基于 STM32 嵌入式开发入门

Release 0.0.1

Oct 05, 2021

关于本项目

本目录文件为 STM32F103VET 教学板相关文件

本教程发布于百度云与 github，

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

  目录 W108_F103_Tech

- github: https://github.com/wujique/stm32_tech

  在 github 上托管的仅仅是 doc 目录，其他资料请从百度云下载。

- gitee 镜像：https://gitee.com/hokgaai/stm32_tech

- 文档同步发布在：https://stm32-tech.readthedocs.io/zh_CN/latest/

- 视频教程：https://www.bilibili.com/video/av83166106?pop_share=1

CHAPTER 1

目录说明

本工程包含代码、资料、文档，目录说明如下。

CHAPTER 2

DOC 目录文档说明

`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，并按照目录结构链接，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

产品说明书

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

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

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

QQ 群：767214262

3.1 关于教学板

本教学板专门针对入门学习与实验室教学设计。

选择高配主控 `STM32F103VET`，片上含 `512K FLASH`、`64K SRAM`，不但能满足学习实验要求，还可支持复杂的综合应用实验。

VET 为 100 脚封装，充足 IO，减少 IO 复用，增加了外扩接口功能。跟低容量型号相比，VET 芯片支持 SDIO 和 FSMC 功能，可以做 TF 卡和 TFT LCD 实验。

硬件外设按照学习实验要求选择，满足循序渐进的教学要求，外扩接口标准化，兼容多种模块。

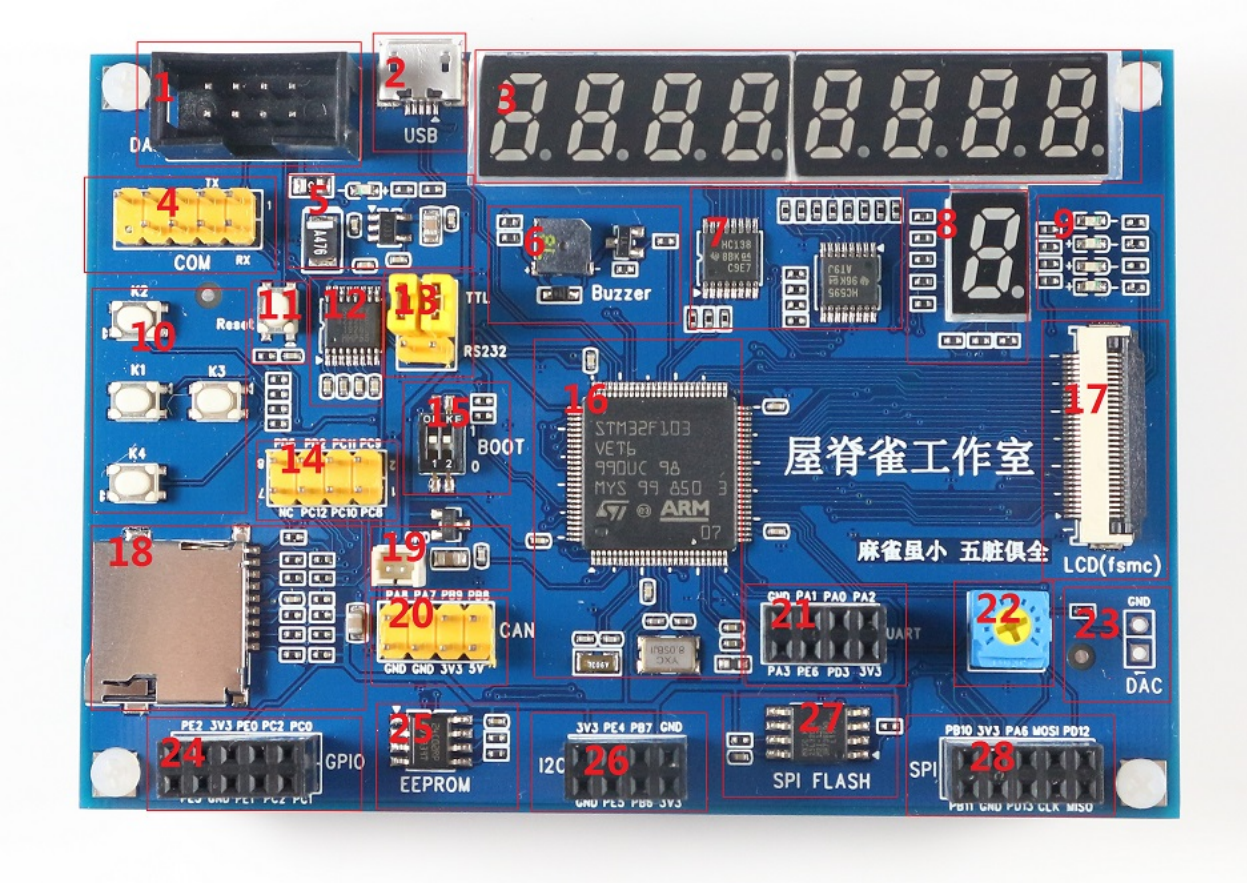
本教学板具有**配置强劲**、**简单易用**、**携带方便**、**扩展性强**、**性价比高**等特点，非常适合入门自学与实验室教学。

3.2 规格说明

工作电压：5V

尺寸：7 厘米 X10 厘米，板厚 1.6 毫米

3.3 模块说明

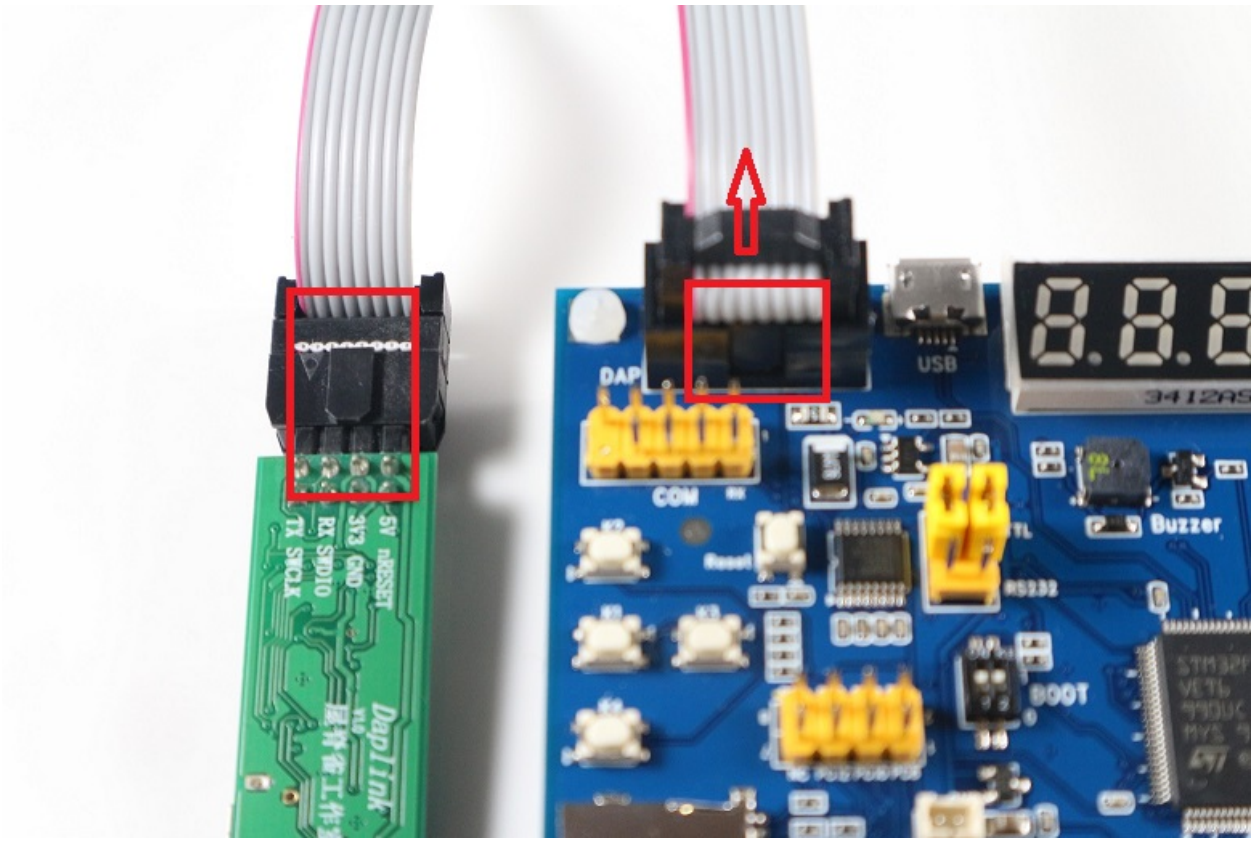


3.4 基本使用

3.4.1 DAP 接法

配套一根 2*4、**同向**、灰排线，插头凸出点在外的一端接开发板，凸出点在内的一端接 DAP，凸出点朝下。

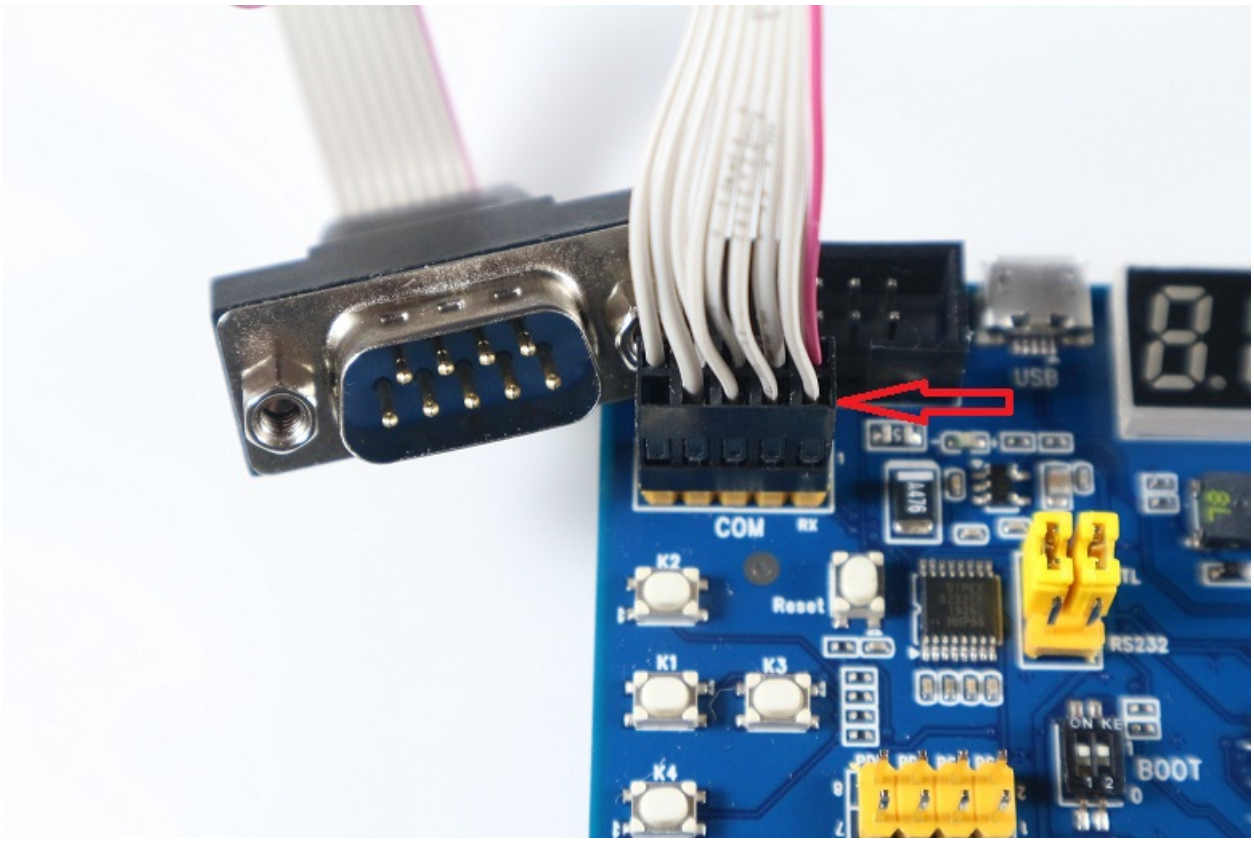
接法如下图



3.4.2 COM 口

配套有一根 DB9 公头灰排线。接线端子一端，红色为 1 脚，对应接开发板 1 脚。10 脚无线，对应接无插针的位置。

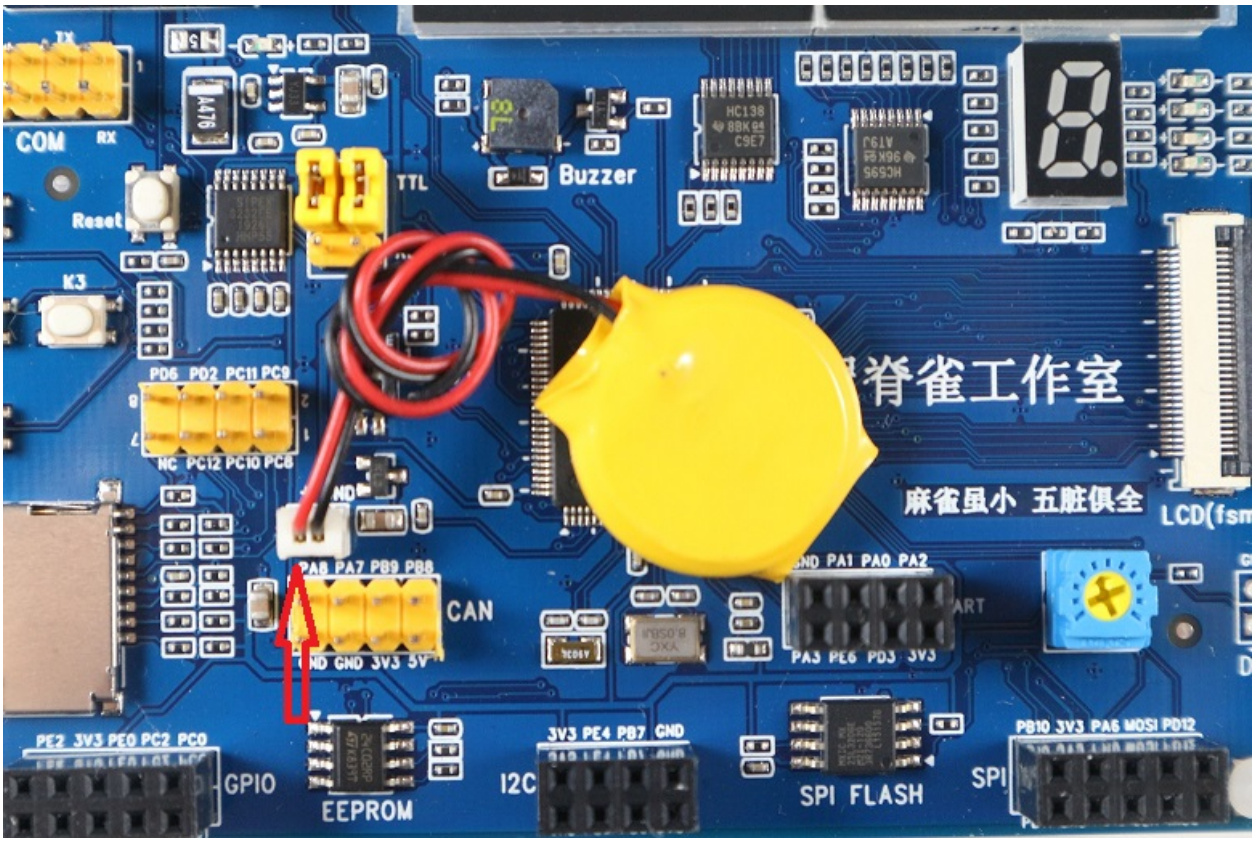
接法如下图



3.4.3 纽扣电池

纽扣电池通过插座连接，兼容电脑主板电池。红线为正极，接丝印 + 极的一边。

接法如下图

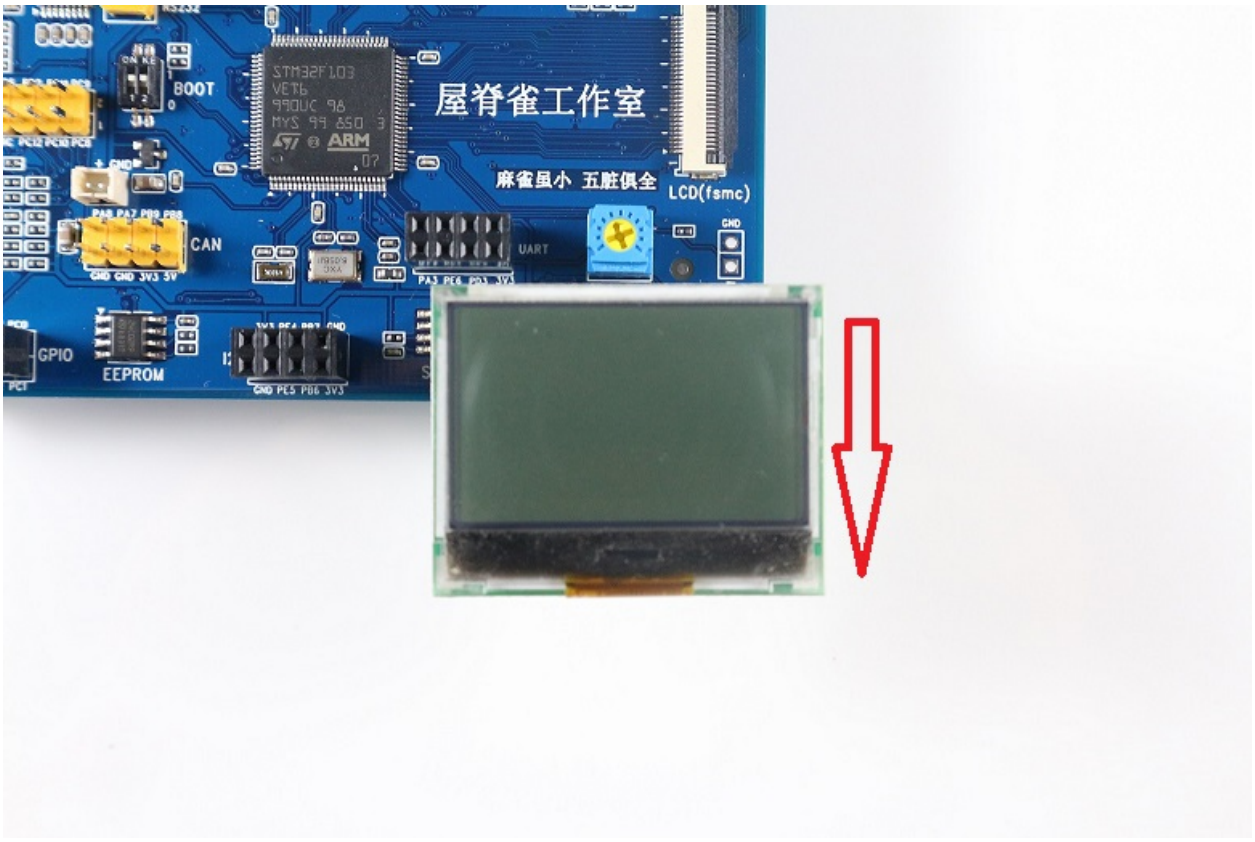


3.4.4 cog lcd

cog lcd 可直接插到外扩的 SPI 接口，使用硬件 SPI 控制器控制。也可以接到外扩 GPIO，使用 IO 模拟 SPI。

LCD 方向朝下，注意看电源与地的位置。

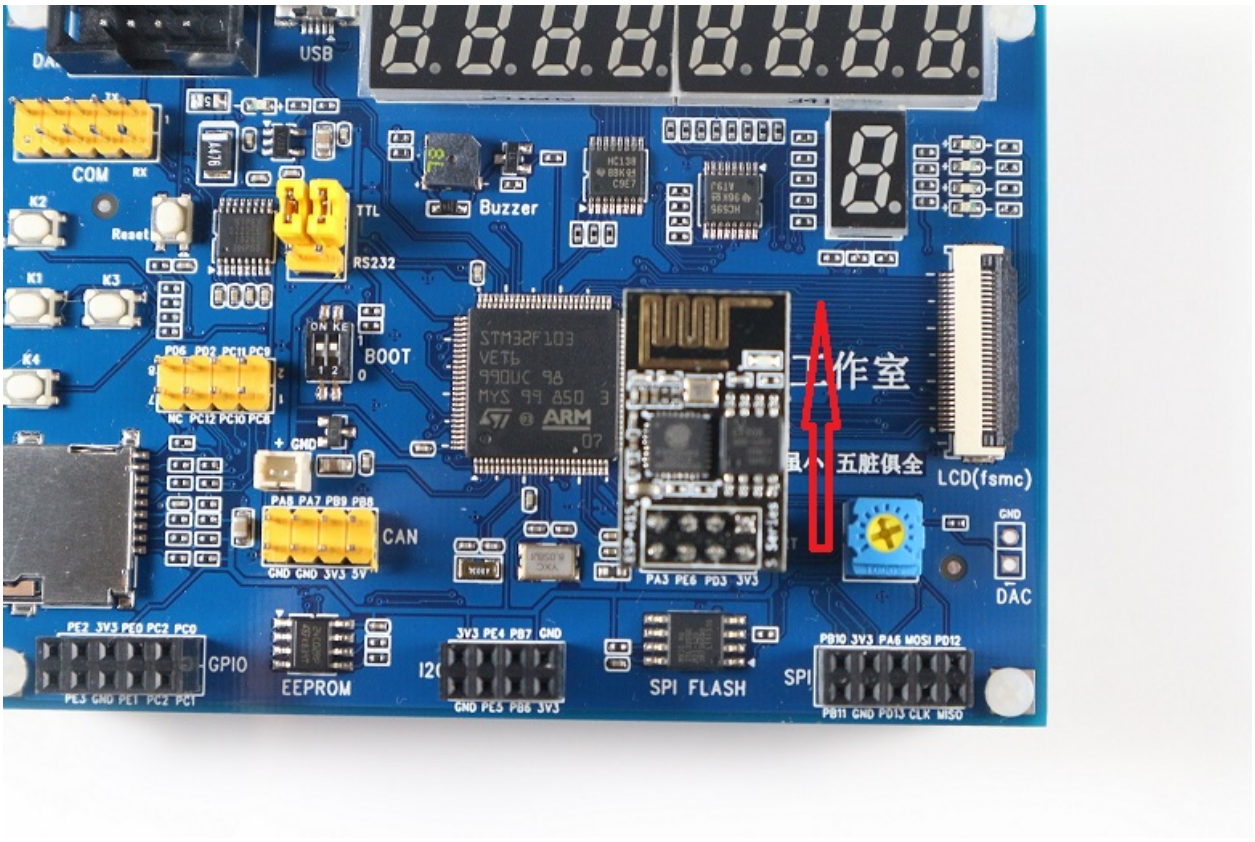
接法如下图



3.4.5 WIFI 模块

外扩串口兼容安信可 WIFI 模块，模组朝上插。

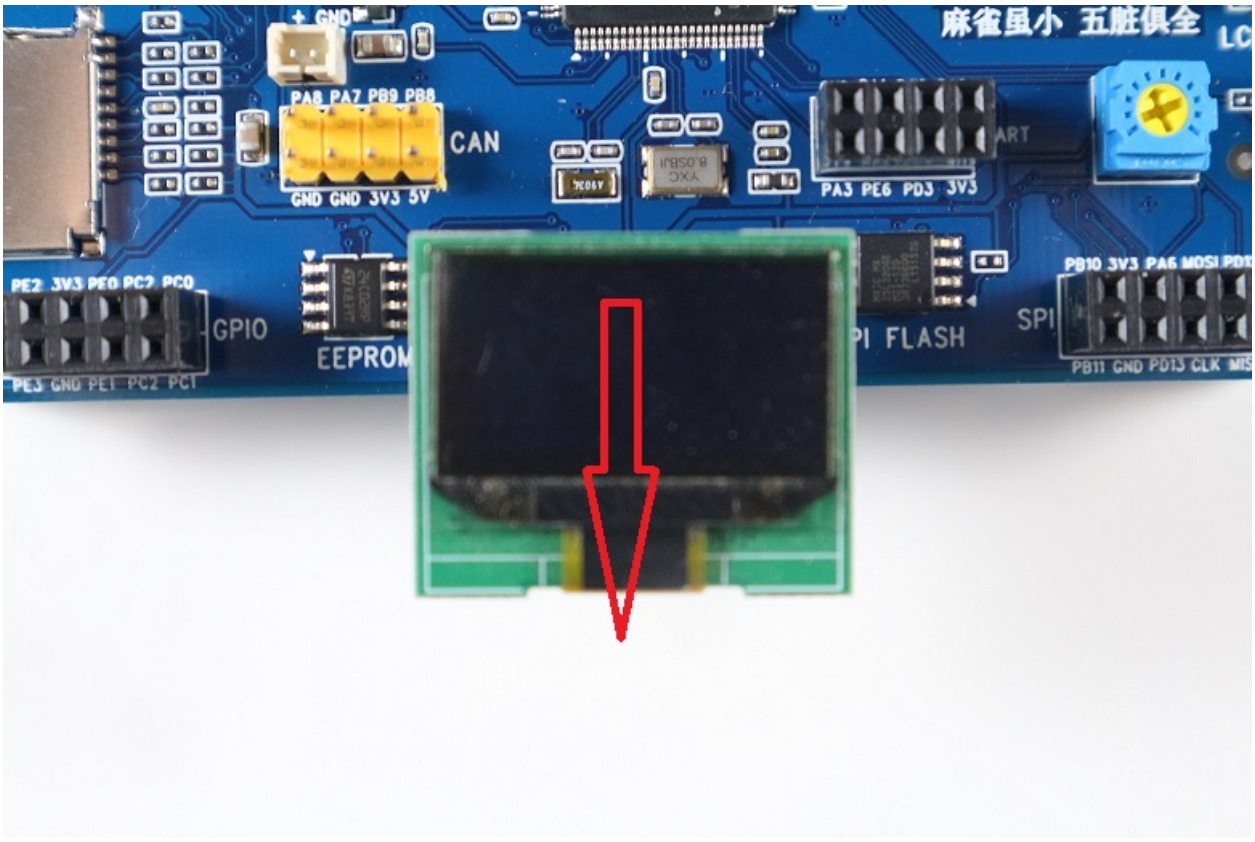
接法如下图



3.4.6 OLED(I2C)

外扩 I2C 接口可接屋脊雀 OLED LCD 模组。方向朝下。

接法如下图



3.4.7 OLED(spi)

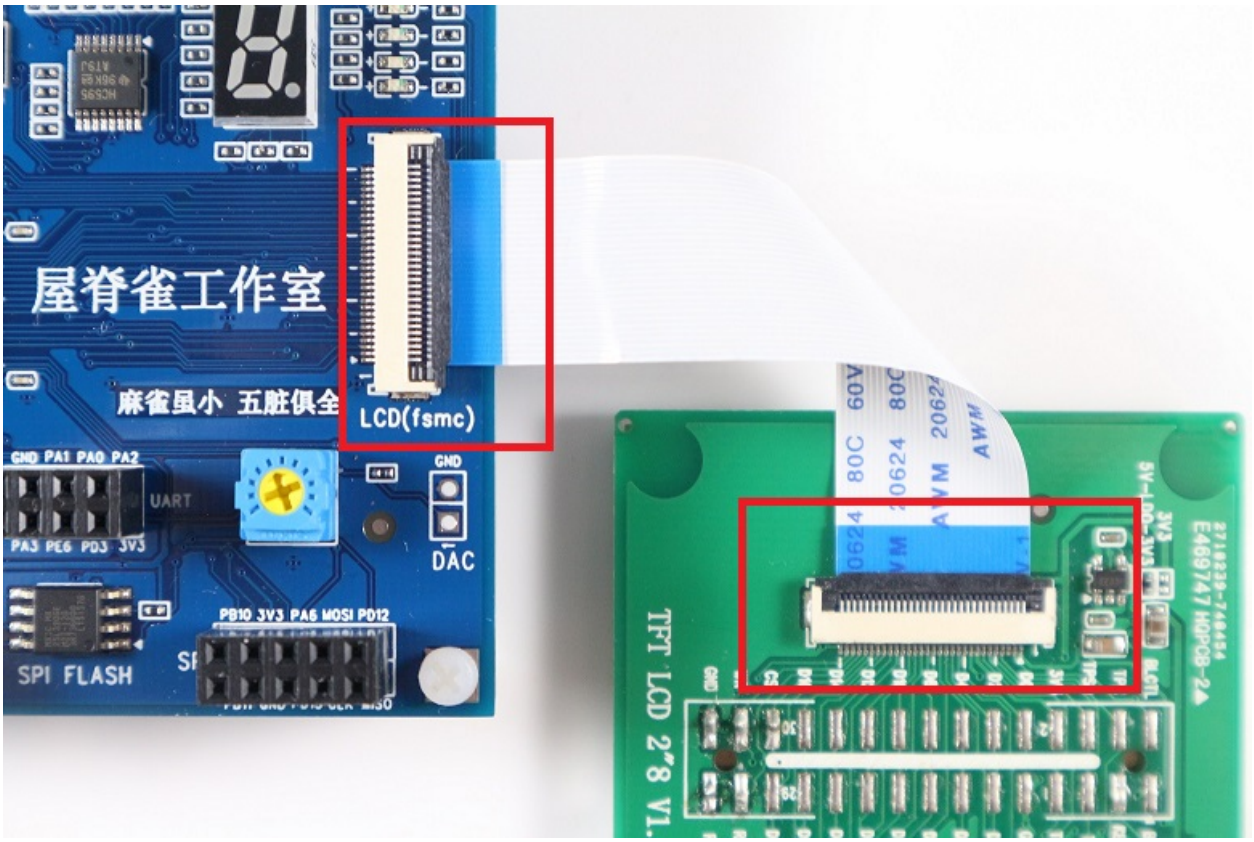
屋脊雀 OELD LCD 模组，可以通过跳线改为 SPI 接口。可以接在外扩的 SPI 或 GPIO 座子。

朝下靠右接（通过 3V3 和 GND 信号确认）

3.4.8 TFT LCD

屋脊雀 TFT LCD 使用 FPC 排线连接，开发板和模组的连接座皆是翻盖下接，也即是排线的金手指朝下。

接法如下图



3.4.9 BOOT 选择

开发板使用拨码开关控制 BOOT 选择。

拨动到上方位置为 1，下方位置为 0。

左边为 BOOT1，右边为 BOOT0。

BOOT0=0 时，在用户 FALSH 运行。

BOOT0=1，BOOT1=0，运行在系统储存器，也即是 ST 的 BOOT。

BOOT0=1，BOOT1=1，运行在 RAM。

20200109

end

CHAPTER 4

前言

欢迎大家参加学习本教程。

先跟大家说明做本教程的原因。

目前在网络上，基于 STM32 的学习教程非常丰富，也非常好。

那为什么还要浪费时间再做一个教程呢?

如果熟悉这些教程的朋友应该了解，这些教程对于入门了解概念，非常合适。

比如，我们要使用一个新外设，根本不知道这个外设是怎么工作的。通过网络上的这些教程，可以快速学会如何使用这个设备。

但是这些教程基本都有一个缺点：没有考虑实际项目工程开发的软件设计!

很多教程，只是官方例程的补充说明。没有考虑软件工程的事、没有考虑系统流程、没有考虑模块化设计、没有考虑接口设计、没有考虑分层设计。

基于这样的考虑，我设计了一套精巧的 STM32F407 开发板。这个开发板的外设、接口，都是基于上述问题设计：**如何模块化**、**如何设计接口**、**如何设计软件分层**。

在这套硬件上，最后形成了一个可复用的模块：`PetiteDrv`。

但是这套软件对于入门者非常不友善。因此决定再设计一个入门的开发板，做一些必要的入门教学。

目的是引导大家走向更专业的软件开发。

因此开发板的性价比极高，整体的设计也非常适合自学和实验室做实验。

编写的入门文档，有以下特点：

1 根据开发板，做 25 个教程。只为引导入门，更多的教程大家可以从其他地方学习。

2 从实用的目的，用实用的方法，教大家如何进行嵌入式开发，不照本宣科。

3 只挑有用的知识点讲解，不做大而全。没有几千页，只有几十页。因此很多知识点并不会涉及。

4 在讲入门知识之外，同时讲一些软件工程问题，引导大家软件设计意识。

4.1 学习说明

- 要不要学 51

  不是必须，学过当然好，有基础，入门更快。

- 要不要学汇编

  不是必须，学过当然后，如果有汇编知识，说明你对芯片底层的操作更加清楚。当遇到一些疑难问题时，你考虑问题就更加全面，也就更容易解决问题。

- 基础要求

  那 0 基础也能学吗?

  我不敢说学不会，但是建议最好有以下基础:

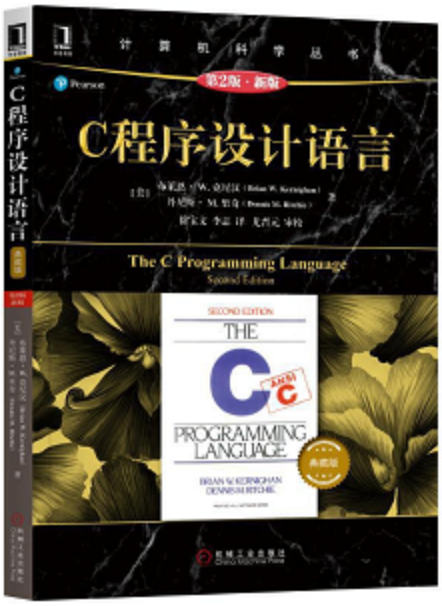
  模电：你总要知道电源、电压、电流、电阻，这些概念吧。

  数电：清楚 1 和 0 是啥、清楚一些常用的逻辑器件、清楚一些常规逻辑操作（与或非等）

  C 语言：程序使用 C 语言编写。

在 <W108_F103_Tech\ref\0 C 语言 > 目录中有一些我收集的 C 语言资料

C 语言推荐下面这本书:



C 语言进阶可读下面这本

4.2 教程说明

本教程分两部分

1. base
2. Advanced

当前教程如下：

提高教程

20200109

END

CHAPTER 5

资料介绍

本章节对本教程的文档进行必要说明。

5.1 开发板模块介绍

硬件介绍请参考 <W108_F103_Tech\doc\source\spec>、或者是 <W108_F103_Tech\data\SPEC> 目录中的 pdf 文档。

5.2 资料介绍

```
|
|--code
|--data
|        |--HDK
|        |--REF
|        |--SDK
|        |--SPEC
|--doc
|        |--build
|        |--source
|                |--spec
```

(continues on next page)

(continued from previous page)

```
|                 |--base
|                 |--Advanced
|--ref
|--readme.md
```

- Code 目录

  这个目录保存了教程的相关代码，全部为压缩包形式，在代码中有丰富的注释。

- data 目录

  这个目录是产品说明文档，包含 4 个子目录：

  HDK 是硬件原理图和丝印说明。REF 是主控相关资料。SDK 为 ST 提供的相关库。开发板产品说明书

- doc 目录

  教程文档。

  `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

- ref 目录

  本目录包含一些 Code 目录相关，或者教程相关的文档和资料。例如一些软件工具、一些从网络获取的文档资料等。

5.3 STM32 库说明

库有多重要?

