STM32 入门简易教程

第一章 STM32 处理器概述

优秀的处理器配合好的开发工具和工具链成就了单片机的辉煌，这是单片机开发者辛勤劳动的结果。也正因为此，ARM 的工具链工程师和 CPU 工程师强强联手，日日夜夜不停耕耘为 ARM7TDMI 设计出了精练、优化和到位的内部结构，终于成就了 ARM7TDMI 的风光无限的辉煌。新的 ARM Cortex-M3 处理器在破茧而出之后，就处处闪耀着 ARM 体系结构激动人心的新突破。它是基于最新最好的 32 为 ARMv7 架构，支持高度成功的 Thumb-2 指令集，并带来了很多前卫崭新的特性。在它优秀，强大的同时，编程模型也更清爽，因而无论你是新手还是骨灰级玩家都会对这样秀外慧中的小尤物爱不释手。

根据 ARM 的统计，2010 年全部 Cortex-M MCU 出货量为 1.44 亿片，2008 年~2011 年第一季度，STM32 累计出货量占 Cortex-M MCU 出货量的 45%。也就是说，两个 Cortex-M 微控制器中有一个就来自 ST。”很多市场分析机构也 ARM 的强劲增长表示认可。2007 年，在 32/64 bit MCU 及 MPU 架构中，ARM 所占市场份额为 13.6%，而 2010 年已经占了 23.5%，击败了 Power Architecture,成为市场占有率最多的架构。

Cortex-M3 内核是 ARM 公司整个 Cortex 内核系列中的微控制器系列（M）内核，还是其他两个系列分别是应用处理器系列（A）与实时控制处理系列（R），这三个系列又分别简称为 A、R、M 系列。当然，这三个系列的内核分别有各自不同的应用场合。

Cortex-M3 内核是为满足存储器和处理器的尺寸对产品成本影响很大的广泛市场和应用领域的低成本需求而专门开发设计的。主要是应用于低成本、小管脚数和低功耗的场合，并且具有极高的运算能力和极强的中断响应能力。Cortex-M3 处理器采用纯 Thumb2 指令的执行方式，这使得这个具有 32 位高性能的 ARM 内核能够实现 8 位和 16 位的代码存储密度。核心门数只有 33K，在包含了必要的外设之后的门数也只有 60K，使得封装更为小型，成本更加低廉。

Cortex-M3 采用了 ARMV7 哈佛架构，具有带分支预测的 3 级流水线，中断延迟最大只有 12 个时钟周期，在末尾连锁的时候只需要 6 个时钟周期。同时具有 1.25DMIPS/MHZ 的性能和 0.19MW/MHZ 的功耗。

意法半导体是 Cortex-M3 内核开发项目的一个主要合作方，现在是第一个推出基于这个内核产品的主要微控制器厂商。ST 也是世界上为数不多的几家能够提供从二极管到处理器的全系列电机控制器件的厂商。意法半导体 STM32 系列是业界最丰富的基于 ARM Cortex M 微控制器系列，共有 250 余款存储容量不同的产品，拥有丰富的外设接口、业界最好的开发生态系统、出色的功耗和整体功能。适用于需要实时控制或联网的任何消费电子产品或电子设备应用。

在性能方面，STM32 系列的处理速度比同级别的基于 ARM7TDMI 的产品快 30%，换句话说，如果处理性能相同，STM32 产品功耗比同级别产品低 75%。同样地，使用新内核的 Thumb 2 指令集，设计人员可以把代码容量降低 45%，几乎把应用软件所需内存容量降低了一半。此外，根据 Dhrystones 和其它性能测试结果，STM32 的性能比最好的 16 位架构至少高出一倍。

意法半导体是市场上第一家提供基于 Cortex-M3 内核的无传感器的磁场定向电机控制

解决方案的厂商。这套工具证明 STM32 的内核和专用电机控制外设都有充足的处理能力来优化驱动器的性能，最小化系统总体成本。STM 微控制器在 25 微秒内即可执行一整套无传感器三相无刷永磁同步电机（PMSM）矢量控制算法，大多数应用任务占用 CPU 资源比率小于 30%，为 CPU 执行其它应用任务（如需要）预留了充足的处理能力。永磁电机同步电机控制解决方案的代码大小少于 16 千字节。

意法半导体在推出 STM32 微控制器之初，也同时提供了一套完整细致的固件开发包，里面包含了在 STM32 开发过程中所涉及到的所有底层操作。尽管库不是那么尽如人意，但是通过在程序开发中引入这样的固件开发包，可以使开发人员从复杂冗余的底层寄存器操作中解放出来，将精力专注应用程序的开发上，这便是 ST 推出这样一个开发包的初衷。事实上也确实带给了我们很大的方便，因而很多人在用。

正是由于上面的特点，我们在此选用了 STM32 系列的 ARM 芯片。

注：因参考资料较多，不一一详述，如涉及作者利益，请联系修改。

湖北师范学院 电工电子实验教学示范中心

第二章 学习建议

一、较低的学习门槛

社会对基于 ARM 的嵌入式系统开发人员的高需求及给予的高回报，催生了很多的培训机构，这也说明嵌入式系统的门槛较高，其主要原因有以下几点。

1、 ARM 本身复杂的体系结构和编程模型，使得我们必须了解详细的汇编指令，熟悉 ARM 与 Thumb 状态的合理切换，才能理解 Bootloader 并对操作系统进行移植，而理解 Bootloader 本身就比较困难，因而对于初学者来说 Bootloader 的编写与操作系统的移植成了入门的第一道难以逾越的门槛；
2、 ARM 芯片，开发板及仿真器的高成本，这样就直接影响了嵌入式开发的普及，使得这方面人才增长缓慢；
3、 高校及社会上高水平嵌入式开发人员的短缺，现实问题使得我们的大学生和公司职工在入门的道路上困难重重，很多人也因此放弃；
4、 培训机构的高费用，虽然有高水平的老师指导，但是高费用就是一道关口，进去的人也只是在短短的几天时间里匆匆了解了一下开发过程，消除了一些畏惧心理而已，修行还是得依靠自己；
5、 好的开发环境需要资金的支持，也直接影响了入门的进度。

基于 Cortex-M3 内核的 ARM 处理器的出现，在优秀的 Keil 开发工具的支持下，可以自动生成启动代码，省去了复杂的 Bootloader 的编写。

Thumb-2 指令集的使用，使得开发人员不用再考虑 ARM 状态与 Thumb 状态的切换，节省了执行时间和指令空间，大大减轻了软件开发的管理工作。

处理器与内存尺寸的减少，大大降低了成本，使得芯片及开发板的价格得以在很大程度上降低。Cortex-M3 内核通过把中断控制器、MPU 及各种调试组件等基础设施的地址固定，很大程度上方便了程序的移植。

源代码是公开的库函数，使得我们可以摒弃晦涩难懂的汇编语言，在不需要了解底层寄存器的操作细节的情况下，用 C 语言就可以完成我们需要的功能。

所有这些特点使得我们学习 ARM 处理器的门槛得以降低。同时 建议大家尽量去用固件库。而不是避开固件库自己写代码。因为在实际的项目中，代码成百上千个，不可能都自己来写，调用固件库中的函数来完成，才是可行的方案。当然我们在深入的情况下，透彻理解寄存器的操作是必要的，也是值得的，高效编程也必须在这方面努力。

