, (void *)SoundBufP[buf_index], rlen, &len);
                }

                //memset(SoundBufP[buf_index], 0, SoundRecBufSize*2);

                playlen += len;

                /*
                u 盘有 BUG，有时候读到的数据长度不对
                稳健的做法是用已经播放的长度跟音源长度比较。
                */
                if(len < rlen)
                {
                    SOUND_DEBUG(LOG_DEBUG, "play finish %d, playlen:%x\r\n", len,␣
↪playlen);
                    fun_sound_stop();

                }
            }
    }

}
```

流程还比较简单，就是通过 fun_sound_get_buff_index 查看是否有缓冲空了，空了就填数据。

34.2.3 测试

在 SD 卡中放几个 WAV 文件。在 main 中，按下按键后调用下面函数播放

```
/**
 *@brief:       fun_sound_test
 *@details:     测试播放
 *@param[in]    void
 *@param[out]   无
 *@retval:
 */
void fun_sound_test(void)
{
        SOUND_DEBUG(LOG_DEBUG, "play sound\r\n");
        fun_sound_play("1:/mono_16bit_8k.wav", "wm8978");
}
```

34.3 总结

WAV 的格式还是很简单的，加上上次已经基本上设计好了语音播放的程序架构，做一个 WAV 播放功能还是很容易的。如何让 DACSOUND 设备也能播放 WAV 文件?

34.4 end

CHAPTER 35

I2SEXT-WM8978-录音

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

在 WM8978 调试章节，我们提到过 STM32 的 I2S_ext 功能。使用这个外扩的 I2S，就能实现 I2S 全双工，就能实现录音功能。现在让我们一起调试录音功能。

35.1 I2S_ext

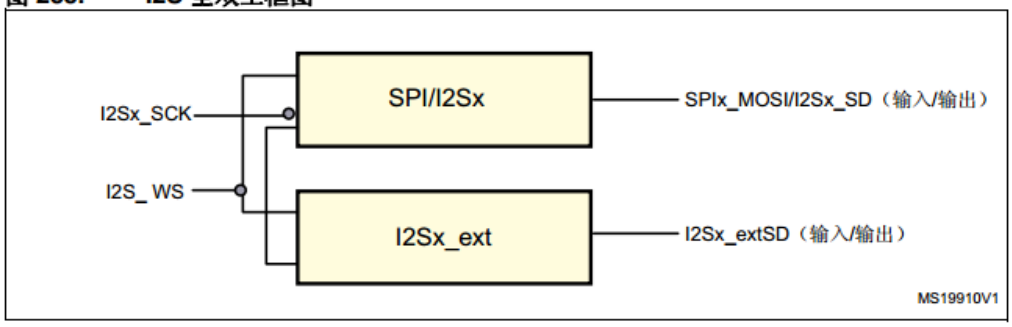
STM32 的 I2S 支持全双工。如下说明。

27.4.2 I2S 全双工

为支持 I2S 全双工模式，除了 I2S2 和 I2S3，还可以使用两个额外的 I^2S，它们称为扩展 I2S（I2S2_ext、I2S3_ext）（参见*图 288*）。因此，第一个 I2S 全双工接口基于 I2S2 和 I2S2_ext，第二个基于 I2S3 和 I2S3_ext。

注意： I2S2_ext 和 I2S3_ext 仅用于全双工模式。

图 288. I2S 全双工框图



1. 其中 X 可以是 2 或 3。

I2Sx 可以在主模式下工作。因此：

- 只有 I2Sx 可在半双工模式下输出 SCK 和 WS
- 只有 I2Sx 可在全双工模式下向 I2S2_ext 和 I2S3_ext 提供 SCK 和 WS。

扩展 I2S (I2Sx_ext) 只能用于全双工模式。I2Sx_ext 始终在从模式下工作。

I2Sx 和 I2Sx_ext 均可用于发送和接收。

WAV

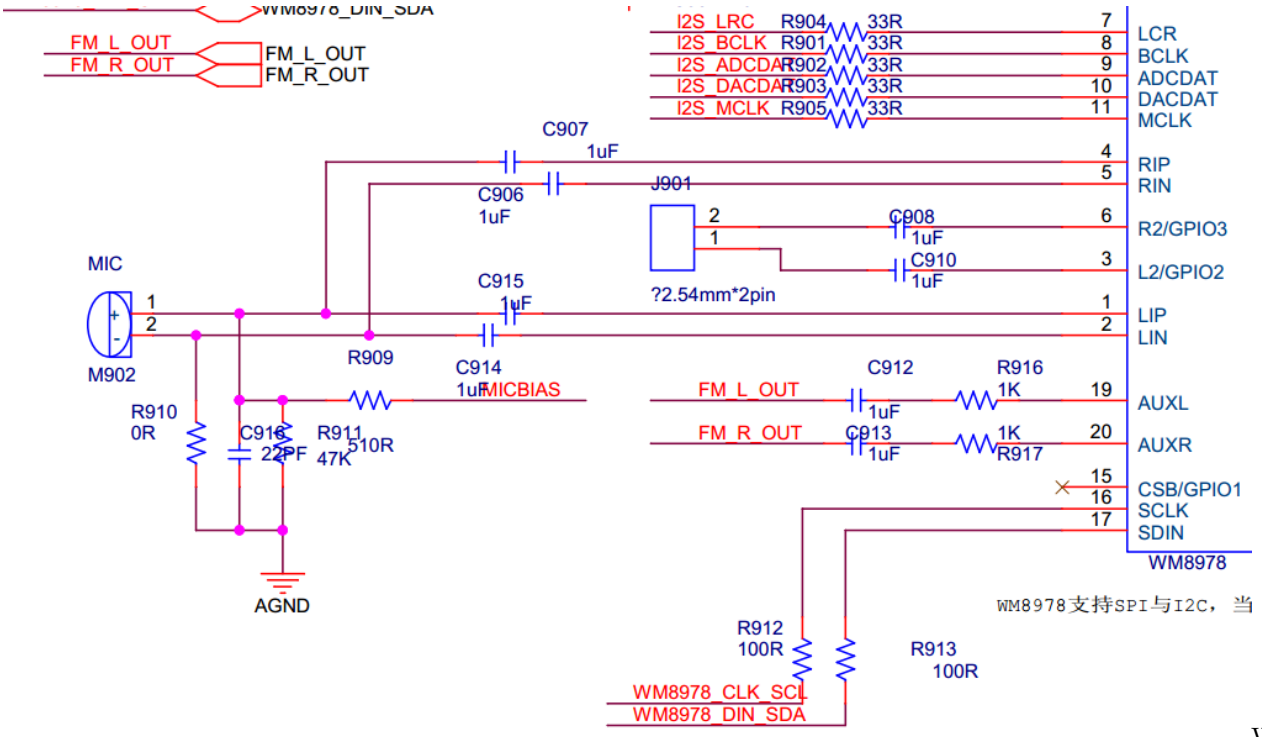
I2S_ext 没有时钟，必须配合 I2S 使用。通常，我们用 I2S 做主设备，输出时钟，发送 I2S 数据到外部设备。I2S_ext 也使用 I2S 的时钟，从外部设备接收 I2S 数据。

35.2 WM8978 录音配置

WM8978 的驱动我们已经完成，现在只需要看看录音需要设置什么寄存器。

35.2.1 硬件

下图是 8978 线路图。WM8978 支持左右声道两个 MIC。板子太小，根本无法区分左右声道，因此将两个 MIC 连在一起。只使用一个 MIC 电路。



WAV

35.2.2 配置

录音功能涉及到 WM8978 的两个模块：

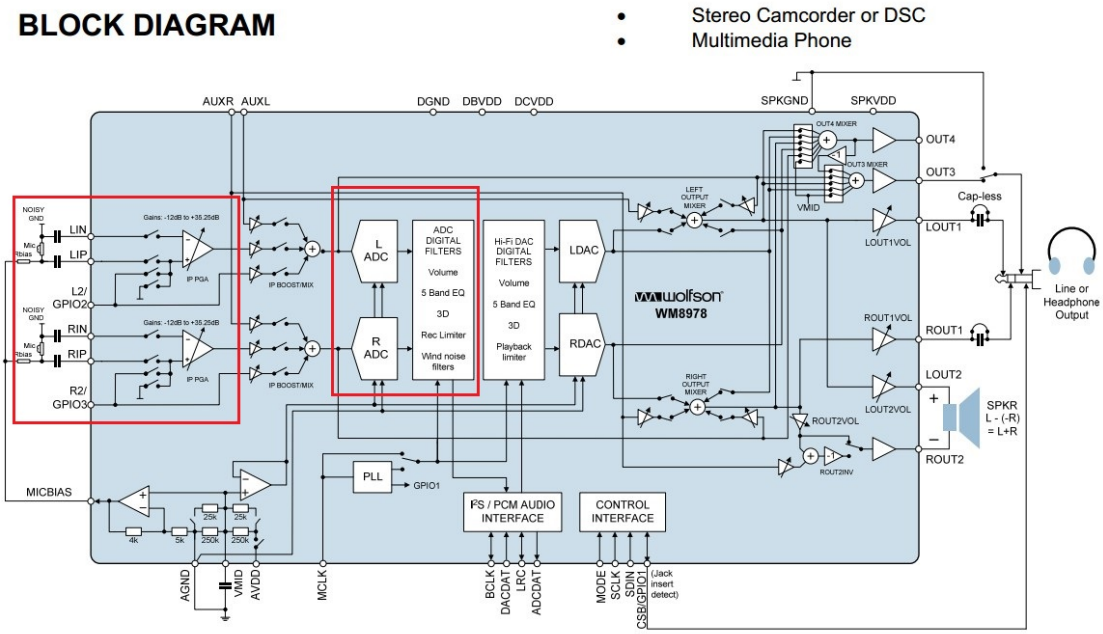
1. MIC 模块。

MIC 模块用于配置是否使用 MIC 输入的音源。只要配置好 MIC 模块，MIC 输入的声音就可以从 WM8978 播放。

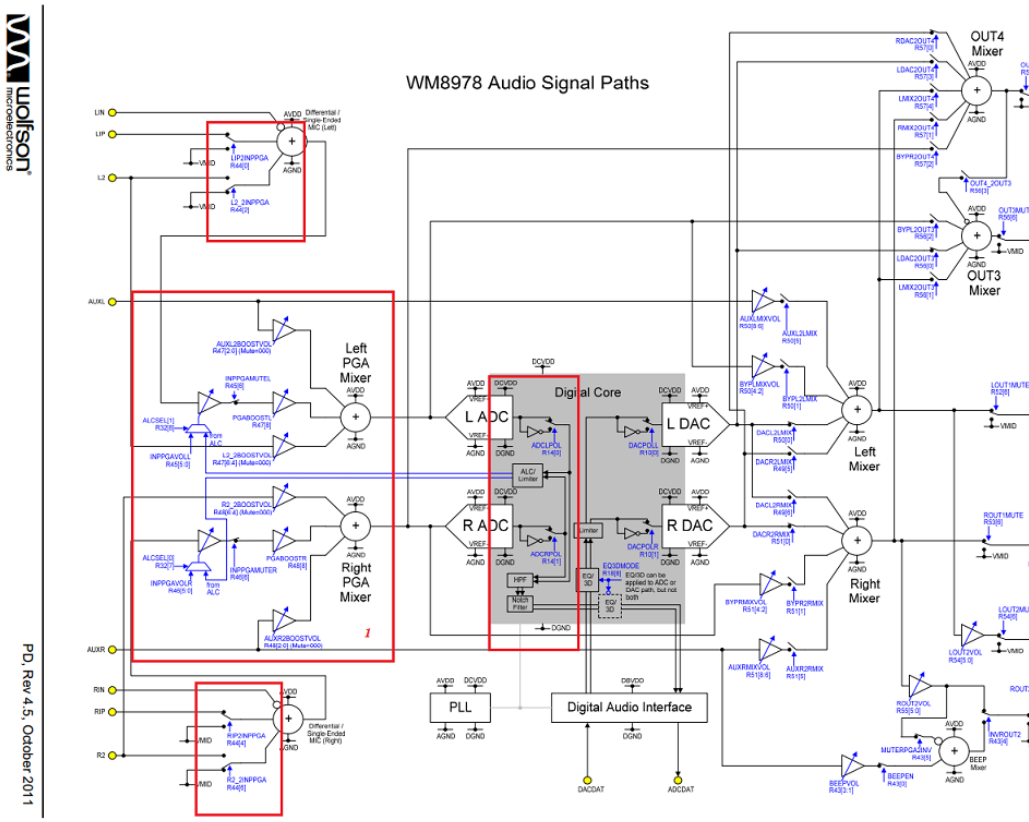
1. ADC 模块。

MIC 模块输入的是模拟信号，要通过 I2S 传输到 STM32，需要经过 ADC 转换。

如下图，左边红框是 MIC 模块，中间红框是 ADC 模块。



WAV



对这两个模块的配置主要有一下：

R45/R46、R47/R48，设置 PGA 增益，也就是 MIC 的输入增益。左边中间红框是设置增益的。

R44，控制 MIC 是否输入到 PGA。左边上下两个红框处。R14，配置 ADC。中间红框。

更多配置请查看代码

35.3 驱动设计

底层驱动设计两部分：I2S_ext、WM8978。

35.3.1 I2S_ext

I2S_ext 驱动类似 I2S。

同样需要配置 I2S 通信格式。同样使用 DMA 传输。同样使用双缓冲 DAM 模式。

- 配置