当你有一定基础时，学习一款新芯片，最好的渠道就是库，就是库中包含的例程。

当你用一款新芯片做项目开发时，最好的资料就是库和例程，对于硬件的底层操作，完全可以拷贝官方例程的代码。我们专心做上层驱动和应用就行了。

可以说，ST 推出库的这个创意，大大方便了开发人员。

我们不用再关心一个外设寄存器的 bit0 是什么功能。我们只需要知道，这个外设有这样的功能，用库接口操作即可。

下面我们看看 STM32 的库到底是如何组成的。

- ST 有很多库，我们现在说的是标准外设库：`STM32F10x_StdPeriph_Lib_V3.5.0`

```
|--_htmresc
|--Libraries
|        |--CMSIS
|        |--STM32F10x_StdPeriph_Driver
|                |--inc
|                |--src
|--Project
|        |--STM32F10x_StdPeriph_Examples
|        |--STM32F10x_StdPeriph_Template
|--Utilities
|        |--STM32_EVAL
|                |--Common
|                |--STM32L152_EVAL
|                |--STM3210B_EVAL
|                |--STM3210C_EVAL
|                ...
|                |--stm32_eval.c
|                |--stm32_eval.h
```

- Libraries

  库文件目录，所谓的库就是用 c 语言将底层操作封装，用户不用再关心寄存器级别的操作。

  包含两部分：

  1. `CMSIS`, 这是内核相关的。
  2. `STM32F10x_StdPeriph_Driver`，这是芯片相关的。在文档中包含了所有的外设操作。src 是源文件，Inc 是 include 的缩写，文件夹中的文件是头文件。

- Project

  例程和模板。

  Examples 中的就是官方例程，对于我们开发程序有非常重要的参考意义。

- Utilities

不同官方开发板的接口封装。什么意思呢?

在 Examples 中有很多例程，官方有很多开发板，那么是不是每个开发板都写一个例程呢?

当然不是，例如，一个操作 SPI FLASH 的例程，操作 SPI FLASH 的程序，在不同的开发板都是一样的，那么这些代码就放在 Examples 目录下。

操作 FLASH 需要 SPI 接口，需要 IO 口，不同的芯片，不同的开发板，代码是有区别的。比如 F103 和 STM32Lxx .

这些不同的操作，就放在 Utilities 目录下。

我们参考一个例程，除了要看 Examples 中的代码，还要看 STM32_EVAL 目录下对应平台的代码。

我们的教学板是 STM32103VET，我们要参考的代码就在 STM3210E_EVAL 目录下。

当然，还有一些是所有平台都通用的，那么就放在 stm32_eval.c 源文件中

20200109

end

CHAPTER 6

最小系统

在将单片机的最小系统之前，我们先来认识认识单片机。

6.1 为什么叫单片机？

纵观计算机的发展，最开始的计算机是一台巨无霸，后面技术更新，变小了，我们称之为微机。

微机，也就是最开始的电脑，发展到目前，电脑通常包含以下几个部件：

1. CPU
2. 主板
3. 硬盘
4. 显示器
5. 键盘鼠标
6. 内存

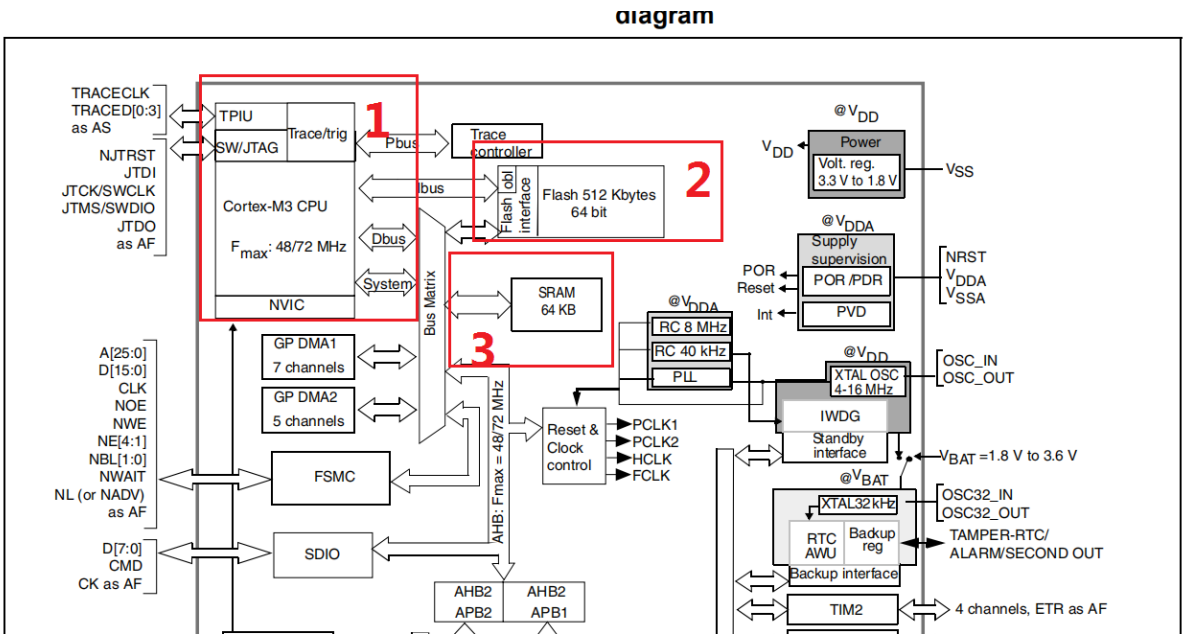
其中主板有包含了一堆外设：

那么单片机，也就是所谓的 MCU，其实就是跟电脑相比。

因为在一颗芯片上，包含了 CPU，硬盘、内存、一堆外设，就像一台小电脑一样。所以叫单片机。

我们看看 STM32F103VET 的框图。在文档中。

如下：



- 红框 1 中就是内核 Cortex-M3，功能相当于电脑上的 CPU。
- 红框 2 中是 Flash，我们选的这款芯片片上包含 512K。相当于电脑的硬盘，用于保存代码和数据。
- 红框 3 中的是 SRAM，片上含 64K。相当于电脑的内存。

  有些同学可能觉得这个比喻不是很恰当，确实，现在的芯片有很多种类，有些芯片内部没有 *FLASH*，有些芯片不含 *RAM*。

  不过大家可以搜索一下苹果最开始的电脑，这些早起电脑就是用 6052 内核的芯片做的，现在在看功能其实相当简单，就像一个单片机一样。

  6052 的内核，在很多台系的单片机上还在使用，就像 8051 内核一样，持久不衰。

6.2 最小系统说明

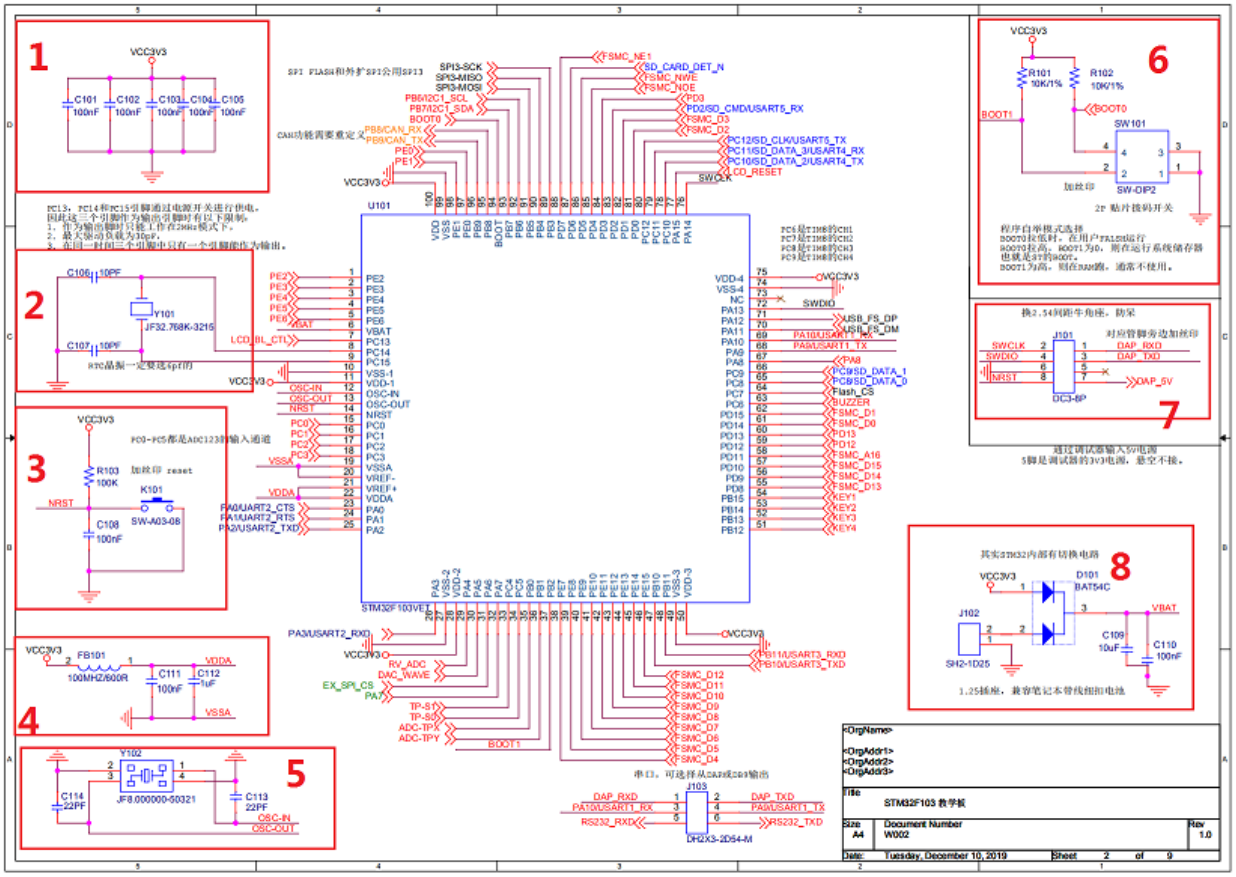
单片机能工作的最小外围电路就叫做最小系统。

通常包含：

1. 时钟，也就是晶振。时钟相当于芯片的心脏，只有时钟，芯片才能工作。
2. 工作电源，这个不用解释了，所有电子设备都需要电源才能工作。不同的芯片工作电源不一样，有 5V 系统，3.3V 系统，1.8V 系统。
3. 其他一些必要的外部设置，例如启动模式选择。

6.2.1 原理图

我们打开原理图，第一页，就是最小系统。居中的就是 STM32F103VET6。



细分的话，整个最小系统分一下部分：

- 电源

  电源又分 3 部分，主电源，也就是上图中框图 1，实际就是一个 3.3V 电源和一些电容。

  模拟电源，红框 4 中的电源就是，模拟电源由主电源通过磁珠等手段隔离而得。

  备份电源，红框 8 就是，用处是，低功耗下维持 CPU 的 RTC 和备份区功能。使用一个二极管并联，有外电时使用外电 3.3V，没有外电时才使用纽扣电池电源。

- 主时钟，也就是 CPU 大部分外设工作的时钟。红框 5 的 8M 晶振电路就是主时钟。晶振电路通常都是由一个晶体和 2 个电容组成。晶体的频率就是主频。电容容值根据芯片设计，要匹配，否则晶体不起振。

- RTC 时钟，红框 2 处即是，频率是 32.768M。

- 复位电路，红框 3 处即是。用于复位 CPU，通常是低电平复位。

  上电时，阻容电路没充电，复位脚是低电平，CPU 进行复位，电容充满电后，复位脚是高电平，CPU 复位完成。这就是常说的上电复位过程。

  正常工作中，按下按键，将复位脚拉低到低电平，进行复位。

- 启动选择电路，红框 6 就是 STM32 的启动选择电路。

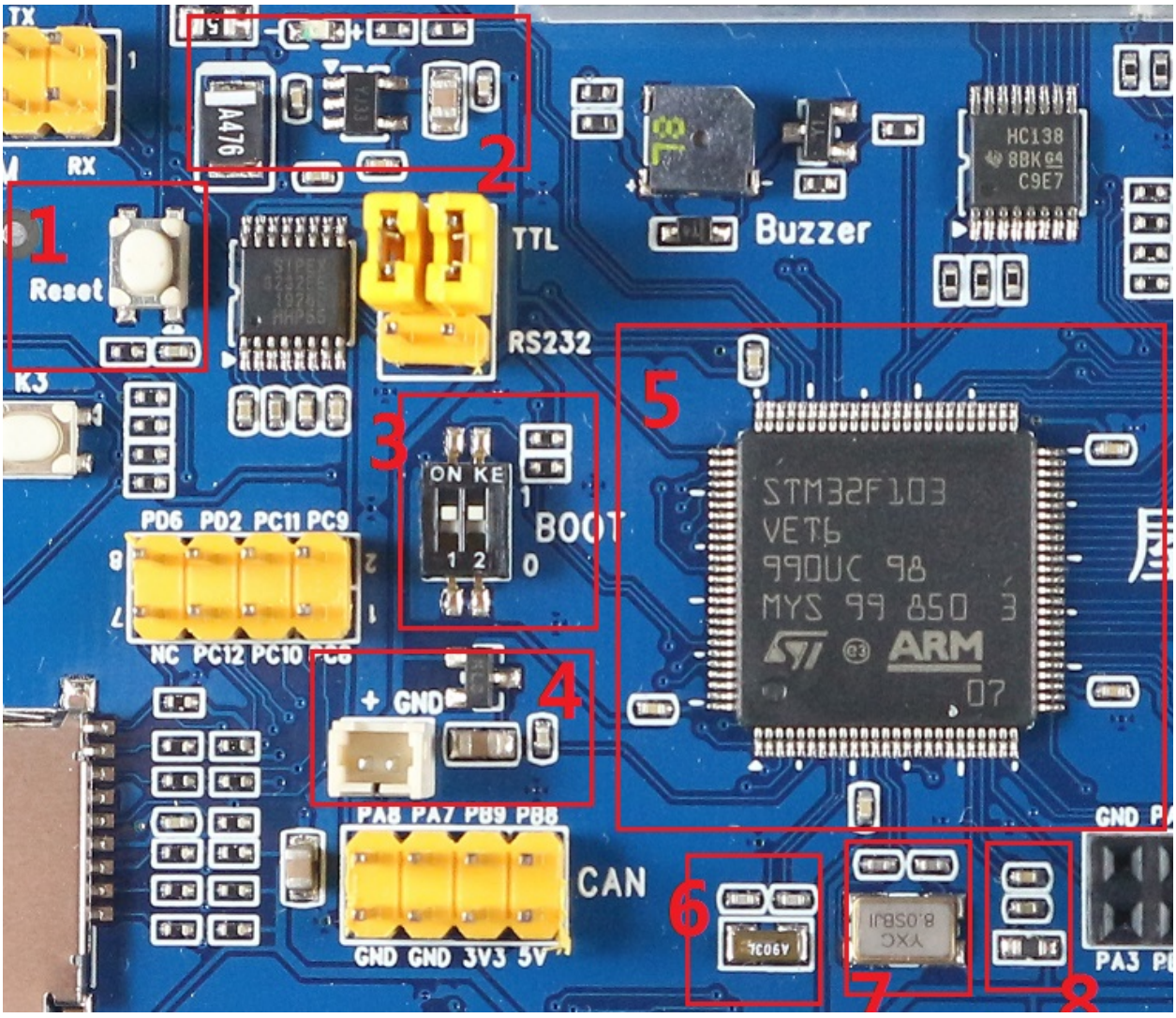
- 调试口，红框 7 处就是调试口。什么是调试? 通过电脑上的工具调试芯片。也就是我们常说的用 JLINK 通过 JTAG 口调试。现在，已经慢慢换成 Daplink+SW 口了，因为 SW 口用的线更少，更方便。

实际上，一个单片机，只需要：主电源、主时钟、复位电路，驱动选择电路，芯片就能运行了。

很多芯片甚至不需要主时钟，使用芯片内部的时钟就可以运行代码。STM32 也有内部时钟，只不过，相对外部晶振来说，内部时钟的精度会差很多。

6.2.2 硬件最小系统

我们看下硬件实物的最小系统。



1. 红框 1，是复位电路，按下复位按键系统就复位了。
2. 红框 2，是电源电路，将外部的 5V 转换为 3.3V。
3. 红框 3，是启动选择电路。
4. 红框 4，是纽扣电池电路。

5. 红框 5，是主芯片和电源电路，4 个电容要靠近芯片电源管脚。
6. 红框 6，是实时时钟。
7. 红框 7，是主时钟，8M 频率。
8. 红框 8，是模拟电源。

20200104

end

CHAPTER 7

建立工程

本章节教大家创建一个 STM32 的工程。

开发环境使用 MDK，也即是 Keil。

本文档只做概要说明，详细操作请参考视频教程

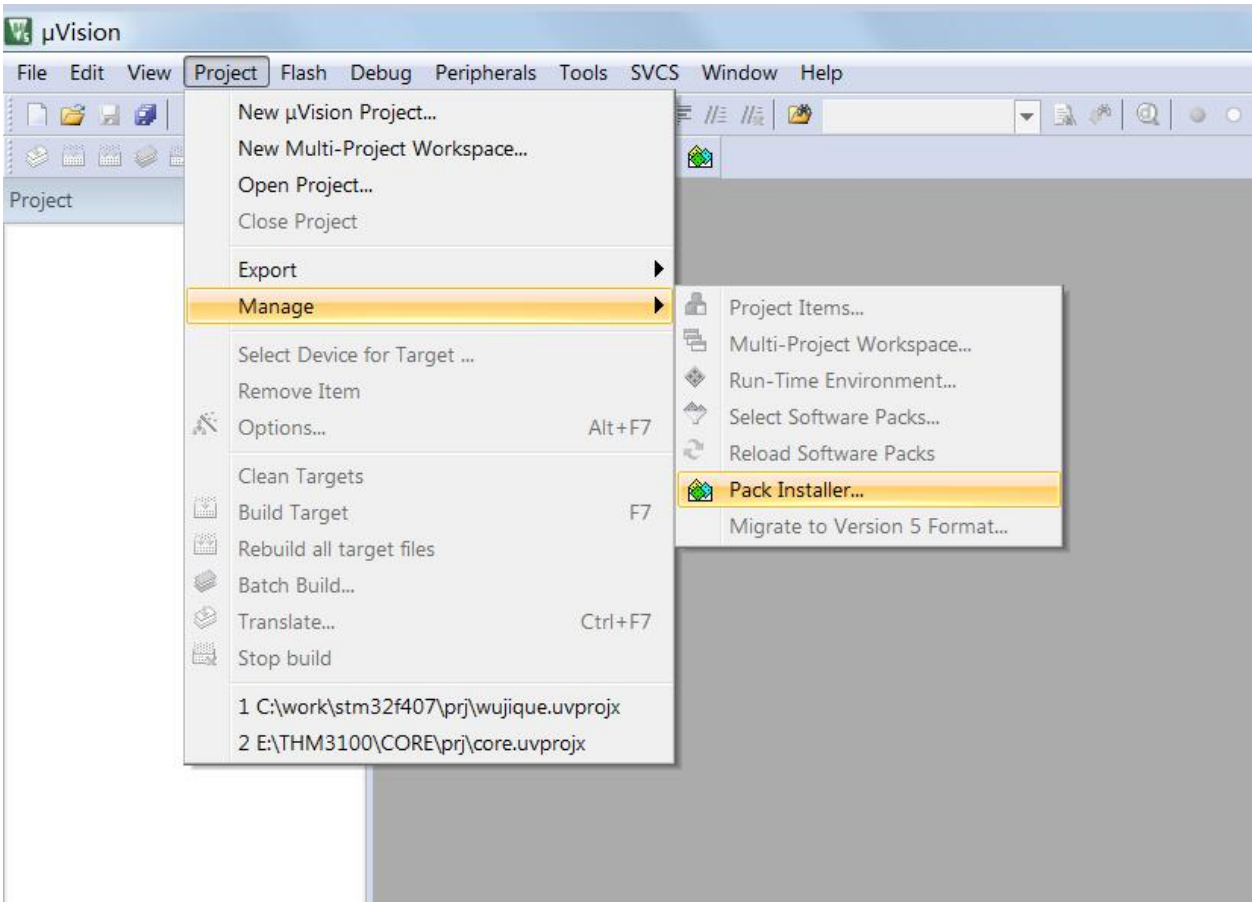
7.1 安装 MDK

自己网上寻找安装方法。

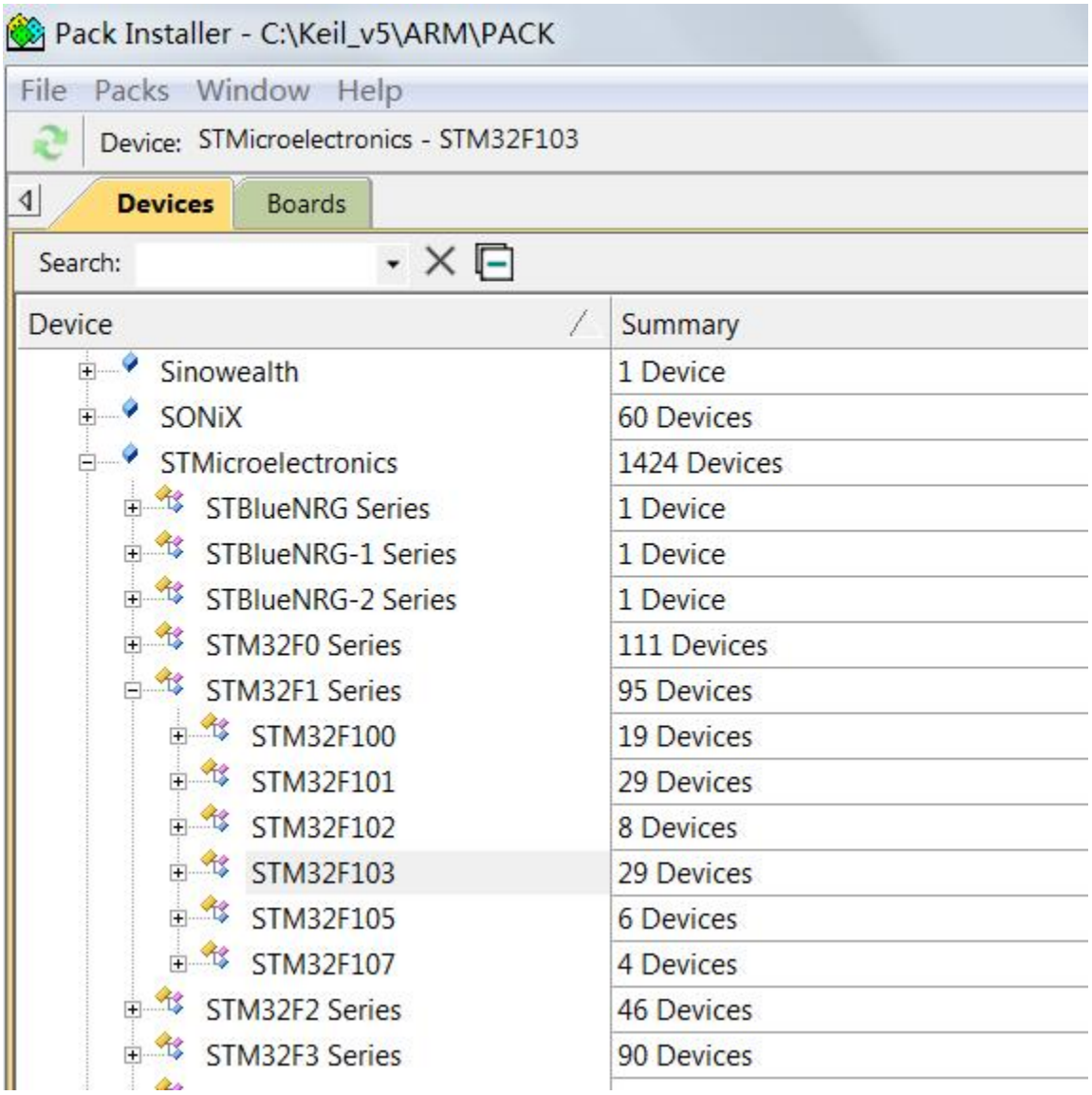
7.2 安装芯片支持包

在创建工程之前，需要先安装芯片支持包，否则在芯片中找不到我们需要的型号。

按照下图打开 Pack 安装页面

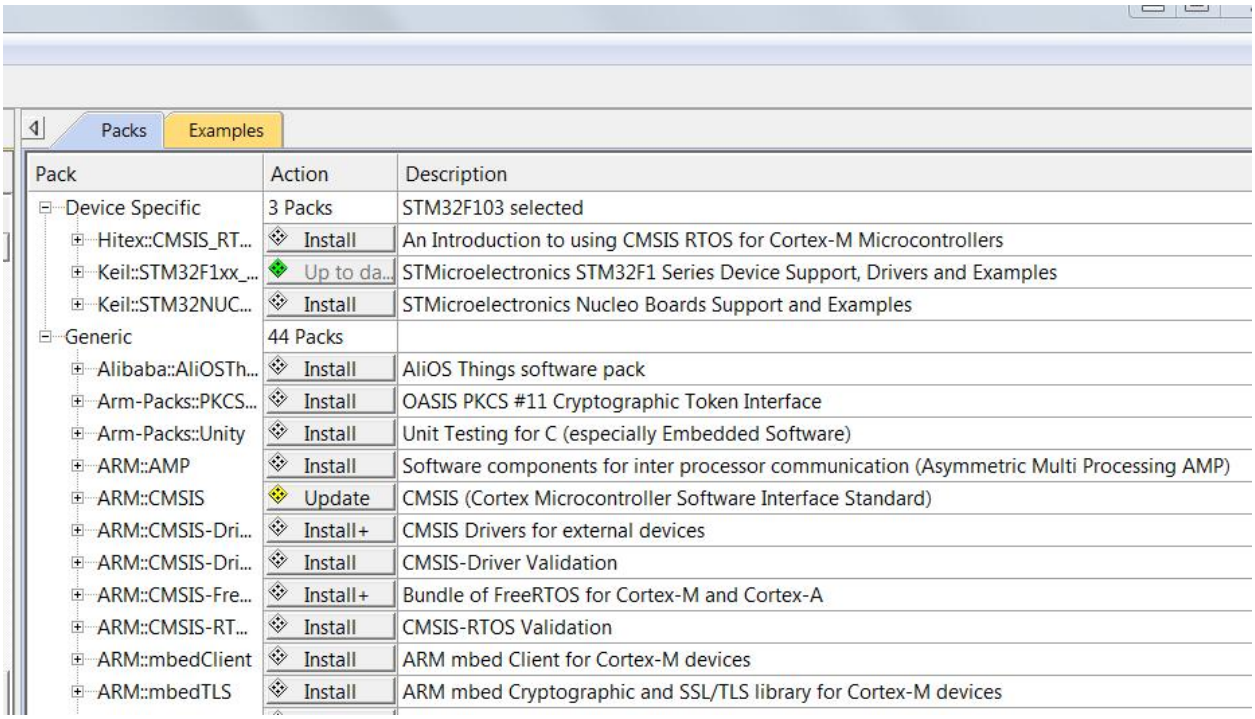


在左边的 Devices 中找到我们要的芯片系列，左键点击选中。

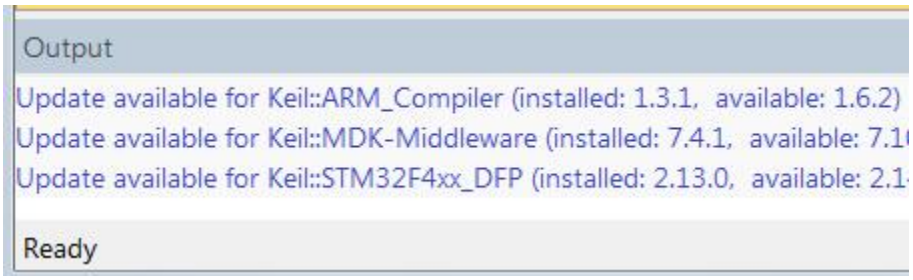


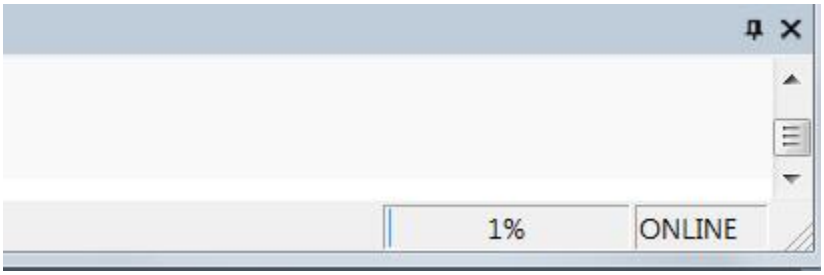
右边就会显示对应的支持包，我们只需要安装基础包。也就是第二行的，截图是我的电脑，已经安装好了。

没安装时和其他 PACK 一样显示 Install，点击 Install 按钮就可以下载安装。



在底部的左边有安装信息，右边有安装进度。‘

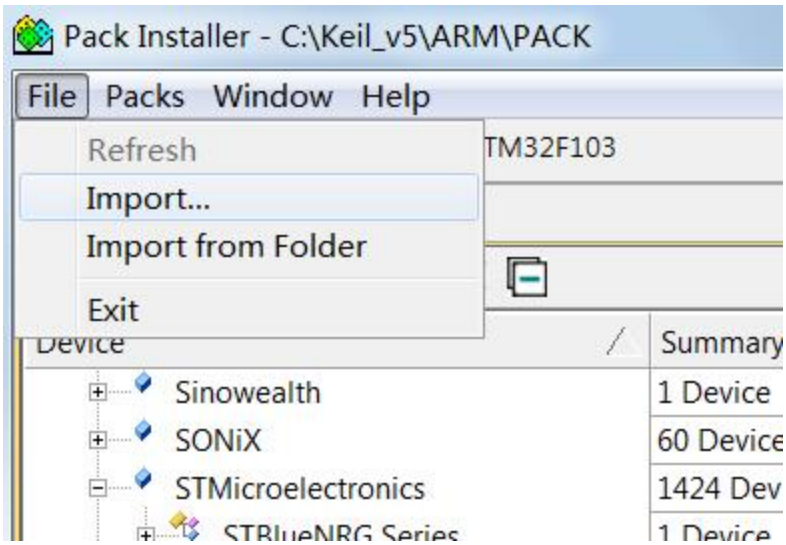




直接下载速度较慢，而且经常会断开。

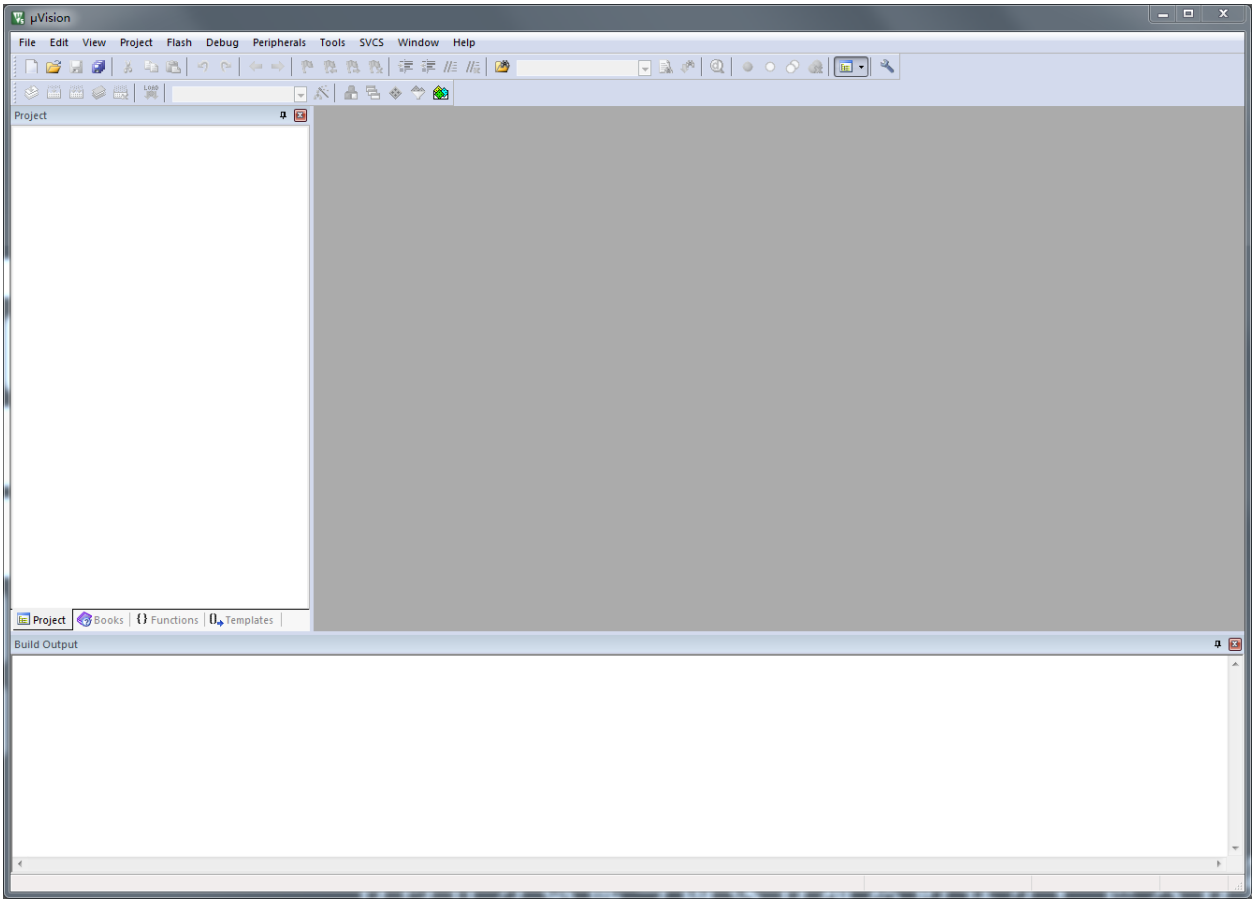
我们可以直接导入下载好的 pack 文件，在我们共享的资料中，`ref\4 mdk` 目录下的 `Keil.STM32F1xx_DFP.2.3.0.pack` 文件就是 F103 的支持包。