二、重要的参考资料

1、 Cortex-M3 权威指南 宋岩 译

权威资料的精简版，思路清晰，有条理，适合学 Cortex-M3 处理器的所有人。

2、 STM32 技术参考手册

a) STM32 微控制器产品的技术参考手册是讲述如何使用该产品的；
b) 包含各个功能模块的内部结构、所有可能的功能描述、各种工作模式的使用和寄存

器配置等详细信息。

3、 STM32F103RB 数据手册

a) 产品的基本配置（内置 FLASH 和 RAM 的容量、外设模块的种类和数量等）；

b) 管脚的数量和分配，电气特性，封装信息和订购代码等。

4、 STM32 开发板手册

与开发板配套的参考资料，有很多经验值得借鉴。

5、 stm32 固件库

a) 相关定义，文档约定和固件库规则；

b) 库的架构，安装指南及使用实例；

c) 每个外围模块的函数及解释。

6、 开发板原理图

必不可少的硬件电路参考。

7、 互联网

取之不尽的知识宝库。

三、必要的学习步骤

对于初学者来说，困难较多，下面是需要了解的一些信息，以供参考。

1、 了解 Cortex-M3 内核

看完 Cortex-M3 权威指南的前 36 页，你就会知道什么是 Cortex-M3。

2、 认识 STM32F103RBT6 处理器

a) 了解 STM32F10xxx 技术参考手册与 STM32F103xB 数据手册；

b) 了解 STM32F10xxx 技术参考手册第二章存储器和总线构架，需要了解外设时，再具体查看具体的功能模块；

c) 芯片选型初期看数据手册以评估该芯片是否能满足功能需求；

d) 基本选定芯片后就需要查看技术参考手册以确定各功能模块的功能是否符合要求；

e) 确定芯片型号，进入编程阶段后需要详细阅读技术参考手册以获知各项功能的具体实现方式和寄存器的配置使用；

f) 在设计硬件时还需要参考数据手册以获得电压、电流、管脚分配、驱动能力等信息。

3、 了解开发板原理图

参考开发板原理图，了解可支配的资源。

4、 了解库的结构与使用

参考 STM32 固件库中文版（UM0427）前三章。

5、 熟悉开发工具的使用

a) KEIL MDK 的使用(本文所有实例均在 Keil v4.10 上编译通过)

b) 程序下载软件 FlyMcu 的使用

c) 串口调试助手的使用

6、 了解时钟系统

时钟的配置涉及到所有的外设资源，所以我们应该对它有更深入的了解。

7、 GPIO

8、 定时器与中断

9、 串口

10、 ADC、SPI、IIC、USB。。。

第三章 编程基础

一、库函数结构与使用

1、 STM32F10XXX V3.4 标准外设库文件夹描述

STM32F10x_StdPeriph_Lib_V3.4.0	_htmresc	本文件夹包含了所有的 html 页面资源
Libraries	CMSIS	
STM32F10x_StdPeriph_Driver	inc	标准外设库驱动头文件
src	标准外设库驱动源文件	
Project	Examples	标准外设库驱动的完整例程
Template	MDK-ARM	KEIL RVMDK 的项目模板示例
RIDE	Raisonance RIDE 的项目模板示例	
EWARM	IAR EWARM 的项目模板示例	
Utilities	STM3210-EVAL	本文件夹包含了用于 STM3210B-EVAL 和 STM3210E-EVAL 评估板的专用驱动

标准外设库的第一部分是 CMSIS 和 STM32F10x_StdPeriph_Driver，CMSIS 是独立于供应商的 Cortex-M 处理器系列硬件抽象层，为芯片厂商和中间件供应商提供了简单的处理器软件接口，简化了软件复用工作，降低了 Cortex-M 上操作系统的移植难度，并减少了新入门的微控制器开发者的学习难度和新产品的上市时间。STM32F10x_StdPeriph_Driver 则包括了分别对应包括了所有外设对应驱动函数，这些驱动函数均使用 C 语言编写，并提供了统一的易于调用的函数接口，供开发者使用。Project 文件夹中则包括了 ST 官方的所有例程和基于不同编译器的项目模板，这些例程是学习和使用 STM32 的重要参考。Utilities 包含了相关评估板的示例程序和驱动函数，供使用官方评估板的开发者使用，很多驱动函数同样可以作为学习的重要参考。

2、 文件功能说明

文件名	功能描述	具体功能说明
core_cm3.h core_cm3.c	Cortex-M3 内核及其设备文件	访问 Cortex-M3 内核及其设备：NVIC，SysTick 等。访问 Cortex-M3 的 CPU 寄存器和内核外设的函数。
stm32f10x.h	微控制器专用头文件	这个文件包含了 STM32F10x 全系列所有外设寄存器的定义（寄存器的基地址和布局）、位定义、中断向量表、存储空间的地址映射等。

system_stm32f10x.h system_stm32f10x.c	微控制器专用系统文件	函数 SystemInit，用来初始化微控制器 函数 Sysem_ExtMemCtl，用来配置外部存储器控制器。它位于文件 startup_stm32f10x_xx .s /.c，在跳转到 main 前调用，SystemFrequncy，该值代表系统时钟频率。
startup_stm32f10x_Xd.s	编译器启动代码	微控制器专用的中断处理程序列表(与头文件一致) 弱定义(Weak)的中断处理程序默认函数(可以被用户代码覆盖) 该文件是与编译器相关的。
stm32f10x_conf.h	固件库配置文件	通过更改包含的外设头文件来选择固件库所使用的外设，在新建程序和进行功能变更之前应当首先修改对应的配置。
stm32f10x_it.h stm32f10x_it.c	外设中断函数文件	用户可以相应的加入自己的中断程序的代码，对于指向同一个中断向量的多个不同中断请求，用户可以通过判断外设的中断标志位来确定准确的中断源，执行相应的中断服务函数。
stm32f10x_ppp.h stm32f10x_ppp.c	外设驱动函数文件	包括了相关外设的初始化配置和部分功能应用函数，这部分是进行编程功能实现的重要组成部分。
Application.c	用户文件	用户程序文件，通过标准外设库提供的接口进行相应的外设配置和功能设计。

3、 基于 CMSIS 标准的软件架构

根据调查研究，软件开发已经被嵌入式行业公认为最主要的开发成本。对于 ARM 公司来说，一个 ARM 内核往往会授权给多个厂家，生产种类繁多的产品，如果没有一个通用的软件接口标准，那么当开发者在使用不同厂家的芯片时将极大的增加了软件开发成本，因此，ARM 与 Atmel、IAR、Keil、hami-nary Micro、Micrium、NXP、SEGGER 和 ST 等诸多芯片和软件厂商合作，将所有 Cortex 芯片厂商产品的软件接口标准化，制定了 CMSIS 标准。此举意在降低软件开发成本，尤其针对新设备项目开发，或者将已有软件移植到其他芯片厂商提供的基于 Cortex 处理器的微控制器的情况。有了该标准，芯片厂商就能够将他们的资源专注于产品外设特性的差异化，并且消除对微控制器进行编程时需要维持的不同的、互相不兼容的标准的需求，从而达到降低开发成本的目的。

如下图所示，基于 CMSIS 标准的软件架构主要分为以下 4 层：用户应用层、操作系统及中间件接口层、CMSIS 层、硬件寄存器层。其中 CMSIS 层起着承上启下的作用：一方面该层对硬件寄存器层进行统一实现，屏蔽了不同厂商对 Cortex-M 系列微处理器核内外设寄存器的不同定义；另一方面又向上层的操作系统及中间件接口层和应用层提供接口，简化了应用程序开发难度，使开发人员能够在完全透明的情况下进行应用程序开发。也正是如此，CMSIS 层的实现相对复杂。