```
void mcu_i2sext_config(u32 AudioFreq, u16 Standard,u16 DataFormat)
{
        I2S_InitTypeDef I2S2ext_InitStructure;
        //I2S_FullDuplexConfig 会进行转换
        I2S2ext_InitStructure.I2S_Mode = I2S_Mode_MasterTx;
        I2S2ext_InitStructure.I2S_Standard=Standard;//IIS 标准
        I2S2ext_InitStructure.I2S_DataFormat=DataFormat;//IIS 数据长度
        //主时钟输出,i2sext 无效
        I2S2ext_InitStructure.I2S_MCLKOutput=I2S_MCLKOutput_Enable;
        I2S2ext_InitStructure.I2S_AudioFreq=AudioFreq;//IIS 频率设置
        I2S2ext_InitStructure.I2S_CPOL=I2S_CPOL_Low;//空闲状态时钟电平
        //初始化 I2S2ext 配置
        I2S_FullDuplexConfig(I2S2ext, &I2S2ext_InitStructure);

        I2S_Cmd(I2S2ext, ENABLE);                //I2S2ext I2S EN 使能.
}
```

- DAM 设置

```
/**
 *@brief:      mcu_i2sext_dma_init
 *@details:    设置 I2S EXT DMA 缓冲
 *@param[in]   u16* buf0
               u16 *buf1
               u32 len
 *@param[out]  无
 *@retval:
 */
void mcu_i2sext_dma_init(u16* buf0, u16 *buf1, u32 len)
{
```

(continues on next page)

(continued from previous page)

```
    NVIC_InitTypeDef   NVIC_InitStructure;
    DMA_InitTypeDef  DMA_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1,ENABLE);//DMA1 时钟使能

    DMA_DeInit(I2S2_EXT_DMA);
    while (DMA_GetCmdStatus(I2S2_EXT_DMA) != DISABLE){}//等待 DMA1_Stream3 可配置
    //清空 DMA1_Stream3 上所有中断标志
    DMA_ClearITPendingBit(I2S2_EXT_DMA,DMA_IT_FEIF3|DMA_IT_DMEIF3
                          |DMA_IT_TEIF3|DMA_IT_HTIF3|DMA_IT_TCIF3);

    /* 配置 DMA Stream */
    DMA_InitStructure.DMA_Channel = DMA_Channel_3;
    DMA_InitStructure.DMA_PeripheralBaseAddr = (u32)&I2S2ext->DR;
    DMA_InitStructure.DMA_Memory0BaseAddr = (u32)buf0;//DMA 存储器 0 地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;//外设到存储器模式
    DMA_InitStructure.DMA_BufferSize = len;//数据传输量
    //外设非增量模式
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;//存储器增量模式
    //外设数据长度:16 位
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
    //存储器数据长度: 16 位
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 使用循环模式
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable; //不使用 FIFO 模
式
    DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_1QuarterFull;
    DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;//外设突发单次传输
    //存储器突发单次传输
    DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
    DMA_Init(I2S2_EXT_DMA, &DMA_InitStructure);//初始化 DMA Stream
    //双缓冲模式配置
    DMA_DoubleBufferModeConfig(I2S2_EXT_DMA, (u32)buf0, DMA_Memory_0);
    //双缓冲模式配置
    DMA_DoubleBufferModeConfig(I2S2_EXT_DMA, (u32)buf1, DMA_Memory_1);

    DMA_DoubleBufferModeCmd(I2S2_EXT_DMA,ENABLE);//双缓冲模式开启
```

(continues on next page)

(continued from previous page)

```
        DMA_ITConfig(I2S2_EXT_DMA,DMA_IT_TC,ENABLE);//开启传输完成中断

        SPI_I2S_DMACmd(I2S2ext, SPI_I2S_DMAReq_Rx, ENABLE);//I2S2ext RX DMA 请求使能.

        NVIC_InitStructure.NVIC_IRQChannel = DMA1_Stream3_IRQn;
        NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority =0x00;//抢占优先级 0
        NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x00;//子优先级 1
        NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//使能外部中断通道
        NVIC_Init(&NVIC_InitStructure);//配置

}
```

看注释基本明白要如何配置，跟播音的设置类似。**要谨记的是，I2S_ext 需要 I2S 配合输出时钟**

35.3.2 WM8978

WM8978 的基本配置在播音例程已经配置好，只需要在 OPEN 的时候打开 MIC 输入。

```
s32 dev_wm8978_open(void)
{
        dev_wm8978_inout(WM8978_INPUT_DAC|WM8978_INPUT_LMIC|WM8978_INPUT_RMIC|WM8978_
↪INPUT_ADC,
                                        WM8978_OUTPUT_SPK|WM8978_OUTPUT_PHONE);
        return 0;
}
```

并且在配置数据格式的时候同时配置 I2S_ext 的数据格式。

```
s32 dev_wm8978_dataformat(u32 Freq, u8 Standard, u8 Format)
{
        u16 standard;
        u16 dataformat;

        dev_wm8978_set_dataformat(Standard, Format);

        ...

        mcu_i2s_config(Freq, standard, dataformat);
        mcu_i2sext_config(Freq, standard, dataformat);
```

(continues on next page)

(continued from previous page)

```
        return 0;
}
```

35.4 录音中间层设计

前面的 I2S_ext 驱动设计，跟 I2S 一样，都是使用双缓冲 DMA 格式，理所当然的我们都能想到：录音的中间层设计跟播音应该差不多。

35.4.1 接口设计

要求和播音基本一致：

开始录音暂停继续停止。

本次测试我们不做暂停功能，只需要实现播放和停止即可。

- 开始录音

```
/**
 *@brief:      fun_sound_rec
 *@details:    启动录音
 *@param[in]   char *name
 *@param[out]  无
 *@retval:
 */
s32 fun_sound_rec(char *name)
{
        FRESULT fres;
        u32 len;

        SOUND_DEBUG(LOG_DEBUG, "sound rec\r\n");
        RecWavSize = 0;
        SoundRecBufSize = SoundBufSize;

        /*  创建 WAV 文件 */
        fres=f_open(&SoundRecFile,(const TCHAR*)name, FA_CREATE_ALWAYS | FA_WRITE);
        if(fres != FR_OK)                          //文件创建失败
        {
                SOUND_DEBUG(LOG_DEBUG, "create rec file err!\r\n");
```

(continues on next page)

(continued from previous page)

```
        return -1;
    }

    recwav_header = (TWavHeader *)wjq_malloc(sizeof(TWavHeader));
    if(recwav_header == NULL)
    {
        SOUND_DEBUG(LOG_DEBUG, "rec malloc err!\r\n");
        return -1;
    }

    recwav_header->rId=0X46464952;
    recwav_header->rLen = 0;//录音结束后填
    recwav_header->wId = 0X45564157;//wave
    recwav_header->fId=0X20746D66;
    recwav_header->fLen = 16;
    recwav_header->wFormatTag = 0X01;
    recwav_header->nChannels = 2;
    //这个采样频率需要特殊处理，暂时不做。
    recwav_header->nSamplesPerSec = SOUND_REC_FRE;
    recwav_header->nAvgBytesPerSec =
        (recwav_header->nSamplesPerSec)*(recwav_header->nChannels)*(16/8);
    recwav_header->nBlockAlign = recwav_header->nChannels*(16/8);
    recwav_header->wBitsPerSample = 16;
    recwav_header->dId = 0X61746164;
    recwav_header->wSampleLength = 0;

    fres=f_write(&SoundRecFile,(const void*)recwav_header,
                        sizeof(TWavHeader), &len);
    if((fres!= FR_OK)
        || (len != sizeof(TWavHeader)))
    {

        SOUND_DEBUG(LOG_DEBUG, "rec write err!\r\n");
        wjq_free(recwav_header);
        return -1;
    }
    else
    {
        SOUND_DEBUG(LOG_DEBUG, "create rec wav ok!\r\n");
    }
```

(continues on next page)

(continued from previous page)

```
        /*  测试录音     */
        SoundRecBufP[0] = (u16 *)wjq_malloc(SoundRecBufSize*2);
        SoundRecBufP[1] = (u16 *)wjq_malloc(SoundRecBufSize*2);

        SOUND_DEBUG(LOG_DEBUG, "%08x, %08x\r\n", SoundRecBufP[0], SoundRecBufP[1]);
        if(SoundRecBufP[0] == NULL)
        {

                SOUND_DEBUG(LOG_DEBUG, "sound malloc err\r\n");
                return -1;
        }

        if(SoundRecBufP[1] == NULL )
        {
                wjq_free(SoundRecBufP[0]);
                return -1;
        }

        dev_wm8978_open();
        dev_wm8978_dataformat(SOUND_REC_FRE, WM8978_I2S_Phillips,
                        WM8978_I2S_Data_16b);

        mcu_i2s_dma_init(RecPlayTmp, RecPlayTmp, 1);
        dev_wm8978_transfer(1);//启动 I2S 传输

        mcu_i2sext_dma_init(SoundRecBufP[0], SoundRecBufP[1], SoundRecBufSize);
        mcu_i2sext_dma_start();