点击 `File` 菜单中的 `Import`，选择 pack 文件即可导入。

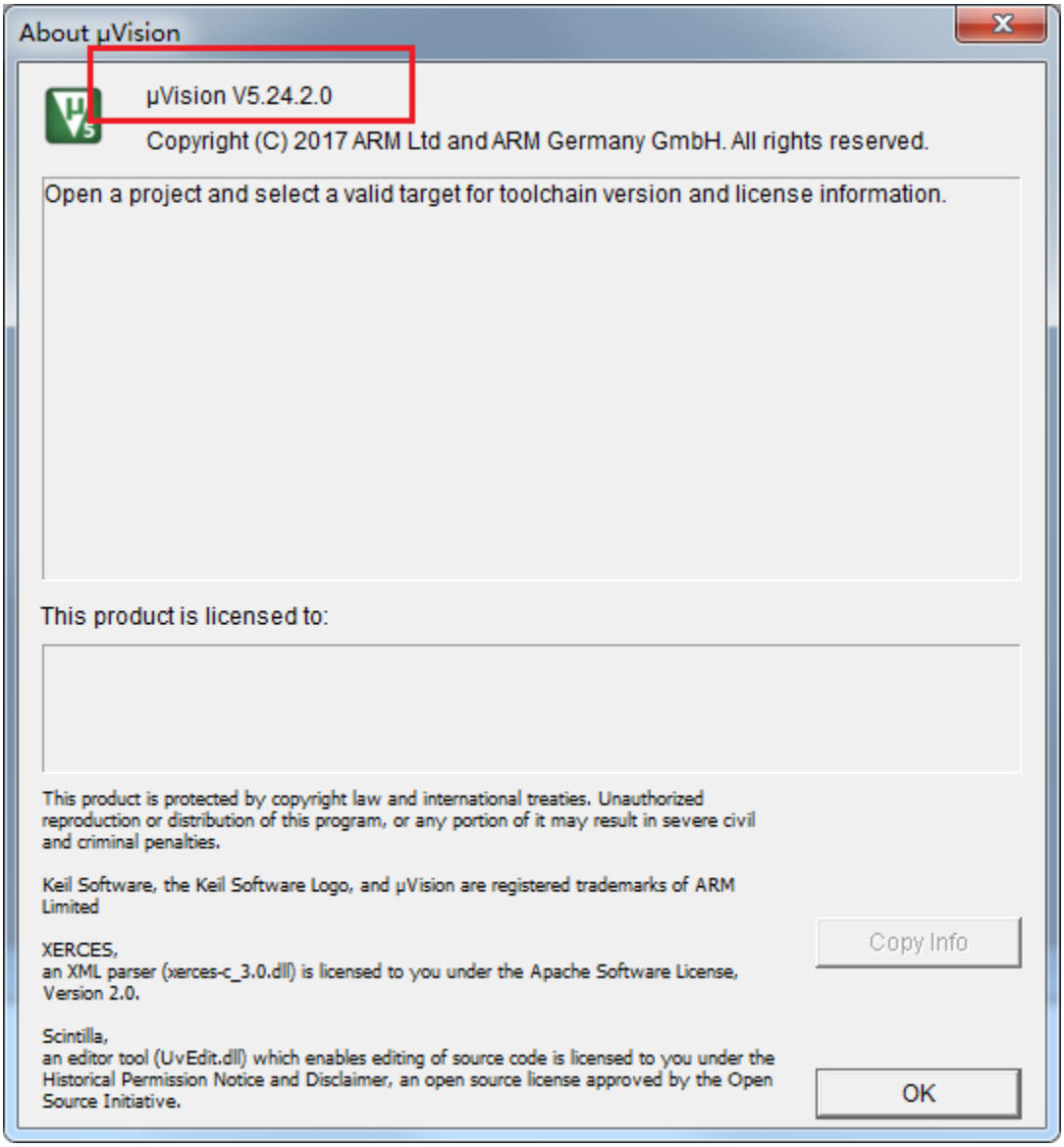


7.3 创建工程

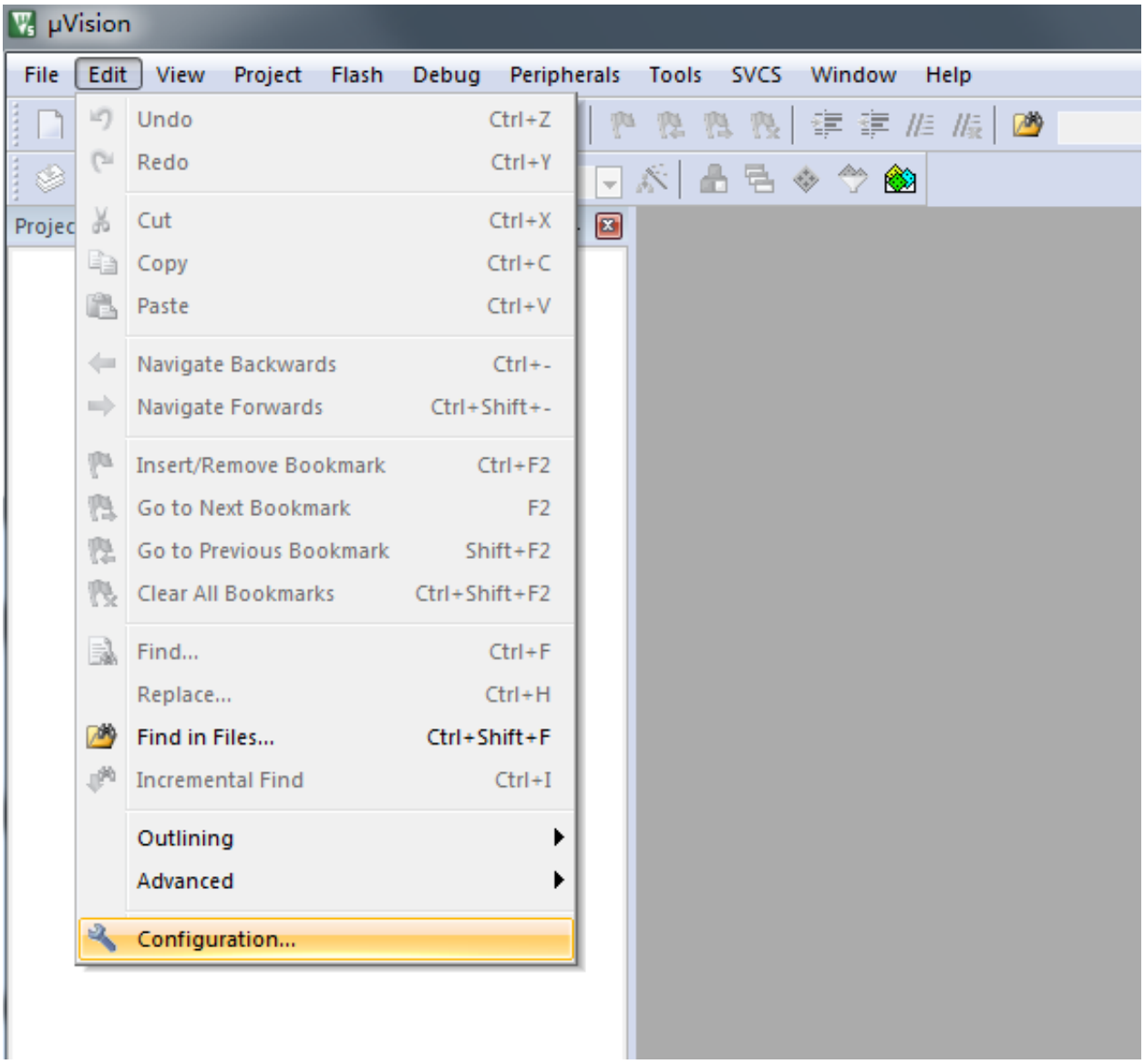
安装完成后，双击图标打开主界面如下。



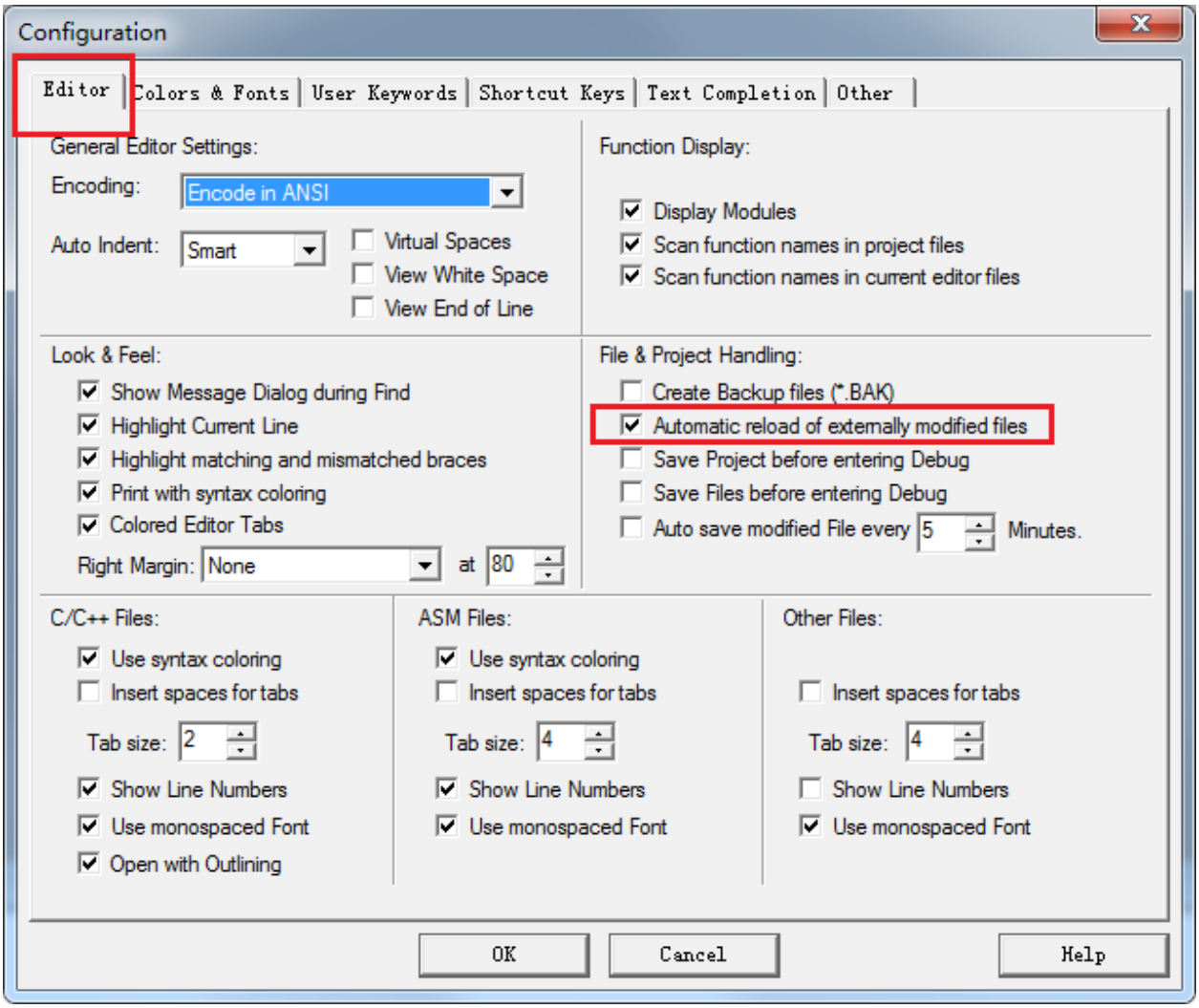
点击菜单栏的 `Help->About uVision`，可以查看到版本信息，我使用的是 V5.24.2.0。



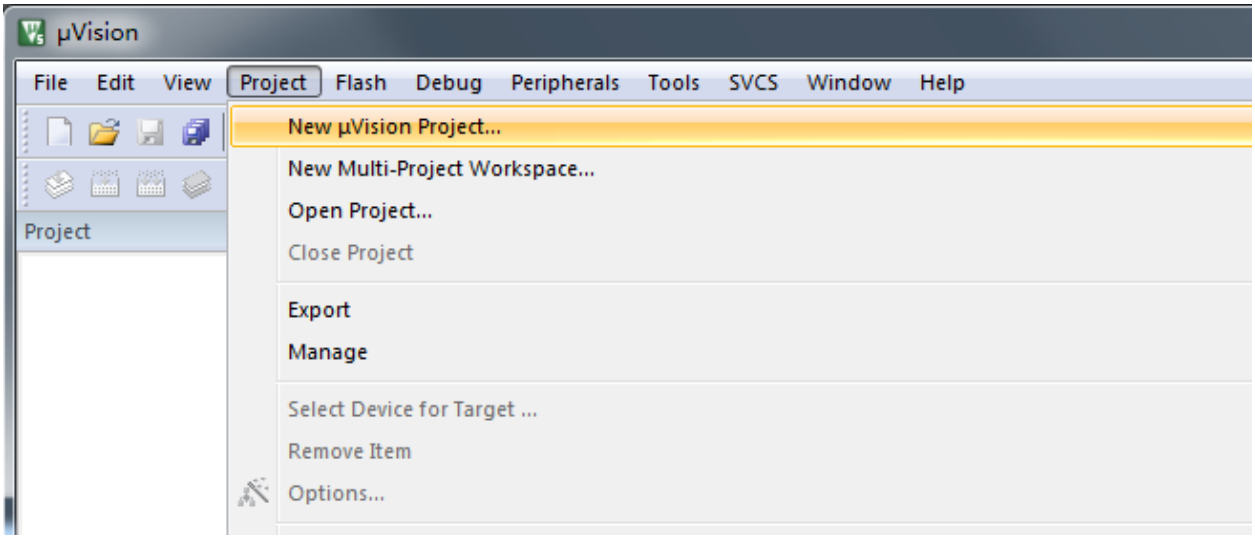
创建工程之前我们先配置 IDE 环境，



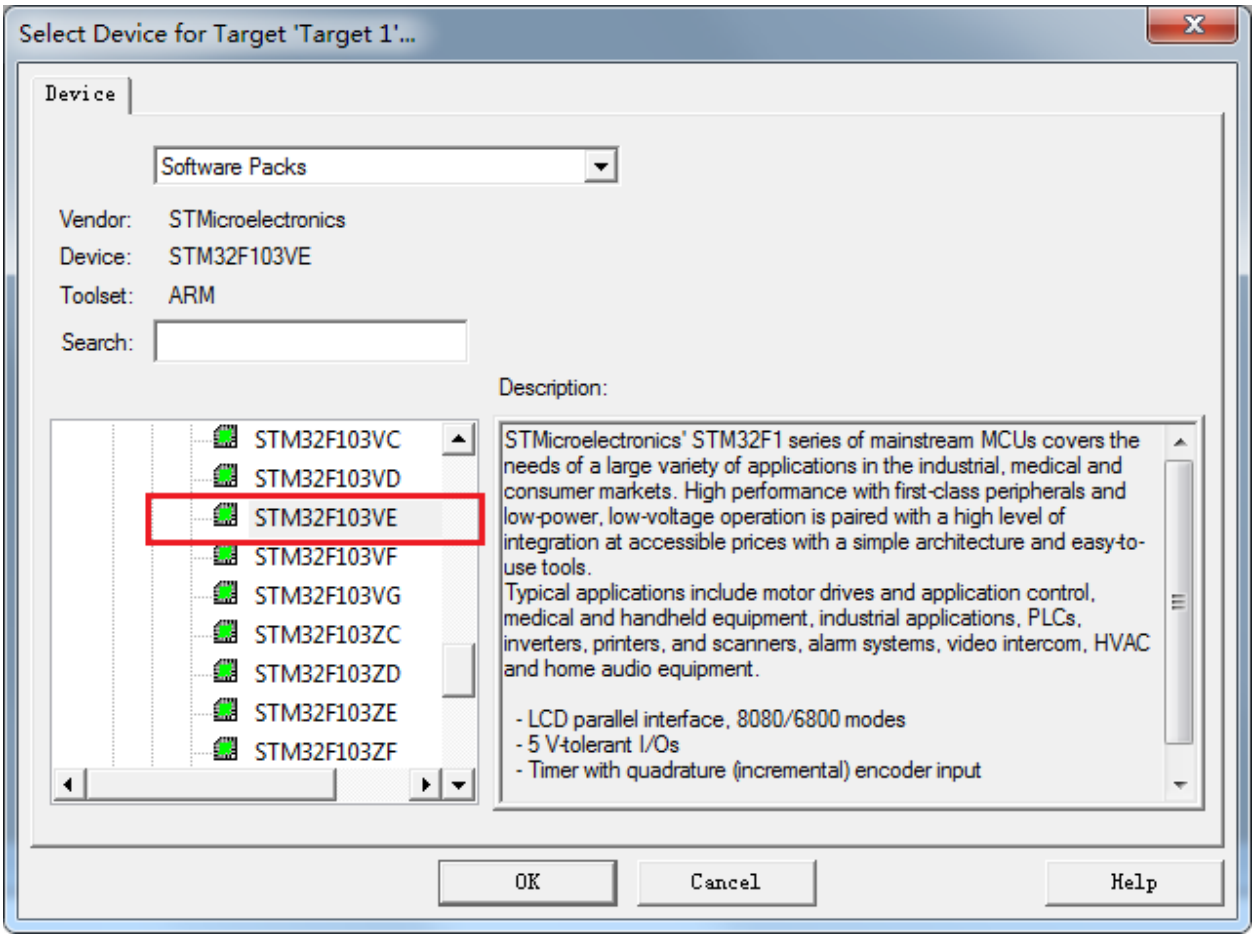
在 Configuration 的 Editor 页中，把 Automatic reload of externally modified files



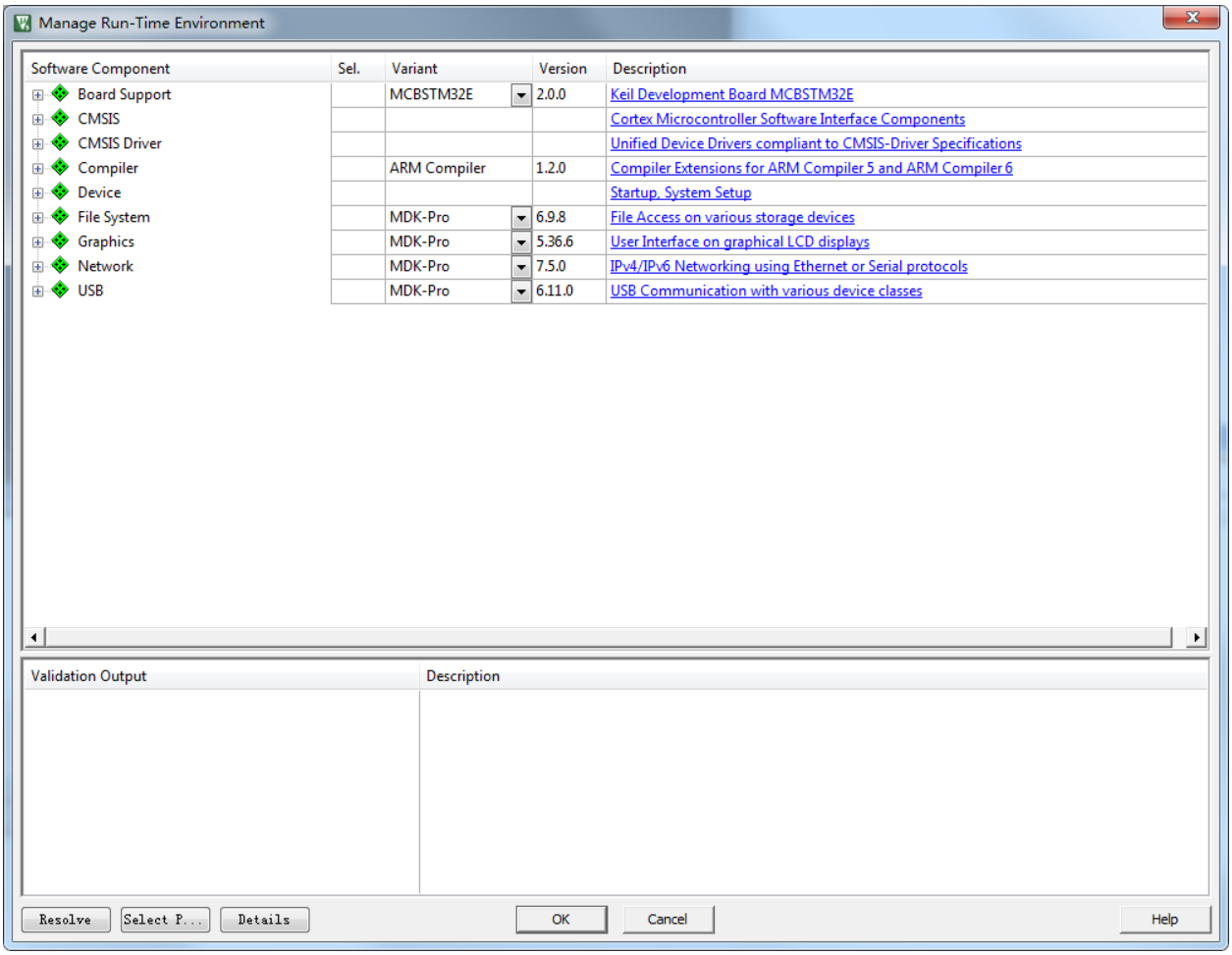
点击 `Project` 中的 `New uVision Project` 创建新工程



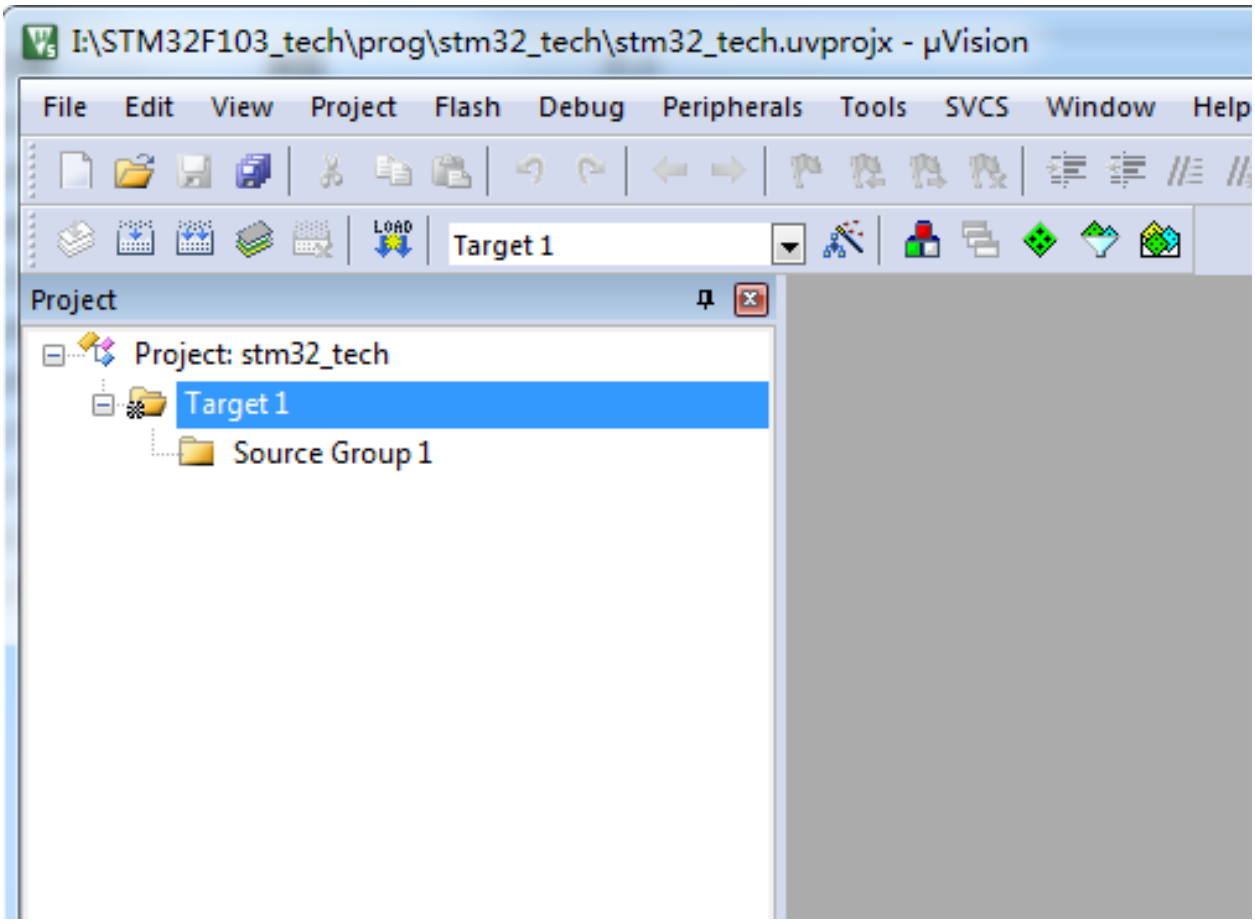
选择芯片 STM32F103VE



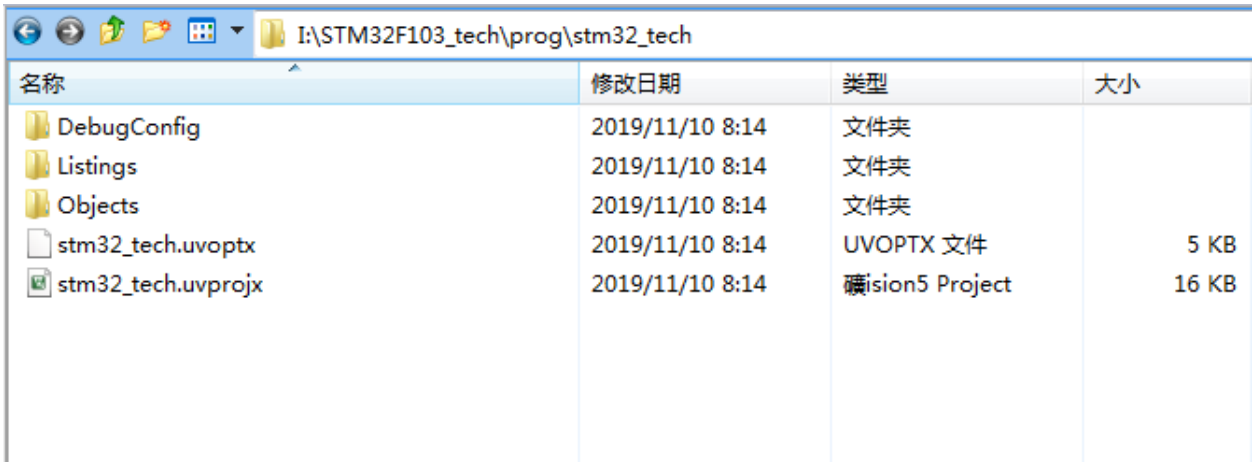
弹出界面让我们选择软件包，我们都不是使用这些软件库。点击 OK 跳过。



创建完成，现在得到一个空的工程。



查看目录，现在只有工程文件，没有代码文件。



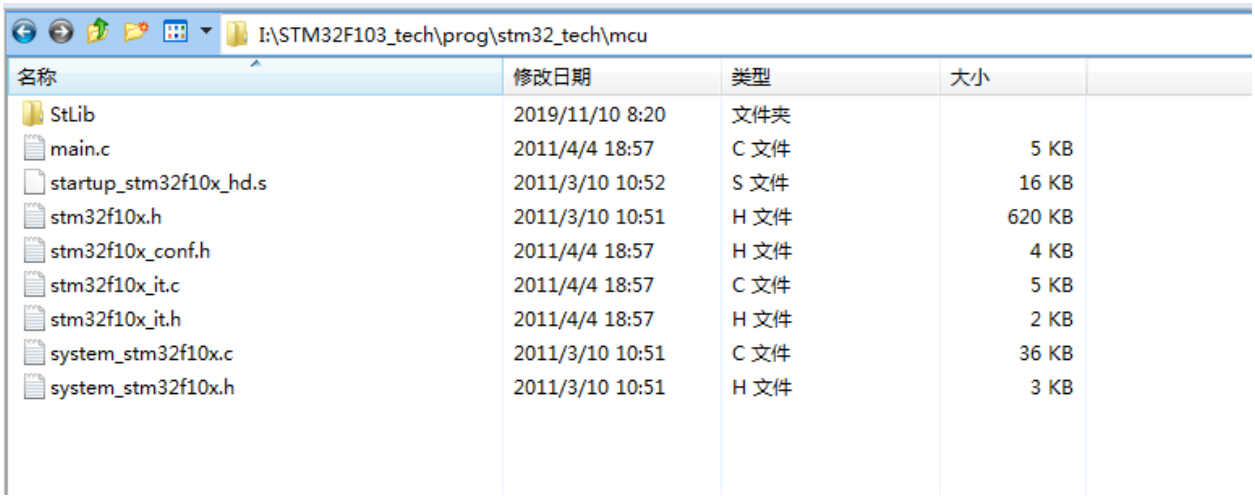
在目录中创建一个 mcu 文件夹。

把 STM32F10x_StdPeriph_Lib_V3.5.0\Libraries 下的 STM32F10x_StdPeriph_Driver 文件夹拷贝到 mcu 目录名字太长，改为 stdlib

拷贝 STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Examples\GPIO\IOToggle 中的文件到 mcu 目录

拷贝 STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x 中的源文件拷贝启动代码，注意选对编译工具和型号 STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm