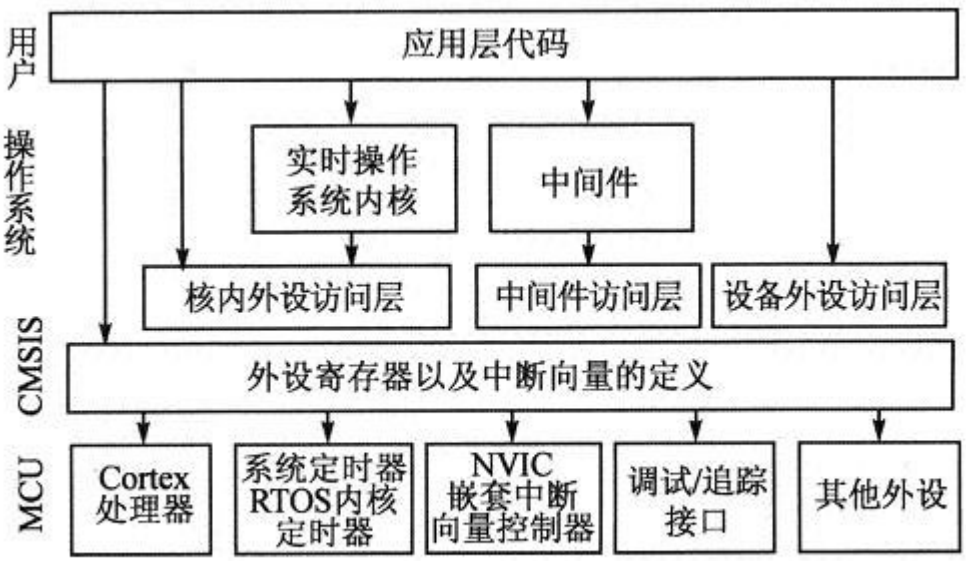


图 2　基于 CMSIS 标准的软件架构

CMSIS 层主要分为以下 3 个部分：

(1) 核内外设访问层（CPAL，Core Peripheral Access Layer）：该层由 ARM 负责实现。包括对寄存器名称、地址的定义，对核寄存器、NVIC、调试子系统的访问接口定义以及对特殊用途寄存器的访问接口（例如：CONTROL，xPSR）定义。由于对特殊寄存器的访问以内联方式定义，所以针对不同的编译器 ARM 统一用来屏蔽差异。该层定义的接口函数均是可重入的。

(2) 片上外设访问层（DPAL, Device Peripheral Access Layer）：该层由芯片厂商负责实现。该层的实现与 CPAL 类似，负责对硬件寄存器地址以及外设访问接口进行定义。该层可调用 CPAL 层提供的接口函数同时根据设备特性对异常向量表进行扩展，以处理相应外设的中断请求。

(3) 外设访问函数（AFP, Access Functions for Peripherals）：该层也由芯片厂商负责实现，主要是提供访问片上外设的访问函数，这一部分是可选的。

对一个 Cortex-M 微控制系统而言，CMSIS 通过以上三个部分实现了：

- 定义了访问外设寄存器和异常向量的通用方法；
- 定义了核内外设的寄存器名称和核异常向量的名称；
- 为 RTOS 核定义了与设备独立的接口，包括 Debug 通道。

这样芯片厂商就能专注于对其产品的外设特性进行差异化，并且消除他们对微控制器进行编程时需要维持的不同的、互相不兼容的标准需求，以达到低成本开发的目的。

4、STM32F10XXX 标准外设库的使用

在实际开发过程中，根据应用程序的需要，可以采取 2 种方法使用标准外设库(StdPeriph_Lib)：

(1) 使用外设驱动：这时应用程序开发基于外设驱动的 API(应用编程接口)。用户只需要配置文件”stm32f10x_conf.h”，并使用相应的文件”stm32f10x_ppp.h/.c”即可。

(2) 不使用外设驱动：这时应用程序开发基于外设的寄存器结构和位定义文件。

这两种方法的优缺点在“使用标准外设库开发的优势”小节中已经有了具体的介绍，这里仍要说明的是，使用使用标准外设库进行开发可以极大的减小软件开发的工作量，也是目前嵌入式系统开发的一个趋势。

标准外设库(StdPeriph_Lib)支持 STM32F10xxx 系列全部成员：大容量，中容量和小容量产品。从表 5- 6 中也可以看出，启动文件已经对不同的系列进行了划分，实际开发中根据使用的 STM32 产品具体型号，用户可以通过文件"stm32f10x.h"中的预处理 define 或者通过开发环境中的全局设置来配置标准外设库(StdPeriph_Lib)，一个 define 对应一个产品系列。

- STM32F10x_LD：STM32 小容量产品
- STM32F10x_MD：STM32 中容量产品
- STM32F10x_HD：STM32 大容量产品

在库文件中这些 define 的具体作用范围是：

- 文件"stm3210f.h"中的中断 IRQ 定义
- 启动文件中的向量表，小容量，中容量，大容量产品各有一个启动文件
- 外设存储器映像和寄存器物理地址
- 产品设置：外部晶振(HSE)的值等
- 系统配置函数

因此通过宏定义这种方式，可以使标准外设库适用于不同系列的产品，同时也方便与不同产品之间的软件移植，极大的方便了软件的开发。

标准外设库中包含了众多的变量定义和功能函数，如果不能了解他们的命名规范和使用规律将会给编程带来很大的麻烦，本节将主要叙述标准外设库中的相关规范，通过这些规范的学习可以更加灵活的使用固件库，同时也将极大增强程序的规范性和易读性，同时标准外设库中的这种规范也值得我们在进行其他相关的开发时使用和借鉴。

5、命名规则

标准外设库遵从以下命名规则 *PPP*表示任一外设缩写，例如：*ADC*。源程序文件和头文件命名都以"stm32f10x_"作为开头，例如：*stm32f10x_conf.h*。常量仅被应用于一个文件的，定义于该文件中；被应用于多个文件的，在对应头文件中定义。所有常量都由英文字母大写书写。寄存器作为常量处理。他们的命名都由英文字母大写书写。在大多数情况下，他们采用与缩写规范一致。外设函数的命名以该外设的缩写加下划线为开头。每个单词的第一个字母都由英文字母大写书写，例如：*SPI_SendData*。在函数名中，只允许存在一个下划线，用以分隔外设缩写和函数名的其它部分。对于函数命名，总的来说有以下规则：

- 名为 PPP_Init 的函数，其功能是根据 PPP_InitTypeDef 中指定的参数，初始化外设 PPP，例如 TIM_Init.
- 名为 PPP_DeInit 的函数，其功能为复位外设 PPP 的所有寄存器至缺省值，例如 TIM_DeInit.
- 名为 PPP_Init 的函数，其功能为通过设置 PPP_InitTypeDef 结构中的各种参数来定义外设的功能，例如：USART_Init .
- 名为 PPP_Cmd 的函数，其功能为使能或者失能外设 PPP，例如： SPI_Cmd.
- 名为 PPP_ITConfig 的函数，其功能为使能或者失能来自外设 PPP 某中断源，例如：RCC_ITConfig.
- 名为 PPP_DMAConfig 的函数，其功能为使能或者失能外设 PPP 的 DMA 接口，例如：TIM1_DMAConfig.
- 用以配置外设功能的函数，总是以字符串"Config"结尾，例如 GPIO_PinRemapConfig.
- 名为 PPP_GetFlagStatus 的函数，其功能为检查外设 PPP 某标志位被设置与否，例如：I2C_GetFlagStatus.
- 名为 PPP_ClearFlag 的函数，其功能为清除外设 PPP 标志位，例如：I2C_ClearFlag.

- 名为 PPP_GetITStatus 的函数，其功能为判断来自外设 PPP 的中断发生与否，例如：I2C_GetITStatus.
- 名为 PPP_ClearITPendingBit 的函数，其功能为清除外设 PPP 中断待处理标志位，例如：I2C_ClearITPendingBit.