        SOUND_DEBUG(LOG_DEBUG, "rec--------------------\r\n");

        return 0;
}
```

17~58，创建 WAV 文件。61~76，准备双缓冲 78~85，配置 WM8978，启动录音。

- 停止录音

```
/**
 *@brief:      fun_rec_stop
 *@details:    停止录音
```

(continues on next page)

(continued from previous page)

```
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 fun_rec_stop(void)
{
        u32 len;
        dev_wm8978_transfer(0);
        mcu_i2sext_dma_stop();

        recwav_header->rLen = RecWavSize+36;
        recwav_header->wSampleLength = RecWavSize;
        f_lseek(&SoundRecFile,0);

        f_write(&SoundRecFile,(const void*)recwav_header,
                sizeof(TWavHeader),&len);//写入头数据
        f_close(&SoundRecFile);

        wjq_free(SoundRecBufP[0]);
        wjq_free(SoundRecBufP[1]);
        wjq_free(recwav_header);
        return 0;
}
```

停止录音，保存 WAV 文件，释放内存。

35.4.2 流程

跟放音一样使用双缓冲，所以录音流程也基本一样。

```
/**
 *@brief:      fun_rec_task
 *@details:    录音线程
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
void fun_rec_task(void)
{
        int buf_index = 0;
```

(continues on next page)

(continued from previous page)

```
        u32 len;
        FRESULT fres;

        buf_index = fun_rec_get_buff_index();
        if(0xff != buf_index)
        {
                //uart_printf("rec buf full:%d!\r\n", buf_index);
                RecWavSize += SoundRecBufSize*2;

                fres = f_write(&SoundRecFile,(const void*)SoundRecBufP[buf_index],
                                2*SoundRecBufSize, &len);
                if(fres != FR_OK)
                {
                        SOUND_DEBUG(LOG_DEBUG, "write err\r\n");
                }

                if(len!= 2*SoundRecBufSize)
                {
                        SOUND_DEBUG(LOG_DEBUG, "len err\r\n");
                }

        }
}
```

在主任务循环中运行 task 函数，发送有缓冲满了，将数据读出来写到 wav 文件。

35.5 测试

增加下面两个函数，fun_rec_test 录音，fun_play_rec_test 播放录音得到的 WAV 文件。

```
/**
 *@brief:      fun_rec_test
 *@details:    开始录音
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
void fun_rec_test(void)
```

(continues on next page)

(continued from previous page)

```
{
        fun_sound_rec("1:/rec9.wav");
}

void fun_play_rec_test(void)
{
        fun_sound_play("1:/rec9.wav", "wm8978");
}
```

在 main 函数的 while 循环中，当按下按键，开始录音，松开按键，播放录音文件。

35.6 总结

在前期播音和 WAV 文件解码两个章节的基础上，实现 I2S 录音可以说不难。要树立的概念就是 I2S_ext 必须使用 I2S 的时钟。我们做的录音和播音，是两个单独的功能。请问如果要在播音的时候能录音，程序要如果修改?

35.7 end

CHAPTER 36

DAC SOUND 驱动改造–播放 WAV 文件

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

前面播放 WAV 音频和 I2S 录音两个小节，我们接触了一种叫做中间件的程序。我可以可以再总结一下：

> 所谓的中间件，通常是实现一种功能的抽象接口。这一层代码，对应用屏蔽了硬件实现，只提供功能接口。例如：LCD 中间件，GUI 也可以算中间件，应用层主要调用 LCD 显示接口，就可以在各种 LCD 伤显示内容。那么，语音播放中间件，就是 APP 播放音乐，可以在多种硬件声音设备上播放。

前面章节我们已经实现了 WM8978 播放，我们硬件正好还有一个 DAC SOUND 的设备。怎么样修改 DAC SOUND 代码，让其在语音播放中间件下也能工作?

36.1 框架设计

不用多想就可以知道，既然都是在 SOUND 中间件下工作的硬件，那么驱动，肯定应该差不多。我们已经完成了 WM8978 的驱动设计，DAC SOUND 按其修改，肯定是最合理的。

36.1.1 DAC SOUND

前面我们做的 DAC SOUND 功能：

> 启动一个定时器，按照音频采样率，定时从数组读取样点，通过 DAC 输出。

DAC SOUND 和 WM8978 的一个最大不同点就是：**它是单声道的**。

36.1.2 双缓冲

如果要根据 I2S 驱动修改 DAC SOUND，只需要简单的将数组改为双缓冲。语音中间件只需要将原来打开 WM8978 设备改为 DAC 设备就可以了。中间件数据处理部分代码都不需要修改。

36.1.3 代码说明

初始化函数，没变，也就是初始化 IO 口。

```
/*

        DAC 播放声音，固定播放 8K 单声道 16BIT 的音源。

*/

u16 *DacSoundSampleP0;
u16 *DacSoundSampleP1;
u16 *DacSoundCrBufP;//当前使用的 BUF

u32 DacSoundSampleBufSize;
u32 DacSoundSampleIndex;

s32 dev_dacsound_init(void)
{
        GPIO_InitTypeDef GPIO_InitStructure;
```

(continues on next page)

(continued from previous page)

```
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//---模拟模式
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//---下拉
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_Init(GPIOA, &GPIO_InitStructure);//---初始化 GPIO

        GPIO_ResetBits(GPIOA, GPIO_Pin_5);

        return 0;
}
```

打开播放函数，除了初始化 DAC 和定时器，还需要初始化两个缓冲的指针。

```
s32 dev_dacsound_open(void)
{
        mcu_dac_open();
        mcu_tim3_init();
        DacSoundSampleIndex = 0;
        DacSoundCrBufP = DacSoundSampleP0;

        return 0;
}
```

空函数，暂时不做太复杂，只播放 8K 采样率的 WAV。

```
/**
 *@brief:      dev_dacsound_dataformat
 *@details:       设置播放配置，DAC 播放固定支持 8K 16BIT 单声-
                  道
 *@param[in]   u32 Freq
               u8 Standard
               u8 Format
 *@param[out]  无
 *@retval:
 */
s32 dev_dacsound_dataformat(u32 Freq, u8 Standard, u8 Format)
{
        return 0;
}
```

初始化缓冲指针，模拟 I2S DMA 配置双缓冲。

```
/**
 *@brief:       dev_dacsound_setbuf
 *@details:     设置播放缓冲
 *@param[in]    u16 *buffer0
                u16 *buffer1
                u32 len
 *@param[out]   无
 *@retval:
 */
s32 dev_dacsound_setbuf(u16 *buffer0,u16 *buffer1,u32 len)
{
        DacSoundSampleP0 = buffer0;
        DacSoundSampleP1 = buffer1;
        DacSoundSampleBufSize = len;

        return 0;
}
```

dev_dacsound_transfer 函数，启动播放，也是模拟 I2S 的函数

```
/**
 *@brief:       dev_dacsound_transfer
 *@details:     启动或停止 DAC 播放
 *@param[in]    u8 sta
 *@param[out]   无
 *@retval:
 */
s32 dev_dacsound_transfer(u8 sta)
{
        if(sta == 1)
        {
                /* 打开定时器，启动播放 */
                DACSOUND_DEBUG(LOG_DEBUG, "dac sound play\r\n");
                mcu_tim3_start();
        }
        else
        {
                /* 停止定时器 */
                mcu_tim3_stop();
        }
```

(continues on next page)

(continued from previous page)

```
        return 0;
}
```

停止播放

```
s32 dev_dacsound_close(void)
{
        dev_dacsound_init();
        return 0;
}
```

定时器中断函数，在这个函数内将缓冲的数据通过 DAC 输出。流程跟原来基本一致。需要修改的是取数据的方法。

```
/**
 *@brief:      dev_dacsound_timerinit
 *@details:    在定时器中断中调用，每 125US 输出一个 DAC 数据
                          能不能改为 DMA？
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 dev_dacsound_timerinit(void)
{
    s16 data = 0;
    u16 tmp;

        if(DacSoundSampleIndex >= DacSoundSampleBufSize)
        {
                if(DacSoundCrBufP == DacSoundSampleP0)
                {
                        DacSoundCrBufP = DacSoundSampleP1;
                        fun_sound_set_free_buf(0);
                }
                else
                {
                        DacSoundCrBufP = DacSoundSampleP0;
                        fun_sound_set_free_buf(1);
                }
                DacSoundSampleIndex = 0;
```

(continues on next page)

(continued from previous page)

```
        }

        /* 要注意，读到的数据是 S16，正负值 */
        data = *(DacSoundCrBufP + DacSoundSampleIndex);
        /*
                先压缩，也就是减少音量，在负数时候压缩（除）
                压缩方向时中位值，如果先将负数调整为正数（抬高直流电平），
                压缩方向就会变成音频的最低值，音效会失真。
        */
        data = data/(16+30);//12 位 DAC，再加上音量设置，
        /* 再调整中位值（直流电平），因为音频数据有负数，DAC 输出没有负数 */
        tmp = (data+0x800);
        mcu_dac_output(tmp);

        DacSoundSampleIndex++;
        return 0;
}
```

36.2 中间件修改

在 int fun_sound_play(char *name, char *dev) 内，原来只有 WM8978 设备，现添加 DAC SOUND 设备。

```
        if(0 == strcmp(dev, "wm8978"))
        {
                dev_wm8978_open();
                dev_wm8978_dataformat(wav_header->nSamplesPerSec,
                        WM8978_I2S_Phillips, format);
                mcu_i2s_dma_init(SoundBufP[0], SoundBufP[1], SoundBufSize);
                SoundDevType = SOUND_DEV_2CH;
                dev_wm8978_transfer(1);//启动 I2S 传输
        }
        else if(0 == strcmp(dev, "dacsound"))
        {
                dev_dacsound_open();
                dev_dacsound_dataformat(wav_header->nSamplesPerSec,
                        WM8978_I2S_Phillips, format);
                dev_dacsound_setbuf(SoundBufP[0], SoundBufP[1], SoundBufSize);
```

(continues on next page)

(continued from previous page)

```
                SoundDevType = SOUND_DEV_1CH;
                dev_dacsound_transfer(1);
        }
```

fun_sound_stop 函数同样添加

```
if(SoundDevType == SOUND_DEV_2CH)
{
  dev_wm8978_transfer(0);
}
else if(SoundDevType == SOUND_DEV_1CH)
{
  dev_dacsound_transfer(0);
  dev_dacsound_close();
}
```

36.3 应用

只需要在播放语音的时候指定 DAC 设备。

```
/**
 *@brief:      fun_sound_test
 *@details:    测试播放
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
void fun_sound_test(void)
{
        SOUND_DEBUG(LOG_DEBUG, "play sound\r\n");
        fun_sound_play("1:/mono_16bit_8k.wav", "dacsound");