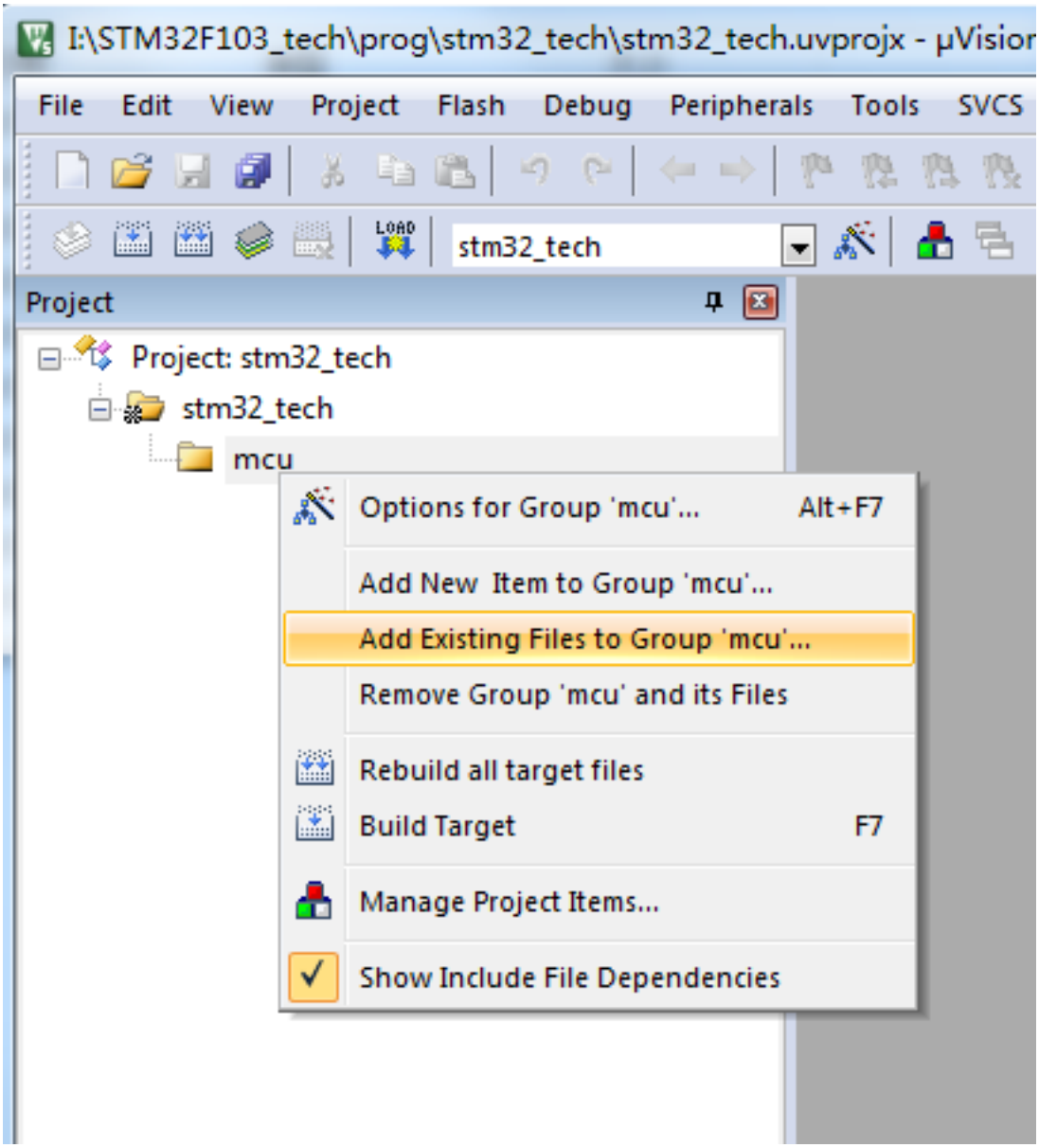
待补充



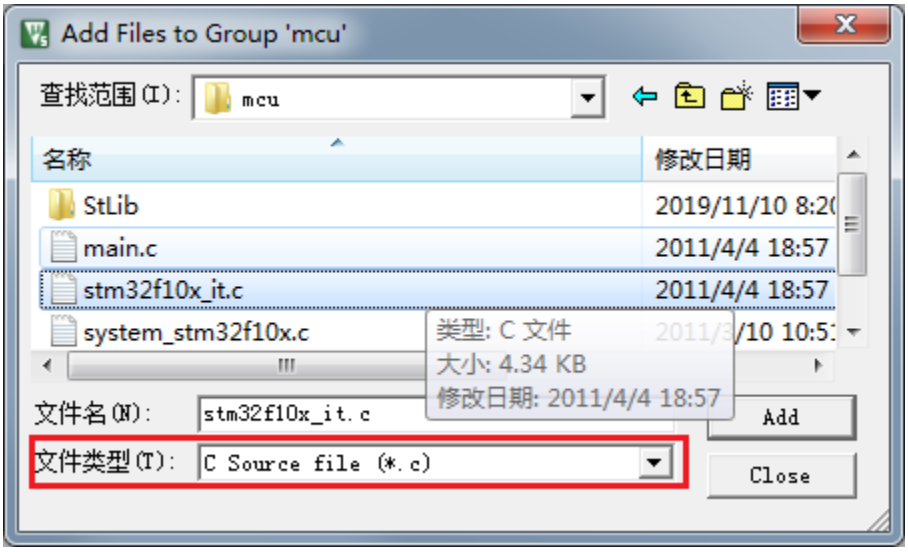
回到 MDK

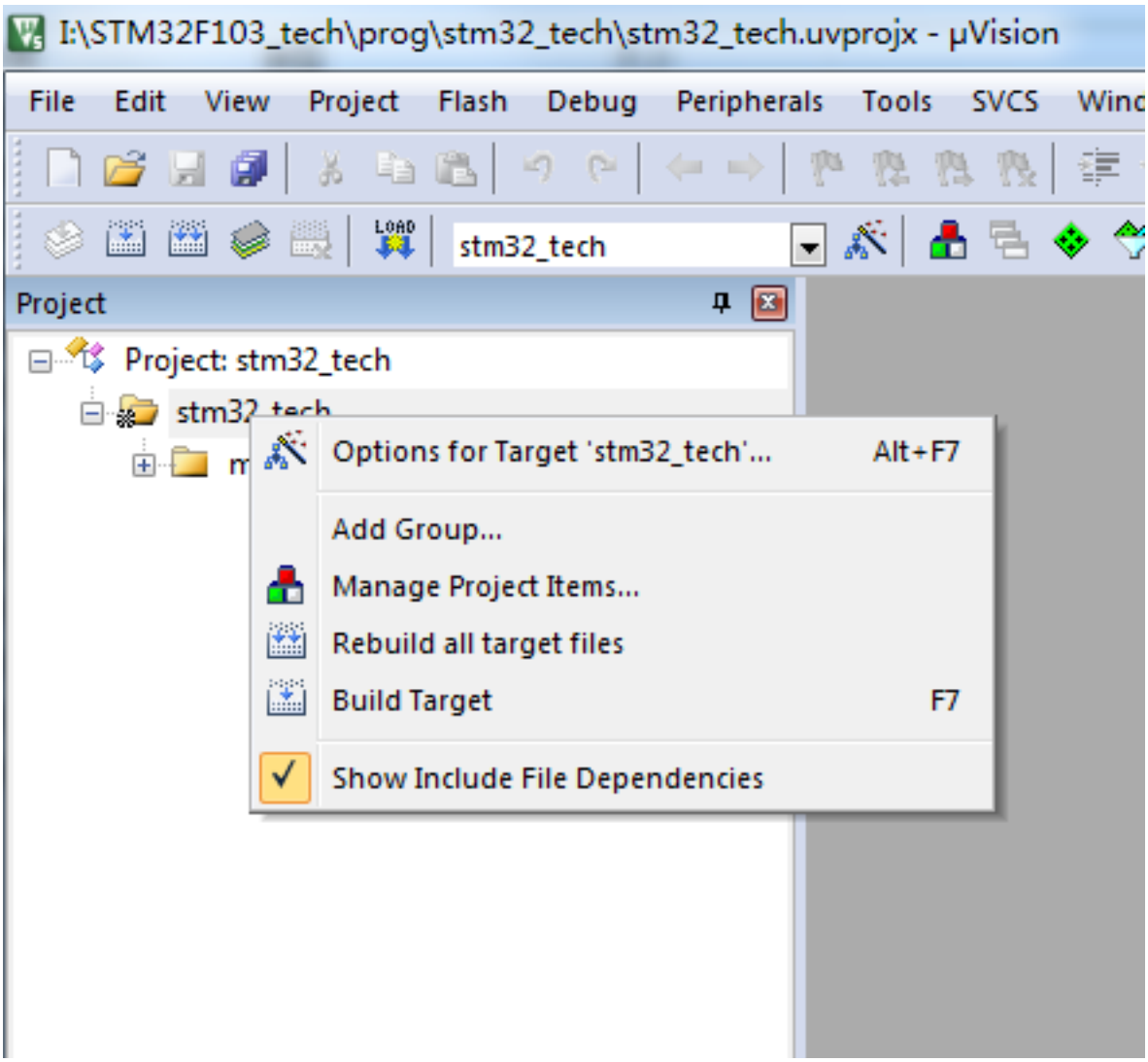
先选中目标名，再单击，修改为 stm32_tech 同样修改源文件目录名

然后右键点击 mcu，将刚刚拷贝到 mcu 目录的文件添加到工程的 mcu 下。

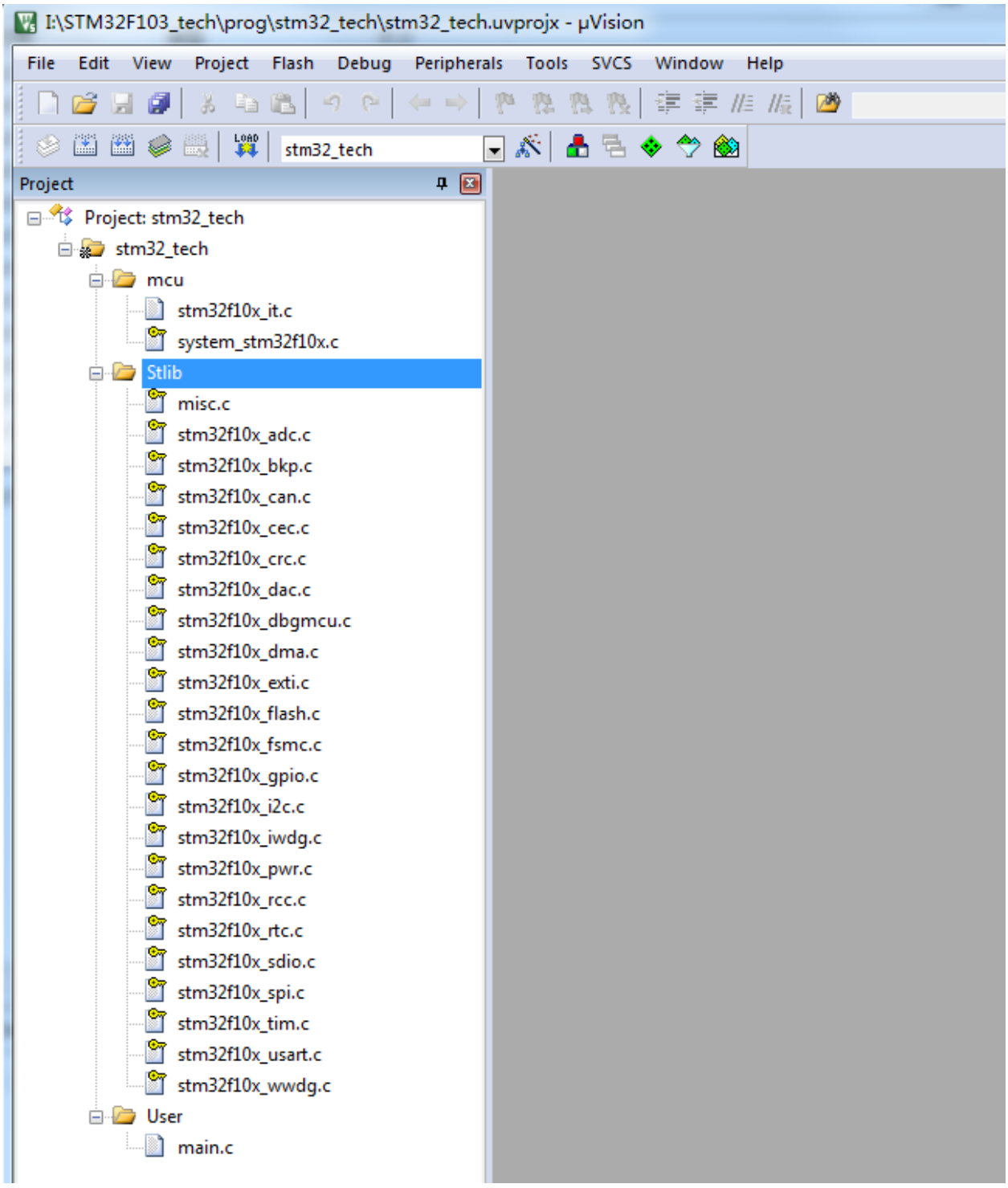


添加文件，添加启动代码的时候要注意，文件类型要选为 ALL，才能看到**汇编文件**。



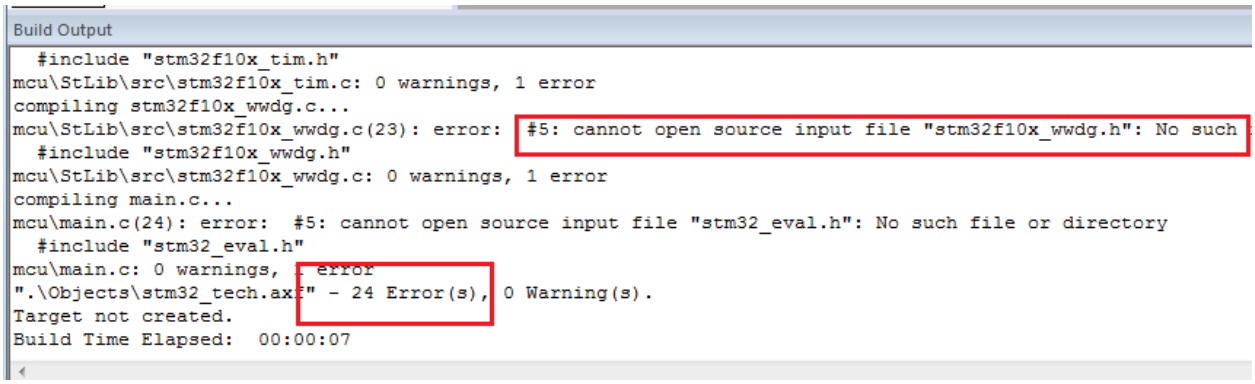


新建一个 Stlib 的文件夹，把库文件全部添加



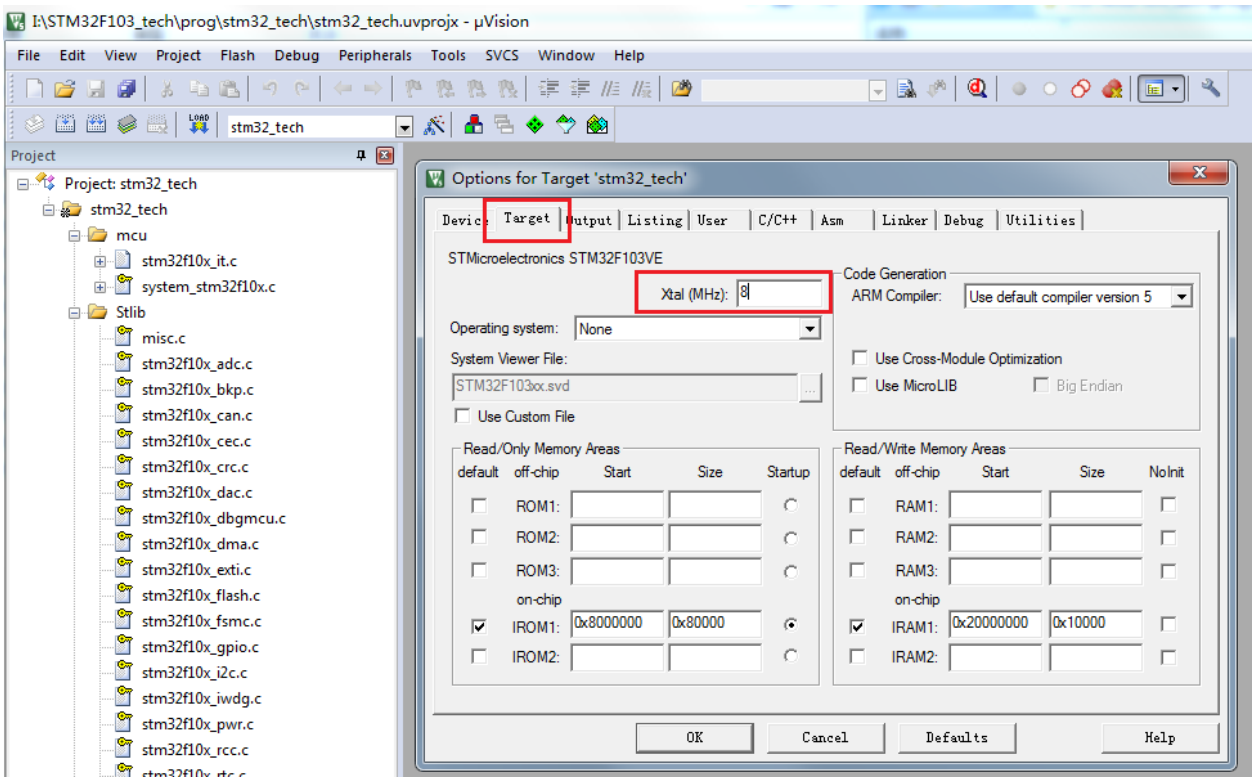
点击编译，在下方框中输出编译过程的信息，编译结果显示 24 个错误。错误都是提示找不到.h 文件。

原因是我们没有添加头文件存放的路径。

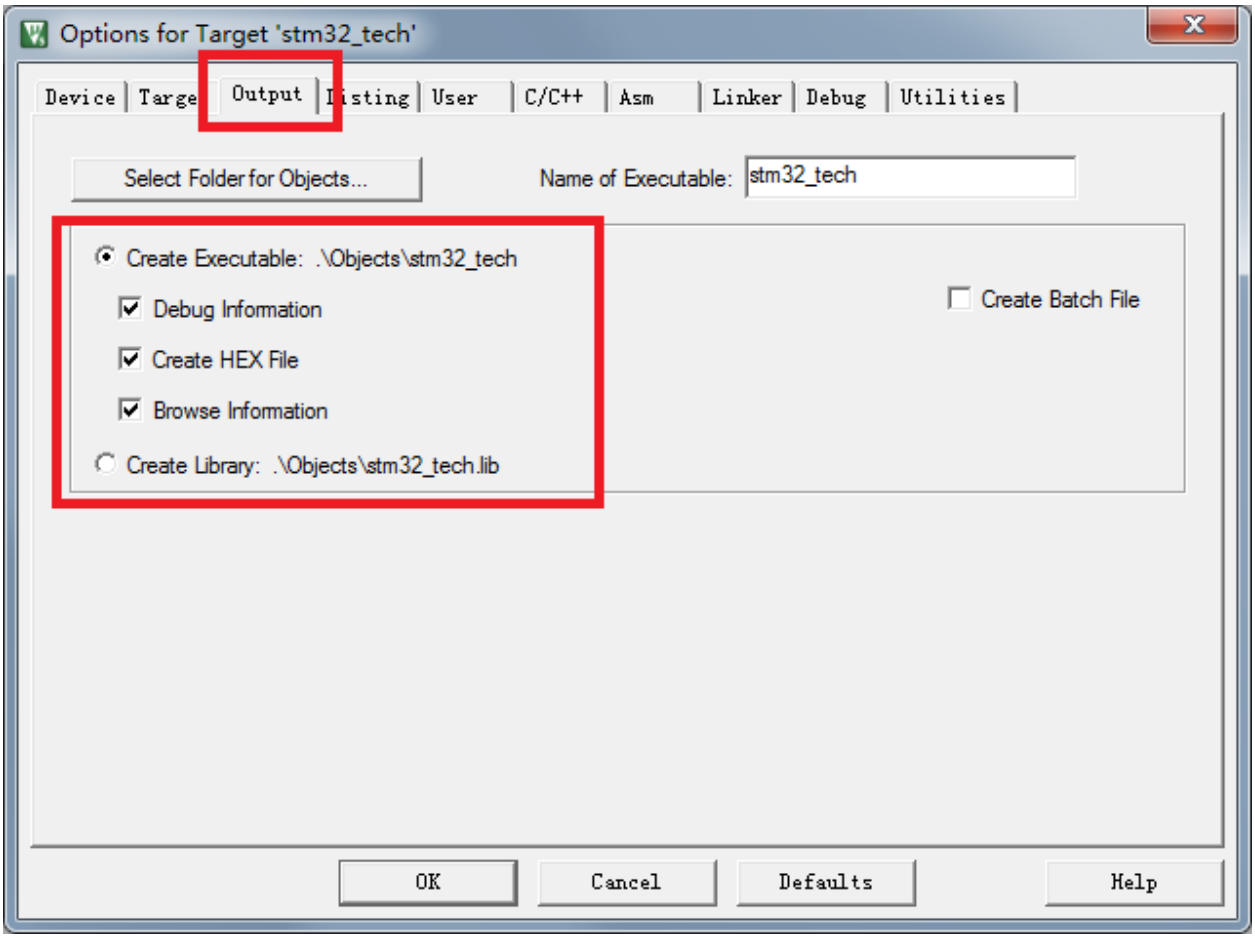


点击魔术棒进入 Option，开始配置工程。

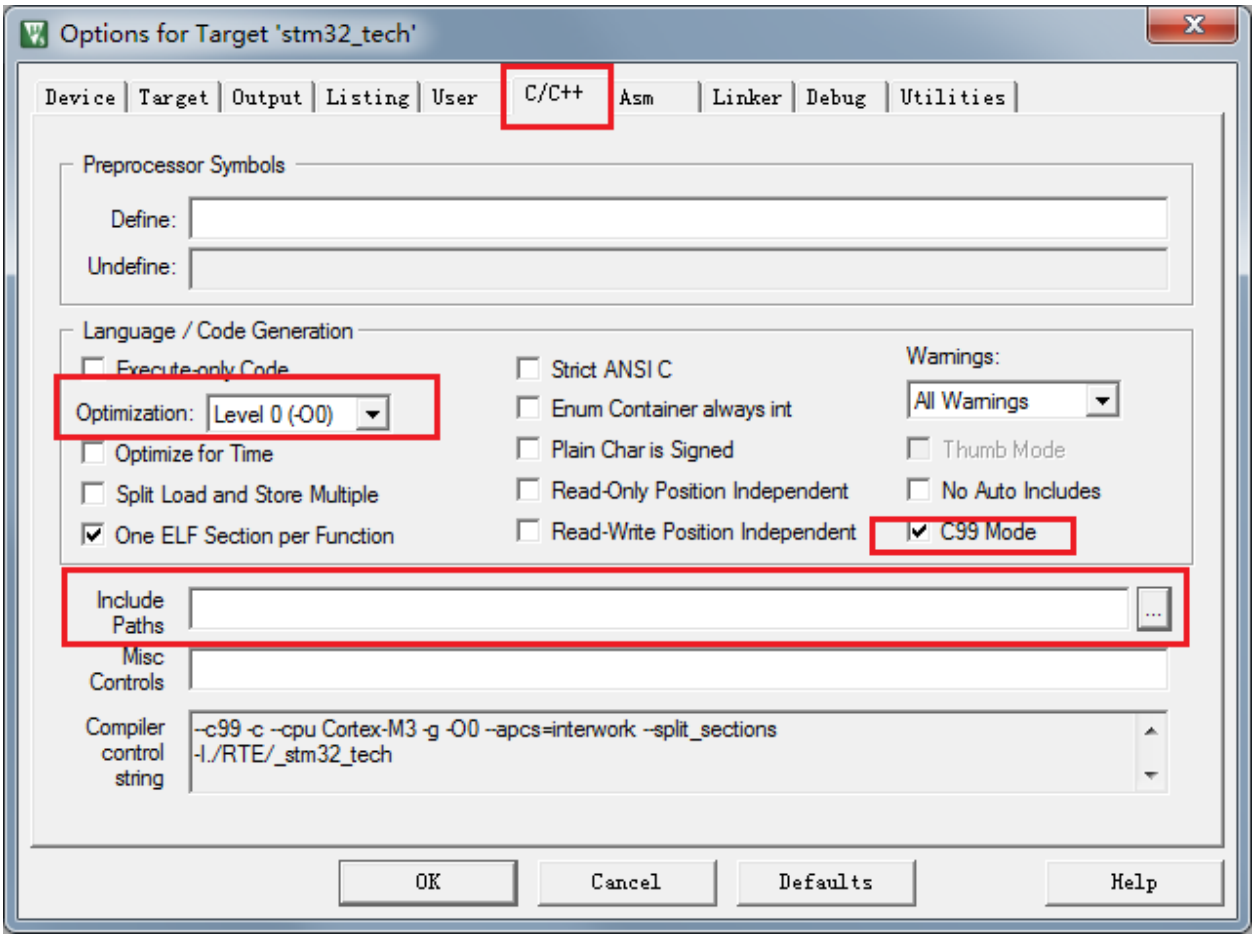
首先将晶振修改为 8M



在 Output 页中勾选如下，选择生成 hex 文件。

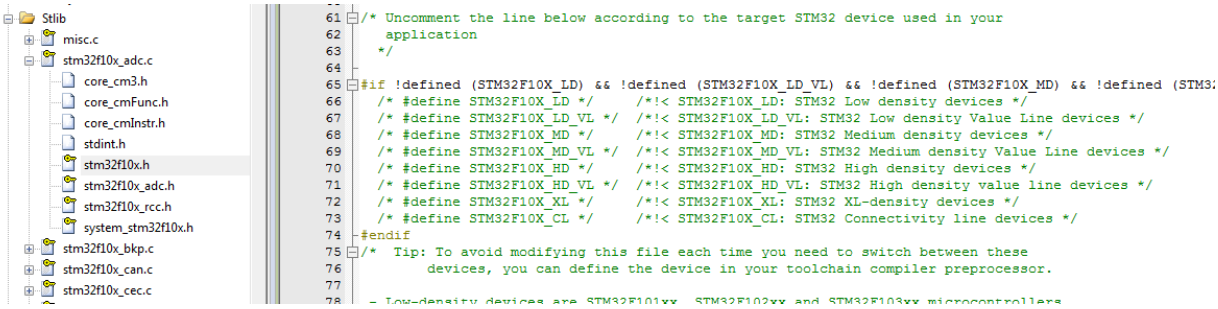


在 C/C++ 页中，勾选上 C99，并且将头文件存在的路径添加到 Include Paths 中。



点击工程目录树中，找到 stm32f10x.h，在 65 开始的地方，默认没有定义任何类型的芯片，我们要定义一个芯片。

有些库文件时只读，在图标上面有一个钥匙标志，需要先修改文件属性才能修改。



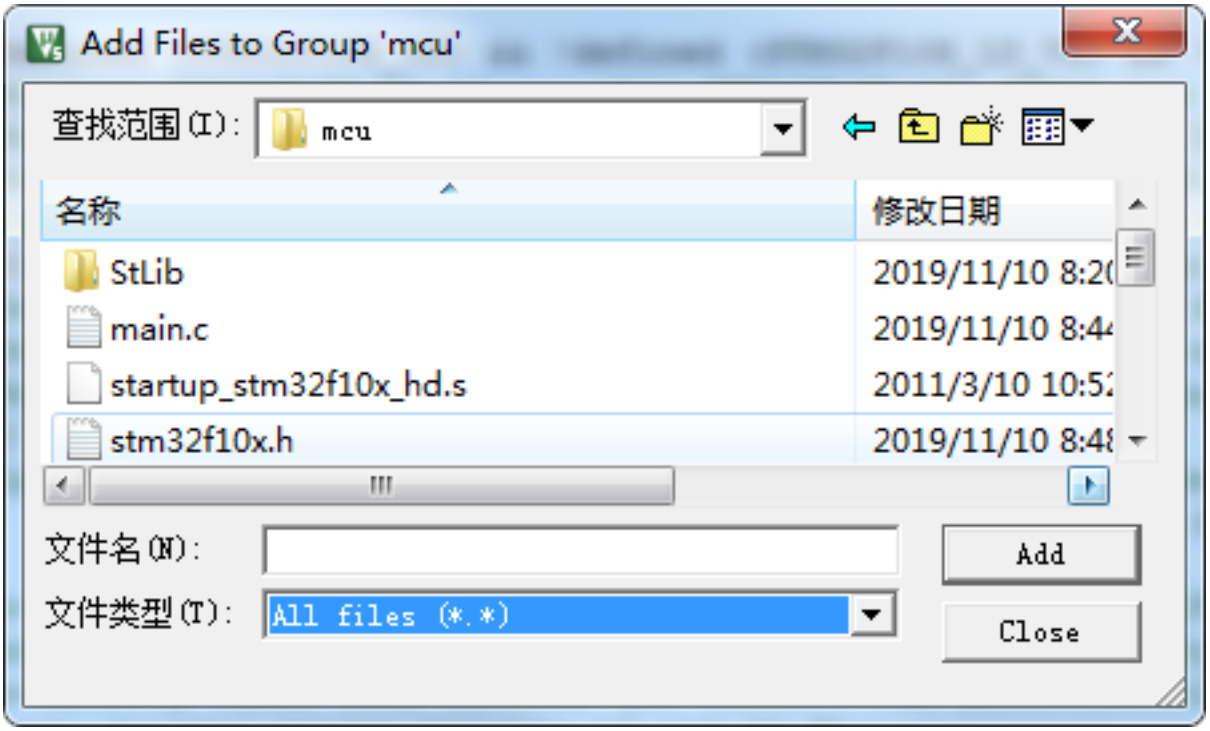
在这个文件的 105 行，打开这个宏，选择使用标准库。

```
 97 #endif
 98
 99 #if !defined  USE_STDPERIPH_DRIVER
100 /**
101   * @brief Comment the line below if you will not use the peripherals drivers.
102     In this case, these drivers will not be included and the application code will
103     be based on direct access to peripherals registers
104     */
105    /*#define USE_STDPERIPH_DRIVER*/
106 #endif
107
108 /**
```

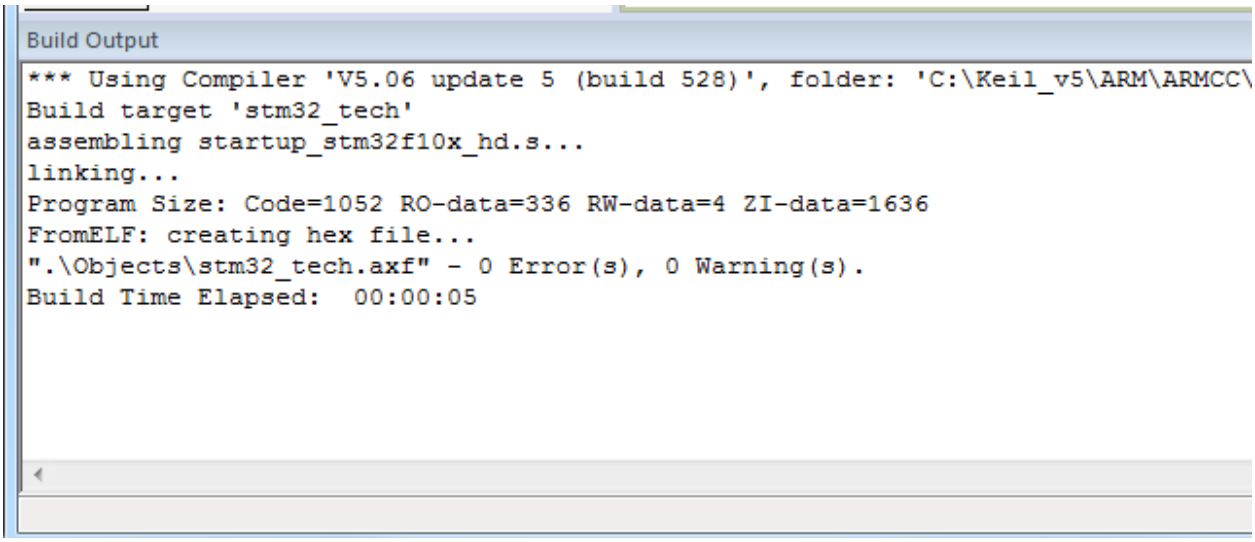
编译，错误，原因是我们没有忘记添加启动文件了。

```
Build Output
compiling stm32f10x_tim.c...
compiling stm32f10x_usart.c...
compiling stm32f10x_wwdg.c...
compiling main.c...
linking...
.\Objects\stm32_tech.sct(7): error: L6236E: No section matches selector - no section to be FIRST/LAST.
Not enough information to list image symbols.
Not enough information to list load addresses in the image map.
Finished: 2 information, 0 warning and 1 error messages.
".\Objects\stm32_tech.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:19
```

添加启动文件，文件类型要选择为 ALL files 才能选中汇编文件。

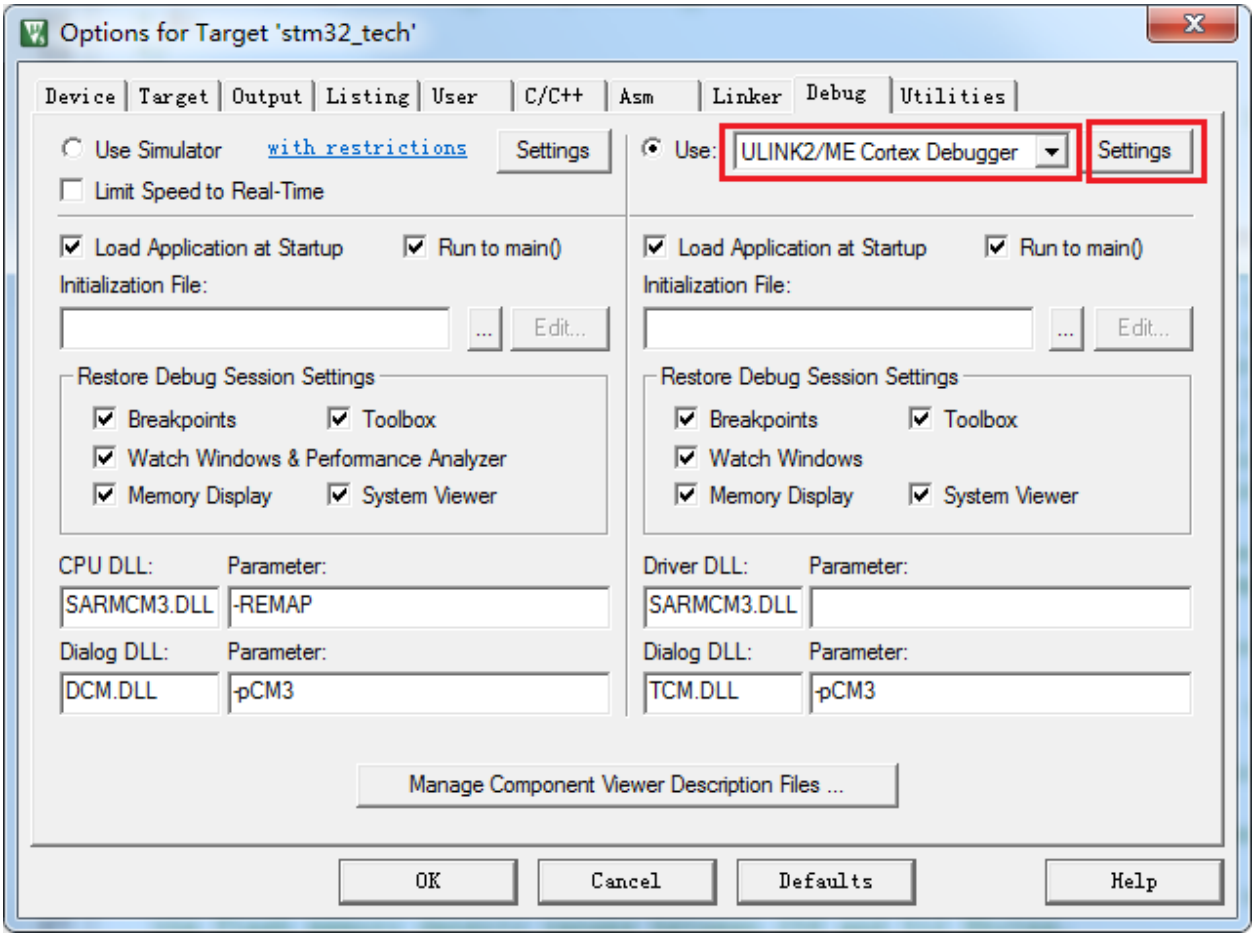


编译通过。

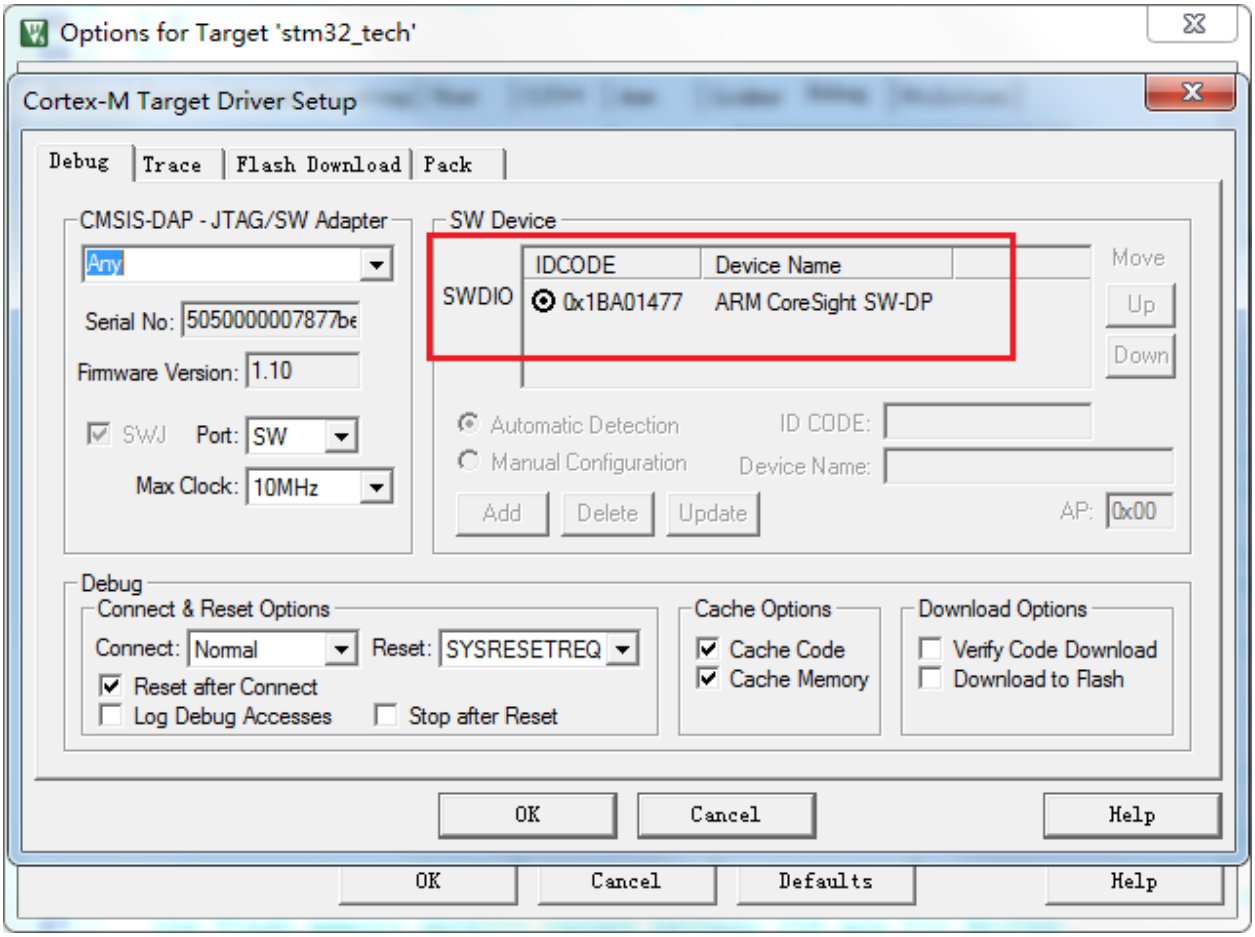


7.4 下载

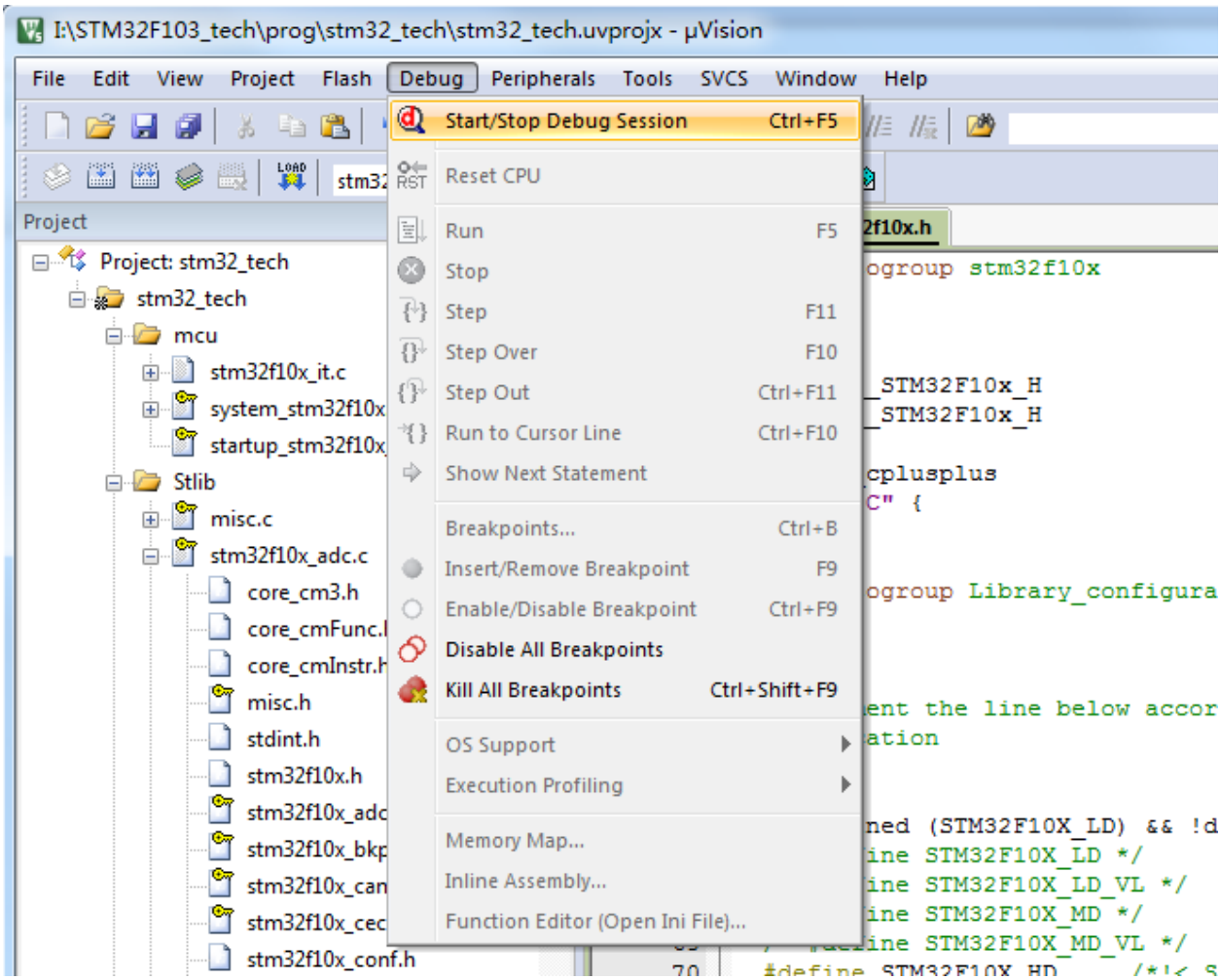
在工程 Option 中，Debug 页，右上角，选择调试器，默认是 ULINK2……，我们选在 CMSIS-DAP。