这样的命名方式非常便于程序的编写和阅读，以标准外设库中的示例函数为例，下面摘录了 STM32F10x_StdPeriph_Examples\ADC\3ADCs_DMA\mian.c 中的一段程序。

```
DMA_InitType Def DMA_InitStructure;
/* DMA1 channel1 configuration ----------------------------------------*/
DMA_DeInit(DMA1_Channel1);
DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADC1ConvertedValue;
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
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_HalfWord;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
DMA_InitStructure.DMA_Priority = DMA_Priority_High;
DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
DMA_Init(DMA1_Channel1, &DMA_InitStructure);
/* Enable DMA1 channel1 */
DMA_Cmd(DMA1_Channel1, ENABLE);
```

这段程序完成了 DMA1 通道的配置，首先定义了 *DMA_InitType DMA_InitStructure*，接着配置 *DMA_InitType* 的各种参数，各参数的命名方式也均使用约定的命名方式，从命名就能够很容易的看出各参数所指代的具体功能。功能参数配置完成后，使用 *DMA_Init(DMA1_Channel1, &DMA_InitStructure);* 完成相应外设的初始化，最后使用 *DMA_Cmd(DMA1_Channel1, ENABLE)* 使能相应外设。从这个例子就能够很容易的看出标准外设库这种规范化的命名规则给编写和阅读程序带来的好处。

6、变量定义

在早期的版本中有 24 个变量定义，在 stm32f10x_type.h 中，可以找到对应的定义。

```
/* Includes ------------------------------------------------------------------*/
/* Exported types ------------------------------------------------------------*/
typedef signed long s32;
typedef signed short s16;
typedef signed char s8;
typedef signed long const sc32; /* Read Only */
typedef signed short const sc16; /* Read Only */
typedef signed char const sc8; /* Read Only */
typedef volatile signed long vs32;
typedef volatile signed short vs16;
typedef volatile signed char vs8;
typedef volatile signed long const vsc32; /* Read Only */
```

```
typedef volatile signed short const vsc16; /* Read Only */
typedef volatile signed char const vsc8; /* Read Only */
typedef unsigned long u32;
typedef unsigned short u16;
typedef unsigned char u8;
typedef unsigned long const uc32; /* Read Only */
typedef unsigned short const uc16; /* Read Only */
typedef unsigned char const uc8; /* Read Only */
typedef volatile unsigned long vu32;
typedef volatile unsigned short vu16;
typedef volatile unsigned char vu8;
typedef volatile unsigned long const vuc32; /* Read Only */
typedef volatile unsigned short const vuc16; /* Read Only */
typedef volatile unsigned char const vuc8; /* Read Only */
```

3.0 以后的版本中使用了 CMSIS 数据类型，变量的定义有所不同，但是出于兼容旧版本的目的，以上的数据类型仍然兼容。在 stm32f10x.h 中可以找到具体的定义，定义如下。

```
/*!< STM32F10x Standard Peripheral Library old types (maintained for legacy purpose) */
typedef int32_t s32;
typedef int16_t s16;
typedef int8_t s8;
typedef const int32_t sc32; /*!< Read Only */
typedef const int16_t sc16; /*!< Read Only */
typedef const int8_t sc8; /*!< Read Only */
typedef __IO int32_t vs32;
typedef __IO int16_t vs16;
typedef __IO int8_t vs8;
typedef __I int32_t vsc32; /*!< Read Only */
typedef __I int16_t vsc16; /*!< Read Only */
typedef __I int8_t vsc8; /*!< Read Only */
typedef uint32_t u32;
typedef uint16_t u16;
typedef uint8_t u8;
typedef const uint32_t uc32; /*!< Read Only */
typedef const uint16_t uc16; /*!< Read Only */
typedef const uint8_t uc8; /*!< Read Only */
typedef __IO uint32_t vu32;
typedef __IO uint16_t vu16;
typedef __IO uint8_t vu8;
typedef __I uint32_t vuc32; /*!< Read Only */
typedef __I uint16_t vuc16; /*!< Read Only */
typedef __I uint8_t vuc8; /*!< Read Only */
```

7、CMSIS IO 类型限定词

IO 类限定词	#define	描述
_I	volatile const	只读访问

_O	volatile	只写访问
_IO	volatile	读和写访问

8、固件库与 CMSIS 数据类型对比

固件库类型	CMSIS 类型	描述
s32	int32_t	易挥发只读有符号 32 位数据
s16	int16_t	易挥发只读有符号 16 位数据
s8	int8_t	易挥发只读有符号 8 位数据
sc32	const int32_t	只读有符号 32 位数据
sc16	const int16_t	只读有符号 16 位数据
sc8	const int8_t	只读有符号 8 位数据
vs32	_IO int32_t	易挥发读写访问有符号 32 位数据
vs16	_IO int16_t	易挥发读写访问有符号 16 位数据
vs8	_IO int8_t	易挥发读写访问有符号 8 位数据
vsc32	_I int32_t	易挥发只读有符号 32 位数据
vsc16	_I int16_t	易挥发只读有符号 16 位数据
vsc8	_I int8_t	易挥发只读有符号 8 位数据
u32	uint32_t	无符号 32 位数据
u16	uint16_t	无符号 16 位数据
u8	uint8_t	无符号 8 位数据
uc32	const uint32_t	只读无符号 32 位数据
uc16	const uint16_t	只读无符号 16 位数据
uc8	const uint8_t	只读无符号 8 位数据
vu32	_IO uint32_t	易挥发读写访问无符号 32 位数据
vu16	_IO uint16_t	易挥发读写访问无符号 16 位数据
vu8	_IO uint8_t	易挥发读写访问无符号 8 位数据
vuc32	_I uint32_t	易挥发只读无符号 32 位数据
vuc16	_I uint16_t	易挥发只读无符号 16 位数据
vuc8	_I uint8_t	易挥发只读无符号 8 位数据

stm32f10x.h 文件中还包含了常用的布尔形变量定义，如：

```
typedef enum {RESET = 0, SET = !RESET} FlagStatus, ITStatus;
typedef enum {DISABLE = 0, ENABLE = !DISABLE} FunctionalState;
#define IS_FUNCTIONAL_STATE(STATE) (((STATE) == DISABLE) || ((STATE) == ENABLE))
typedef enum {ERROR = 0, SUCCESS = !ERROR} ErrorStatus;
```

不同版本的标准外设库的变量定义略有不同，如 3.4 版本中就没有之前版本的 TRUE 和

FALSE 的定义，用户也可以根据自己的需求按照上面的格式定义自己的布尔形变量。在使用标准外设库进行开发遇到相关的定义问题时应首先找到对应的头文件定义。

9、使用步骤

前面几个小节已经详细介绍了标准外设库的组成结构以及部分主要文件的功能描述，那么如果在开发中使用标准外设库需要哪些描述呢？下面就进行简要的介绍，这儿介绍的使用方法是与开发环境无关的，在不同的开发环境中可能在操作方式上略有不同，但是总体的流程都是一样的，下一小节将介绍在 MDK ARM 开发环境下使用标准外设库的详细过程。

首先新建一个项目并设置工具链对应的启动文件，可以使用标准外设库中提供的模板，也可以自己根据自己的需求新建。标准外设库中已经提供了不同工具链对应的文件，位于 STM32F10x_StdPeriph_Lib_V3.4.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup 目录下。

其次按照使用产品的具体型号选择具体的启动文件，加入工程。文件主要按照使用产品的容量进行区分，根据产品容量进行选择即可。每个文件的具体含义可以在“stm32f10x.h”文件中找到对应的说明。

“stm32f10x.h”是整个标准外设库的入口文件，这个文件包含了 STM32F10x 全系列所有外设寄存器的定义（寄存器的基地址和布局）、位定义、中断向量表、存储空间的地址映射等。为了是这个文件适用于不同系列的产品，程序中是通过宏定义来实现不同产品的匹配的，上面这段程序的注释中已经详细给出了每个启动文件所对应的产品系列，与之对应，也要相应的修改这个入口文件，需要根据所使用的产品系列正确的注释/去掉相应的注释 define。在这段程序的下方同样有这样的一个注释程序 */*#define USE_STDPERIPH_DRIVER*/* 用于选择是否使用标准外设库，如果保留这个注释，则用户开发程序可以基于直接访问“stm32f10x.h”中定义的外设寄存器，所有的操作均基于寄存器完成，目前不使用固件库的单片机开发，如 51、AVR、MSP430 等其实都是采用此种方式，通过在对应型号的头文件中进行外设寄存器等方面的定义，从而在程序中对相应的寄存器操作完成相应的功能设计。
如果去掉*/*#define USE_STDPERIPH_DRIVER*/*的注释，则是使用标准外设库进行开发，用户需要使用在文件“stm32f10x_conf.h”中，选择要用的外设，外设同样是通过注释/去掉注释的方式来选择。几乎所有的应用都需要使用复位与时钟以及通用 I/O，因此这两项是必须的，而多数程序同样要使用 NVIC 中断 IRQ 设置和 SysTick 时钟源设置，那么 “misc.h”这一项也是必须的。