}
```

36.4 总结

简单吧？确实简单。这么简单的原因是，我们良好的架构设计。但愿我们能教会大家写好代码。

36.5 end

CHAPTER 37

详解矩阵按键扫描

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

按键输入是人接交互的一个重要输入途径。电脑键盘，就是一个最常见的矩阵按键。在其他电子消费产品中，经常见到 4*4 矩阵按键。用户按下按键，程序如何知道用户按下哪个键？现在就让我们一起学习按键扫描的原理。

37.1 单键扫描

单键，并不是指一个按键，而是指一个 IO 口控制一个按键，原理图如下：

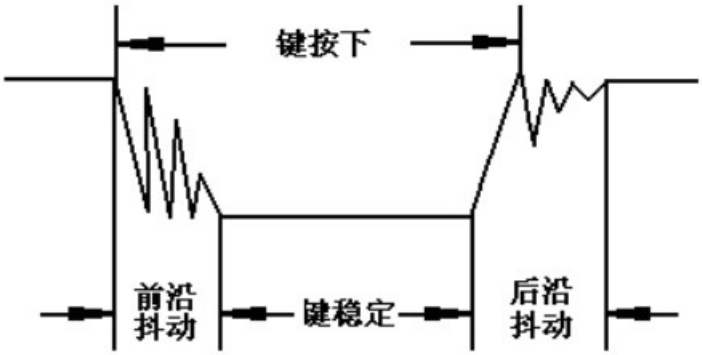
单键原理图

PA0 这个 IO 口接到按键的一端，按键另外一端接到地。当按键按下，两端短路，IO 口就接到地，就是低电平。那没按下时，IO 口啥都没接，为啥是高电平呢？因为 IO 口在芯片内部可以配置连接一个内部上拉电阻。如果使用了没有内部上拉电阻的 IO，就只能在外部接一个电阻将 IO 口上拉到高电平。

37.2 按键扫描方式

首先我们要记住的以下常识：

1. 芯片跑得很快。从一个 IO 口读取输入电平，只是一瞬间的事情。
2. 手可能会抖动。
3. 机械按键可能会抖动。

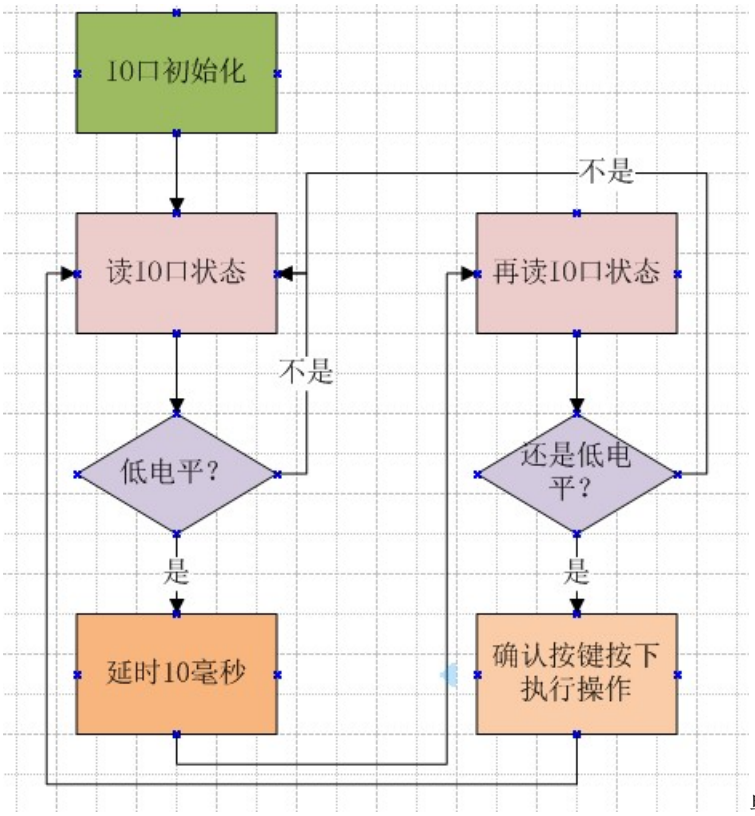


用示波器抓按键的波形，可能如下图：

按键抖动

很明显，如果我们在抖动时间段读 IO 状态，得到的值将是**随机**的。因此，按键扫描最重要的一个功能就是**去抖动**。去抖动的原理很简单：

> 间隔一定时间读几次，电平连续相同则认为状态是可靠的。



通常大家看到的教程，按键流程如下：单键扫描流程这样的单键扫描流程有以下问题：

1. 去抖动通过硬延时实现。
2. 只对按下去抖动，没有对松开去抖动。

硬延时在真正的项目开发中，是绝对不建议使用的，无论是按键扫描还是其他功能，都不应该用硬延时。无论按下还是松开，都应该有去抖动功能。就算一些没有抖动的按键，也要加上，毕竟产品销售出去后，环境变化，可能会受到外部干扰。如果按键被干扰造成误动作，将是一个失败的产品。

那么如何优化掉硬延时呢？在功能稍微复杂的单片机系统中，常用的是**轮询模式**。轮询模式的**代码模式**大概如下：

```
main(void)
{

        初始化

        while(1)
        {
                扫描按键 ();
                扫描串口 ();
                ........
        }
```

(continues on next page)

(continued from previous page)

```
}
```

在 while 循环中，轮流执行各个驱动的代码。我们通常把这些函数叫做任务，TASK。在这些任务函数中，一般都是检测状态或执行一些简单的代码，就会退出，不会在任务中卡太久。如果某个任务卡太久，其他任务的响应时间就会很差。

但是很多驱动或者任务，经常是需要等待某种状态，或者是要延时一定时间再判断状态。就像按键一样，要间隔一定时间再次去读 IO 状态。怎么做呢？在轮询模式下，驱动常用的一种编码手段就是**步骤拆分**。什么叫步骤拆分？假设轮询按键扫描的间隔是 2 毫秒（2 毫秒执行一次扫描按键）。那么我们就在按键扫描里面增加一个防抖计数和一个**步骤计数**。

```
scan_key(void)
{
        第一步，检测按键是否按下

        第二步，防抖计数自加，判断防抖计数，
                记到 5 次，就到 10 毫秒了。
                再判断按键有没有按下
}
```

程序流程如上面的伪代码。每次进入 scan_key 这个函数，只会执行一个步骤。这样，陷入 scan_key 函数的时间将会很短，仅仅执行几条代码，没有延时。在 while 中需要轮询的其他功能就可以很快得到执行。

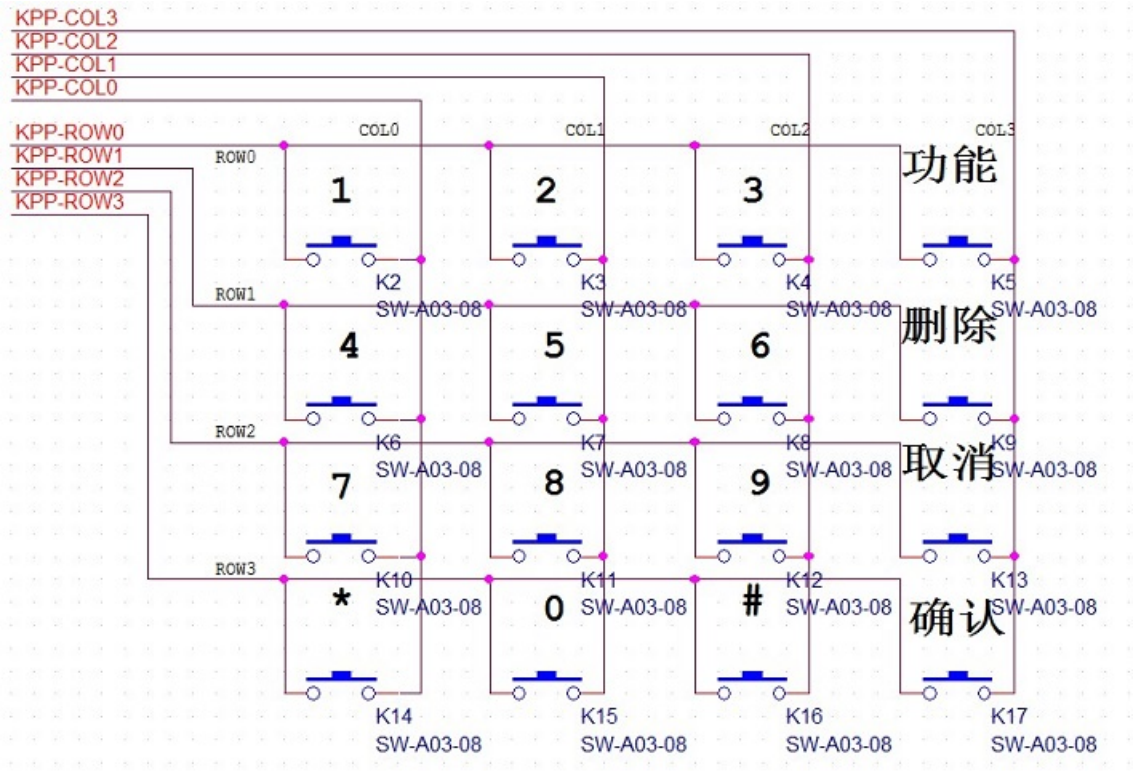
对于第二个问题：松开没有去抖动。我们的解决方法并不是松开的时候也增加去抖动，而是将松开和按下合并处理。这就牵涉到如何对某种事物进行抽象。前面的做法，是将一个按键抽象为按下和松开两个状态（也可以叫两种变化，或两种事件）。我们的做法是将按键抽象为一个电平变化的事件，无论按下还是松开，都是从一种电平变化为另外一种电平。那么，按下和松开，就可以用同一种扫描方法，用同一段代码扫描。得到变化事件后，在根据 IO 状态识别松开还是按下。

这个小观念的转变，对代码架构有很大影响。如果要总结为什么这样做，可以称之为 “统一”。将几种不同的东西，提取他们的共同点，编写一段代码共用。延伸开说，也就是模块化。

对于单键扫描的程序，在此就不展开了，大家可以根据原理自己尝试编写。

37.3 矩阵按键的扫描原理

单键模式有一个很大缺陷：当需要较多的按键时，需要的 IO 口就多。例如 16 个按键的时候，就需要 16 个 IO 口。如果使用矩阵按键，只需要 8 根 IO。下面的接法就叫做矩阵按键。



矩阵按键原理图

在 8 根 IO 上要串上限流电阻，上图没画出。

从图上可以看出，只需要 8 根 IO 口，就可以实现 16 个按键的输入。原理是什么呢？我们通过分析扫描方法解释原理。矩阵按键扫描通常有两种方法：**交叉扫描**、**逐行扫描**。

- 交叉扫描速度快，程序简单，扫描结果通过**查表**获取。但是缺陷也多。
- 逐行扫描需要轮询所有行，程序也稍微复杂，但是可以识别键盘的多种状态。

通常我们说的按键扫描都是用逐行扫描。就像字面意思说的，逐行-逐行-扫描。例如上面原理图：

> KPP-ROW0 输出 0，其他 ROW 输出 1，读取 4 根 COL IO 的状态，就可得到第一行四个按键（1、2、3、功能）的状态了。（当然，还需要去抖动）。然后 KPP-ROW1 输入 0，其他 ROW 输出 1，读 4 根 COL IO 的状态。。。。。。。。。。不断重复从 ROW0 到 ROW3。

这就叫做矩阵按键逐行扫描。上面仅仅说明原理，真正的逐行扫描当然没那么简单。下面我们就用一个完整的逐行扫描说明程序应该如何写。