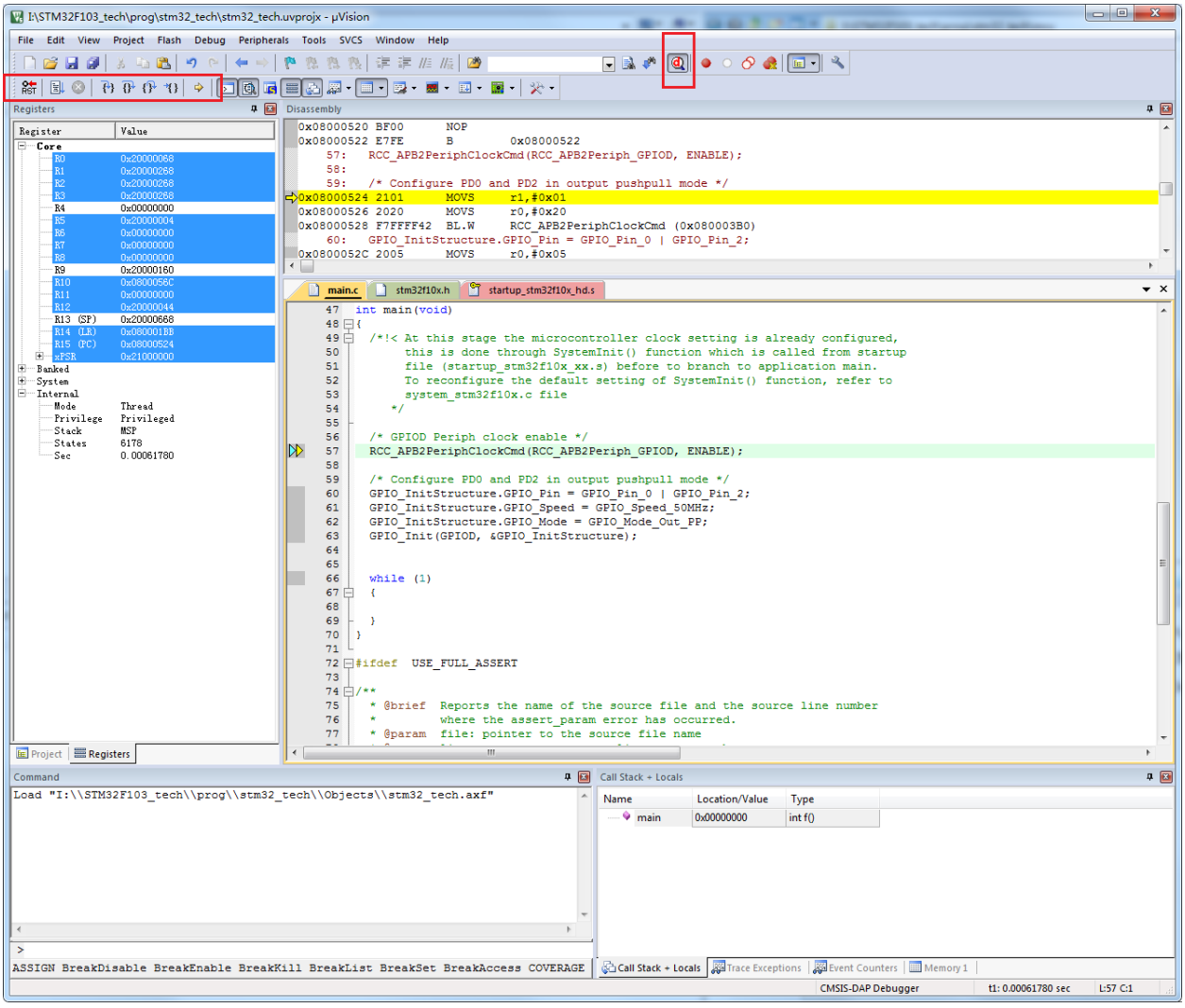
硬件连接正确的情况下，点击 Settings，可以读到芯片。



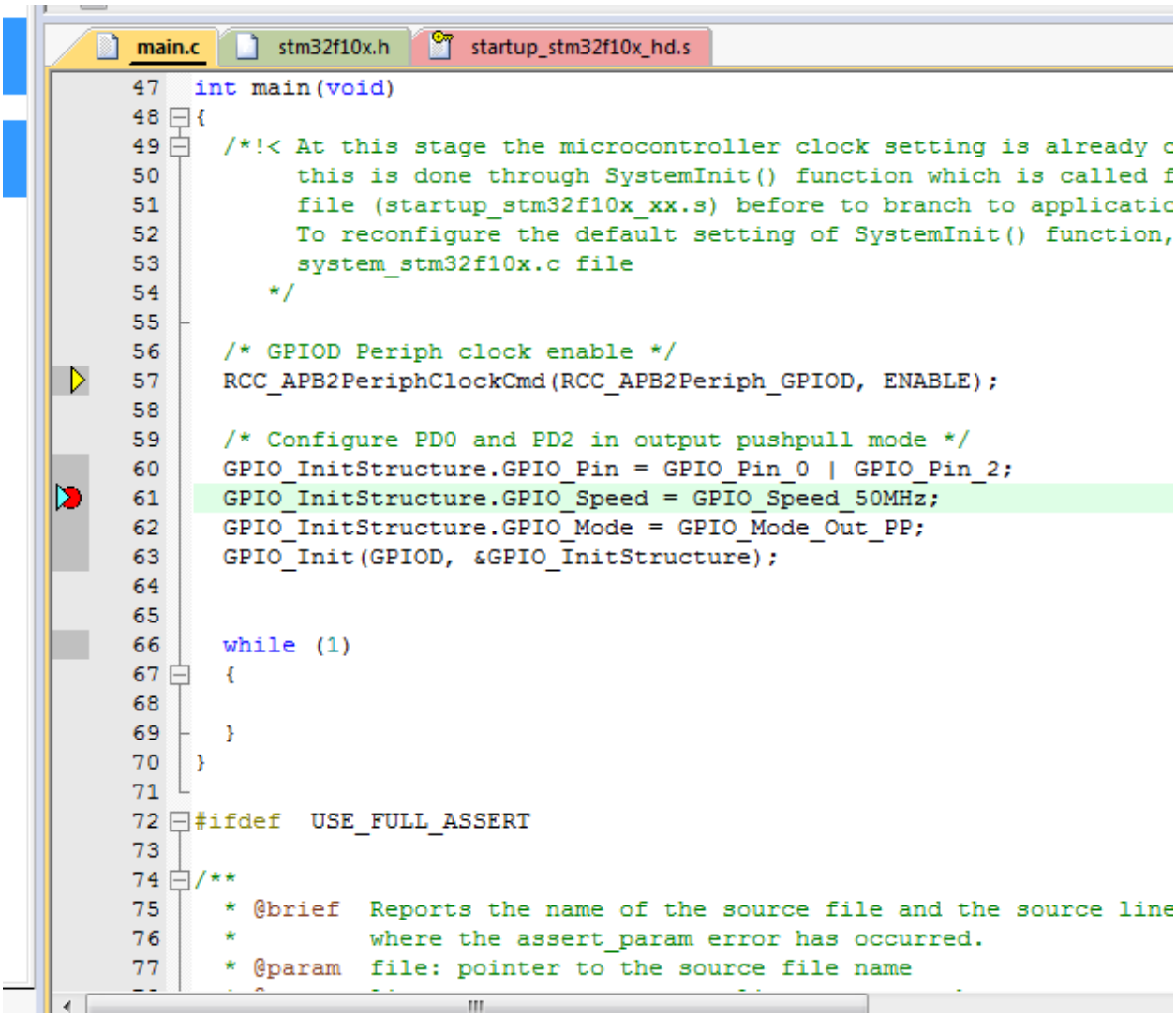
回到主界面，点击 Debug，开始调试。


I:\STM32F103_tech\prog\stm32_tech\stm32_tech.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Start/Stop Debug Session Ctrl+F5
Reset CPU
Run F5
Stop
Step F11
Step Over F10
Step Out Ctrl+F11
Run to Cursor Line Ctrl+F10
Show Next Statement
Breakpoints... Ctrl+B
Insert/Remove Breakpoint F9
Enable/Disable Breakpoint Ctrl+F9
Disable All Breakpoints
Kill All Breakpoints Ctrl+Shift+F9
OS Support
Execution Profiling
Memory Map...
Inline Assembly...
Function Editor (Open Ini File)...
Project
Project: stm32_tech
stm32_tech
mcu
stm32f10x_it.c
system_stm32f10x
startup_stm32f10x
Stlib
misc.c
stm32f10x_adc.c
core_cm3.h
core_cmInstr.h
misc.h
stdint.h
stm32f10x.h
stm32f10x_conf.h
2f10x.h
ogroup stm32f10x
_STM32F10x_H
_STM32F10x_H
cplusplus
C" {
ogroup Library_configura
ent the line below accor
ation
ned (STM32F10X_LD) && !d
ine STM32F10X_LD */
ine STM32F10X_LD_VL */
ine STM32F10X_MD */
ine STM32F10X_MD_VL */

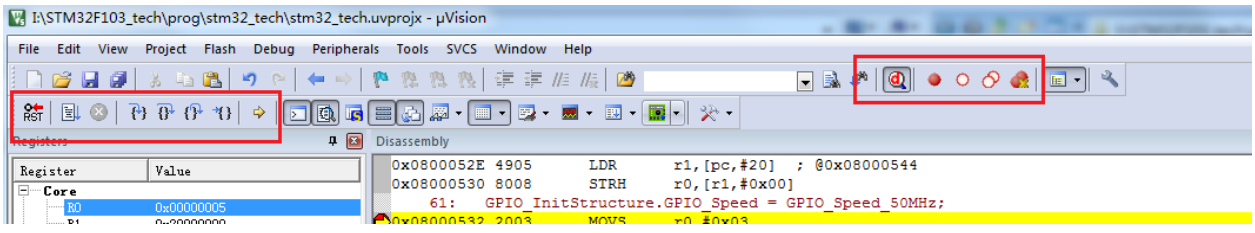
进入调试后界面如下。



左键点击行号旁边可以添加断点，黄色三角符号是当前执行位置。



各种调试按钮在菜单栏。



20200114

end

CHAPTER 8

IO& 点亮 LED

8.1 概述

芯片如何控制外部电路?

从本课开始，让我们一点一点把单片机系统的知识框架搭建起来。

本课，学习最简单的、也是最基本的：GPIO 口。

本课讲以下几个问题：

1. IO 是什么？GPIO 是什么?
2. STM32 的 IO 有什么特点?
3. LED 如何使用?
4. 编码和调试过程（这段看视频更好）

8.2 IO 是什么？

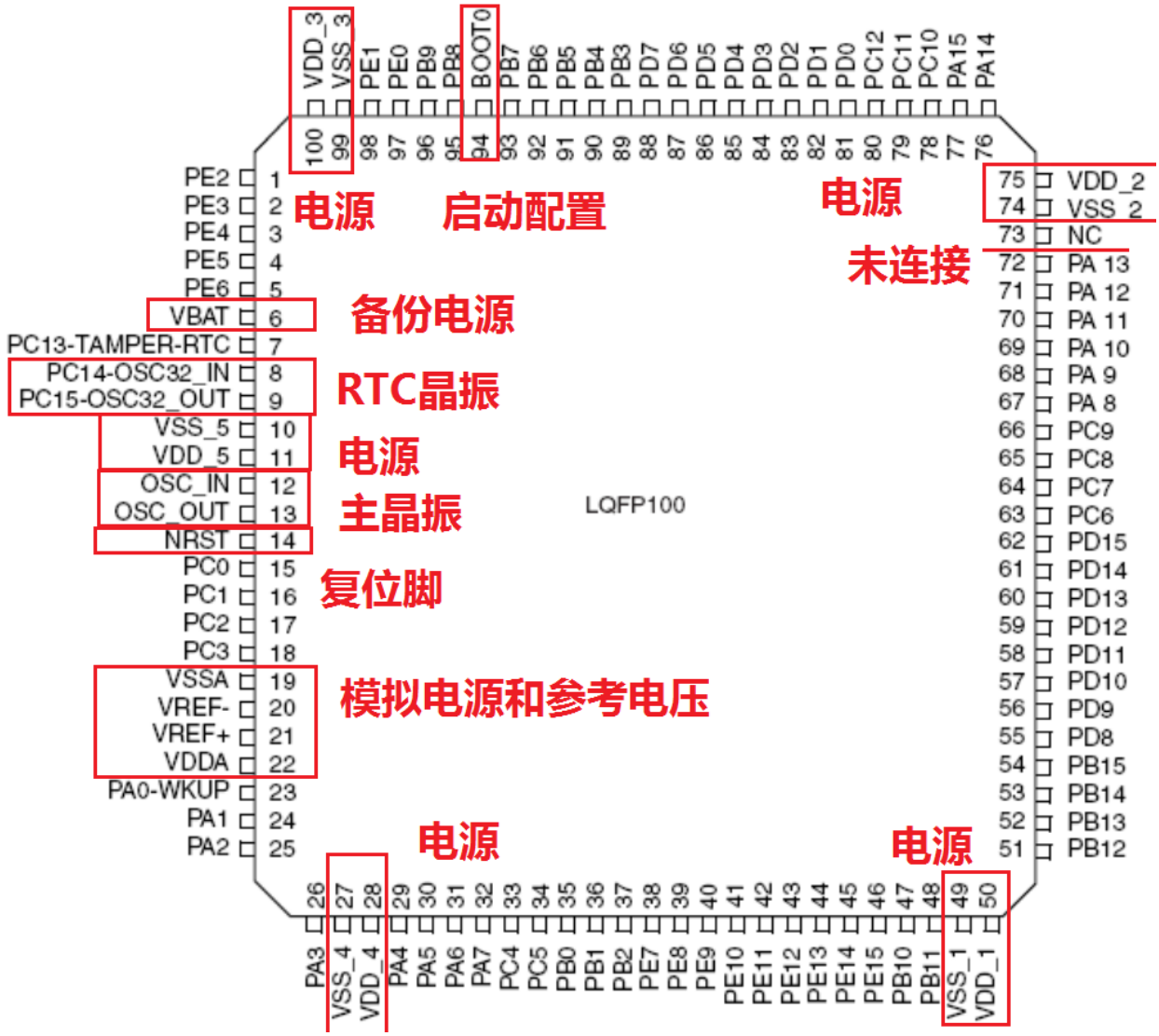
IO 是 Input&Output，也就是输入和输出。

我们选的 STM32F103VET6 这款芯片的，管脚总共有 100 根。

但是《STM32F103 数据手册.pdf》中有写：IO 口只有 80，为什么?

- 多达112个快速I/O口
  - 51/80/112个多功能双向的I/O口
  - 所有I/O口可以映像到16个外部中断
  - 除了模拟输入口以外的IO口可容忍5V信号输入

在数据手册的第 14 页有管脚定义图，如下：



从管脚名称看到，P 前缀的的管脚，就是 IO 口。100 脚的 STM32，有 PA、PB、PC、PD、PE，一共 5 组 IO。

除了 IO，还有电源和地、晶振输入、BOOT 配置等其他功能的管脚。

1. 普通电源有 5 组。
2. 模拟电源有 1 组。
3. 备份电源一组。
4. 参考电压 VREF 有一组。
5. 复位脚和启动配置 BOOT0 各一根管脚。
6. 晶振两组，其中 RTC 晶振可做普通 IO 使用。
7. 73 脚是空的，没有连接。

有些朋友可能有疑问，怎么都没看到 SPI、I2C 等功能的管脚?

通常 IO 口都能复用作其他功能，比如 SPI、I2C 等。在数据手册的管脚定义中有几页说明管脚可以复用做什么功能。

比如:

脚位					管脚名称	类型(1)	I/O电平(2)	主功能(复位后)(3)	可选功能	
BGA144	BGA100	LQFP64	LQFP100	LQFP144					默认功能	重定义功能
J1	G1	12	19	30	V_{SSA}	S		V_{SSA}		
K1	H1	-	20	31	V_{REF-}	S		V_{REF-}		
L1	J1	-	21	32	V_{REF+}	S		V_{REF+}		
M1	K1	13	22	33	V_{DDA}	S		V_{DDA}		
J2	G2	14	23	34	PA0-WKUP	I/O		PA0	WKUP/USART2_CTS(6) ADC123_IN0/TIM5_CH1 TIM2_CH1_ETR TIM8_ETR	
K2	H2	15	24	35	PA1	I/O		PA1	USART2_RTS(6) ADC123_IN1 TIM5_CH2/TIM2_CH2(6)	

PA0，主功能是 PA0，可用作串口的 CTS、ADC、TIM6、TIM2、TIM8 等功能。

从此可见，我们说 IO 功能，通常只是一个 IO 口的基本功能：**GPIO**，通用输入输出的意思。

8.2.1 GPIO 功能

GPIO 能用来做什么呢?

要理解这个 GPIO，先要定下一个概念：**除了 DA 转换和 AD 转换，其他的 IO 口都是数字逻辑功能**。

对 GPIO 来说，功能很简单，分两个:

1. 输出-在管脚上输出数字逻辑电平。
2. 输入-检测管脚的逻辑电平

   芯片用的是 TTL 电平：

   数字电平有两种，高电平和低电平。

   高电平是逻辑 1，低电平是逻辑 0.

   高电平是芯片的 IO 电压，STM32 没有独立 IO 电压，IO 电压就是是芯片工作电压，低电平是就地电平。

   实际上呢，高电平和低电平是有一个范围的，并不仅仅是 *3.3V* 和 *0V*

8.2.2 GPIO 驱动能力

一个 IO 口输出电流或输入电流的能力。

在中文版数据手册第 31 页中，有下面这个表格：

表8　电流特性

符号	描述	最大值(1)	单位
I_{VDD}	经过V_{DD}/V_{DDA}电源线的总电流(供应电流)(1)	150	mA
I_{VSS}	经过V_{SS}地线的总电流(流出电流)(1)	150	
I_{IO}	任意I/O和控制引脚上的输出灌电流	25	
	任意I/O和控制引脚上的输出电流	-25	
$I_{INJ(PIN)}$ (2) (3)	NRST引脚的注入电流	+/-5	
	HSE的OSC_IN引脚和LSE的OSC_IN引脚的注入电流	+/-5	
	其他引脚的注入电流(4)	+/-5	
$\Sigma I_{INJ(PIN)}$(2)	所有I/O和控制引脚上的总注入电流(4)	+/-25	

灌电流和拉电流都是 25mA

在设计外围电路时，电流不能超过这个值，否则会烧芯片。比如大电流的 LCD 背光，就需要在外部添加三极管驱动电路。

还有一个需要注意的，有些芯片，灌电流和拉电流的最大值不一样，灌电流可能只有 5ma。

8.3 STM32 IO 特点

以前的单片机，比如 8051，GPIO 口非常简单，只要设置 GPIO 口的方向是输出还是输入，就可以工作了。用起来虽然简单，却无法满足各种应用场景。因此，高级的芯片，IO 口通常有很多功能可以配置。

STM32 的 GPIO 就是这样，如果操作寄存器操作的话，相当复杂。在《STM32F10x 微控制器参考手册.pdf》中，第七章就是讲 GPIO 功能的。

- 首先要知道，IO 口有 8 种模式：

- 输入浮空
- 输入上拉
- 输入下拉
- 模拟输入
- 开漏输出
- 推挽式输出
- 推挽式复用功能
- 开漏复用功能

其中 GPIO 用到的有 5 种。模拟输入是 AD 转换使用。推挽复用和开漏复用模式用于 GPIO 外的其他外设功能，比如 SPI。

GPIO 五种模式，要区分输入和输出：浮空、上拉、下拉，都是说输入。推挽、开漏，是输出。

驱动 LED 用输出功能。那选推挽还是开漏模式呢？要先搞清楚推挽和开漏输出的区别。

1. 开漏模式，IO 口内部没有上拉电阻，没有接 MOS 管，所以开漏电路不能输出高电平；要输出高电平，需要外部接上拉电阻，如果外部上拉电阻接的电压不是芯片 IO 电压，就相当于实现了 IO 口电平转换功能。

   比如，推挽模式下，IO 口输出高电平就是 3.3V，用开漏模式，外部接电阻上拉到 5V，那么输出高电平就是 5V。

2. 电平转换速度指芯片 0/1 翻转的速度。
3. 线与，两根 IO 直接连接在一起，电平按照与逻辑。通常用在一些可挂载多设备的总线上，比如 I2C。

原理参考：https://www.cnblogs.com/lweleven/p/mcuioout.html

因此，驱动 LED 用推挽还是开漏？**除了必须用开漏的场合，我们都习惯用推挽输出**

- 第二，SMT32 的 IO 有 IO 速度需要配置。

表16　输出模式位

MODE[1:0]	意义
00	保留
01	最大输出速度为10MHz
10	最大输出速度为2MHz
11	最大输出速度为50MHz

如果不知道如何选，全部用 50M，功能肯定正常。但是可能会增加电流，增加 EMC 辐射。

经验：

普通功能的 IO，通常 2M 就可以了。

如果一个 IO 用作 I2C 通信，速度通常就 10K 到 400K，选 10M 就好了。

如果是用作 SPI 功能，可能会到 20M 速度，那就要选 50M 了。

到此，我们基本了解了 STM32 GPIO 的功能。下面看看 ST 的库都提供了什么函数给我们用。

8.4 ST 库函数

打开上一节我们创建的工程。在库函数中找到 stm32f10x_gpio.c 和 stm32f10x_gpio.h

函数有下面这些：

```
void GPIO_DeInit(GPIO_TypeDef* GPIOx);
void GPIO_AFIODeInit(void);
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
void GPIO_StructInit(GPIO_InitTypeDef* GPIO_InitStruct);
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_PinLockConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void GPIO_EventOutputConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
void GPIO_EventOutputCmd(FunctionalState NewState);
void GPIO_PinRemapConfig(uint32_t GPIO_Remap, FunctionalState NewState);
void GPIO_EXTILineConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
void GPIO_ETH_MediaInterfaceConfig(uint32_t GPIO_ETH_MediaInterface);
```

GPIO_Init：初始化 IO 口，

GPIO_SetBits：IO 口输出 1

GPIO_ResetBits：IO 口输出 0

GPIO_WriteBit：IO 口输出状态，相当于 GPIO_SetBits 和 GPIO_ResetBits 组合。

GPIO_Write：输出 IO 口状态。

GPIO_WriteBit 是在指定的 IO 口上输出**相同**的状态，GPIO_Write 是在一组 IO 上输出需要的状态，。

我们看参数：

```
GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

将 GPIOx 这组 IO 口中，GPIO_Pin 指定的 IO 口，输出高电平。

GPIO_ResetBits 功能和 GPIO_SetBits 相反。

```
GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, BitAction BitVal);
```

将 GPIOx 这组 IO 口中 GPIO_Pin 指定的 IO 口设置为 BitVal 的状态。

```
GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
```

将 GPIOx 这组 IO 口设置为 PortVal 的状态。**注意，是一次设置一组 IO**。

我们在来看看初始化的接口

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
```

关键是第二个参数，这是一个结构体，定义如下：

```
typedef struct
{
  uint16_t GPIO_Pin;             /*!< Specifies the GPIO pins to be configured.
                                      This parameter can be any value of @ref GPIO_pins_
↪define */

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

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

第 1 个参数 GPIO_Pin 指定要配置的 IO 口。

第 2 个参数 GPIO_Speed 配置 IO 口速度。

第 3 个参数 GPIO_Mode 配置 IO 口模式。

其中 GPIO_Speed 和 GPIO_Mode 类型是**枚举**，如下：

- 速度

```
typedef enum
{
  GPIO_Speed_10MHz = 1,
  GPIO_Speed_2MHz,
  GPIO_Speed_50MHz
}GPIOSpeed_TypeDef;
```

- 模式

```
typedef enum
{ GPIO_Mode_AIN = 0x0,
  GPIO_Mode_IN_FLOATING = 0x04,
  GPIO_Mode_IPD = 0x28,
  GPIO_Mode_IPU = 0x48,
  GPIO_Mode_Out_OD = 0x14,
  GPIO_Mode_Out_PP = 0x10,
  GPIO_Mode_AF_OD = 0x1C,
  GPIO_Mode_AF_PP = 0x18
}GPIOMode_TypeDef;
```

配置 IO 的时候，选用这里的定义即可。

我们看下例程,在目录 STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Examples\GPIO\IOToggle 中的 main 函数，初始化 IO 口的代码如下：

```
int main(void)
{
  /*!< At this stage the microcontroller clock setting is already configured,
       this is done through SystemInit() function which is called from startup
       file (startup_stm32f10x_xx.s) before to branch to application main.
       To reconfigure the default setting of SystemInit() function, refer to
       system_stm32f10x.c file
     */

  /* GPIOD Periph clock enable */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);

  /* Configure PD0 and PD2 in output pushpull mode */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_2;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
```

(continues on next page)

(continued from previous page)

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

1. 调用 RCC_APB2PeriphClockCmd 函数打开 GPIOD 时钟。

这个是需要注意的，ST 的芯片每个设备都有时钟，使用之前都需要打开。

1. GPIO_Pin 设置为 GPIO_Pin_0 | GPIO_Pin_2，说明过一次配置两个 IO 口。
2. 速度配置为 GPIO_Speed_50MHz，也就是 50M
3. GPIO_Mode 设置为 GPIO_Mode_Out_PP，也就是输出推挽模式。
4. 调用函数 GPIO_Init 进行配置。

8.5 如何使用 LED

LED 是什么?

发光二极管简称为 LED。因化学性质又分有机发光二极管 OLED 和无机发光二极管 LED。

它本质上是一个二极管，所以就有正负极。

加上电压就会亮，但是 LED 其实是一个电流器件，有电流了才会亮。

电流越大，亮度越大，但是不能超过规格。

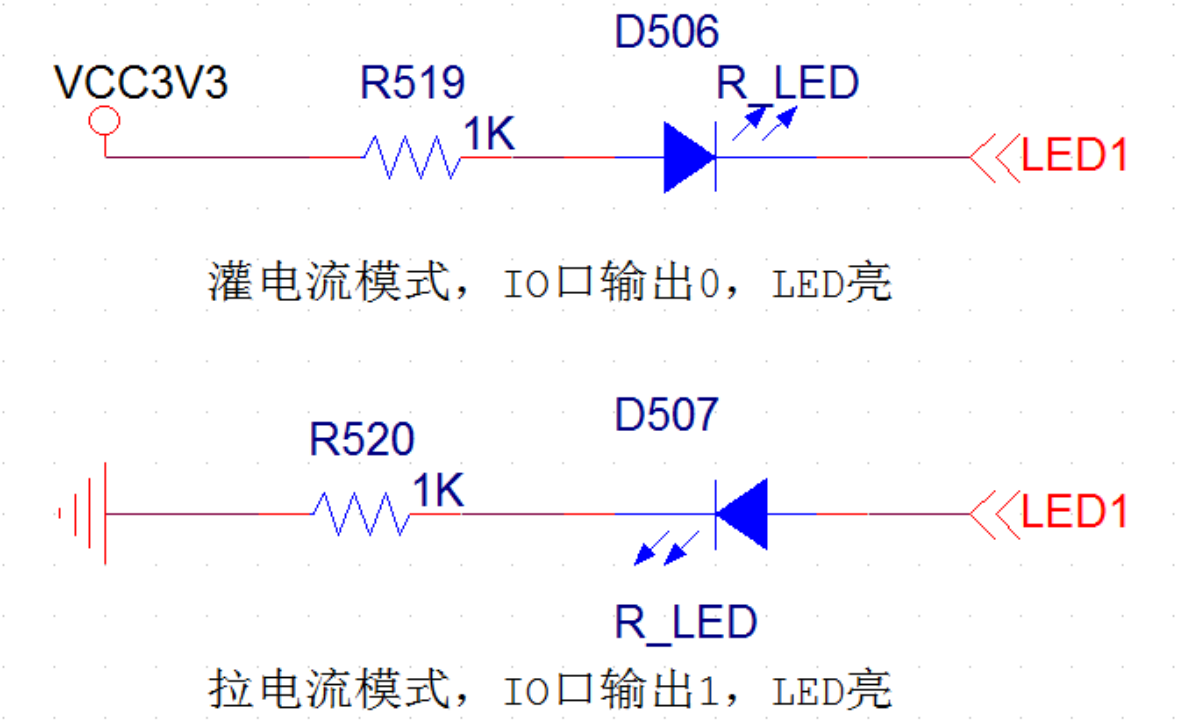
在资料文件夹内有一个发光二极管的规格书。LED 的规格书中有一个很重要的参数。《黄绿 0603 (33_40mcd)_PDF_C2289_2015-07-23.pdf》

3、最大绝对标称值（环境温度=25℃）

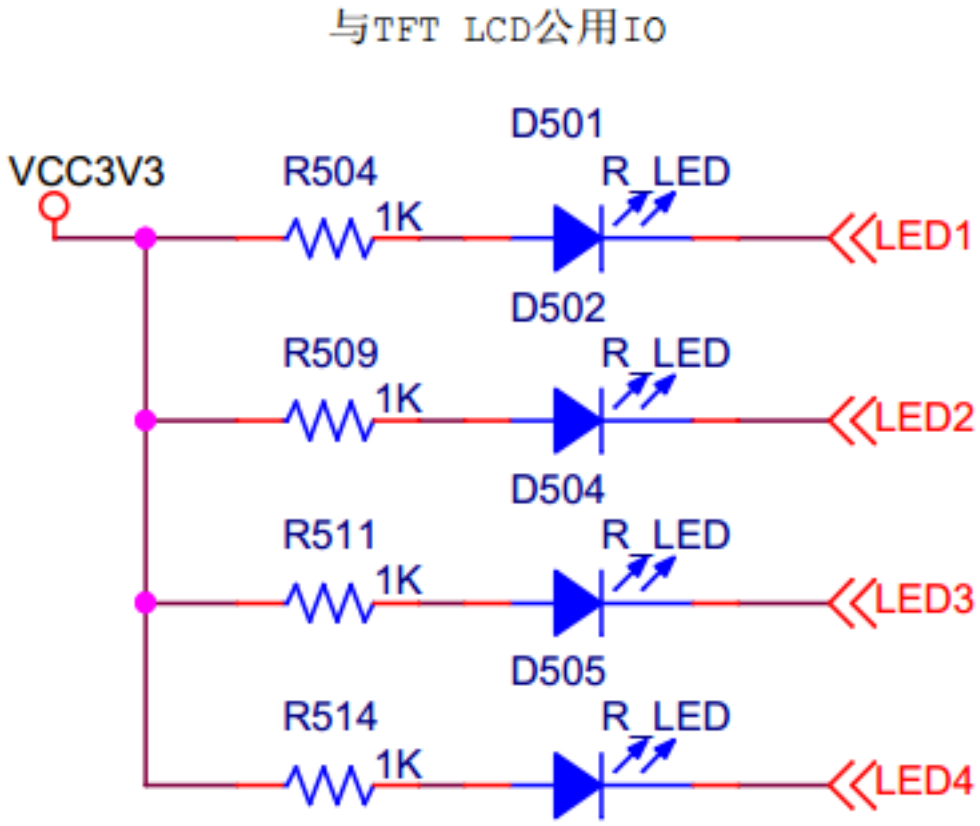
顺向电流	I_F	20	mA
顺向峰值电流 *1	I_{FP}	100	mA
反向电压	VR	5	V
焊接温度	Tsol	回流焊：250 ° C，8sec. 手工焊：300 ° C，3sec.	
使用温度	Topr	-40° C~+85	
储存温度	Tstg	-40° C~+85	

第 1 行，顺向电流，就是 LED 的工作电流，不能超过 20mA。

LED 驱动有两种方式：低电平（灌电流）和高电平（拉电流），如下图



我们的板子选用灌电流模式，4 位 LED 电路原理图如下：



LED 正极通过一个限流电阻接到 VCC3V3，也就是高电平。负极接到 IO 口。

当 IO 口输出低电平，电流从电阻流过，流过 LED，流入 IO 口。LED 就会发光。

限流电阻是防止流过 LED 的电流大于 LED 的**顺向电流**最大值。

电流的计算可以粗略如下：