上面已经针对具体的产品信号和程序功能进行了针对性的配置，接下来需要配置系统所使用的时钟，系统时钟在“system_stm32f10x.c”同样通过注释的方式来配置，程序如下：

```
#if defined (STM32F10X_LD_VL) || (defined STM32F10X_MD_VL) || (defined STM32F10X_HD_VL)
/* #define SYSCLK_FREQ_HSE HSE_VALUE */
#define SYSCLK_FREQ_24MHz 24000000
#else
/* #define SYSCLK_FREQ_HSE HSE_VALUE */
/* #define SYSCLK_FREQ_24MHz 24000000 */
/* #define SYSCLK_FREQ_36MHz 36000000 */
/* #define SYSCLK_FREQ_48MHz 48000000 */
/* #define SYSCLK_FREQ_56MHz 56000000 */
#define SYSCLK_FREQ_72MHz 72000000
```

#endif
如果这儿没有明确的定义那么 HSI 时钟将会作为系统时钟。

至此，已经配置了系统的主要外部参数，这些参数主要是通过更改相关的宏定义来实现的，有些开发环境，例如 Keil 支持在软件设置中加入全局宏定义，因此像芯片系列定义，是否使用固件库定义等也可以通过软件添加来实现。

完成了主要参数配置以后就可以进行程序的开发了，标准外设库开发就可以使用标准外设库中提供的方便的 API 函数进行相应的功能设计了。前面已经介绍了基于标准外设库开发的优势，配置完成后，程序中仍然可以直接更改相应寄存器的配置，通过对寄存器的操作可以提高程序的效率，因此可以使用标准外设库和寄存器操作两种相结合的方式。

二、系统时钟

1、时钟源分类

在 STM32 芯片中中，有五个时钟源，分别为 **HSI**、**HSE**、**LSI**、**LSE**、**PLL**。

- **HSI** 是高速内部时钟，RC 振荡器，频率为 8MHz。
- **HSE** 是高速外部时钟，可接石英/陶瓷谐振器，或者接外部时钟源，频率范围为 4MHz~16MHz。
- **LSI** 是低速内部时钟，RC 振荡器，频率为 40kHz。
- **LSE** 是低速外部时钟，接频率为 32.768kHz 的石英晶体。
- **PLL** 为锁相环倍频输出，其时钟输入源可选择为 HSI/2、HSE 或者 HSE/2。倍频可选择为 2~16 倍，但是其输出频率最大不得超过 72MHz。

2、时钟系统框图

片上总线标准种类繁多，而由 ARM 公司推出的 AMBA 片上总线受到了广大 IP 开发商和 SoC 系统集成者的青睐，已成为一种流行的工业标准片上结构。AMBA 规范主要包括了 AHB(Advanced High performance Bus)系统总线和 APB(Advanced Peripheral Bus)外围总线。二者分别适用于高速与相对低速设备的连接。

一般性的时钟设置需要先考虑系统时钟的来源，是内部 RC 还是外部晶振还是外部的振荡器，是否需要 PLL。然后考虑内部总线和外部总线，最后考虑外设的时钟信号。遵从先倍频作为 CPU 时钟，然后在由内向外分频，下级迁就上级的原则有点儿类似 PCB 制图的规范化要求，在这里也一样。

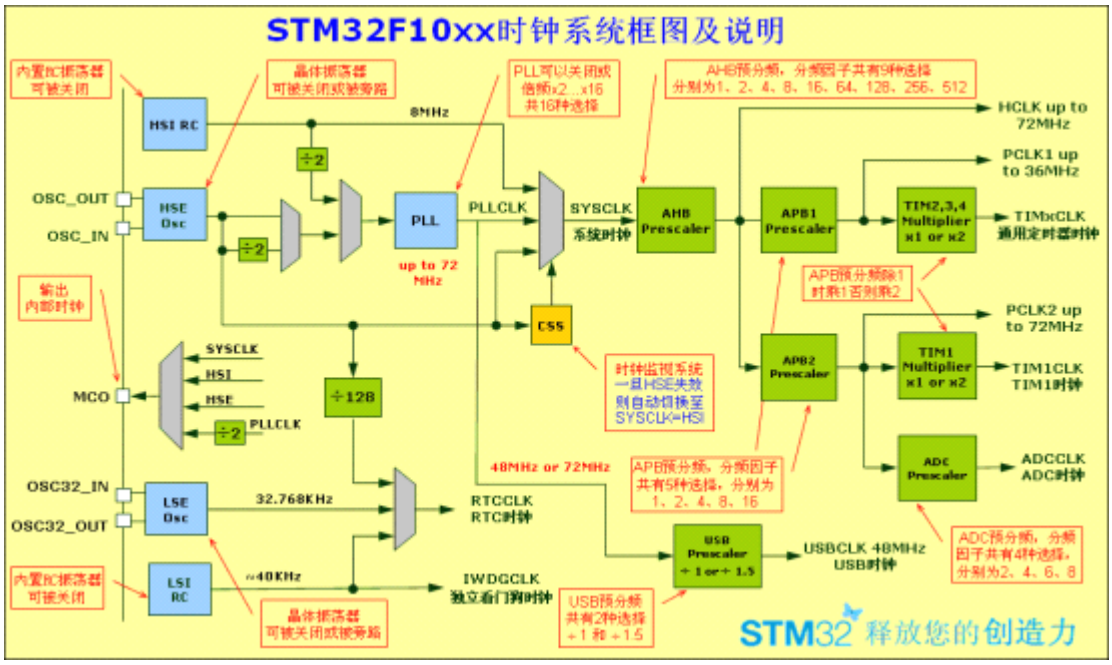


图 1 时钟系统时钟结构框图

STM32 中有一个全速功能的 USB 模块，其串行接口引擎需要一个频率为 48MHz 的时钟源。该时钟源只能从 PLL 端获取，可以选择为 1.5 分频或者 1 分频，也就是，当需使用到 USB 模块时，PLL 必须使能，并且时钟配置为 48MHz 或 72MHz。

另外 STM32 还可以选择一个时钟信号输出到 MCO 脚(PA.8)上，可以选择为 PLL 输出的 2 分频、HSI、HSE 或者系统时钟。

系统时钟 SYSCLK，它是提供 STM32 中绝大部分部件工作的时钟源。系统时钟可以选择为 PLL 输出、HSI、HSE。系统时钟最大频率为 72MHz，它通过 AHB 分频器分频后送给各个模块使用，AHB 分频器可以选择 1、2、4、8、16、64、128、256、512 分频，其分频器输出的时钟送给 5 大模块使用：

（1）送给 AHB 总线、内核、内存和 DMA 使用的 HCLK 时钟；

（2）通过 8 分频后送给 Cortex 的系统定时器时钟；

（3）直接送给 Cortex 的空闲运行时钟 FCLK；

（4）送给 APB1 分频器。APB1 分频器可以选择 1、2、4、8、16 分频，其输出一路供 APB1 外设使用（PCLK1，最大频率 36MHz），另一路送给定时器(Timer)2、3、4 倍频器使用。该倍频器可以选择 1 或者 2 倍频，时钟输出供定时器 2、3、4 使用。

（5）送给 APB2 分频器。APB2 分频器可以选择 1、2、4、8、16 分频，其输出一路供 APB2 外设使用（PCLK2，最大频率 72MHz），另外一路送给定时器(Timer)1 倍频使用。该倍频器可以选择 1 或 2 倍频，时钟输出供定时器 1 使用。另外 APB2 分频器还有一路输出供 ADC 分频器使用，分频后送给 ADC 模块使用。ADC 分频器可选择为 2、4、6、8 分频。需要注意的是定时器的倍频器，当 APB 的分频为 1 时，它的倍频值为 1，否则它的倍频值就为 2。

连接在 APB1(低速外设)上的设备有：电源接口、备份接口、CAN、USB、I2C1、I2C2、UART2、UART3、SPI2、窗口看门狗、Timer2、Timer3、Timer4。注意 USB 模块虽然需要一个单独的 48MHz 的时钟信号，但是它应该不是供 USB 模块工作的时钟，而只是提供给串行接口引擎(SIE)使用的时钟。USB 模块的工作时钟应该是由 APB1 提供的。