```
/**
 *@brief:      dev_keypad_init
 *@details:    初始化矩阵按键 IO 口
 *@param[in]   void
 *@param[out]  无
 *@retval:
```

(continues on next page)

(continued from previous page)

```
 */
s32 dev_keypad_init(void)
{
        /*
        c:PF8-PF11   当做输输入
        r:PF12-PF15  当做输出
        */

        GPIO_InitTypeDef  GPIO_InitStructure;

        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE);

        /* r */
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
        GPIO_Init(GPIOF, &GPIO_InitStructure);

        /* c */
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_11;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
        GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
        GPIO_Init(GPIOF, &GPIO_InitStructure);

        GPIO_SetBits(GPIOF, GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15);

        u8 i;
        for(i = 0; i< KEY_PAD_ROW_NUM; i++)
        {
                KeyPadCtrl[i].dec = 0;
                KeyPadCtrl[i].oldsta = KEYPAD_INIT_STA_MASK;
                KeyPadCtrl[i].newsta = KEYPAD_INIT_STA_MASK;
        }

}
/**
 *@brief:      dev_keypad_scan
 *@details:    按键扫描, 在定时器或者任务中定时执行
```

(continues on next page)

(continued from previous page)

```
 *@param[in]   void
 *@param[out]  无
 *@retval:
 */
s32 dev_keypad_scan(void)
{
        u16 ColSta;
        u8 chgbit;
        static u8 scanrow = 0;
        u8 keyvalue;

        if(DevKeypadGd == -1)
                return -1;

        /* 读输入的状态，如果不是连续 IO，先拼成连续 IO*/
        ColSta = GPIO_ReadInputData(GPIOF);
        ColSta = (ColSta>>8)&KEYPAD_INIT_STA_MASK;

        /* 记录新状态，新状态必须是连续稳定，否则重新计数 */
        if(ColSta != KeyPadCtrl[scanrow].newsta)
        {
                KeyPadCtrl[scanrow].newsta = ColSta;
                KeyPadCtrl[scanrow].dec = 0;
        }

        /* 如新状态与旧状态有变化，进行扫描判断 */
        if(ColSta != KeyPadCtrl[scanrow].oldsta)
        {
                uart_printf(" chg--");
                KeyPadCtrl[scanrow].dec++;
                if(KeyPadCtrl[scanrow].dec >= KEY_PAD_DEC_TIME)//大于防抖次数
                {
                    /* 确定有变化 */
                    KeyPadCtrl[scanrow].dec = 0;
                    /* 新旧对比，找出变化位 */
                    chgbit = KeyPadCtrl[scanrow].oldsta^KeyPadCtrl[scanrow].newsta;
                    uart_printf("row:%d, chage bit:%02x\r\n",scanrow,chgbit);

                    /* 根据变化的位，求出变化的按键位置 */
                    u8 i;
```

(continues on next page)

(continued from previous page)

```
                    for(i=0;i<KEY_PAD_COL_NUM;i++)
                    {
                        if((chgbit & (0x01<<i))!=0)
                        {
                            keyvalue =          scanrow*KEY_PAD_COL_NUM+i;
                            /* 添加通断 (按下松开) 标志 */
                            if((KeyPadCtrl[scanrow].newsta&(0x01<<i)) == 0)
                            {
                                uart_printf("press\r\n");
                            }
                            else
                            {
                                uart_printf("rel\r\n");
                                keyvalue += KEYPAD_PR_MASK;
                            }
                            /**/
                            KeyPadBuff[KeyPadBuffW] =keyvalue+1;//+1, 从 1 开始, 不从 0 开
始
                            KeyPadBuffW++;
                            if(KeyPadBuffW>=KEYPAD_BUFF_SIZE)
                                KeyPadBuffW = 0;
                        }
                    }

                        KeyPadCtrl[scanrow].oldsta = KeyPadCtrl[scanrow].newsta;

                }
        }

        /* 将下一行的 IO 输出 0*/
        scanrow++;
        if(scanrow >= KEY_PAD_ROW_NUM)
                scanrow = 0;

        GPIO_SetBits(GPIOF, GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15);

        switch(scanrow)
        {
                case 0:
                        GPIO_ResetBits(GPIOF, GPIO_Pin_12);
```

(continues on next page)

(continued from previous page)

```
                        break;
                case 1:
                        GPIO_ResetBits(GPIOF, GPIO_Pin_13);
                        break;
                case 2:
                        GPIO_ResetBits(GPIOF, GPIO_Pin_14);
                        break;
                case 3:
                        GPIO_ResetBits(GPIOF, GPIO_Pin_15);
                        break;
        }

}
```

1. dev_keypad_init 初始化函数，完成 IO 口初始化，并初始化扫描过程用到的变量。
2. dev_keypad_scan 就是扫描函数，这个函数可以放到定时器，也可以放在 main 函数的 while 状态轮询，**轮询间隔会影响防抖效果**。
3. 62/63 行，读取 4 根 col 的状态，如果你的 col 不是连续的 IO，最好在这里拼成连续的，方便下面处理。估计有同学会问，怎么没有输出对应的 ROW 就读 COL 状态？前面我们说过步骤拆分，在这里就是将 “对应 ROW 输出 0 电平”，“读 COL”，拆分为两个步骤。上一次退出扫描函数的时候，将下一行对应 ROW 输出 0，等下一次进入扫描的时候才读取 COL。为什么？我们一直强调，CPU 很快，如果你对应的 ROW 输出 0，然后立马读 COL，间隔很短，**IO 口电平变化可是要时间的**。有同学又会说了，我知道，IO 口电平变化需要可能几百纳秒，那我输出 0 后，延时 1us，再读，应该可以读到真正电平了吧？硬延时 1us，通常也是可以接受的。是的，很多情况下我们也可以这么做，但是，会有意外。原因就是，电平变化时间除了跟 CPU 本身性能有关，还跟外部硬件有关，例如 PCB 板材，按键材料，PCB 走线，甚至是环境温度湿度都会影响电平变化。**曾经发生过换了 *PCB* 板厂，发到东北的产品发生按键无效事件**。

   IO 电平变化时间由 ns 上升到 us。我们在上一轮扫描就将对应 ROW 输出 0，下一次轮询的时候读 COL 状态，间隔通常是 ms 级的，可以避免上面问题。

1. 66-70 行，意思就是不仅仅是跟上一次的稳定状态不一样（有变化），而且需要在防抖过程中多次读取的状态一样，不一样就重新开始去抖动计数。
2. 73 行到 113 行就是去抖动跟按键识别。
3. 73-78，状态变化，而且连续变化次数达到去抖动计数，我们就认为是一个稳定的变化了。
4. 82，新旧状态异或，找出变化的位。（这样处理的好处就是，当同一行的两个按键同时按下时，我们都可以识别）
5. 87 行这个 for 循环的意思就是，每一个 col 的变化我们都要识别。
6. 91 行，识别到变化按键的物理位置编码。

我们提供的是位置编码，能不能提供功能编码？例如，第一个键是按键 1，最后一个键是确认。可以，不建议，而且是非常不建议。功能是谁的定义？谁关心？应用关心，那就让它去管，反正我按键扫描就告诉你，第一个按键按下，至于是 1 还是确认，你自己定。如果扫描给出的键值是功能键值，那就麻烦了，因为这个按键是什么功能，只要客户说一声，然后，改一下丝印。怎么的？要改改底层驱动？

1. 93-101 行，判断是按下还是松开，并在键值上添加标志位。
2. 103-106，将键值填入缓冲。关于这个缓冲，又是一个**驱动设计的要点**。

   很多人写的按键扫描，都是直接通过 return 返回键值。我就问你，当一个系统比较复杂的时候，合适吗？明显不合适，假如一个系统，有两个菜单，每个菜单都需要按键功能，难道这两个菜单都调用扫描函数？这种现象叫什么？叫做耦合：两个不同模块扯在一起。耦合，是大忌；解耦合才是正道。解耦合的一个手段就是数据缓冲。扫描模块扫到键值，丢入缓冲，至于谁要这个按键？I Don't care! 再提供一个函数用于读缓冲内的键值。谁想要谁读，按键扫描程序自己运行到天荒地老。

1. 115-135 行，就是将下一 ROW 输出 0。

一个按键矩阵按键扫描流程就是这样。怎么用这个按键扫描呢？这样用：

```
main()
{
                dev_keypad_init();
                dev_keypad_open();
                其他初始化

                while(1)
                {
                        dev_keypad_scan();
                        其他轮询任务
                        。。。。。。
                        应用处理
                        dev_keypad_read(&key,1);
                        读到按键就处理。
                }
}
```

37.4 问题

这个矩阵按键的硬件设计是有缺陷的，会产生幽灵键。当你按下 124 三个按键，按键 5 即使没按下，程序扫描出来的结果也是按下。要解决这个问题，需要在所有按键上增加二极管。通常我们会放弃三键按下这种情况，通过软件判断，将三件按下时产生幽灵键的情况抛弃。具体可以参考文档**《矩阵式键盘的先天缺陷与解决方案.docx》**

37.5 总结

矩阵按键扫描曾是我毕业后第一个产品用到的程序。在领导细心关怀下，花了两周时间才算想清楚各种细节。我认为如果你能搞清楚矩阵扫描的各种细节，对其他驱动的编写，大有帮助。

37.6 end

CHAPTER 38

FreeRtos 移植

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

待补充完善

很多同学可能从一开始就盼着这天了。其实我想说的是，什么操作系统都是纸老虎，等你学会了，就知道，不过如此。学嵌入式的路程，基本上是：裸奔-RTOS-LINUX。我们现在就开始结束裸奔。怎么学 RTOS 呢? 我觉得按照下面四个步骤学习，效果不错。

1. 了解 RTOS 的概念。
2. 找一个合适的 RTOS 了解概况。

3. 用 RTOS 跑起来。
4. 再回头研究 RTOS 的实现。

38.1 RTOS

什么是 RTOS？RTOS = **Real Time Operating System**，实时操作系统。

38.1.1 基本概念

操作系统，我们电脑上的 WINDOW 就是一个操作系统。与 WINDOWS 对应的，就是我们经常说的 linux。但是我们经常有一个误解，其实 linux 不是一个操作系统，而是**操作系统内核**。基于 linux 内核的操作系统很多，各种 LINUX 发行版都是，例如红帽，ubuntu 等。安卓系统也是基于 linux 内核的。

那实时操作系统和普通电脑手机上的操作系统有什么区别呢？先看看百度百科定义：

> 实时操作系统（RTOS）是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统做出快速响应，调度一切可利用的资源完成实时任务，并控制所有实时任务协调一致运行的操作系统。提供及时响应和高可靠性是其主要特点。

上面是定义，很多同学学习时，都会认为实时操作系统就是一个**响应很快**的操作系统。这个是片面的。实时，确实比 WIDOW 这些 PC 操作系统快，但快，不是我们的指标。实时，更准确的描述是，响应时间确定，稳定。如果一个响应，说是十秒钟之内会响应，就永远不会 11 秒才响应，这才叫实时。WINDOW 就不是一个实时系统，很多操作经常会被其他应用卡住，甚至死机。

说到这里，都是书面内容，我们还是不明白 RTOS 到底是什么鬼。RTOS 通常有以下特点：

> 小，通常只有几个文件，代码量也很小，需要的内存也很小，通常几 K 就能跑起来。如果功能不复杂，任务不多，在一个增强型 8051 都能运行起来。

还是不懂什么是 RTOS，怎么办？

38.1.2 大循环任务架构

我们要一个 RTOS 来做什么？我们用 RTOS 的一个最重要功能，是任务管理。经过前面的实验，我们知道一个裸奔的嵌入式程序，就是一个大循环，一个 while(1) 里面的大循环。所有放在 while(1) 循环里面的任务，都不能卡死。只要一个任务卡死，其他任务就都不执行了。为了达到大循环的目的，我们介绍一个程序架构方法：**步骤拆分**。不知道大家还记得吗？代码框架如下：