```
I= (3.3-0.6) /1K = 2.7mA
```

这个电路的接法，电流是流入 IO 口的，就是灌电流。

8.6 编码与调试

请看视频，文档待补充

1. 对原理图找到对应的 IO, PE15 PD8 PD9 PD10
2. 拷贝例程初始化代码，修改为我们的 IO 口。
3. 单步运行，发现还没设置 IO 口，只是初始化就亮了，为什么?

4. 修改，先设置输入输出的值，再初始化 IO 口。
5. 写流水灯，无延时，单步运行，流水灯功能正常。全速，没有出现流水，但是亮度变暗了。为什么？

   讲程序，简一点 C 语言的编码知识。

   十进制十六进制的数值定义。

   加延时关键字 volatile

   C 语言知识点: 宏定义

8.7 作业问题

问题，为什么代码能控制 IO 口？

看库函数到底做了什么

END

20200120

CHAPTER 9

提高效率的工具

si

beyondcompare

参考百度网盘，目录 W108_F103_Tech\ref\5 tool 目录中的文档。

CHAPTER 10

点亮数码管

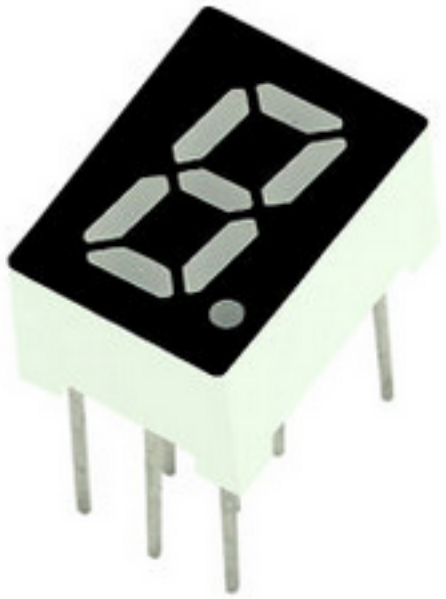
本节课利用已经学习的 LED 知识去控制一个 8 位数码管。

本节的原理比较简单。不需要多少时间讲。

更多时间是跟大家一起编码调试，从中学习一些编码思路和学习方法。

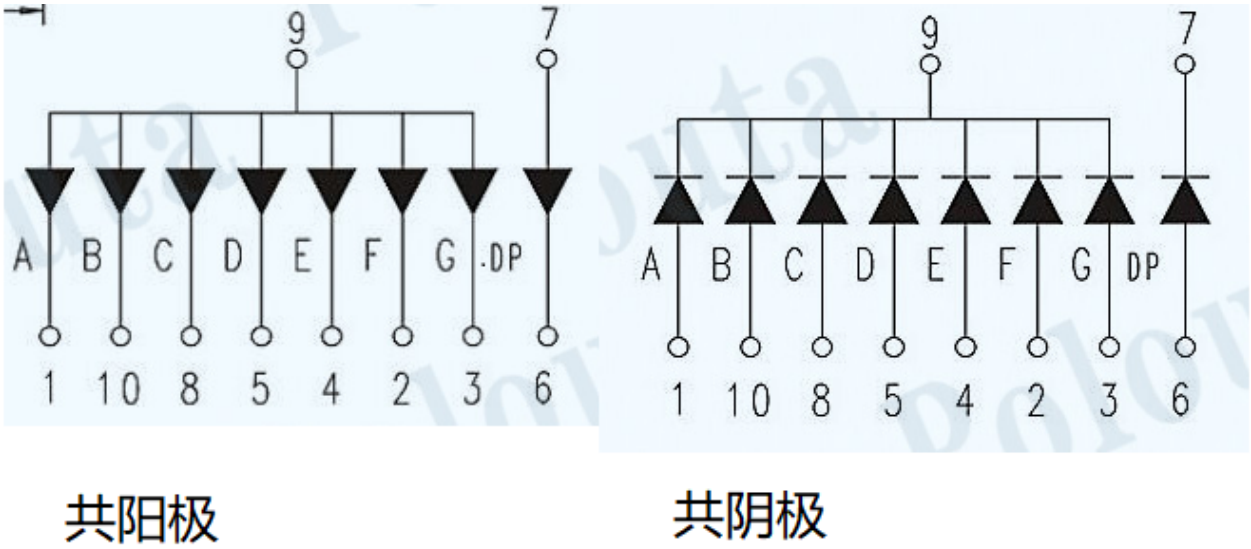
10.1 什么是数码管

数码管是什么？下图就是一个数码管



从硬件上个看，其实就是 8 个 LED 组合在一起。8 个 LED 应该有 16 个引脚，但是数码管上只有 10 个引脚。为什么呢?

请看下图:



1 个 LED 有两个引脚，要控制 LED，1 个引脚接控制信号，另外一个引脚接电源或者地（高驱动或低驱动，下同）。

那么，当有 8 个 LED，只需要 8 根 IO 口控制状态，其他 IO 全部接到地或者电源即可。

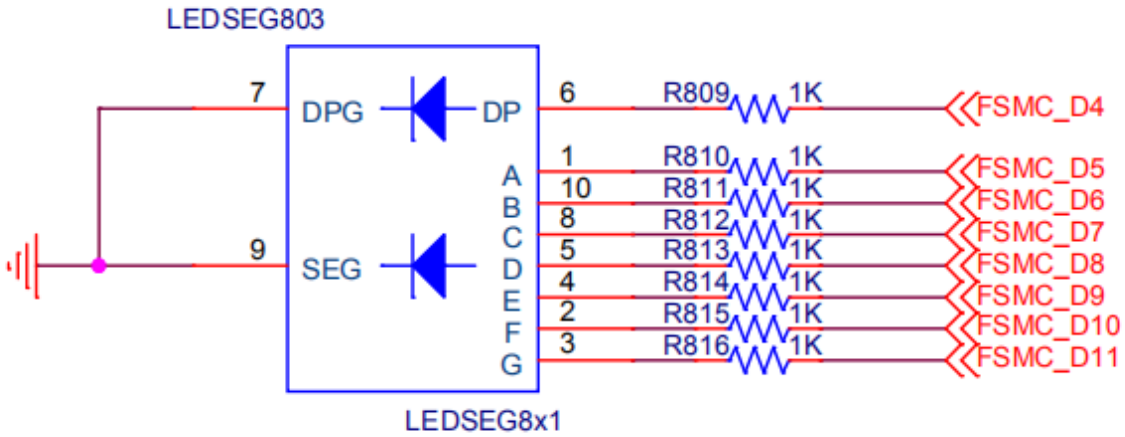
当用高驱动时，LED 负极全部接到地，这种数码管就叫做共阴极数码管。

当用低电平驱动时，LED 正极全部接到电源，这种数码管就叫做共阳极数码管。

数码管实物中，小数点 LED 通常单独引出两个引脚，由我们在电路图上连接在一起。

10.2 原理图

从原理可知，控制数码管需要 8 根 IO 口。下图就是原理图。



IO 口选择 PE7—PE14，这 8 个 IO 口是连续的，方便代码控制。

共阴极数码管，所有负极接到地。正极通过 1 个限流电阻接到控制 IO 口。

10.3 接口设计

什么是接口?

1. 两件事物之间的交互通道叫做接口。
2. 软件中，应用程序控制硬件用的函数，就是接口。
3. 从上往下看，即用户角度看硬件，用户想要什么功能?
4. 从下往上看，硬件有什么功能？能提供什么功能?（注意二者区别）

   刚刚开始学编程，这些设计理念可以了解，慢慢实践

一位数码管有什么功能?

1. 首先，有 8 个 LED 可以点亮。

2. 然后，8 个数码管可以组成数字。

从用户角度看，我们用数码管做什么呢？通常我们需要的功能是显示数字，而不是点亮某个段。

所以，我们就定义数码管的功能是：**显示数字**

函数接口如下：

```
/*
        定义一个 seg_display
        输入参数有 2 个，分别是 char 型的 num，char 型的 dot
        没有返回值。
*/
void seg_display(char num, char dot)
```

num 就是要显示的数字：0~9

dot 表示要不要点亮小数点

到此，我们编码前的学习和设计就完成了，下面开始实现功能。

10.4 编码调试

1. 第一步，点亮 LED

   这一步在上一节调试 LED 时已经学过，代码如下：

```
 /*
        调用库函数 RCC_APB2PeriphClockCmd
       传入两个参数 RCC_APB2Periph_GPIOD, ENABLE
       RCC_APB2Periph_GPIOD 是一个宏定义，
       ENABLE 是一个新定义的枚举类型 FunctionalState
 */
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
       RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);

       GPIO_ResetBits(GPIOE, GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_
↪11|GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14);
       /* Configure PD0 and PD2 in output pushpull mode */
       /* Configure PD0 and PD2 in output pushpull mode */
       /* 赋值给结构体变量 GPIO_InitStructure 的成员，
              注意，GPIO_InitStructure 是实体，所以用点，
              如果是一个结构体指针，就用->
              GPIO_InitStructure->GPIO_Pin
```

(continues on next page)

(continued from previous page)

```
      */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_
↪Pin_11|GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOE, &GPIO_InitStructure);

      GPIO_SetBits(GPIOE, GPIO_Pin_7);
      GPIO_SetBits(GPIOE, GPIO_Pin_8);
      GPIO_SetBits(GPIOE, GPIO_Pin_9);
      GPIO_SetBits(GPIOE, GPIO_Pin_10);
      GPIO_SetBits(GPIOE, GPIO_Pin_11);
      GPIO_SetBits(GPIOE, GPIO_Pin_12);
      GPIO_SetBits(GPIOE, GPIO_Pin_13);
      GPIO_SetBits(GPIOE, GPIO_Pin_14);
```

几个关键点：

1. 记得打开 IO 口时钟。
2. 先设置状态，再配置 IO 口为输出。**防止 IO 口配置完后 LED 闪一下**。

   （特别是在控制电机时，一定要配置为确定状态后再将 GPIO 外设连接到 IO 口）
3. 然后调用 GPIO_SetBits 控制 IO 口。
4. 用调试器一个一个 LED 数码管轮流测试，看是不是能点亮、熄灭。

   为什么要一个一个调试？因为这样测试可以要测试出硬件上 IO 口短路的情况。

   如果 8 个 IO 口一起控制亮灭，就无法知道 IO 口有没有短路。

2. 用直接控制 IO 的方法实现接口

   接口函数原型我们已经定义好：

   void seg_display(char num, char dot)

   既然我们都会控制 LED 了，那么，就用控制 LED 的方法实现这个函数。

```
GPIO_ResetBits(GPIOE, GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_11|GPIO_
↪Pin_12|GPIO_Pin_13|GPIO_Pin_14);

      if(dot  == 1)
      {
                          GPIO_SetBits(GPIOE, GPIO_Pin_7);
      }
```

(continues on next page)

(continued from previous page)

```
        switch(num)
        {
                case 0:
                        GPIO_SetBits(GPIOE, GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_
↪Pin_11|GPIO_Pin_12|GPIO_Pin_13);
                        break;

                case 1:
                        GPIO_SetBits(GPIOE, GPIO_Pin_9|GPIO_Pin_10);
                        break;

                case 2:
                        GPIO_SetBits(GPIOE, GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_11|GPIO_
↪Pin_12|GPIO_Pin_14);
                        break;

                case 3:
                        GPIO_SetBits(GPIOE,GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_
↪Pin_11|GPIO_Pin_14);
                        break;

                case 4:
                        GPIO_SetBits(GPIOE, GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_13|GPIO_
↪Pin_14);
                        break;

                case 5:
                        GPIO_SetBits(GPIOE, GPIO_Pin_8|GPIO_Pin_10|GPIO_Pin_11|GPIO_
↪Pin_13|GPIO_Pin_14);
                        break;

                case 6:
                        GPIO_SetBits(GPIOE, GPIO_Pin_8|GPIO_Pin_10|GPIO_Pin_11|GPIO_
↪Pin_12|GPIO_Pin_13|GPIO_Pin_14);
                        break;

                case 7:
                        GPIO_SetBits(GPIOE, GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10);
                        break;
```

(continues on next page)

(continued from previous page)

```
                case 8:
                        GPIO_SetBits(GPIOE, GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_
↪Pin_11|GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14);
                        break;

                case 9:
                        GPIO_SetBits(GPIOE, GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_
↪Pin_13|GPIO_Pin_14);
                        break;

                default:
                        break;
        }
```

进入函数后，先把所有的 LED 都熄灭。

然后用 if 语句判断 dot 参数，如果等于 1，就点亮小数点。

再用 switch 语句，根据 num 的值，点亮不同的 LED，组成 num 指定的数字。

如此，就是实现了功能，在 main 函数中调用这个函数，就能显示指定的数字了。(先用调试器加断点运行查看执行结果)

3. 用同时操作一组 IO 口的方法

上面的方法尽管实现了功能，但是你有没有觉得，这么简单的功能用了这么复杂的代码，是不是很不美？代码也很🈚嗦。

其实我们有更简洁的方法。很多同学可能想到直接把 GPIO_SetBits 函数的第二个参数定义为一个数字，看起来就更简洁了。

我们说的不是这种优化，我们能把代码优化得很简单。(**用 GPIO_SetBits 也可以优化，大家自己实验**)

先看下 GPIO_SetBits 和 GPIO_ResetBits 函数。这两个函数操作不同的寄存器，对指定的 IO 进行置位或清零。

如果我们去查规格书，可以发现我们可以直接设置一个寄存器输出 0 或 1，而不是置位或清零操作。不用看寄存器，直接看库函数也能看到。

注意额，不是 GPIO_WriteBit，这个函数只是将 GPIO_SetBits 和 GPIO_ResetBits 组合使用。

我们说的是 GPIO_Write，这个函数直接写 ODR 寄存器，你写入什么值，IO 口就输出什么值。

用这个函数还有一个好处：将 8 个 LED 当做一个整体。

代码如下：

```
if(dot  == 1)
                {
                        GPIO_SetBits(GPIOE, GPIO_Pin_7);
                }

                switch(num)
                {
                        case 0:
                           GPIO_Write(GPIOE, 0x3f00);
                           break;

                        case 1:
                           GPIO_Write(GPIOE, 0x0600);
                           break;

                        case 2:
                           GPIO_Write(GPIOE, 0x5b00);
                           break;

                        case 3:
                           GPIO_Write(GPIOE,0x4f00);
                           break;

                        case 4:
                           GPIO_Write(GPIOE, 0x6600);
                           break;

                        case 5:
                           GPIO_Write(GPIOE, 0x6d00);
                           break;

                        case 6:
                           GPIO_Write(GPIOE, 0x7d00);
                           break;

                        case 7:
                           GPIO_Write(GPIOE, 0x0700);
                           break;

                        case 8:
```

(continues on next page)

(continued from previous page)

```
                                GPIO_Write(GPIOE,0x7f00);
                                break;

                        case 9:
                                GPIO_Write(GPIOE, 0x6700);
                                break;

                        default:
                                break;
                }
```

这种方法跟置位方法的区别是：

置位需要两步，先清所有 IO。

再置位要点亮的 IO。

GPIO_Write 一步就可以将所有 IO 设置为指定值。

4. 用查表法

表是什么？**表，就是数组**。

从上面的 switch 我们可以看出，**不同数字对应不同的值**。而且：

switch 的参数 num 是连续的值 0~9

因此我们可以用表获取要写到 IO 口的值，然后，抛弃 switch。

```
/*
        定义一个全局数组 SegTab，数组成员类型是 uint16_t
        并初始化数组。
        这个数组是数码管显示 0-9 的段定义。
        请看 seg_display 函数，
        例如，第一个值是 0x3f00
        在 seg_display，取这个数，输出到 IO 口，LED 就能显示 0。
*/
uint16_t SegTab[10]={0x3f00, 0x0600, 0x5b00, 0x4f00, 0x6600, 0x6d00, 0x7d00, 0x0700,
↪ 0x7f00, 0x6700};


..........



if(dot  == 1)
                {
```

(continues on next page)

(continued from previous page)

```
                GPIO_SetBits(GPIOE, GPIO_Pin_7);
        }

        if(num >= 10)
                return;


        GPIO_Write(GPIOE, SegTab[num]);
```

用了表，SegTab 表中的值就是对应的数码管点亮值。

很长的 switch 变为一行代码。

5. 修复同时控制一组 IOBUG

用上面的函数做实验，我们会发现，其他 IO 口也被我们控制了，不应该这样。

原因是 GPIO_Write 一次性写一组 IO，但是我们只是用了其中的 8 个 IO。另外 8 根 IO 也被我们输出为 0 了。

解决这个问题的方法就是：

读回–> 修改–> 写进

记住，这是一个重要方法。

代码如下：

```
uint16_t tmp;

        if(dot  == 1)
        {
                GPIO_SetBits(GPIOE, GPIO_Pin_7);
        }

        if(num >= 10)
                return;

        tmp = GPIO_ReadOutputData(GPIOE);
        tmp = tmp&0x80ff;
        tmp = tmp | SegTab[num];
        GPIO_Write(GPIOE, tmp);
```

GPIO_ReadOutputData 读回当前 GPIOE 的输出值，注意，**是读输出值，而不是输入值**。

位与上 0x80ff，意思是将为 0 的位清零，这些位就是我们准备要设置的 IO 口。

位或上要写的值 SegTab[num]，或的功能是有 1 为 1。

再输出。

如此，GPIO_Write 操作就只会改变数码管的 IO 口。

请先理解位与和位或。

和逻辑与逻辑或是不一样的。

6. 小数点也合进来。

小数点的 IO 正好也在 GPIOE，同一组 IO 口，可以合并进来。

最终代码

```
                /*
                        写一个 IO 口，回读--写模式
                        为什么呢？因为我们只是使用了一个 IO 口中的几个管脚
                        比如 GPIOE，一共有 16 个脚， 我们只是用了 8 个脚。
                        GPIO_Write 函数是一次性设置 16 个脚。
                        如果不回读直接设置，那么，除了我们使用的 8 个脚之外的脚就会被意
外改变。
                */
                tmp = GPIO_ReadOutputData(GPIOE);
                /* 清空我们使用的几个管脚对应的位 */
                tmp = tmp&0x807f;//位与，注意和&& 的区别，&& 是逻辑比较
                /* 将我们要使用的几个管脚设置为我们需要的值，
                        比如，显示 0，那么值就是 SegTab[0]， 也就是 0x3f00，
                        或操作是有 1 为 1.
                        那么，经过下面的或操作，
                        我们的管脚，需要设置为 1 的位，就会是 1，
                        我们不使用的管脚，原来是 1 的，现在也不会被改变，还是 1.
                */
                tmp = tmp | SegTab[num];//位或，注意和 || 的区别

                if(dot  == 1)
                {
                        /*
                                如果需要显示数码管的小数点，就将对应位设置为 1
                                0x0080， 为 1 的位是 bit7，因为数码管的小数点接在
GPIOE.7 上。
                        */
                        tmp = tmp | 0x0080;
                }
                GPIO_Write(GPIOE, tmp);
```

本文档没列出所有代码，请查看例程代码获取完整版本。

10.5 结束

SegTab 这个数组，就是在数码管这个现实设备上显示数字的点阵字库。

END

2020-03-10

CHAPTER 11

程序各种要素说明

这节课我们用一个最简单的程序跟大家讲清楚程序的构成。（请看视频）

11.1 概述

- 硬件

首先要知道硬件的组成。

在前面章节我们说过，芯片包含 **Flash** 和 **RAM**。

他们虽然不是相同的东西，但是都属于同一个地址空间，32 位芯片的地址空间大小是 4G。

比如 ST32，FLASH 通常从 0X8000000 开始，而 RAM 就从 0x20000000 开始。

高级点的芯片，可能会有外部 SDRAM，内核也会为这 SDRAM 分配一段地址。

地址，就是地址，比如你们家的门牌号，酒店的房间号。

TODO 添加 STM32 芯片地址映射图。

- 程序

程序包含什么?

写代码的时候包含**函数过程**和**变量**。

编译得到的目标文件包含**函数过程**和变量的**初始化值**。

- 变量

变量有很多种：全局变量，局部变量、静态变量。。。

变量保存在哪里?

下面我们就从一个简单的程序来分析上面问题。

11.2 包罗万象的小程序

11.2.1 程序入口

程序入口，程序启动执行的第一条代码就叫程序入口。

或者说，芯片上电开始执行的第 1 条用户代码。

这条代码在哪?

我们写代码，通常都是从 main 函数开始写，我们也会把 main 函数叫做函数入口。

那么 main 函数是芯片复位的第一条代码吗?

实际不是，在执行 main 函数之前，已经执行了很多代码了。

其中最早执行的，也就是芯片复位的第一条代码，就是我们经常说的启动代码。

在我们的 STM32 工程中，启动代码就是 startup_stm32f10x_hd.s。

这是一个汇编文件。我们一起来看看这个启动代码。这个文件是一个汇编文件。

```
; 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
                DCD     0                          ; Reserved
```

这就是启动代码的入口。但是这里放的并不是代码，而是函数指针，这些函数指针就是中断向量。

DCD 的意思是分配一个空间来保存后面的值。

___Vectors 是一个标号，等下在分散加载文件中会提到。

现在我们只要知道这里保存的是中断向量，并且，复位也是一个中断。

当芯片复位时，芯片从这里找到对应的函数指针 Reset_Handler，然后跳到这个函数执行。

这个函数同样在启动文件中，如下：

```
; Reset handler
Reset_Handler   PROC
                EXPORT  Reset_Handler             [WEAK]
                IMPORT  __main
                IMPORT  SystemInit
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

复位后芯片做了什么呢?

1. 调用 SystemInit 函数。
2. 调用 ___main。

   SystemInit 在 system_stm32f10x.c 文件中，这个函数完成芯片的时钟配置。

   ___main 函数在哪呢？在工程中找不到的。是不是 main？不是。

   这是一个编译系统根据不同芯片生成的一个库函数。

   在这个库函数中完成变量（RAM）的初始化，居然后跳到真正的 main 函数执行。

11.2.2 函数

int main(void) 是我们接触的第一个函数。

函数的定义包含名称、参数、返回值。

我们可以定义一些子函数。

11.2.3 变量

- 全局变量

在函数外定义的叫做全局变量。

比如 main 函数中，SegTab 就是一个全局变量，这个变量的类型是一个 uint16_t 数组。

```
/*
        定义一个全局数组 SegTab，数组成员类型是 uint16_t
        并初始化数组。
        这个数组是数码管显示 0-9 的段定义。
        请看 seg_display 函数，
        例如，第一个值是 0x3f00
        在 seg_display，取这个数，输出到 IO 口，LED 就能显示 0。
*/
uint16_t SegTab[10]={0x3f00, 0x0600, 0x5b00, 0x4f00, 0x6600, 0x6d00, 0x7d00, 0x0700,␣
↪0x7f00, 0x6700};
```

变量是保存在 RAM 中的，我们都知道 RAM 是易失性存储，掉电数据就没了，那数组的些值是如何赋值给数组的呢?

这问题有两个方面:

1. 编译的时候，这些值会保存在代码中。同时还保存这些值和变量的关系。(细节暂时不研究)
2. 在启动代码中，执行 ___main 函数时，会根据这些关系执行初始化变量的过程，然后才执行用户的 main 函数。
3. 这个过程就是编译器生成的，如果你用一些很便宜的单片机，比如台湾的一些小单片机，这个过程就需要自己写代码实现，通常是用汇编写。

- 局部变量

在函数内定义的变量就是局部变量，例如 seg_display 函数中的 tmp 就是一个局部变量。

```
/*
   定义一个 seg_display
   输入参数有 2 个，分别是 char 型的 num，char 型的 dot
   没有返回值。
*/
void seg_display(char num, char dot)
{
          uint16_t tmp;
```

局部变量同样也是在 RAM 上。但是具体在哪呢? 地址是哪里?

局部变量的地址是不固定的。当调用函数时，从栈上分配。函数退出后就释放了。

- 变量有效域

  局部变量只在函数中有效。

  全局变量呢?

这个不是芯片的知识，是 C 语言的知识。和编译系统也有关系，在 MDK 中，全局变量在声明之后的 C 代码中都可以调用。

还可以通过 EXTERN 在外部文件中声明后调用。

局部变量可以通过 static 定义成类似全局变量，但仅限本函数使用。

static 还可以限制全局变量只在本文件有效。

11.3 分散加载文件

为什么启动代码就是上电执行的第一条指令呢?

因为我们用分散加载文件（链接文件）指定启动代码保存在芯片复位时指向的位置。

分析分散加载文件

; ** ;
*** Scatter-Loading Description File generated by uVision *** ;
****

LR_IROM1 0x08000000 0x00080000 { ; load region size_region ER_IROM1 0x08000000 0x00080000 { ; load address = execution address *.o (RESET, +First) *(InRoot$$Sections) .ANY (+RO) } RW_IRAM1 0x20000000 0x00010000 { ; RW data .ANY (+RW +ZI) } }

定义了 IROM1，地址和范围就是芯片 Flash 的定义。

其中，放在最全面的是 reset，也就是启动代码中定义的 `AREA RESET, DATA, READONLY`。

紧接则放置的是 InRoot$$Sections，这些代码是编译器链接时根据芯片和内核自动添加的。

我们可以认为这就是 ___main。

最后放其他代码，也就是 RO 段。

定义 IRAM1，就是 RAM，内存。

所有的 RW 段和 ZI 段都放在 RAM 中。

11.4 编译结果如何看？

1. 在 MDK IDE 界面有编译过程和最终结果：

```
compiling stm32f10x_sdio.c...
compiling stm32f10x_rcc.c...
compiling stm32f10x_usart.c...
compiling stm32f10x_spi.c...
compiling stm32f10x_tim.c...
```

(continues on next page)

(continued from previous page)

```
compiling main.c...
compiling stm32f10x_wwdg.c...
linking...
Program Size: Code=1336 RO-data=336 RW-data=24 ZI-data=1632
FromELF: creating hex file...
".\Objects\stm32_tech.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:19
```

Program Size: Code=1336 RO-data=336 RW-data=24 ZI-data=1632

这一句说明生成的目标文件大小，代码 1336 字节，RO（只读变量）336 字节，RW（读写变量）24 字节，ZI 数据 1632 字节。

1. 更细的情况，可以通过 map 文件查看。map 文件在 `Listings\`目录下，名字叫 `stm32_tech.map`。

   用文件编辑器打开就能看到内容。

   map 文件最开始是最细的地方，最后是整体情况。拖到最后，就能看到下面内容：

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

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

      1336         96        336         24       1632     236688   Grand Totals
      1336         96        336         24       1632     236688   ELF Image Totals
      1336         96        336         24          0          0   ROM Totals

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

    Total RO  Size (Code + RO Data)                 1672 (   1.63kB)
    Total RW  Size (RW Data + ZI Data)              1656 (   1.62kB)
    Total ROM Size (Code + RO Data + RW Data)       1696 (   1.66kB)

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

这些内容跟 IDE 中看到的基本类似。这是程序的总体情况。有一个地方需要注意：

Total RO Size (Code + RO Data)：

Total RW Size (RW Data + ZI Data)

Total ROM Size (Code + RO Data + RW Data)

Total ROM Size 就是最终的目标文件，也就是写到 FLASH 上的内容，请问，为什么包含 RW Data 的大小?

因为 RW 数据需要一个初始化值，这个值并不是凭空而来，而是代码中定义了，编译后保存在 ROM 中。

所以 ROM 会包含 RW。

往回看，则是 Image component sizes。map 文件每个大段之间用等号分开。

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

Image component sizes


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

       304         20          0         24          0       1705   main.o
         0          0          0          0          0     203136   misc.o
        64         26        304          0       1536        792   startup_
↪stm32f10x_hd.o
       298          0          0          0          0      12407   stm32f10x_gpio.o
        26          0          0          0          0      16706   stm32f10x_it.o
        32          6          0          0          0        557   stm32f10x_rcc.o
       328         28          0          0          0       1845   system_
↪stm32f10x.o

    ------------------------------------------------------------------------
      1058         80        336         24       1536     237148   Object Totals
         0          0         32          0          0          0   (incl.␣
↪Generated)
         6          0          0          0          0          0   (incl. Padding)

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

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

         8          0          0          0          0         68   __main.o
```

本段说明了组成程序的各个文件的信息，每一个.o 文件对应一个.c 文件。

从这我们还能看到程序暗地里使用了多少个函数库。

再往上：

`Memory Map of the image`，说明各文件使用的 RAM 分类情况。

`Image Symbol Table`，这是个文件中使用的函数和 RAM 情况。

在往上的内容我们基本也不会看了。

2. 这里我们关键看下函数入口的情况。

```
 RESET                                   0x08000000   Section      304  startup_
↪stm32f10x_hd.o(RESET)
    !!!main                              0x08000130   Section        8  __main.
↪o(!!!main)
    !!!scatter                           0x08000138   Section       52  __
↪scatter.o(!!!scatter)
    !!handler_copy                       0x0800016c   Section       26  __
↪scatter_copy.o(!!handler_copy)
    !!handler_zi                         0x08000188   Section       28  __
↪scatter_zi.o(!!handler_zi)
```

在 0x08000000，放的确实是向量表。芯片复位时就会从这里开始执行代码。

除了 ___main，还有一些我们不知道是什么东西的代码放在启动代码后面。

11.5 为什么能控制外设？

因为有外设寄存器。

外设寄存器是跟 RAM 一样的存在。(RAM 是可以读写的，外设寄存器有些不能写)。

这些寄存器链接到对应的硬件。

TOTO 请看规格书**地址空间 map 图**。

我们只要写这些寄存器，就能实现对应外设的功能。

我们看 ST 提供的库，比如下面函数

```
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->BSRR = GPIO_Pin;
}
```

要设置一个 GPIO 的输出，就是配置 GPIOx->BSRR = GPIO_Pin;

这时什么意思呢?

```
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

我们可以看到，BSRR 是结构体 GPIO_TypeDef 的内容。

```
GPIO_SetBits(GPIOE, GPIO_Pin_7);
```

使用这个函数的时候我们会传入一个 GPIOE，这是一个 GPIO_TypeDef 结构体指针。

而 GPIOE 的定义是下面这些宏定义：

```
#define PERIPH_BASE           ((uint32_t)0x40000000) /*!< Peripheral base address in the␣
↪alias region */

#define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)

#define GPIOE_BASE            (APB2PERIPH_BASE + 0x1800)

#define GPIOE                 ((GPIO_TypeDef *) GPIOE_BASE)
```

意思是：

在 0x40000000 这个地址上，是 PERIPH_BASE，也就是 PERIPH 外设的基地址，起始地址。

在 PERIPH_BASE 偏移 0x10000 的地方，放的是 APB2PERIPH，也就是 APB2 总线上的外设。

在 APB2PERIPH_BASE 偏移 0x1800 的地方，是 GPIOE 外设寄存器地址。

第四行则是把一个 uint32_t 的值强行类型转换为 GPIO_TypeDef 指针。

如此，就能通过 GPIOE 这个宏定义找到 GPIOE 外设的相关寄存器。

---

end

2020-04-18

CHAPTER 12

动态扫描数码管

前面我们学习了如何使用一位 LED 显示数字，很简单是吧?

现在我们加点难度。一位数码管只能显示一位数字，现在我们要显示 8 位数字（或者显示时间）。

那么我们就需要 8 位数码管，如果按照 1 位数码管的硬件接法，8 位数码管就需要 64 根 IO。

相当于 1 个 LED 使用 1 根 IO 口控制。

大家觉得可行吗? 当然可行，我们芯片有 100 根管脚，80 多根 IO。

但是你只打算用芯片控制 8 位数码管吗? 肯定不是嘛！这样的方案肯定是非常浪费 IO 的。

那怎么办嗯? 要解决这个问题，要用到一个原理，两个芯片。

12.1 一个原理

不知道大家是否了解过以前的胶片电影，一张一张的画片，连续播放就能看到活生生的，会动的人。

这是为什么呢?

> 原理是“视觉暂留”。
>
> 科学实验证明，人眼在某个视像消失后，仍可使该物像在视网膜上滞留 0.1 - 0.4 秒左右。电影胶片以每秒 24 格画面匀速转动，一系列静态画面就会因视觉暂留作用而造成一种连续的视觉印象，产生逼真的动感。

我们在数码管上能不能用这个原理呢?

8 个数码管都用同样的 IO 控制亮灭，轮流显示。

只要同一个 LED 的点亮间隔不大于 0.4 秒（实际要比这个小），那么我们就会一直看到这个数码管是亮着的。（和真正一直亮会有什么差别?）

这样我们就只要 8 根 IO 了?

如何选择 8 个数码管该点亮哪个?