连接在 APB2（高速外设）上的设备有：UART1、SPI1、Timer1、ADC1、ADC2、GPIOx(PA~PE)、第二功能 IO 口。

3、HSE 时钟设置

由于现在所用的开发板已经外接了一个 8MHz 的晶振，因此将采用 HSE 时钟，在 MDK 编译平台中，程序的时钟设置参数流程如下：

（1）将 RCC 寄存器重新设置为默认值：RCC_DeInit;

（2）打开外部高速时钟晶振 HSE： RCC_HSEConfig(RCC_HSE_ON);

（3）等待外部高速时钟晶振工作： HSEStartUpStatus = RCC_WaitForHSEStartUp();

（4）设置 AHB 时钟(HCLK)： RCC_HCLKConfig;

（5）设置高速 AHB 时钟(APB2)： RCC_PCLK2Config;

（6）设置低速 AHB 时钟(APB1)： RCC_PCLK1Config;

（7）设置 PLL： RCC_PLLConfig;

（8）打开 PLL： RCC_PLLCmd(ENABLE);

（9）等待 PLL 工作： while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET);

（10）设置系统时钟： RCC_SYSCLKConfig;

（11）判断 PLL 是否是系统时钟： while(RCC_GetSYSCLKSource() != 0x08);

（12）打开要使用的外设时钟： RCC_APB2PerphClockCmd()….

某些函数的详细的使用方法，可以参考 ST 公司出版的《STM32F10xxx_Library_Manual》

4、RCC 寄存器结构

RCC_TypeDeff, 在文件“stm32f10x_map.h”中定义如下：

```
typedef struct
{
vu32 CR; //时钟控制寄存器(RCC_CR)
vu32 CFGR; //时钟配置寄存器(RCC_CFGR）
vu32 CIR;    //时钟中断寄存器 (RCC_CIR)
vu32 APB2RSTR; //外设复位寄存器 (RCC_APB2RSTR)
vu32 APB1RSTR; //外设复位寄存器 (RCC_APB1RSTR)
vu32 AHBENR;    // AHB 外设时钟使能寄存器 (RCC_AHBENR)
vu32 APB2ENR;   //外设时钟使能寄存器(RCC_APB2ENR)
vu32 APB1ENR;   //外设时钟使能寄存器(RCC_APB1ENR)
vu32 BDCR;       //备份域控制寄存器 (RCC_BDCR)
vu32 CSR;        //控制/状态寄存器 (RCC_CSR)
} RCC_TypeDef;
```

5、RCC 配置函数

下面是 STM32 软件固件库的程序中对 RCC 的配置函数(使用外部 8MHz 晶振)

```
/*******************************************************************************
* Function Name  : RCC_Configuration
* Description    :  RCC 配置(使用外部 8MHz 晶振)
* Input          : 无
* Output         : 无
* Return         : 无
*******************************************************************************/
void RCC_Configuration(void)
{
  /*将外设 RCC 寄存器重设为缺省值 */
  RCC_DeInit();
  /*设置外部高速晶振（HSE）*/
  RCC_HSEConfig(RCC_HSE_ON);    // HSE 晶振打开(ON)
  /*等待 HSE 起振*/
  HSEStartUpStatus = RCC_WaitForHSEStartUp();
  if(HSEStartUpStatus == SUCCESS)     //SUCCESS: HSE 晶振稳定且就绪
  {
      /*设置 AHB 时钟（HCLK）*/
      RCC_HCLKConfig(RCC_SYSCLK_Div1);  // AHB 时钟 = 系统时钟
      /* 设置高速 AHB 时钟（PCLK2）*/
      RCC_PCLK2Config(RCC_HCLK_Div1);   // APB2 时钟 = HCLK
```

```
        /*设置低速 AHB 时钟（PCLK1）*/
        RCC_PCLK1Config(RCC_HCLK_Div2);      // APB1 时钟 = HCLK / 2
        /*设置 FLASH 存储器延时时钟周期数*/
        FLASH_SetLatency(FLASH_Latency_2);    //FLASH_Latency_2  2 延时周期
        /*选择 FLASH 预取指缓存的模式*/
        FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);// 预取指缓存使能
        /*设置 PLL 时钟源及倍频系数*/
        RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);//8*9 = 72MHz
        /*使能 PLL */
        RCC_PLLCmd(ENABLE);
        /*检查指定的 RCC 标志位(PLL 准备好标志)设置与否*/
        while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET){ };
        /*设置系统时钟（SYSCLK） */
        RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);// 选择 PLL 作为系统时钟
        /* PLL 返回用作系统时钟的时钟源*/
        while(RCC_GetSYSCLKSource() != 0x08) { };//0x08：PLL 作为系统时钟
    }

  /*使能或者失能 APB2 外设时钟*/
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB |
                           RCC_APB2Periph_GPIOC , ENABLE);
      //RCC_APB2Periph_GPIOA      GPIOA 时钟
      //RCC_APB2Periph_GPIOB      GPIOB 时钟
      //RCC_APB2Periph_GPIOC      GPIOC 时钟
      //RCC_APB2Periph_GPIOD      GPIOD 时钟
}
```

三、SysTick 定时器

1、SysTick 介绍

NVIC 中，捆绑着一个 SysTick 定时器，它是一个 24 位的倒数计数定时器，当计到 0 时，将从 RELOAD 寄存器中自动重装载定时初值并继续计数，同时内部的 COUNTFLAG 标志会置位，触发中断 (如果中断使能情况下)。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。Cortex-M3 允许为 SysTick 提供 2 个时钟源以供选择，系统时钟 HCLK 与 HCLK/8。

2、SysTick 寄存器

下面介绍一下 STM32 中的 SysTick，它属于 NVIC 控制部分，一共有 4 个寄存器：

STK_CSR，　　0xE000E010：　　控制寄存器
STK_LOAD，　　0xE000E014：　　重载寄存器
STK_VAL，　　0xE000E018：　　当前值寄存器
STK_CALRB，　　0xE000E01C：　　校准值寄存器

STK_CSR 控制寄存器，有 4 个 bit 具有意义：
第 0 位：ENABLE，SysTick 使能位（0：关闭 SysTick 功能，1：开启 SysTick 功能）；
第 1 位：TICKINT，SysTick 中断使能位（0：关闭 SysTick 中断，1：开启 SysTick 中断）；
第 2 位：CLKSOURCE，SysTick 时钟选择（0：使用 HCLK/8 作为时钟源，1：使用 HCLK）；
第 3 为：COUNTFLAG，SysTick 计数比较标志，如果在上次读取本寄存器后，SysTick 已经数到 0 了，则该位为 1，如果读取该位，该位自动清零。

STK_LOAD 重载寄存器：
Systick 是一个递减的定时器，当定时器递减至 0 时，重载寄存器中的值就会被重装载，继续开始递减。STK_LOAD 重载寄存器是个 24 位的寄存器最大计数 0xFFFFFF。

STK_VAL 当前值寄存器：
也是个 24 位的寄存器，读取时返回当前倒计数的值，写它则使之清零，同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志。

STK_CALRB 校准值寄存器：
其中包含着一个 TENMS 位段，具体信息不详。暂时用不到。

3、SysTick 库函数

在 MDK 开发环境中，我们不必要非得去操作每一个寄存器，可以通过调用 ST 函数库中的函数来进行相关的操作，其步骤如下：

（1）调用 SysTick_CounterCmd()　　失能 SysTick 计数器
（2）调用 SysTick_ITConfig()　　失能 SysTick 中断
（3）调用 SysTick_CLKSourceConfig()　　设置 SysTick 时钟源
（4）调用 SysTick_SetReload()　　设置 SysTick 重装载值

（5）调用 NVIC_SystemHandlerPriorityConfig()　　设置 SysTick 定时器中断优先级
（6）调用 SysTick_ITConfig()　　使能 SysTick 中断
（7）在 stm32f10x_it.c 中 SysTickHandler()下写中断服务函数。
（8）在需要的时候调用 SysTick_CounterCmd()　开启 SysTick 计数器