```
void n_task(void)
{
        switch(步骤)
        {
```

(continues on next page)

(continued from previous page)

```
                case 步骤 1:
                        break;
                case 步骤 2:
                        break;
                case 步骤 3:
                        break;
        }
}

void b_task(void)
{
        switch(步骤)
        {
                case 步骤 1:
                        break;
                case 步骤 2:
                        break;
                case 步骤 3:
                        break;
        }
}

void app_task(void)
{
        switch(步骤)
        {
                case 步骤 1:
                        break;
                case 步骤 2:
                        break;
                case 步骤 3:
                        break;
        }
}

void main(void)
{
        初始化 ();
        while(1)
        {
```

(continues on next page)

(continued from previous page)

```
                /*  底层驱动 */
                n_task();
                b_task();

                /* 应用流程 */
                app_task();
        }
}
```

但是，我要说的是：**大循环**，是有点反人类的。每一个任务（驱动流程），都要去管别人，自己怎么执行的，也会受到别人影响。要很小心的设计步骤，才能解决所有任务的运行矛盾。如果，一个程序，有复杂的 APP，跟底层的矛盾，就更难解决了。而且，复杂的应用层，很难改为大循环。

38.1.3 小循环代码架构

与大循环对应的，就是**小循环**。什么是小循环？每个功能，都写成一个 while(1)，就是小循环。在想用 while(1) 的地方，就用 while(1)，就是小循环。例如，我们在按键扫描驱动中，在 scan 函数中用 while(1) 一直扫描。代码框架如下：

```
void n_task(void)
{
        while(1)
        {
                switch(步骤)
                {
                        case 步骤 1:
                                break;
                        case 步骤 2:
                                break;
                        case 步骤 3:
                                break;
                }
        }
}

void b_task(void)
{
        while(1)
        {
                while(1)
```

(continues on next page)

(continued from previous page)

```
                {
                        if(条件符合)
                                break;
                }
                while(1)
                {
                        if(条件符合)
                                break;
                }
                while(1)
                {
                        if(条件符合)
                                break;
                }
        }
}

void app_task(void)
{
        while(1)
        {
                while(1)
                {
                        if(条件符合)
                                break;
                }
                while(1)
                {
                        if(条件符合)
                                break;
                        else
                        {
                                while(1)
                                {
                                        等某种状态
                                                break;
                                }
                        }
                }
        }
```

(continues on next page)

(continued from previous page)

```
}

void main(void)
{
        初始化 ();
        while(1)
        {
                /*  底层驱动 */
                n_task();
                b_task();

                /* 应用流程 */
                app_task();
        }
}
```

这样每个驱动都是一个死循环，程序怎么跑呢？肯定跑不起来。嵌入式 OS，就是让各个死循环跑起来的代码。

每个 TASK 我们就叫做任务，OS，就是实现多任务运行。在每个 task 里面，可以随便按照你自己的需求写多个小循环。

38.1.4 基本原理

RTOS 多任务是怎么实现的呢？如果你学过汇编，可能就容易理解。首先，如何打断代码执行？中断，对，就是中断。每一个 RTOS 都会有一个定时器中断，我们叫做系统滴答，例如，系统滴答是 2ms，那么就是 2ms 就会产生一次定时中断，这个中断打断了正在执行的程序。在这个定时中运行 RTOS 的管理代码，这时，控制权在 RTOS 手上，想怎么搞就怎么搞。普通的中断一般会返回原来的程序位置运行。系统滴答中断，故意不回到原来的地方，而是切到其他地方运行，那么，就完成了一次任务切换。为了回到上次的任务，就需要将上次的运行状态记住。等需要的时候，就可以继续执行。

任务管理，是 RTOS 的最基本功能。为了支持多任务，一个 RTOS 还需要其他功能，例如任务间通信的各种手段：互斥锁，邮箱，信号等。

38.2 FreeRtos

很早之前，说到 RTOS，都是说 UCOS。其他，RTOS 有很多很多：UCOS、freertos、rtems。。。。以前 UCOS 流行，因为他说开源的。但是，其实 UCOS 开源不免费，商业使用是需要授权费的。而 freertos，是免费的。现在物联网兴起，很多 zigbee 和 wifi 芯片都选择免费的 freertos。所以这几年 freertos 爆发了。我们的代码都是免费的，我们当然选择免费的 freertos。

百度百科

> FreeRTOS 是一个迷你的实时操作系统内核。作为一个轻量级的操作系统，功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能、软件定时器、协程等，可基本满足较小系统的需要。由于 RTOS 需占用一定的系统资源 (尤其是 RAM 资源)，只有 μC/OS-II、embOS、salvo、FreeRTOS 等少数实时操作系统能在小 RAM 单片机上运行。相对 μC/OS-II、embOS 等商业操作系统，FreeRTOS 操作系统是完全免费的操作系统，具有源码公开、可移植、可裁减、调度策略灵活的特点，可以方便地移植到各种单片机上运行，其最新版本为 10.0.1 版。

官网：https://www.freertos.org/

38.3 移植 FreeRtos

如何移植？规范的方法：

1. 看 freertos 的程序包，参考他的移植。一个程序要推广，官方会移植到很多平台。freertos 的 demo，有 170 多个芯片的范例，总有一个适合你。
2. 看平台的例程，我们可以从 ST 官网找到 freertos 的移植范例。

我们更多时候，是网络搜索别人移植好的。

下面我们不管参考哪个例程，只说说移植需要考虑的一些细节问题。

1. 移植前，把所有 freertos 文件增加 ft 前缀。

我们已经移植了 LWIP，有一些源码文件和 freertos 重名。但是我的程序是在所有硬件都编写了驱动之后才移植 freertos，代码包含了 LWIP。

> 在移植过程出现了很多错误，例如

```
..\Utilities\FreeRTOS\Source\timers.c(76): error:  #20: identifier
"TimerCallbackFunction_t" is undefined
          TimerCallbackFunction_t        pxCallbackFunction;
                /*<< The function that will be called when the timer expires. */
..\Utilities\FreeRTOS\Source\timers.c(104): error:  #20: identifier
"PendedFunction_t" is undefined
          PendedFunction_t        pxCallbackFunction;
                /* << The callback function to execute. */
..\Utilities\FreeRTOS\Source\timers.c(220): error:  #20: identifier
"TimerCallbackFunction_t" is undefined
 ␣
↪                                                                    TimerCallbackFunction_
↪t pxCallbackFunction,
..\Utilities\FreeRTOS\Source\timers.c(279): error:  #20: identifier
"TimerHandle_t" is undefined
```

但是这些定义命名全部都在 timers.h 里面有定义。点开工程 timer.c，查看里面的头文件，你会发现竟然包含的是 LWIP 的头文件。头文件包含错误为了避免更多麻烦，决定在 freertos 所有头文件加上 ft 前缀。

还有就是我们选择屏蔽下面代码 FreeRTOSConfig.h

```
//#define vPortSVCHandler SVC_Handler
//#define xPortPendSVHandler PendSV_Handler
//#define xPortSysTickHandler SysTick_Handler
```

而是直接将这些系统用的函数塞到 stm32f4xx_it.c 文件中的中断内。1 保持所有中断响应都在这个文件内处理 2 我们要用与 freertos 系统无关的延时函数 Delay。3 freeRTOS 通常会将 systick 定义 10 毫秒，比较粗，我们可能需要将系统滴答精确到毫秒。

同时在 FreeRTOSConfig.h 文件最后定义，也就是内存分配用我们自己的，不用 freertos 提供的。

```
#define pvPortMalloc wjq_malloc
#define vPortFree        wjq_free
```

尽快启动任务，这样就可以将 startup_stm32f40_41xxx 里面定义的堆跟栈设置为最少，甚至设置为 0

38.4 总结

38.5 end

CHAPTER 39

简易菜单

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

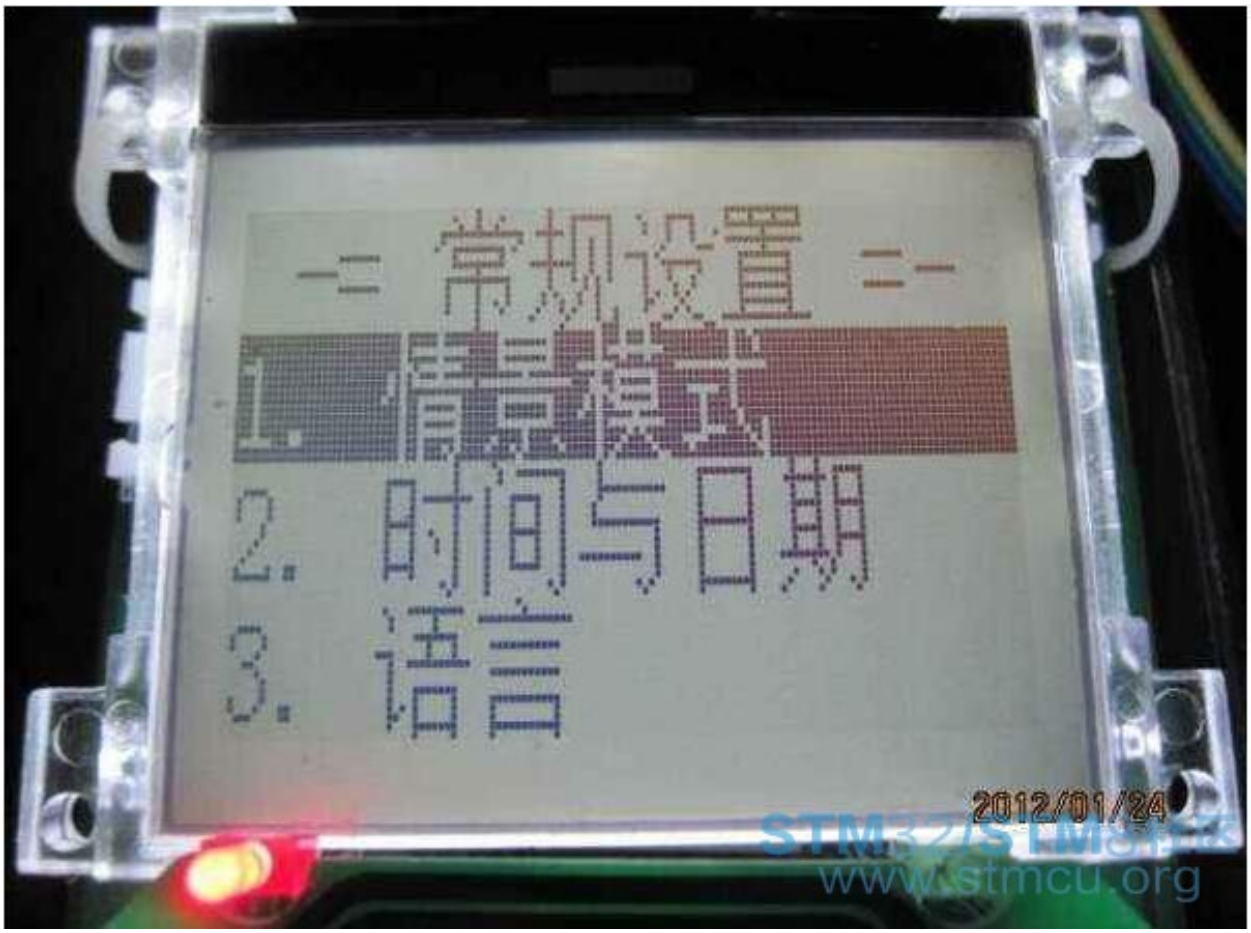
淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

声明：本处所说的菜单是用在 128*64 这种小屏幕的菜单，例如下面这种，不是彩屏上的 GUI。



作为一个底层驱动工程师，驱动写完了，是要写硬件测试程序的。这个测试程序，一般给测试部/硬件工程师用来测试硬件，也会给工厂产线测试准成品。

开始的人偷懒，不想一秒就直接上，所有菜单都这样做，一层套一层