前面我们用共阴极数码管，阴极是接到地线的。

我们可以用 IO 口控制阴极，只有对应的 IO 是低电平，这个数码管才有亮。如果阴极是高电平，数码管就不会亮。

如此，我们需要 8+8 根 IO 就够了。省去了 48 根 IO，太有成就了。

12.2 两个芯片

我们都是高兴太早，通常 IO 口还是不够，使用 16 根 IO 也是很浪费的。

那这么办呢？利用数字电路，有两个芯片能帮上我们的忙。

74HC138 和 **74HC595**。

138 是三八译码器，595 是 8 位串行输入、并行输出的位移缓存器。

> 74 是一系列数字功能芯片，注意中间字母的区别。我们选用的是 HC 类型，HC 表示是 CMOS 电平，或者简单说就是 3.3V 电压。

- 三八译码器

  三八译码器是什么？我们从数据手册一探究竟。

  打开数据手册，标题：**SNx4HC138 3-Line To 8-Line Decoders/Demultiplexers**

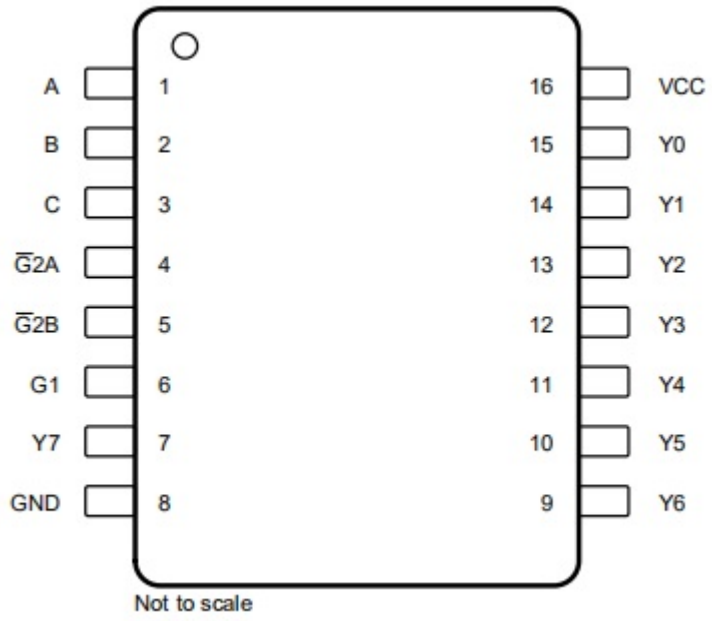
  翻译为中文就是：**SNx4HC138 3 线转 8 线译码器/多路分配器**，怎么转呢？往下看。

  我们选用的型号是 **SN74HC138PWR**。型号这些数字和字母都是什么意思呢?

  > SN74 是芯片系列。
  >
  > HC 是芯片种类。
  >
  > 138 是芯片具体型号。
  >
  > PW 是封装，TSSOP16。
  >
  > R 包装形式，编带。

  138 的功能：**用 3 根线的电平，选择 8 根线中的一根线输出低电平，其他输出高电平**。

  芯片电气信号



真值表如下：

Table 1. Function Table

INPUTS						OUTPUTS							
ENABLE			SELECT										
G1	$\overline{G}$2A	$\overline{G}$2B	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	X	H	H	H	H	H	H	H	H
X	X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	L	H	H	H	H	H	L	H	H	H	H
H	L	L	H	L	L	H	H	H	H	L	H	H	H
H	L	L	H	L	H	H	H	H	H	H	L	H	H
H	L	L	H	H	L	H	H	H	H	H	H	L	H
H	L	L	H	H	H	H	H	H	H	H	H	H	L

左边是输入，右边是输出。

ENABLE 信号通常我们输出 H-L-L，也就是默认使能，不进行控制。

C/B/A，38 译码器 3 根输入线，一共有 8 种组合。

输出信号 8 根，根据 3 根输入线的状态，选择其中 1 根输出**低电平**，其他线输出高电平。

因此，**选中的数码管是低电平，那么就只能用共阴极数码管**。

- 595 功能

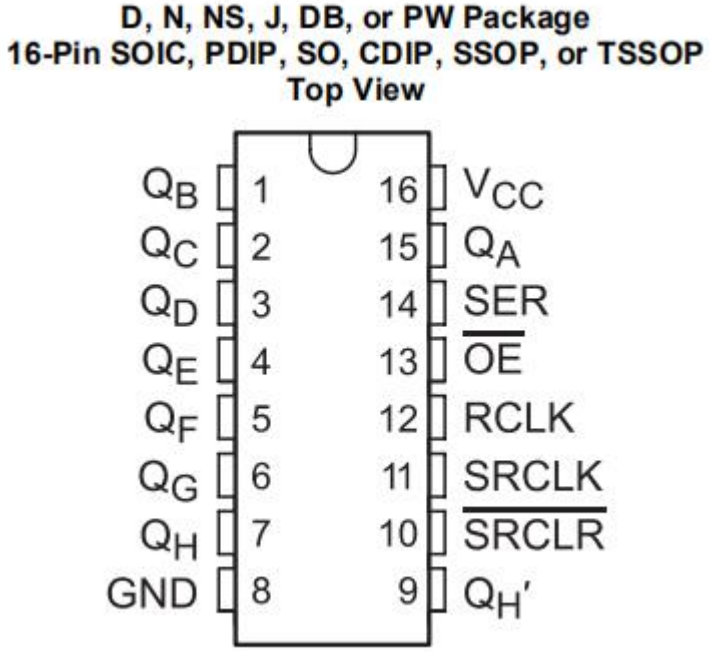
打开 595 手册

标题：**8-Bit Shift Registers With 3-State Output Registers**

意思：8 位移位寄存器，具有 3 态输出。

我们选用的型号是：SN74HC595PWR，名称含义与 138 类似。

芯片电气信号



时序图

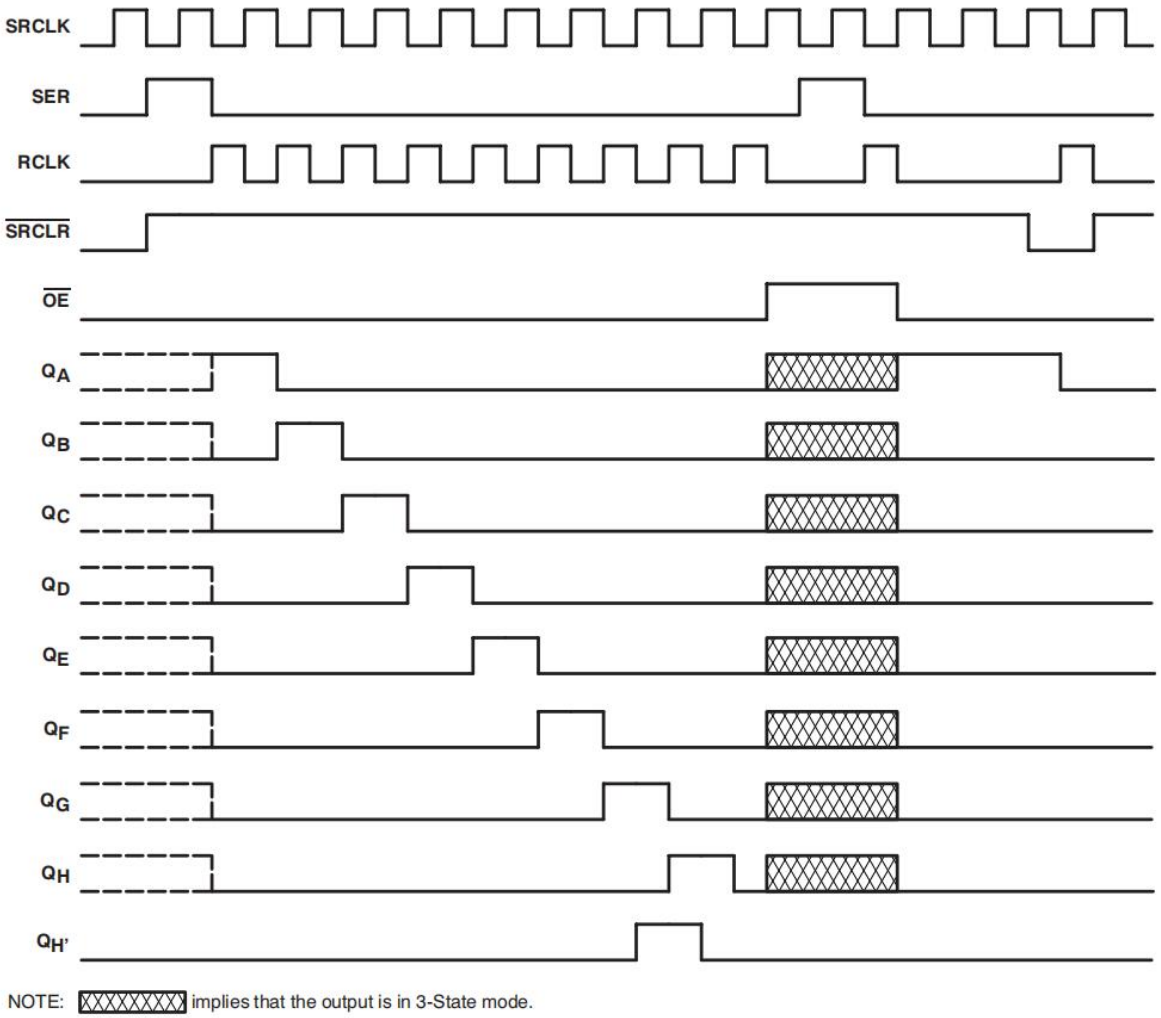


Figure 1. Timing Diagram

14 脚 SER 输入，11 脚 SRCLK 上升沿，从 14 脚输入 1 位数据。8 次之后，就有一个 BYTE 的数据保存在 595 中。当时钟继续输出，数据将从 9 脚输出，因此，可以通过多个 595 串联实现更多的移位位数。两个 595 就可以组成 16 位移位寄存器。

12 脚 RCLK 上升沿，保存在 595 中的 8 位数据，从 595 的 8 个并行输出引脚输出（OE 需要低电平）

10 脚 SRCLR 是复位脚，低电平有效，上电后输出高即可。

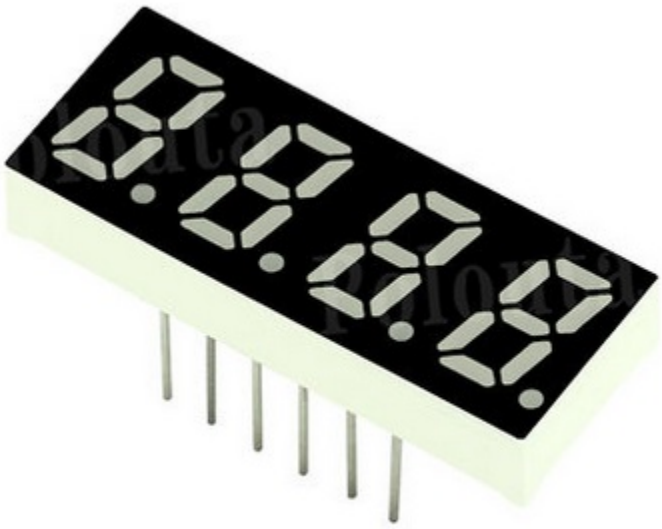
更多细节可参考：https://baike.baidu.com/item/74HC595/9886491

我们用三八译码器控制刷管的共阴极，595 控制数码管的正极。三八译码器决定哪个数码管亮，595 决定亮的内容。如此，我们就只需要 7 个 IO 口就搞定了。

12.3 硬件原理

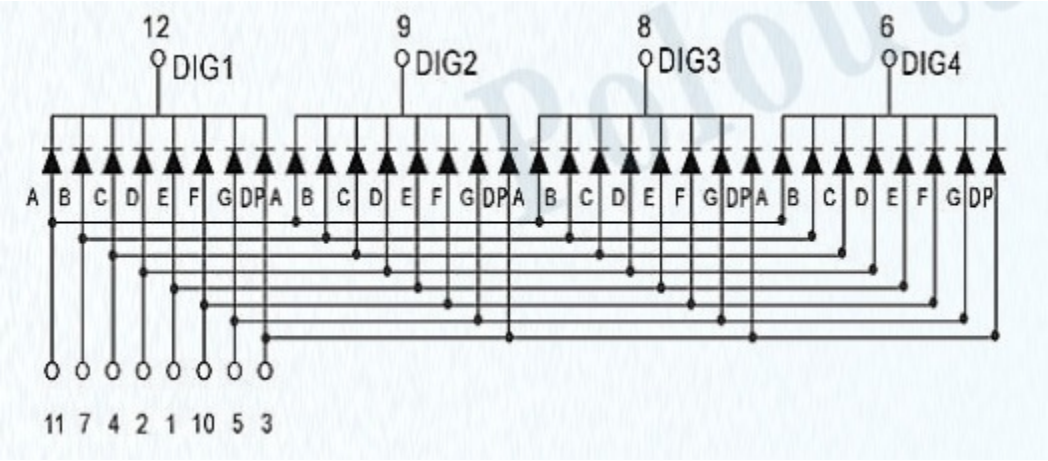
节省 IO 是一种共识，所以要用 8 位数码管时，我们不需要用 8 位单独的数码管组成。

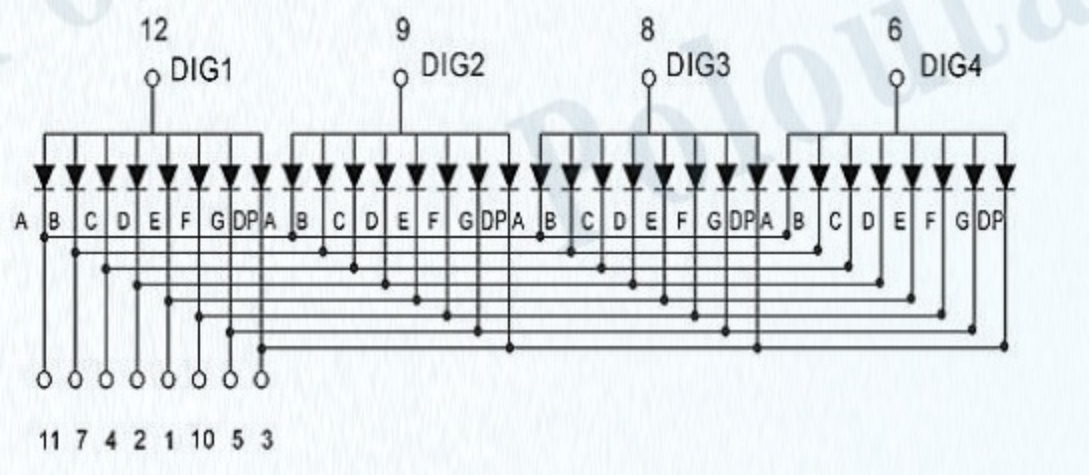
而是用 2 个内部连接好信号的 4 位数码管。如下图:



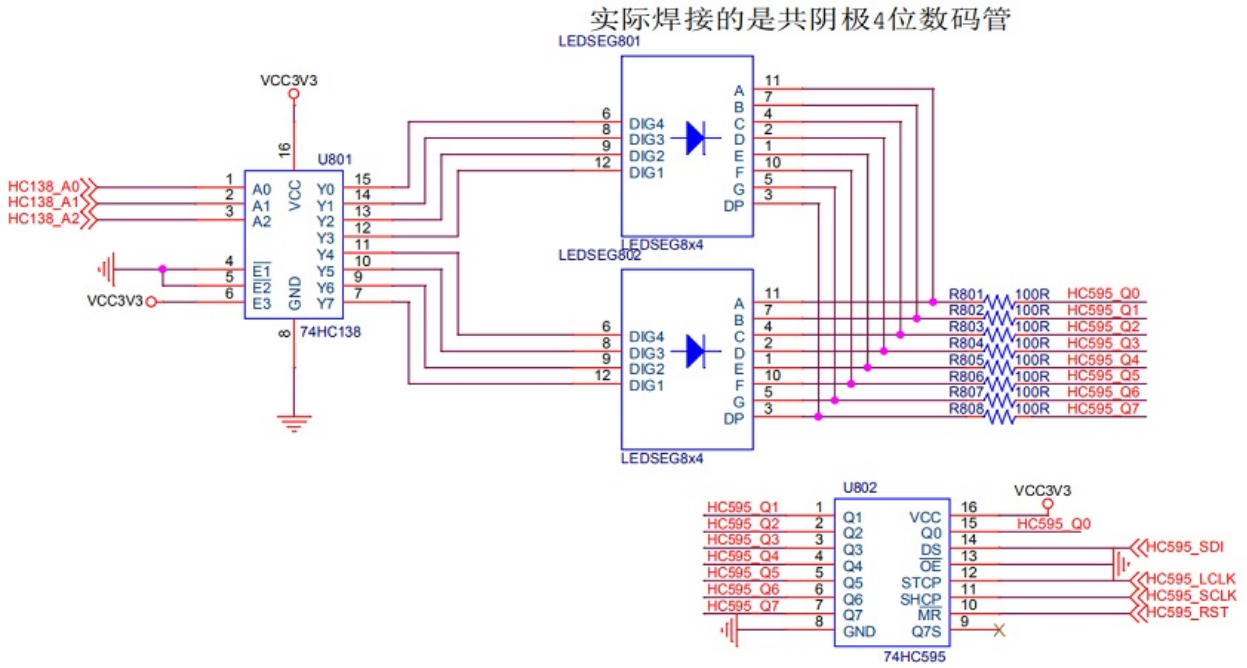
这种数码管内部已经将共用的信号连在一起。同样，也有共阴极和共阳极数码管之分。

内部连接信号如下:





电路图根据前面分析的原理设计，如下图：



12.4 调试

12.4.1 第一步

静态显示，38 译码器设定一个固定输出，选中一个数码管，控制 595 输出，让数码管显示不同数字。

- 初始化硬件

```
/*
        595_SDI--- ADC-TPX---PB0---数据输入
        595_LCLK---ADC-TPY---PB1---数据锁存---上升沿锁存
        595_SCLK---TP-S0---PC5---数据移位---上升沿移位
        595_RST---TP-S1---PC4---芯片复位--低电平复位

        A138_A0---FSMC_D2---PD0
        A138_A1---FSMC_D1---PD15
        A138_A2---FSMC_D0---PD14
*/
void seg_init(void)
{
        /* GPIOD Periph clock enable */
          RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
          RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
          RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);

          /* 38 译码器输入 0 , 选中第 4 个数码管 */
          GPIO_ResetBits(GPIOD, GPIO_Pin_0|GPIO_Pin_14|GPIO_Pin_15);
          /* Configure PD0 and PD2 in output pushpull mode */
          GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_14|GPIO_Pin_15;
          GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
          GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
          GPIO_Init(GPIOD, &GPIO_InitStructure);

          GPIO_ResetBits(GPIOB, GPIO_Pin_0|GPIO_Pin_1);
          /* Configure PD0 and PD2 in output pushpull mode */
          GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1;
          GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
          GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
          GPIO_Init(GPIOB, &GPIO_InitStructure);

          GPIO_ResetBits(GPIOC, GPIO_Pin_4|GPIO_Pin_5);
          /* Configure PD0 and PD2 in output pushpull mode */
          GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4|GPIO_Pin_5;
          GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
          GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
          GPIO_Init(GPIOC, &GPIO_InitStructure);

          /* 拉高复位信号 */
```

(continues on next page)

(continued from previous page)

```
        GPIO_SetBits(GPIOC, GPIO_Pin_4);

}
```

- 138 驱动

```
/*
        选择数码管, 控制 138 选中对应数码管
        pos 参数就是位置
*/
void seg_select(uint8_t pos)
{
        if (pos == 1) {
                GPIO_SetBits(GPIOD, GPIO_Pin_14);
                GPIO_ResetBits(GPIOD, GPIO_Pin_0|GPIO_Pin_15);
        } else if (pos == 2) {
                GPIO_SetBits(GPIOD, GPIO_Pin_0|GPIO_Pin_14);
                GPIO_ResetBits(GPIOD, GPIO_Pin_15);
        } else if(pos == 3) {
                GPIO_SetBits(GPIOD, GPIO_Pin_15|GPIO_Pin_14);
                GPIO_ResetBits(GPIOD, GPIO_Pin_0);
        } else if(pos == 4) {
                GPIO_SetBits(GPIOD, GPIO_Pin_0|GPIO_Pin_15|GPIO_Pin_14);
        } else if (pos == 5) {
                GPIO_ResetBits(GPIOD, GPIO_Pin_0|GPIO_Pin_15|GPIO_Pin_14);
        } else if(pos == 6) {
                GPIO_ResetBits(GPIOD, GPIO_Pin_15|GPIO_Pin_14);
                GPIO_SetBits(GPIOD, GPIO_Pin_0);
        } else if(pos == 7) {
                GPIO_ResetBits(GPIOD, GPIO_Pin_0|GPIO_Pin_14);
                GPIO_SetBits(GPIOD, GPIO_Pin_15);
        } else if(pos == 8) {
                GPIO_ResetBits(GPIOD, GPIO_Pin_14);
                GPIO_SetBits(GPIOD, GPIO_Pin_0|GPIO_Pin_15);
        }
}
```

- 595 驱动

```
/*
        输出一位数码管显示数据
```

(continues on next page)

(continued from previous page)

```
*/
void seg_display_1seg(uint8_t segbit)
{
        uint8_t tmp;
        uint8_t cnt = 0;

        tmp = segbit;

        cnt = 0;
        /* 拉低 595_LCLK*/
        GPIO_ResetBits(GPIOB, GPIO_Pin_1);

        while(1) {
                /* 拉低 595_SCLK*/
                GPIO_ResetBits(GPIOC, GPIO_Pin_5);
                /* 将数据从 SDI 发出去*/
                if((tmp & 0x80)== 0x00)//注意操作符的优先级
                {
                        GPIO_ResetBits(GPIOB, GPIO_Pin_0);
                } else        {
                        GPIO_SetBits(GPIOB, GPIO_Pin_0);
                }

                tmp = tmp<<1; //移位

                delay(100);
                /* 拉高 595_SCLK 移位数据 */
                GPIO_SetBits(GPIOC, GPIO_Pin_5);
                delay(100);

                cnt++;
                if(cnt >= 8)
                        break;
        }

        GPIO_SetBits(GPIOB, GPIO_Pin_1);
        delay(100);

}
```

- 应用

在 main 中初始化数码管，138 固定输出值，调用 595 驱动函数输出各种数字。

```
seg_init();

        /*
                第一步，调试 595 和 138 功能
                在第 1 个数码管显示 0-9
        */
        seg_select(1);
        seg_display_1seg(0x3f);
        seg_display_1seg(0x06);
        seg_display_1seg(0x5b);
        seg_display_1seg(0x4f);
        seg_display_1seg(0x66);
        seg_display_1seg(0x6d);
        seg_display_1seg(0x7d);
        seg_display_1seg(0x07);
        seg_display_1seg(0x7f);
        seg_display_1seg(0x67);
        seg_display_1seg(0x3f|0x80);
```

输出数字对应的数码管段值，列入一个数组，索引就是数字，比如显示数字 1，输出的数码管段值就是 SegTab1，也就是 0x06。

```
uint8_t SegTab[10]={0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x67};
```

单步运行看效果。

12.4.2 第二步

固定显示，595 输出固定值，调试 38 译码器，让数字在数码管上轮流显示相同的数字。

代码和第一步类似。

经过第一第二步调试后，138 和 595 驱动就完成了。

12.4.3 第三步

第一第二步只是实现了单个数码管显示，属于静态显示。

前面讲原理时讲过，8 位数码管使用动态显示方法。需要将 38 译码器和 595 配合，才能动态刷 8 位数据，我们现在尝试固定显示 12345678。

动态刷新是一个循环，因此放在 while 循环中实现。

代码如下：

```
        seg_select(1);
        seg_display_1seg(0x3f);

        seg_select(2);
        seg_display_1seg(0x06);

        seg_select(3);
        seg_display_1seg(0x5b);

        seg_select(4);
        seg_display_1seg(0x4f);

        seg_select(5);
        seg_display_1seg(0x66);

        seg_select(6);
        seg_display_1seg(0x6d);

        seg_select(7);
        seg_display_1seg(0x7d);

        seg_select(8);
        seg_display_1seg(0x07);
```

单步运行，看效果。发现一个问题，在调用 138 切换数码管时，会将前面显示的内容显示到下一个位置。

比如，数码管 1 显示 1，调用数码管 138 切换显示位置到数码管 2，这时，数码管 2 会显示 1。这是个问题。

我们全速运行程序看看效果。显示的内容并不是 87654321，而是 76543218，而且有重影，影子隐隐约约是我们要的效果：87654321。

如何解决这个问题呢?

方法是：**在切换显示位置前，将显示内容清零**，也就是将 595 的输出内容输出为 0。

增加一个 seg_clear 函数实现这个功能。实现如下：

```
void seg_clear(void)
{
        uint8_t cnt = 0;

        cnt = 0;
```

(continues on next page)

(continued from previous page)

```
        /* 拉低 595_LCLK*/
        GPIO_ResetBits(GPIOB, GPIO_Pin_1);

        while(1) {
                /* 拉低 595_SCLK*/
                GPIO_ResetBits(GPIOC, GPIO_Pin_5);
                /* 将数据从 SDI*/
                GPIO_ResetBits(GPIOB, GPIO_Pin_0);
                delay(100);
                /* 拉高 595_SCLK 移位数据 */
                GPIO_SetBits(GPIOC, GPIO_Pin_5);
                delay(100);

                cnt++;
                if(cnt >= 8)
                        break;
        }

        GPIO_SetBits(GPIOB, GPIO_Pin_1);
        delay(100);

}
```

在前面的测试函数中所有 seg_select 函数之前都添加本函数。

编译下载全速运行，效果正常。

12.4.4 第四步

经过第三步调试，8 位数码管的功能已经实现了。

那，驱动算完成了吗？没有。为什么？

先介绍一个很重要的概念：**时间片**。

> 什么是时间片呢？拿 LED 和 8 位数码管进行对比。
>
> LED，只要将 IO 置位，就能点亮，之后如果不改变 LED 状态，不需要再管它。
>
> 8 位数码管呢？因为我们用动态扫描方法，不能仅仅将 8 位数码管输出一次内容之后就不管了，要一直刷新。
>
> 前面原理也说过，每个数码管刷一次的时间间隔不能小于 24ms。
>
> 这种需要定时操作的，我们通常就说这个功能需要时间片。

好，了解了时间片。那么应用程序要如何使用呢?

> 应用程序只是想在数码管上显示一些数字而已，数码管怎么显示的，它是不管的。
>
> 为了显示数字，让应用程序间隔 24ms 就调用你的程序刷新显示，这明显不合理，专业术语叫强耦合，本来不相关的。

讲到这，不知道大家是否明白。不了解也没关系，后面再慢慢理解。

总之，矛盾就是：**应用只是想显示一数字，数码管驱动要时间片维持显示**。

怎么实现呢？用**缓冲**。缓冲就是一个组数，这个数组是应用和驱动之间的联系。

应用程序将要显示的内容放到缓冲。驱动将缓冲中的内容显示到数码管。

如此，就达到了最简单的**模块分离**。

> **程序设计中有一个理论：生产者和消费者**。
>
> 数码管驱动和应用虽然不是真正的生产者和消费者，但是使用缓冲的逻辑是相似的。

- 有 8 位数码管，就定义包含 8 个空间的数组。

```
/* 动态扫描 添加缓冲功能 */
/* 8 位数码管的显示内容 */
char BufIndex = 0;
/* 缓冲，保存的是对应数码管段值 */
char Seg8DisBuf[8]={0x7f,0x07,0x7d,0x6d,0x66,0x4f,0x5b,0x06};
```

- 定义一个函数，用于动态刷新数码管。这个函数最好放在定时或者 RTOS 的定时任务中执行。现在我们还没学会，可以放在 main 函数中的 while 运行。

```
/*
        动态刷新
        定时调用本函数,
        本函数对应用层屏蔽，意思是：应用层不知道我是通过动态刷新实现 8 位数码管功能。
*/
void seg_display_task(void )
{
        seg_clear();
        seg_select(BufIndex+1);
        seg_display_1seg(Seg8DisBuf[BufIndex]);

        BufIndex++;
        if(BufIndex >=8)
                BufIndex = 0;
}
```

- 定义一个函数，给应用程序调用，改变数码管缓冲的值。

```
/*
        segbit 数码管段值，为 1 的 bit 点亮
        seg 数码管位置，1~8
*/
void seg_fill_disbuf(uint8_t segbit, uint8_t seg)
{
        Seg8DisBuf[seg-1] =         segbit;
        return;
}
```

- 在 main 函数中调用数码管刷新功能。

```
    /*-----------------驱动--------------*/
        /* 使用显示缓冲方法，要改变显示内容，
        调用函数 seg_fill_disbuf 改变 Seg8DisBuf 中的内容即可 */
        seg_display_task();

        /*-----------------应用-----------------*/
        cnt++;
        if(cnt >= 1000) {
                cnt=0;
                disnum ++;
                if(disnum > 9)
                        disnum = 0;

                seg_fill_disbuf(SegTab[disnum], 1);

        }
        /*----------------------------------*/
        delay(1000);
```

驱动是数码管的内容，while 循环最后 delay 1000，也就是刷新间隔。现在没定时器，暂时定一个值，数码管不闪烁即可。

应用就是延时 1000 次个 delay(1000) 后，改变数码管 1 显示的数字，从 0 显示到 9。

编译下载看效果。

---

end

20200418

CHAPTER 13

代码结构调整

传说有人写的程序只有一个 main.c，一万行代码，这是一个神奇的故事

本节主要通过代码讲解如何模块化代码。

13.1 概述

代码结构调整有很多方式，今天只说最简单的。

1. 源码模块化—**接口标准化**
2. 硬件相关宏定义

13.2 源码模块化

模块化步奏：

1. 将一个功能、一个模块、一种设备的相关代码封装在同一个.c 源文件中。
2. 内部使用的函数用 static 宏控制，不允许外部使用。
3. 内部的定义，比如宏、结构体，定义在 c 文件。
4. 这个源文件有一个相同名字的头文件。
5. 对外的定义，宏、结构体等，定义在头文件。
6. 变量只能定义在源文件，不对外直接暴露变量。

我们就拿上一节的代码整理。

数码管，就是一个设备。这个设备提供的功能是什么呢？点亮对应的段？显示数字和小数点？

我认为：**点亮对应的段，才是八段数码管的根本功能**。

显示数字属于应用层功能。

为什么这样划分呢？有以下原因：

设备驱动尽量只提供自己能实现的功能本质，数码管的功能本质就是每个段点亮。

不同的段点亮后，组成的到底是数字还是字母，最好不要放在设备驱动中。

当项目越大，程序越复杂，参与开发的工程师多时，越能感觉到这样划分的合理性。

假如你数码管驱动只提供显示数字的接口，但是新项目需要显示一些自定义的字符，

那到底是改驱动还是改应用呢？

不过在实践上，数码管是一个小驱动，将显示数字接口放在设备驱动中，也并不是不可。

但设备较复杂，例如 LCD，可以在驱动和应用间封装一层 pannel 层，用于实现各种应用需要的接口。

- 建立一个 dev 目录

  在 dev 目录下建立两个文件：seg.c、seg.h

  把 `seg_init`, `seg_display_1seg`、`seg_select`、`seg_clear`、`seg_display_task`、`seg_fill_disbuf` 这几个函数拷贝到 seg.c 文件。

  同时拷贝 `BufIndex`、`Seg8DisBuf` 这两个变量。

- 对外的函数是：`seg_display_task`、`seg_fill_disbuf`、`seg_init`，这三个函数拷声明贝到头文件 seg.h，并且加上 extern 前缀。如下：

```
#ifndef __SEG_H__
#define __SEG_H__

extern void seg_init(void);
extern void seg_display_task(void );
extern void seg_fill_disbuf(uint8_t segbit, uint8_t seg);



#endif
```

  其中 `#ifndef` 等三个宏定义是为了解决重复包含的问题。每个头文件都会有这三行指令，后面的宏不一样而已。

- 为了防止外部函数调用内部函数，对没有在头文件声明的函数加上 `static`，在外部调用此函数时，编译会出错。

- 把调试过程的函数剪切到 seg.c，定义一个函数 seg_test。
- 把变量 `BufIndex`、`Seg8DisBuf` 也加上 static 前缀，seg.c 文件外部就不能直接操作这两个变量了。
- 在 main.c 头部增加 `#include "seg.h"`，包含 seg 驱动的头文件。
- 打开 MDK 工程，在工程目录新建一个目录，并添加 seg.c 到目录，并修改工程头文件路径。
- 重新编译下载。

13.3 宏定义

宏定义的第一个好处：将某个定义在一个地方定义，后续要修改这个定义的时候，只要改一个地方即可。

在数码管的驱动中，硬件 IO 口的定义在 `seg_init`、`seg_display_1seg`、`seg_select`、`seg_clear` 四个函数中都有使用，如果我们要修改某个 IO 的硬件连接，可能就需要修改这四个函数。

如果我们将这些函数中硬件相关的定义统一到一个宏定义，只要修改一个地方，就能改变整个数码管驱动的硬件连接。

```
#define SEG_138_A0_PIN        GPIO_Pin_0
#define SEG_138_A1_PIN        GPIO_Pin_15
#define SEG_138_A2_PIN        GPIO_Pin_14
#define SEG_138_A_PORT  GPIOD

#define SEG_595_SDI_PIN          GPIO_Pin_0
#define SEG_595_SDI_PORT  GPIOB

#define SEG_595_LCLK_PIN        GPIO_Pin_1
#define SEG_595_LCLK_PORT   GPIOB

#define SEG_595_SCLK_PIN        GPIO_Pin_5
#define SEG_595_SCLK_PORT   GPIOC

#define SEG_595_RST_PIN                GPIO_Pin_4
#define SEG_595_RST_PORT   GPIOC
```

宏定义如上，相关函数修改为宏即可，具体见代码。

编译下载验证

13.4 推荐

《林锐高质量 C-C++ 编程指南》

《嵌入式 C 精华》

end

20210819

CHAPTER 14

IO 输入与按键扫描

输出输入是 GPIO 的两种功能，前面我们点亮 LED、控制数码管，用的是 IO 口输出。现在我们开始学习 IO 口输入功能。

14.1 概述

IO 口输入的意思就是：读取 IO 口上的电平：高电平为 1，低电平为 0。

通常我们默认默认高电平就是 CPU 工作电压，比如 STM32 就是 3.3V。低电平就是 GND 电压，也就是 0V。

> 但是其实这并不严格，在 CPU 的规格书中标有输入电压电平范围，比如 0~0.6V，就认为是低电平，2.7V~3.3V 就是高电平。这些细节除非特殊应用场合，平常不用特别注意。
>
> 还有一个要点，一些复杂的芯片，会有多种输入电压，比如一些 ARM9 芯片，会有内核电压、内存电压、IO 电压，GPIO 的高低电平对应的是 IO 电压。

14.2 IO 口配置

一个 IO 口做为输入，通常有哪些可以选择的配置呢?