第四章 外设使用

一、GPIO

1、GPIO 介绍

（1）8 种可选模式

- 带上拉输入
- 带下拉输入
- 模拟输入
- 开漏输出
- 推挽输出
- 复用功能的推挽输出
- 复用功能的开漏输出

模式 7 和模式 8 需根据具体的复用功能决定。

（2）专门的寄存器(GPIOx_BSRR 和 GPIOx_BRR)实现对 GPIO 口的原子操作，即回避了设置或清除 I/O 端口时的“读-修改-写”操作，使得设置或清除 I/O 端口的操作不会被中断处理打断而造成误动作。

（3）每个 GPIO 口都可以作为外部中断的输入，便于系统灵活设计。

（4）I/O 口的输出模式下，有 3 种输出速度可选(2MHz、10MHz 和 50MHz)，这有利于噪声控制。

（5）所有 I/O 口兼容 CMOS 和 TTL，多数 I/O 口兼容 5V 电平。

（6）大电流驱动能力：GPIO 口在高低电平分别为 0.4V 和 VDD-0.4V 时，可以提供或吸收 8mA 电流；如果把输入输出电平分别放宽到 1.3V 和 VDD-1.3V 时，可以提供或吸收 20mA 电流。

（7）具有独立的唤醒 I/O 口。

（8）很多 I/O 口的复用功能可以重新映射。

（9）GPIO 口的配置具有上锁功能，当配置好 GPIO 口后，可以通过程序锁住配置组合，直到下次芯片复位才能解锁。此功能非常有利于在程序跑飞的情况下保护系统中其他的设备，不会因为某些 I/O 口的配置被改变而损坏——如一个输入口变成输出口并输出电流。

2、GPIO 在外设中的使用说明

1、配置步骤

（1）配置输入的时钟

（2）初始化后即被激活(开启)；

（3）如果使用该外设的输入输出管脚，则需要配置相应的 GPIO 端口（否则该外设对应的输入输出管脚可以做普通 GPIO 管脚使用）；

（4）再对外设进行详细配置。

2、对应到外设的输入输出功能有下述三种情况：

（1）外设对应的管脚为输出：需要根据外围电路的配置选择对应的管脚为复用功能的推挽

输出或复用功能的开漏输出。

（2）外设对应的管脚为输入：则根据外围电路的配置可以选择浮空输入、带上拉输入或带下拉输入。

（3）ADC 对应的管脚：配置管脚为模拟输入。

如果把端口配置成复用输出功能，则引脚和输出寄存器断开，并和片上外设的输出信号连接。将管脚配置成复用输出功能后，如果外设没有被激活，那么它的输出将不确定。

3、GPIO 初始化

```
void GPIO_Configuration(void)
{
    GPIO_InitTypeDef    GPIO_InitStructure;// 定义 GPIO 初始化结构体
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;//选择管脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;//推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//设置 IO 口翻转速度
    GPIO_Init(GPIOA, &GPIO_InitStructure);//初始化 GPIO A 口
}
```

二、中断

1、简介

NVIC 共支持 1 至 240 个外部中断。Cortex‐M3 在内核水平上搭载了一个异常响应系统，这个系统规定为 0 到 256 个中断异常，支持为数众多的系统异常和外部中断。其中，编号为 1－15 的对应系统异常，大于等于 16 的则全是外部中断。除了个别异常的优先级被定死外，其它异常的优先级都是可编程的。异常的优先级在这里我们先不说，对于所有的 Cortex‐M3 内核处理器（包括 STM32）256 个异常中的 0~15 号异常都是一样的，内核规定好的，这就是 Cortex‐M3 的特点。0~15 号异常如下：

表 7.1　系统异常清单

编号	类型	优先级	简介
0	N/A	N/A	没有异常在运行
1	复位	-3（最高）	复位
2	NMI	-2	不可屏蔽中断（来自外部 NMI 输入脚）
3	硬(hard)fault	-1	所有被除能的 fault，都将“上访”(escalation)成硬 fault。只要 FAULTMASK 没有置位，硬 fault 服务例程就被强制执行。Fault 被除能的原因包括被禁用，或者 FAULTMASK 被置位。
4	MemManage fault	可编程	存储器管理 fault，MPU 访问犯规以及访问非法位置均可引发。企图在“非执行区”取指也会引发此 fault
5	总线 fault	可编程	从总线系统收到了错误响应，原因可以是预取流产（Abort）或数据流产，或者企图访问协处理器
6	用法(usage) Fault	可编程	由于程序错误导致的异常。通常是使用了一条无效指令，或者是非法的状态转换，例如尝试切换到 ARM 状态
7-10	保留	N/A	N/A
11	SVCall	可编程	执行系统服务调用指令（SVC）引发的异常
12	调试监视器	可编程	调试监视器（断点，数据观察点，或者是外部调试请求
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的“可悬挂请求”（pendable request）
15	SysTick	可编程	系统滴答定时器（也就是周期性溢出的时基定时器）

EEworld.com.cn

从 16 开始的外部中断类型，是制造商做成芯片后，支持的中断源数，自然各种芯片的中断源数目常常不到 240 个，并且优先级的位数也由芯片厂商最终决定。如下表：

表 7.2　外部中断清单

编号	类型	优先级	简介
16	IRQ #0	可编程	外中断#0
17	IRQ #1	可编程	外中断#1
…	…	…	…
255	IRQ #239	可编程	外中断#239

EEworld.com.cn

这 240 个中断的使能与除能分别使用各自的寄存器来控制——这与传统的，使用单一比特的两个状态来表达使能与除能是不同的。CM3 中可以有 240 对使能位／除能位，每个中

断拥有一对。这 240 个对子分布在 8 对 32 位寄存器中（最后一对没有用完）。欲使能一个中断，你需要写 1 到对应 SETENA 的位中；欲除能一个中断，你需要写 1 到对应的 CLRENA 位中；如果往它们中写 0，不会有任何效果。通过这种方式，使能 / 除能中断时只需把“当事位”写成 1，其它的位可以全部为零。再也不用像以前那样，害怕有些位被写入 0 而破坏其对应的中断设置（写 0 没有效果），从而实现每个中断都可以自顾地设置，而互不侵犯——只需单一的写指令，不再需要读 - 改 - 写。

如上所述，SETENA 位和 CLRENA 位可以有 240 对，对应的 32 位寄存器可以有 8 对，因此使用数字后缀来区分这些寄存器，如 SETENA0, SETENA1…SETENA7，如表 8.1 所示。但是在特定的芯片中，只有该芯片实现的中断，其对应的位才有意义。因此，如果你使用的芯片支持 32 个中断，则只有 SETENA0/CLRENA0 才需要使用。SETENA/CLRENA 可以按字/半字/字节的方式来访问。又因为前 16 个异常已经分配给系统异常，故而中断 0 的异常号是 16。

表 8.1　SETENA/CLRENA 寄存器族　（此表参考官方技术参考手册作了些改编——译者注）

SETENAs: xE000_E100 – 0xE000_E11C　；　CLRENAs:0xE000E180 - 0xE000_E19C

名称	类型	地址	复位值	描述
SETENA0	R/W	0xE000_E100	0	中断 0-31 的使能寄存器，共 32 个使能位 位[n]，中断#n 使能（异常号 16+n）
SETENA1	R/W	0xE000_E104	0	中断 32-63 的使能寄存器，共 32 个使能位
…	…	…	…	…
SETENA7	R/W	0xE000_E11C	0	中断 224-239 的使能寄存器，共 16 个使能位
CLRENA0	R/W	0xE000_E180	0	中断 0-31 的除能寄存器，共 32 个除能位 位[n]，中断#n 除能（异常号 16+n）
CLRENA1	R/W	0xE000_E184	0	中断 32-63 的除能寄存器，共 32 个除能位
…	…	…	…	…
CLRENA7	R/W	0xE000_E19C	0	中断 224-239 的除能寄存器，共 16 个除能位

EEworld.com.cn

2、优先级判断

STM32 目前支持的中断共为 84 个（16 个内核+68 个外部（互联型 8 个保留，STM32F10x 为 60 个），注：不是 68 个外部中断），16 级可编程中断优先级的设置（仅使用中断优先级设置 8bit 中的高 4 位）和 16 个抢占优先级（因为抢占优先级最多可以有四位数）。