```
void test_main(void)
{
        while(1)
        {
                get_key(&key);
                switch(key)
                {
                        case 1:
                                test_key();
                                break;
                        case 2:
                                test_lcd();
                                break;
                        ....
```

(continues on next page)

(continued from previous page)

```
                }
        }
}
```

当菜单越来越多，就开始纠结了，这样写维护不便，看起来也不美，还浪费程序空间。

作为一个天天看《编程之美》的码农，决定改变现状。酷狗百度一番，找到了两个参考：《基于二叉树的多层的液晶菜单界面设计》《基于节点编号的通用树状菜单设计方法与实现.pdf》按照他们的设计方法，鼓捣了一个版本，能用，挺好，但是也纠结。因为他们用了树这种数据结构。对于程序运行来说，非常好，效率高。但是对于我来说，菜单代码是一次性的，但是菜单内容，却是会经常改的。让我用人脑去维护一个包含几十个上百个菜单的树，不容易。

想来想去，这些菜单到底有什么不好？对于我来说，为什么不好用？得出下面结论：

1. 管得太宽菜单，你就管菜单切换就行了，到了最低一层，也就是实际的测试功能，就不要管了。菜单切换是类似的，实际测试都是不同的。比如在菜单中，按键 1，是进入第一个菜单。但是在测试中，按键 1，功能都不一样。如果菜单连这个也要管，相同动作功能太多，无法进行统一抽象，就很难模块化。
2. 出发点不一样上面说到的菜单，出发点都是如何设计一个好的菜单数据结构，让程序快速，高效运行。我想要的却是一个容易维护的菜单结构，至于菜单的代码有多乱多纠结，没关系，而且，几百上千个菜单，就算用轮询的方法，也不过几百 us 吧，没关系。

根据需求，我重新设计了一个菜单结构体

```
/**
 * @brief  菜单对象
*/
typedef struct _strMenu
{
    MenuLel l;      ///< 菜单等级
    char cha[MENU_LANG_BUF_SIZE];   ///中文
    char eng[MENU_LANG_BUF_SIZE];   ///英文
    MenuType type;  ///菜单类型
    s32 (*fun)(void);  ///测试函数

} MENU;
```

是的，就这么简单，每一个菜单都是这个结构体用这个结构体填充一个列表，就是我们的菜单了

```


const MENU EMenuListTest[]=
{
        MENU_L_0,//菜单等级
        "测试程序",//中文
```

(continues on next page)

(continued from previous page)

```
        "test",         //英文
        MENU_TYPE_LIST,//菜单类型
        NULL,//菜单函数，功能菜单才会执行，有子菜单的不会执行

                MENU_L_1,//菜单等级
                "LCD",//中文
                "LCD",          //英文
                MENU_TYPE_LIST,//菜单类型
                NULL,//菜单函数，功能菜单才会执行，有子菜单的不会执行
                        MENU_L_2,//菜单等级
                        "VSPI OLED",//中文
                        "VSPI OLED",          //英文
                        MENU_TYPE_FUN,//菜单类型
                        test_oled,//菜单函数，功能菜单才会执行，有子菜单的不会执行

                        MENU_L_2,//菜单等级
                        "I2C OLED",//中文
                        "I2C OLED",          //英文
                        MENU_TYPE_FUN,//菜单类型
                        test_i2coled,//菜单函数，功能菜单才会执行，有子菜单的不会执行

                MENU_L_1,//菜单等级
                "声音",//中文
                "sound",          //英文
                MENU_TYPE_LIST,//菜单类型
                NULL,//菜单函数，功能菜单才会执行，有子菜单的不会执行
                        MENU_L_2,//菜单等级
                        "蜂鸣器",//中文
                        "buzzer",          //英文
                        MENU_TYPE_FUN,//菜单类型
                        test_test,//菜单函数，功能菜单才会执行，有子菜单的不会执行

                        MENU_L_2,//菜单等级
                        "DAC 音乐",//中文
                        "DAC music",          //英文
                        MENU_TYPE_FUN,//菜单类型
                        test_test,//菜单函数，功能菜单才会执行，有子菜单的不会执行

                        MENU_L_2,//菜单等级
                        "收音",//中文
```

(continues on next page)

(continued from previous page)

```
                    "FM",        //英文
                    MENU_TYPE_FUN,//菜单类型
                    test_test,//菜单函数，功能菜单才会执行，有子菜单的不会执行

                MENU_L_1,//菜单等级
                "触摸屏",//中文
                "tp",        //英文
                MENU_TYPE_LIST,//菜单类型
                NULL,//菜单函数，功能菜单才会执行，有子菜单的不会执行

                    MENU_L_2,//菜单等级
                    "校准",//中文
                    "calibrate",        //英文
                    MENU_TYPE_FUN,//菜单类型
                    test_cal,//菜单函数，功能菜单才会执行，有子菜单的不会执行

                    MENU_L_2,//菜单等级
                    "测试",//中文
                    "test",        //英文
                    MENU_TYPE_FUN,//菜单类型
                    test_tp,//菜单函数，功能菜单才会执行，有子菜单的不会执行

                MENU_L_1,//菜单等级
                "按键",//中文
                "KEY",        //英文
                MENU_TYPE_FUN,//菜单类型
                test_key,//菜单函数，功能菜单才会执行，有子菜单的不会执行

        /* 最后的菜单是结束菜单，无意义 */
        MENU_L_0,//菜单等级
        "END",//中文
        "END",        //英文
        MENU_TYPE_NULL,//菜单类型
        NULL,//菜单函数，功能菜单才会执行，有子菜单的不会执行
};
```

这个菜单列表有什么特点和要求呢？1 需要一个根节点和结束节点 2 子节点必须跟父节点，类似下面结构

```
-----------------------------------------------
根节点
```

(continues on next page)

(continued from previous page)

```
        第 1 个 1 级菜单
                        第 1 个子菜单
                        第 2 个子菜单
                        第 3 个子菜单
        第 2 个 1 级菜单
                        第 1 个子菜单
                                        第 1 个孙菜单
                                        第 2 个孙菜单
                        第 2 个子菜单
                        第 3 个子菜单
        第 3 个 1 级菜单
        第 4 个 1 级菜单
        第 5 个 1 级菜单
结束节点
-------------------------------------------------
```

第 2 个 1 级菜单有 3 个子菜单，子菜单是 2 级菜单，其中第 1 个子菜单下面又有 2 个孙菜单（3 级菜单）。

维护菜单，就是维护这个列表，添加删除修改，非常容易。那菜单程序怎么样呢？管他呢。定义好菜单后，通过下面函数运行菜单，