不同的 CPU 会有差异，也有共同点。现在我们看看 STM32 配置一个 IO 口为输出，有哪些配置。

不知道大家还是否记得 `GPIOMode_TypeDef` 枚举定义。请看下面：

```
typedef enum
{ GPIO_Mode_AIN = 0x0,
  GPIO_Mode_IN_FLOATING = 0x04,
  GPIO_Mode_IPD = 0x28,
  GPIO_Mode_IPU = 0x48,
  GPIO_Mode_Out_OD = 0x14,
  GPIO_Mode_Out_PP = 0x10,
  GPIO_Mode_AF_OD = 0x1C,
  GPIO_Mode_AF_PP = 0x18
}GPIOMode_TypeDef;
```

上面就是 STM32 IO 的模式，输入的模式我们讲过了。哪些是输入模式呢?

GPIO_Mode_IN_FLOATING、GPIO_Mode_IPD、GPIO_Mode_IPU 这三种模式就是输入模式。

从名字看，三种模式的区别仅仅是上下拉电阻配置不一样。分别是：FLOATING-浮空、IPD-下拉、IPU-上拉。

所以，如果用 ST 的库函数配置一个 IO 位输入，是非常简单的。

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
           GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
           GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
           GPIO_Init(GPIOB, &GPIO_InitStructure);
```

第 1 行代码选择要配置的 IO，可以同时配置多个。

第 2 行选择速度。

第 3 行关键，输入模式。

然后调用 GPIO_Init 函数初始化即可。

那么如何获取输入状态呢？看库函数头文件都提供了哪些功能函数就可以知道了。

只有两个函数:

```
uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx)
...
uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
```

这两个函数的功能，注释说的很清楚。

第 1 个函数是读整组 IO 的状态，所以返回值是一个 uint16_t，也就是一个 16 位的数，正好对应一组 IO 口。

第 2 个函数是读一个 Bit，也就是一个指定的 IO 口的值。返回值是一个 uint8_t，但是并不是会返回 8 位。看函数：返回值有两种，Bit_SET 和 Bit_RESET，高电平，返回 Bit_SET，低电

平返回 Bit_RESET。

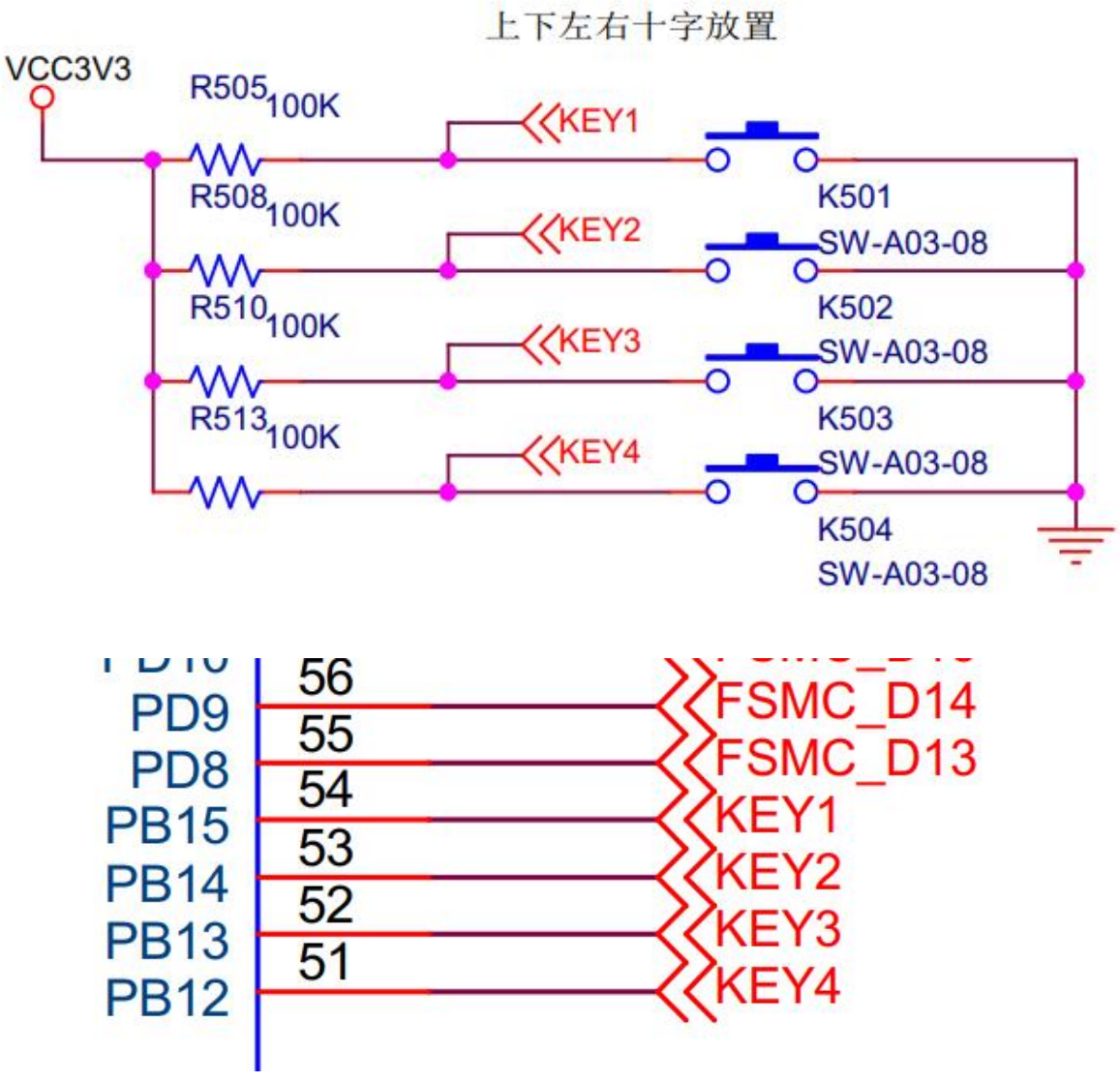
14.3 按键原理

理解了 IO 输入原理，按键原理就简单了。

14.3.1 按键硬件接法

一个 IO 只能输入两种状态，接在上面的按键，当然也只会有两种状态。

我们教学板上有 4 个按键接在 IO 口上，请看原理图：



4 个按键的原理图是一样的。

按键一端接到地，另外一端接到 IO 口。

在 IO 口这端，通过一个电阻接到 VCC，这个电阻就是我们常说的上拉电阻。

按键按下，IO 口接到地，输入低电平。按键松开，IO 口通过电阻接到 VCC，输入高电平。

上拉电阻

如果我们把 IO 口配置为上拉模式，内部已经有一个上拉电阻。但是这个电阻阻值通常是固定的。

在某些情况，我们需要灵活配置上拉电阻。

上拉电阻作用是当按键没有按下时，把 IO 口的状态维持在文档的高电平，防止程序读到意外状态。

但是，这个电阻还有一个重要的要点，就是电流。当按键按住，VCC 将通过这个电阻连接到地线。

流过的电流 =VCC/R。

所以这个电阻不能选太小的，太小电流大。比如在一些低功耗设备，按键会一直按住，进入睡眠，如果电阻很小，电流就很大。有时我们会用 1M 的电阻，这样一个按键的电流就只有 3uA。

但是电阻也不能选太大，越大的电阻本身寄生的电容电感就很大，很容有受到外部干扰。

通常，如果不是要求超低功耗的设备，用几 K 几十 K 的电阻。

超低功耗设备选择 1M 电阻，最大不超过 3M。

14.4 调试

14.4.1 第一步

使用单步调试，确定 IO 口输入有反应。

初始化代码：

```
/*  按键接在 PB12 PB13 PB14 PB15*/
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
           GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15;
           GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
           GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
           GPIO_Init(GPIOB, &GPIO_InitStructure);
```

测试代码

```
        while(1)
        {
                /* 读一组 IO 口输入 */
```

(continues on next page)

(continued from previous page)

```
                sta = GPIO_ReadInputData(GPIOB);
                /* 16 位 IO，按键用高四位，通过与操作 屏蔽其他低位
                如果高四位不等于 0xf，说明有按键按下 */
                if ((sta & 0xf000) != 0xf000) {
                        /* 没用的语句，用来测试是否进入这里 */
                        sta = 1;
                }
        }
```

编译后下载，进入 debug 模式。

在 sta=1 的地方加上断点，然后全速运行。

分别按下四个按键，看是否在断点处停止。

测试正常进入下一步。

14.4.2 第二步

IO 口能检测到按键按下了，但是还不能确认按键能用。

我们现在是检测到按下就停住了，正常程序是不会的吧?

如果全速运行，会发生什么呢?

前面我们调试了 8 位数码管，我们现在可以将 IO 口的状态显示在数码管上，看看全速运行时按键是什么状态。

在 8 位数码管的应用中，将定时刷新数码管应用改为下面

```
io_sta = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_12);
        if (io_sta == Bit_RESET) {
                seg_fill_disbuf(SegTab[0], 1);
        } else {
                seg_fill_disbuf(SegTab[1], 1);
        }

        io_sta = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_13);
        if (io_sta == Bit_RESET) {
                seg_fill_disbuf(SegTab[0], 2);
        } else {
                seg_fill_disbuf(SegTab[1], 2);
        }
```

(continues on next page)

(continued from previous page)

```
        io_sta = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_14);
        if (io_sta == Bit_RESET) {
                seg_fill_disbuf(SegTab[0], 3);
        } else {
                seg_fill_disbuf(SegTab[1], 3);
        }

        io_sta = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_15);
        if (io_sta == Bit_RESET) {
                seg_fill_disbuf(SegTab[0], 4);
        } else {
                seg_fill_disbuf(SegTab[1], 4);
        }
```

按下对应按键时将对应数码管显示为 1，松开按键，数码管显示 0。

下载测试，看起来还很好，能反应 IO 口状态变化。那是不是真的呢?

14.4.3 第三步

上一步测试，数码管能正常显示 IO 口状态，但是其实是不稳定的。

我们改一个程序试下：把第一个按键的程序改为下面：

```
        io_sta = GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_12);
        if (io_sta == Bit_RESET) {
                cnt++;
                if(cnt > 9)
                        cnt = 0;
                seg_fill_disbuf(SegTab[cnt], 1);
        } else {

        }
```

按键按下一次，显示数字加 1，到 9 后返回 0，循环显示。

编译下载测试。什么效果?

数码管 1 数字确实在加，但是并不是我们要的效果，我们想象的效果是按下一次，数码管加 1。现在按下一次，数码管变化几个数字。

为什么?

14.4.4 第四步

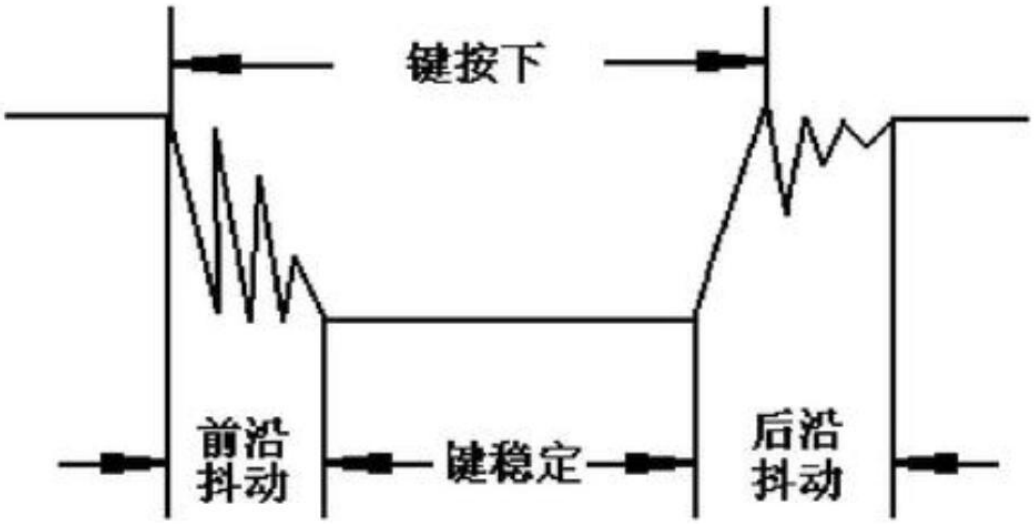
第三步的效果跟我们想象不一样。两个原因：

1. 按键抖动。
2. CPU 跑的很快，读 IO 口的状态也很快。

- 按键抖动

  按键，是机械结构，按下时接触点之间会弹动。我们认为按键按下，电平就立刻从高电平转为低电平，实际并不是这样。

  实际的图如下：



  无论按键按下还是松开，IO 口的状态都不是立刻、稳定变化。电平状态会来回抖动。

  当 CPU 读 IO 口的速度很快时，将会读都多个低电平。

  这个图只是说明了按键造成的抖动问题

  实际上，还有一个问题：IO 口的状态变化也是要时间的。

  IO 口从 1 变为 0，并不是立刻变化，0 变为 1 也不是立刻变化。是要时间的，尽管这个时间很快很快，那还是要时间的。

  普通的 IO 口速度只有 10M，高速的可以做到 100M 以上。

  这个知识点在这里暂时不会有影响，记住就好。

如何去抖动呢?

原理很简单：

重复读，当连续多次都是低电平时，按键就是低电平了。

这是很多例程去抖动的方法。方法是对的，但是状态抽象不对，状态抽象不对，程序写的逻辑就不好。

我认为按键去抖动，或者说按键扫描的逻辑是：

当状态连续多次变化到另外一种状态，说明按键状态变化了。

确定变化后，再根据状态判断是按下还是松开。

如此一来，松开和按下都有防抖动，而程序只有一个防抖动流程。

代码如下：

```
/*-----------------应用-----------------*/
new_sta = GPIO_ReadInputData(GPIOB);
new_sta = new_sta&0xf000;

/*  这里其实有一个提高可靠性的问题,
        old_sta 是稳定状态,
        sta 是本次读到的状态,
        如果本次读到的状态跟上次读到的状态不一样呢？                    */
if (sta != new_sta) {
        /* 不相等, 说明状态变化 */
        debcnt++;
        if (debcnt >= 10) {
                /* 已经连续 10 次状态变化, 说明变化已经稳定 */
                debcnt = 0;

                /*        求出变化的位        */
                chg_bit = sta ^ new_sta;
                /* 检测变化 IO 现在的状态 */
                if ((new_sta & chg_bit) != 0) {
                        /* 按键松开 */
                } else {
                        /* 按键按下 */

                        /* 根据变化的位, 判断是哪个按键按下 */
                        if(chg_bit == 0x8000) {
                                /* 应用 */
                                cnt++;
                                if(cnt > 9)
                                        cnt = 0;
                                seg_fill_disbuf(SegTab[cnt], 1);
                        }
                }
```

(continues on next page)

(continued from previous page)

```
                    /* 更新状态 */
                    sta = new_sta;
                }
            } else {
                debcnt = 0;
            }
```

new_sta 是当前读到的 IO 口状态。

两个状态比较，是否有变化?

有变化，debcnt 自加，去抖动。(这有个隐患，大家能想到吗?)

通过异或求出变化的 bit

用变化的 bit 和 new_sta 比较，判断到底是按键按下还是按键松开。

根据 cha_bit，确定是哪个按键按下。(用相等比较，有隐患？能想到吗?)

如果是最高位的按键按下，显示数字就自加。

编译下载测试，效果杠杠的，按下按键数字就加 1，没有误按键。

14.4.5 总结

这种扫描方法，有一个很容易理解的切入点：滑窗法

14.4.6 最后

那，现在按键驱动算完成了吗?

还差点，差什么呢？差模块化。

上面的程序，应用功能是数码管数字自加。这几行代码属于应用，却镶嵌在按键驱动中。

所以我们要把按键驱动整理为一个模块。按键驱动扫描到按键后，放在缓冲。

应用从缓冲中提取按键。

按键就是生产者，应用就是消费者。

---

end

20200419

CHAPTER 15

外部中断与按键触发

15.1 概述

- 中断是啥?

  中断是指芯片运行过程中，发生特殊事件，需芯片干预时，芯片自动停止正在运行的程序，并转入处理刚刚中断发生的事件，处理完后返回被打断暂停的程序继续运行。

在前面的程序中，程序没有使用中断，执行流程就只有 main 函数中的 while(1) 循环执行。

比如前面一章节按键扫描:

```
while(1) {
                 /*-----------------驱动--------------*/
                /* 使用显示缓冲方法，要改变显示内容，
                调用函数 seg_fill_disbuf 改变 Seg8DisBuf 中的内容即可 */
                seg_display_task();

                /*-----------------应用-----------------*/
                new_sta = GPIO_ReadInputData(GPIOB);
                new_sta = new_sta&0xf000;

                ......

                /*----------------------------------*/
```

(continues on next page)

(continued from previous page)

```
            delay(1000);
        }
```

这就是程序的全部流程，驱动和应用都放在这个流程中，循环执行。

- 为什么要中断?

在按键的 while 循环中，不断查询 IO 口的状态，判断是否有按键按下。有按键按下时，就进行去抖动。

这里有两个事情是要记住的:

1. **循环查询，速度再快，也是有间隔的**。
2. 连续不断查询，是要使用 CPU 时间片的。

那么，查询的越频繁，需要的 CPU 时间越多，那就不能去干其他事情了。

查询的不频繁呢? 间隔就大了，响应的速度就不及时了。

现在我们拿一个**按键输入**和你在家**等快递**对比。

按键按下，就相当于快递敲门。

在 while 循环中查询 IO 口，就相当于你想知道有没有人敲门，跑到门口看看有没有人。

如果你频繁的去门口查看是否有人来，甚至一直坐在门口等，你就没时间干其他事情。

如果你一个小时才去看一次，那很可能就会让快递员等一个小时，或者，快递员都不等你，没人开门就走了。**这样，你就会丢掉一个快递**。

如果使用中断呢

相当于在门口装个门铃，你在家里做其他事情，快递来了，按下门铃，然后你出去开门，拿了快递后，回家接着做刚才做的事。

15.2 STM32 中断管理

我们常说某某芯片的中断管理，就像本节标题。

实际上，中断是内核在管理。所以，要搞懂芯片的中断，需要学习两部分知识。

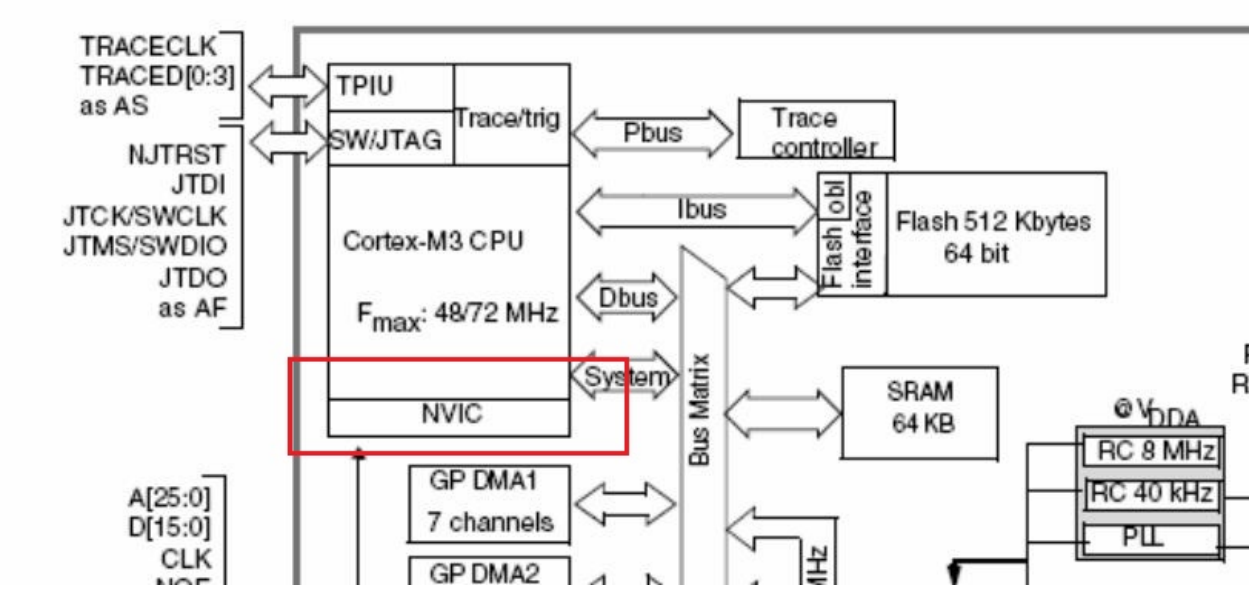
1. 内核如何管理中断。(中断响应)
2. 芯片中断如何连接到内核。

15.2.1 NVIC

NVIC 的全称是 Nested vectoredinterrupt controller，即嵌套向量中断控制器。

NVIC 属于内核的一部分，这东西怎么实现的我们不需要管。只需要知道:

1. 要在 NVIC 中使能对应中断，才会产生中断
2. 通过 NVIC 可以配置中断的优先级，优先级有两种：响应优先级和嵌套优先级。



内核还决定了中断向量表。

什么是中断向量？这个问题在知乎有一个讨论。

我的观点是何必浪费时间。

15.2.2 中断向量表

在 `startup_stm32f10x_hd.s` 文件中，有以下这段代码:

```
; 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
                DCD     0                         ; Reserved
                DCD     0                         ; Reserved
```

(continues on next page)

(continued from previous page)

```
                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

                ; External Interrupts
                DCD     WWDG_IRQHandler         ; Window Watchdog
                DCD     PVD_IRQHandler          ; PVD through EXTI Line detect
                DCD     TAMPER_IRQHandler       ; Tamper
                DCD     RTC_IRQHandler          ; RTC
                DCD     FLASH_IRQHandler        ; Flash
                DCD     RCC_IRQHandler          ; RCC
                DCD     EXTI0_IRQHandler        ; EXTI Line 0
                DCD     EXTI1_IRQHandler        ; EXTI Line 1
                DCD     EXTI2_IRQHandler        ; EXTI Line 2
                DCD     EXTI3_IRQHandler        ; EXTI Line 3
                DCD     EXTI4_IRQHandler        ; EXTI Line 4
```

这就是中断向量表，表内是什么呢?

> 第 2 行, AREA RESET, DATA, READONLY, 指定了这段代码放在 RESET 段, 这个 RESET 会在分散加载文件中指定在芯片启动的位置 0x8000000。所以芯片能在复位时执行 Reset_Handler 函数。从这里可见，复位也是一个中断。
>
> 后面一堆 DCD，DCD 的意思是，分配一个 4 个字节，内容是后面跟着的标志，通常是一个函数名，也就是一个函数指针，一个函数的地址。比如：DCD EXTI4_IRQHandler，意思就是把 EXTI4_IRQHandler 这个函数的地址放在这个地方。如此一来，当产生 EXTI4 中断时，芯片就会自动到这个位置取这个函数地址，然后执行这个函数。

15.2.3 外设中断

一个芯片有很多外设，每个外设都有各种中断。NVIC 决定了芯片能响应多少中断，但是，芯片能在一个中断入口中实现多种中断。

比如，串口 5 在中断向量表中只有一个中断入口，但是串口 5 有很多中断，接收到数据中断，发送数据完成中断，接收溢出中断等。

有个更特别的，EXTI15_10_IRQHandler，从名字就知道，这个中断入口时外部中断 15~ 外不中断 10，总共 6 个外不中断的入口。

所以，要正确配置一个中断，除了 NVIC 之外，对应外设的中断配置相当重要。

下面我们看看 STM32F103 这款芯片的 IO 口外部中断。

15.2.4 STM32 EXIT 中断

1. 所有 IO 都支持中断。
2. 中断方式有电平、边沿（上升沿、下降沿）
3. 中断线只有 16 根，所有 PORT 的相同编号的 IO，共用中断线，只能用同时有一个 IO 具有中断功能。
4. 中断入口并没有 16 个，EXTI15_10_IRQHandler，EXTI9_5_IRQHandler。

15.3 外部中断例程

参考官方例程:STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Examples\EXTI\EXTI_Config

配置中断如下，分 4 部分:

```
void EXTI0_Config(void)
{
  /* Enable GPIOA clock */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

  /* Configure PA.00 pin as input floating */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  /* Enable AFIO clock */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);

  /* Connect EXTI0 Line to PA.00 pin */
  GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);

  /* Configure EXTI0 line */
  EXTI_InitStructure.EXTI_Line = EXTI_Line0;
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);
```

(continues on next page)

(continued from previous page)

```
  /* Enable and set EXTI0 Interrupt to the lowest priority */
  NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0F;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

1. IO 口初始化，将 IO 配置为浮空输入。(纯粹的浮空是不行的，外部应该带上下拉)
2. 使能 AFIO 时钟，并用函数 GPIO_EXTILineConfig 将 GPIOA 的 GPIO_Pin_0 连接到中断源 GPIO_PinSource0。
3. 配置外部中断，调用 EXTI_Init 函数配置 EXTI_Line0 为中断模式，Rising 上升沿触发，ENABLE。
4. 配置 NVIC，调用函数 NVIC_Init 配置 EXTI0_IRQn，Pre 优先级 () 为 0x0f，Sub 优先级 () 也配置为 0x0f，使能 (ENABLE)

   对于 NVIC 的配置，特别是优先级的配置，需要先设置优先级分组。不同的分组支持不同的 pre 和 sub 优先级。

   此处我们暂时不管优先级问题。

中断配置好后，还需要编写中断服务函数。通常我们统一将中断函数放在 `stm32f10x_it.c`

```
void EXTI0_IRQHandler(void)
{
  if(EXTI_GetITStatus(EXTI_Line0) != RESET)
  {
    /* Toggle LED1 */
     STM_EVAL_LEDToggle(LED1);

    /* Clear the  EXTI line 0 pending bit */
    EXTI_ClearITPendingBit(EXTI_Line0);
  }
}
```

函数名 EXTI0_IRQHandler 跟中断向量表中的名字一致。

进入中断后，先判断是不是 EXTI_Line0 中断，是才进入处理，STM_EVAL_LEDToggle 转换 LED 状态，这就是用户功能。然后 EXTI_ClearITPendingBit，清楚中断标志，如果不清楚，中断就会重复进入。

由此可知一个中断的响应流程:

IO 状态变化–> 产生 EXIT 事件标志–> 如果使能了中断，就会产生设备中断标志，并产生信号

发送给 NVIC–>NVIC 使能了中断，芯片自动操作，从当前执行位置跳转到中断向量指定的函数–> 执行中断服务程序–> 执行完之后就返回原来的位置继续执行。

15.4 按键中断

板子的 4 个按键连接在 GPIOB 的 12/13/14/15 四个 IO。

参考例程配置即可，代码如下：

```
void EXTI15_10_Config(void)
{
        EXTI_InitTypeDef   EXTI_InitStructure;
        GPIO_InitTypeDef   GPIO_InitStructure;
        NVIC_InitTypeDef   NVIC_InitStructure;


  /* Configure PG.08 pin as input floating */
 /*  外部中断的 IO，属于输入 IO，先配置未输入
        输入 IO 有 3 中模式，浮空，上拉，下拉。做为按键通常是上拉，按下按键就是接地。
  */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  /* Enable AFIO clock */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);

  /* Connect EXTI8 Line to PG.08 pin */
  /*  配置中断线，中中断源接到制定的 IO 口 */
  GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource12);
  GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource13);
  GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource14);
  GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource15);

  /* Configure EXTI8 line */
  /* 使能外部中断 */
  EXTI_InitStructure.EXTI_Line = EXTI_Line12|EXTI_Line13|EXTI_Line14|EXTI_Line15;
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);
```

(continues on next page)

(continued from previous page)

```
  /* Enable and set EXTI9_5 Interrupt to the lowest priority */
  /* 配置 NVIC */
  NVIC_InitStructure.NVIC_IRQChannel = EXTI15_10_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0F;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

中断服务函数也是一样的。

```
/*
        EXTI15_10_IRQHandler 中断服务函数名，要跟中断限量表一致。
*/
void EXTI15_10_IRQHandler(void)
{
        /* EXTI15_10_IRQHandler 是 10~15 中断线的入口。
           产生中断后，只能通过查询 EXTI 状态来判断到底是那根 IO 产生了中断。
        */
  if(EXTI_GetITStatus(EXTI_Line12) != RESET)
  {
        app_display_num(4, 0, 1);

        /* Clear the  EXTI line 8 pending bit */

        /* 请中断标志 */
        EXTI_ClearITPendingBit(EXTI_Line12);
  }

......
```

响应中断后，调用 app_display_num 在数码管上显示数字。

编译下载，按 4 个按键，对应在最后一位数码管上显示 1~4 四个数字。

15.5 使用中断的按键处理

1. 中断作为启动，不需要一直查询 IO 状态。
2. IO 口产生中断后，禁止中断，开始用原来的方式扫描按键。

3. 扫描结束后，开启中断。

2020-05-04

end

CHAPTER 16

定时器

在前面调试数码管时，我们使用了 delay 函数进行延时，延时后更新数码管上的数字。

通过 delay 的延时有两个特点：

1. 如果要延时比较准确，需要根据主频计算芯片机器指令执行时间，根据指令周期安排 delay 函数中的代码。

   还要考虑中断的影响。无法灵活定时，代码安排也很麻烦，吃力不讨好。

2. 使用 delay，是一种硬延时，也就是常说的死等。通常我们需要的是定时，而不是死等延时。

因此，只会在一些很短的死等延时才会用 delay 的方式。比如某个外部芯片需要一个 1us 的低电平复位信号，我们就可以在代码中拉低 IO，硬延时 1us，再拉高 IO。

定时功能在程序中基本是不可避免的。例如：

1. 间隔扫描按键。
2. 数码管动态刷新。
3. RTOS 任务调度。
4. 等等

定时功能是程序的基本功能，因此芯片基本都标配了定时器，奢侈的芯片甚至配套了十几个。

程序本质是流程，是时间流，是时间线。

扩展

定时，实际也可以说是计时/计数。

拥有计时能力后，我们就可以做很多事情，为了方便我们实现功能，减少开发难度，芯片进一步将我们要实现的功能集成到定时器。

比如:

1. 将 IO 口输入的信号做位时钟，用定时器统计 IO 口的脉冲数，这叫做**输入计数器**
2. 用定时器统计 IO 口上的脉冲宽度，这是**输入捕获**
3. 用定时器配套 IO，输出 PWM。比如输出 1K 方波推动蜂鸣器；比如更复杂的 PWM 控制电机。这是定时器 **PWM 输出**

定时器的基本功能组成

STM32 的定时器框图

定时器中断

中断中不能进行太多延时

定时器的应用

计算定时，定时功能其实就是一个闹钟。

讲清楚定时器框图

我们只配置定时 1S，刷新数码管

说清楚中断不能执行太多程序。

先用调试器测试能跑到定时中断，再详细计算定时时间。

引出系统时钟设置问题。

测试增加单独的 LED 灯显示

CHAPTER 17

PWM 输出与蜂鸣器

PWM 输出是定时器的一个功能。

什么是 PWM?

配合一根 IO 才能输出。

查原理图和 datasheet，看 IO 是哪个定时的的哪个通道。

PC6 TIM8 CH1

官方例程 STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Examples\TIM\PWM_Output

除了设置定时器，还需要配置 PWM，还需要配置 IO 口。

查蜂鸣器响应频率，4K

调试时，可以用示波器测试是否有 PWM 输出。

CHAPTER 18

串口与调试信息输出

串口 UART 通用异步, USART 增加了同步功能。TX RX 讲时序，提流控。串口有附加功能，红外，IC 卡协议。在 ST 的例程，有 IrDA，Smartcard，这就是串口的附件例程。

讲框图

学习，最开始通常先移植 polling，也就是通过查询方式收发数据。然后使用串口中断 Interrupt，还可以考虑使用 DMA 方式

那我们先移植 POLLING，查看例程，是一个双串口收发的，

我们改一下，只用单串口，连接电脑收发。宏定义在 platform_config.h

先移植初始化，时钟要记得开，IO 初始化，读例程，然后修改

讲 sizeof，讲宏

讲硬件

试验多输出一个\0，讲字符串结束符的知识。

开始讲接收，

加上发送，，加断点，发 4 个，有断点就收不到了。

门铃与来客，中断，的对比说明。

中断

调试信息参考 printf 例程，不成功。需要把 microlib 勾上。

CHAPTER 19

ADC 模数转换

ADC 是什么？有什么关键参数?

STM32 的 ADC 硬件上又是什么样子的？讲框图。

分析例程 ADC1_DMA

PA4 ADC12_IN4

ADC 操作过程：1 配置

如何转换为电压?

CHAPTER 20

DAC 数模转换

波形输出

CHAPTER 21

时钟和日历

讲解掉电保存区域 RTC

例程 STM32F10x_StdPeriph_Examples\RTC\Calendar

CHAPTER 22

I2C 与 EEPROM

为了讲清楚 I2C 的时序，我们先用 IO 口模拟 I2C

1 讲时序图，讲如何看时序图

2 讲 EPROM 的功能。

讲 I2C 流程怎么写，地址要定义为 7 位，定义传输函数。

讲 24C 的接口函数怎么写。接口函数是面对用户的，要对用户屏蔽特性，提供共性。屏蔽细节，提供功能。

地址要说明，不要定义为 8 位，按照惯例定义为 8 位。24C02 的地址有点不一样，最低位要用来当地址位了。因为我们这个是 512 字节的，一个地址字节只能寻址到 256，所有要一个 A8 当寻址。

先降低速度调试。

解决 3 个 BUG ，测试程序成功。

如何验证？1 在写之前先读全部的出来。2 写多种数据。3 断电

芯片支持 page 模式，请自己实现。

ST 例程 STM32F10x_StdPeriph_Lib_V3.5.0\Utilities\STM32_EVAL\Common

CHAPTER 23

SPI 与 SPI FLASH

官方有例程

STM32F10x_StdPeriph_Lib_V3.5.0\Project\STM32F10x_StdPeriph_Examples\SPI\SPI_FLASH
STM32F10x_StdPeriph_Lib_V3.5.0\Utilities\STM32_EVAL\Common

CHAPTER 24

OLED

i2c 控制 OLED

CHAPTER 25

SPI 控制 COG LCD

COG 和 OLED 在硬件上完全不用。

但是 COG 和 OLED 的显示方式很像。

驱动，除了初始化，没什么大区别。

当然，硬件控制比较复杂一点。

这里有一个问题需要思考，CS 由谁控制

通过对比代码，说明和 OLED 的区别。

CHAPTER 26

FSMC 与 TFT LCD

FSMC 是啥？TFT LCD 是啥？LCD 的关键功能是啥?

CHAPTER 27

触摸屏与 Tslib

用 IO 模拟 SPI 控制 XTP2046，触摸屏校准用 tslib

CHAPTER 28

SDIO 通信

如何移植官方 SD 卡例程

要改大栈到 0x2000

CHAPTER 29

USB

不做讲解，只讲如何移植官方例程。

CHAPTER 30

串口中断和环形缓冲

TODO

CHAPTER 31

矩阵按键

矩阵按键扫描

CHAPTER 32

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。
```