```
emenu_run(WJQTestLcd, (MENU *)&WJQTestList[0], sizeof(WJQTestList)/sizeof(MENU), FONT_
↪SONGTI_1616, 2);
```

-第 1 个参数是在哪个 LCD 上显示菜单，-第 2 个是菜单列表，-第 3 个是菜单长度，-第 4 个四字体，-第 5 则是行间距

注意：运行这个菜单需要有 rtos，因为菜单代码是 while(1) 的，陷进去就不出来了。需要有其他线程 (TASK) 维护系统，例如按键扫描。

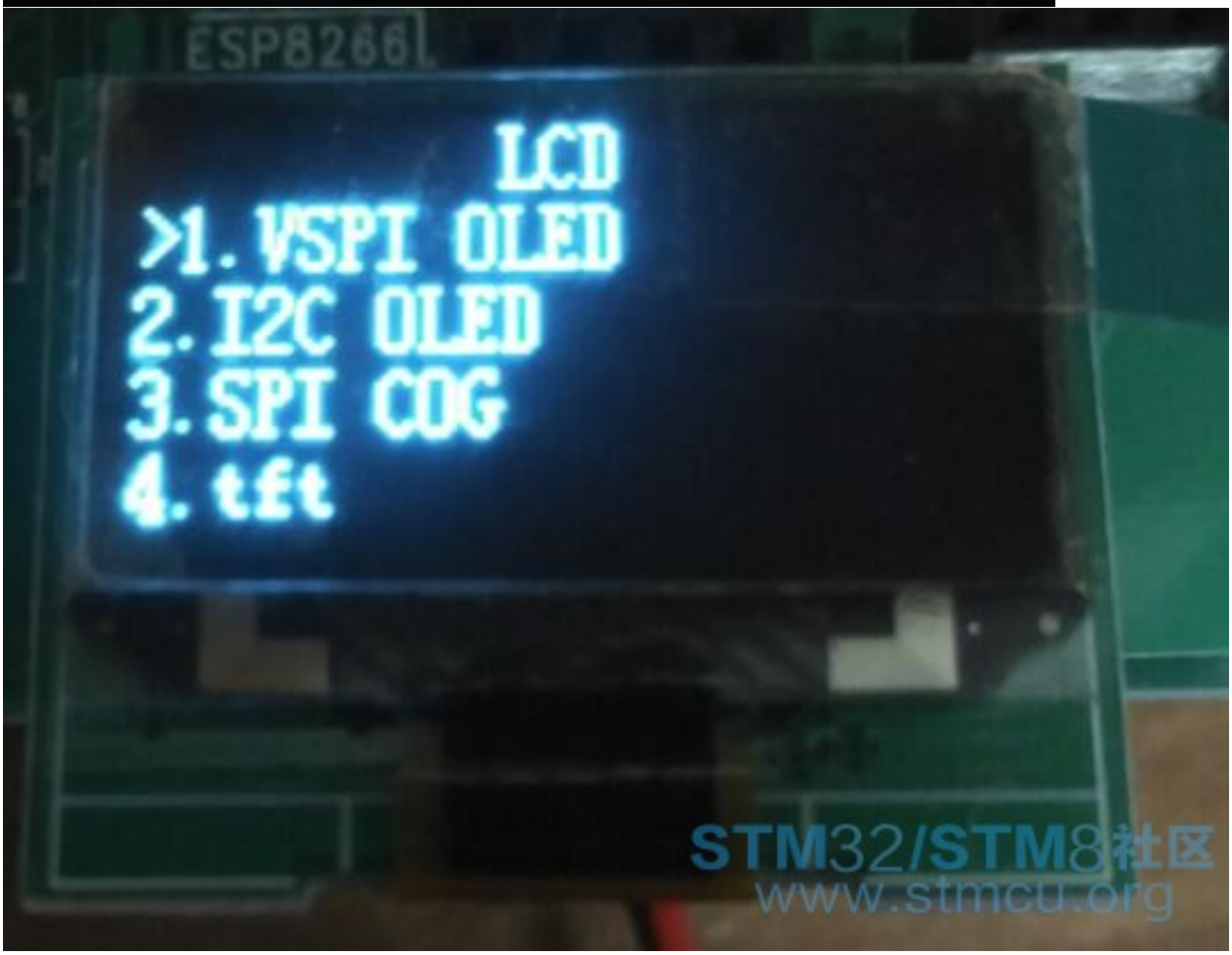
代码托管在 github：https://github.com/wujique/stm32f407/tree/sw_arch 相关文件：emenu.c、emenu.h、emenu_test.c

当前代码：1 实现了双列菜单，用数字键选择进入下一层。每页最多显示 8 个菜单（4*4 键盘用 1-8 键）2 实现了单列菜单，通过上下翻查看菜单，确认键进入菜单。3 天顶菜单未实现，谁有兴趣可以加上。3 基于 LCD 驱动架构，这个简易菜单自适应于多种 LCD。

效果如下，有需要的尽管拿去，不用谢。

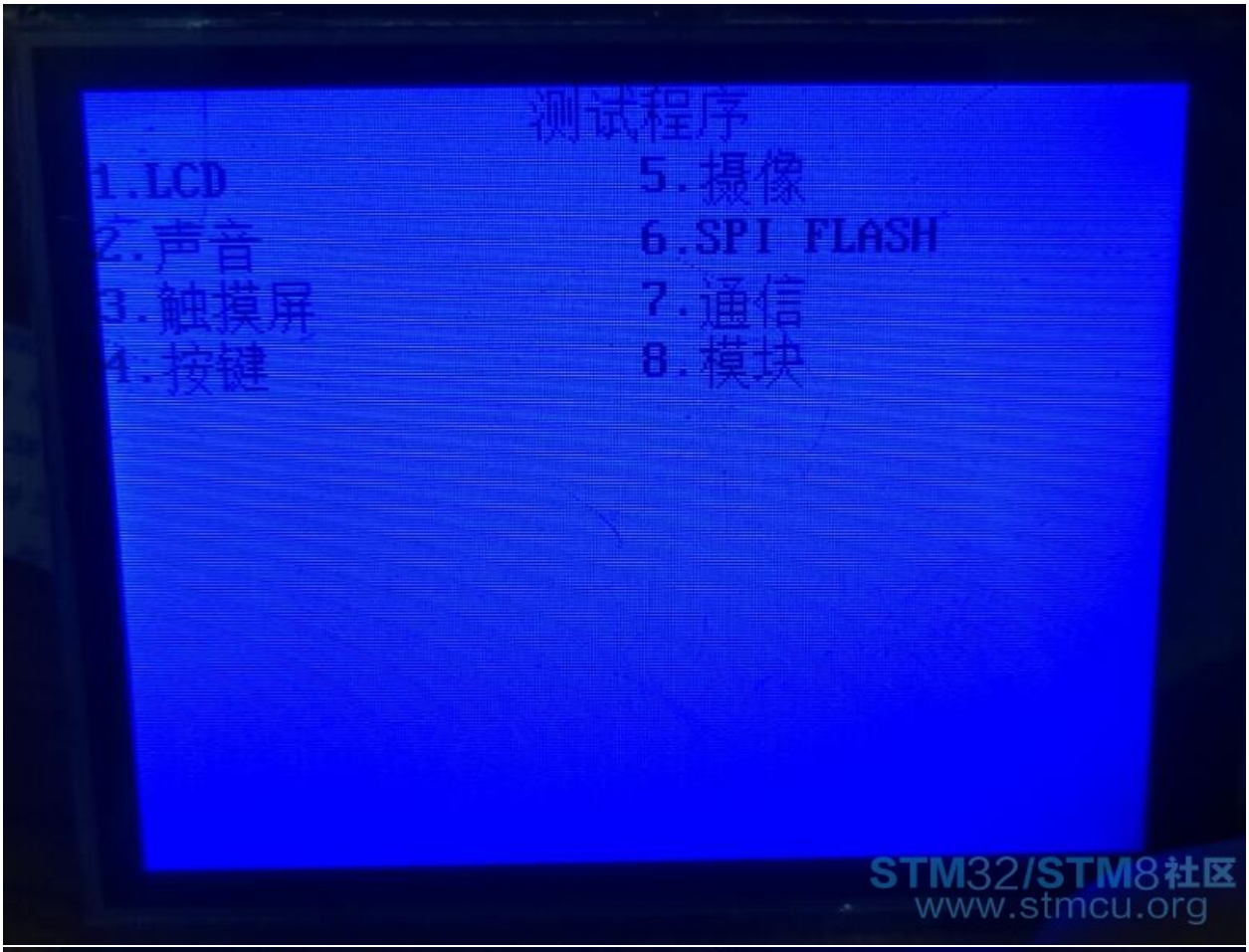
39.1 显示效果

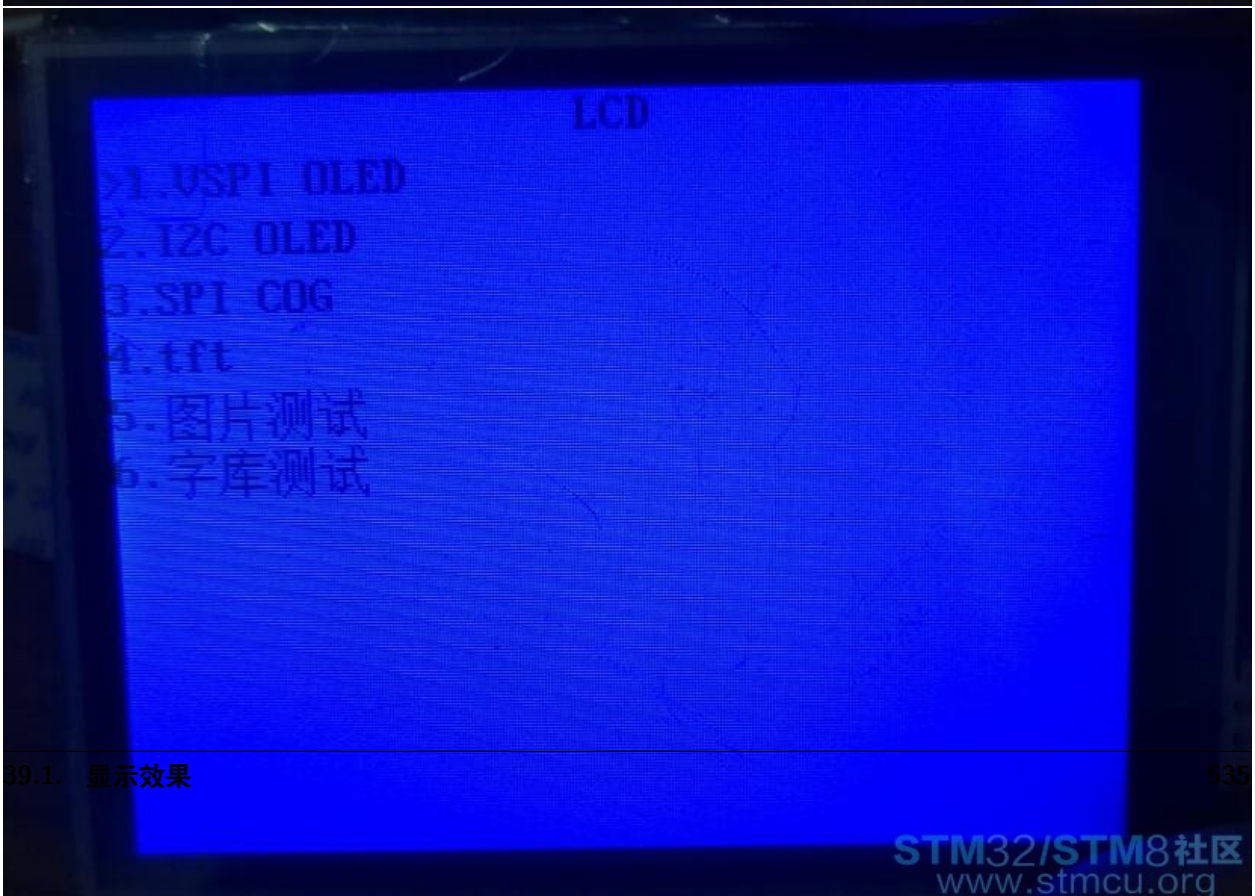
39.1.1 128*64 OLED



39.1.2 128*128 tft lcd

39.1.3 320*240 tft lcd





39.2 总结

类似菜单在我开发的产品上已经推广使用。经测试，可以明显减少测试程序代码量，节省程序空间。并且易于修改和维护。

39.3 end

CHAPTER 40

系统测试程序

够用的硬件

能用的代码

实用的教程

屋脊雀工作室编撰 -20190101

愿景：做一套能用的开源嵌入式驱动（非 LINUX）

官网：www.wujique.com

github: https://github.com/wujique/stm32f407

淘宝：https://shop316863092.taobao.com/?spm=2013.1.1000126.2.3a8f4e6eb3rBdf

技术支持邮箱：code@wujique.com、github@wujique.com

资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

QQ 群：767214262

待补充完善

在提供底层驱动程序给 APP 使用前，需要完成底层测试。驱动测试程序一般由对应驱动人员编写，类似白盒测试。并且将部分测试程序用于硬件生产测试。主要有以下几点要求：

1. 测试程序要使用提供给 APP 的接口编写。
2. 要测试到软件接口的各种边界。

3. 测试硬件的步骤，测试案例要能充分暴露硬件问题。

下面我们对我们硬件的测试程序做简要说明。

40.1 1 Lcd

1. 彩屏 LCD，需要测试显示图像，仅仅显示三原色，不能保证并口是连接完好。

40.2 2 矩阵键盘

1. 在做生产测试时，要限制流程，不测试完按键不能直接退出，需要进一步确认退出才能退出。防止测试人员遗漏测试部分按键。

40.3 end

CHAPTER 41

版权说明

```
1 源码归屋脊雀工作室所有。
2 源码可以用于的其他商业用途（配套开发板销售除外），不须授权。
3 屋脊雀工作室不对代码功能做任何保证，请使用者自行测试，后果自负。
4 可随意修改源码并分发，但不可直接销售本代码获利，并且保留版权说明。
5 如发现 BUG 或有优化，欢迎发布更新。请联系：code@wujique.com
6 使用本源码则相当于认同本版权说明。
7 如侵犯你的权利，请联系：code@wujique.com
8 保留所有文档所有权利。
9 一切解释权归屋脊雀工作室所有。
```