STM32(Cortex-M3)中有两个优先级的概念——抢占式优先级和响应优先级，有人把响应优先级称作'亚优先级'或'副优先级'，每个中断源都需要被指定这两种优先级。具有高抢占式优先级的中断可以在具有低抢占式优先级的中断处理过程中被响应，即中断嵌套，或者说

高抢占式优先级的中断可以嵌套低抢占式优先级的中断。

当两个中断源的抢占式优先级相同时，这两个中断将没有嵌套关系，当一个中断到来后，如果正在处理另一个中断，这个后到来的中断就要等到前一个中断处理完之后才能被处理。如果这两个中断同时到达，则中断控制器根据他们的响应优先级高低来决定先处理哪一个；如果他们的抢占式优先级和响应优先级都相等，则根据他们在中断表中的排位顺序决定先处理哪一个。

3、优先级分组

既然每个中断源都需要被指定这两种优先级，就需要有相应的寄存器位记录每个中断的优先级；在 Cortex-M3 中定义了 8 个比特位用于设置中断源的优先级，这 8 个比特位在 NVIC 应用中断与复位控制寄丛器（AIRCR）的中断优先级分组域中，可以有 8 种分配方式，如下：

所有 8 位用于指定响应优先级
最高 1 位用于指定抢占式优先级，最低 7 位用于指定响应优先级
最高 2 位用于指定抢占式优先级，最低 6 位用于指定响应优先级
最高 3 位用于指定抢占式优先级，最低 5 位用于指定响应优先级
最高 4 位用于指定抢占式优先级，最低 4 位用于指定响应优先级
最高 5 位用于指定抢占式优先级，最低 3 位用于指定响应优先级
最高 6 位用于指定抢占式优先级，最低 2 位用于指定响应优先级
最高 7 位用于指定抢占式优先级，最低 1 位用于指定响应优先级
这就是优先级分组的概念。

Cortex-M3 允许具有较少中断源时使用较少的寄存器位指定中断源的优先级，因此 STM32 把指定中断优先级的寄存器位减少到 4 位（AIRCR 高四位），这 4 个寄存器位的分组方式如下：

第 0 组：所有 4 位用于指定响应优先级
第 1 组：最高 1 位用于指定抢占式优先级，最低 3 位用于指定响应优先级
第 2 组：最高 2 位用于指定抢占式优先级，最低 2 位用于指定响应优先级
第 3 组：最高 3 位用于指定抢占式优先级，最低 1 位用于指定响应优先级
第 4 组：所有 4 位用于指定抢占式优先级

可以通过调用 STM32 的固件库中的函数 NVIC_PriorityGroupConfig()选择使用哪种优先级分组方式，这个函数的参数有下列 5 种：

NVIC_PriorityGroup_0 => 选择第 0 组
NVIC_PriorityGroup_1 => 选择第 1 组
NVIC_PriorityGroup_2 => 选择第 2 组
NVIC_PriorityGroup_3 => 选择第 3 组
NVIC_PriorityGroup_4 => 选择第 4 组

中断优先级分组是为了给抢占式优先级和响应优先级在中断优先级寄丛器的高四位分配各个优先级数字所占的位数，在一个程序中只能设定一次。

4、中断源的优先级

接下来就是指定中断源的优先级，中断源优先级是在中断优先级寄存器中设置的，只能设置及高四位，必须根据中断优先级分组中设置好的位数来在该寄存器中设置相应的数值。

假如你选择中断优先级分组的第 3 组：最高 3 位用于指定抢占式优先级，最低 1 位用于指定响应优先级，那么抢占式优先级就有 000-111 共八种数据选择，也就是有八个中断嵌套，而响应优先级中有 0 和 1 两种，总共有 8*2=16 种优先级。

中断源优先级具体的设置了该中断源的优先级别

在一个程序中可以设定多个（最多 16 个）优先级，每个中断源只能设定的一个。

每写一个关于中断优先级的程序必须包含下列两个函数：

（1）*void NVIC_PriorityGroupConfig(u32 NVIC_PriorityGroup)*中断分组设置

（2）*void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct)*中断优先级设置

三、串口

1、串口的基本概念

在 STM32 的参考手册中，串口被描述成通用同步异步收发器(USART)，它提供了一种灵活的方法与使用工业标准 NRZ 异步串行数据格式的外部设备之间进行全双工数据交换。USART 利用分数波特率发生器提供宽范围的波特率选择。它支持同步单向通信和半双工单线通信，也支持 LIN（局部互联网），智能卡协议和 IrDA（红外数据组织）SIR ENDEC 规范，以及调制解调器(CTS/RTS)操作。它还允许多处理器通信。还可以使用 DMA 方式，实现高速数据通信。

USART 通过 3 个引脚与其他设备连接在一起，任何 USART 双向通信至少需要 2 个引脚：接受数据输入(RX)和发送数据输出(TX)。

RX: 接受数据串行输入。通过过采样技术来区别数据和噪音，从而恢复数据。

TX: 发送数据输出。当发送器被禁止时，输出引脚恢复到它的 I/O 端口配置。当发送器被激活，并且不发送数据时，TX 引脚处处于高电平。在单线和智能卡模式里，此 I/O 口被同时用于数据的发送和接收。

2、串口工作方式

一般有两种方式，查询和中断。

（1）查询：串口程序不断地循环查询，看看当前有没有数据要它传送。如果有，就帮助传送（可以从 PC 到 STM32 板子，也可以从 STM32 板子到 PC）。

（2）中断：平时串口只要打开中断即可。如果发现有一个中断来，则意味着要它帮助传输数据——它就马上进行数据的传送。同样，可以从 PC 到 STM3 板子，也可以从 STM32 板子到 PC。

3、串口配置

USART1 在系统存储区启动模式下，将通过该口通过 PC 对板上的 CPU 进行 ISP，该口也可作为普通串口功能使用。

在 RCC 配置的时候，要使能 USART 时钟，并且要打开管脚功能复用时钟。

在 GPIO 配置中，将发送端的管脚配置为复用推挽输出，将接收端的管脚配置为浮空输入。

在 USART 的配置中，通过 USART_InitTypeDef 结构体对 USART 进行初始化操作，按照自己所需的功能配置好就可以了。注意，在超级终端的设置中，需要和这个里面的配置相对应。由于我是采用中断接收数据的方式，所以记得在 USART 的配置中药打开串口的中断，同时最后还要打开串口。

在 NVIC 的配置中，主要是 USART1_IRQChannel 的配置，和以前的笔记中讲述的中断配置类似，不会配置的可以参考以前的笔记。

全部配置好之后就可以开始发送/接收数据了。发送数据用 USART_SendData()函数，接收数据用 USART_ReceiveData()函数。具体的函数功能可以参考固件库的参考文件。根据 USART 的配置，在发送和接收时，都是采用的 8bits 一帧来进行的，因此，在发送的时候，先开辟一个缓存区，将需要发送的数据送入缓存区，然后再将缓存区中的数据发送出去，在接收的时候，同样也是先接收到缓存区中，然后再进行相应的操作。

注意在对数据进行发送和接收的时候，要检查 USART 的状态，只有等到数据发送或接收完毕之后才能进行下一帧数据的发送或接收。采用 USART_GetFlagStatus()函数。

同时还要注意的是，在发送数据的最开始，需要清除一下 USART 的标志位，否则，第 1 位数据会丢失。因为在硬件复位之后，USART 的状态位 TC 是置位的。当包含有数据的一帧发送完成之后，由硬件将该位置位。只要当 USART 的状态位 TC 是置位的时候，就可以进行数据的发送。然后 TC 位的置零则是通过软件序列来清除的，具体的步骤是“先读 USART_SR，然后写入 USART_DR”，只有这样才能够清除标志位 TC，但是在发送第一帧数据的时候，并没有进行读 USART_SR 的操作，而是直接进行写操作，因此 TC 标志位并没有清空，那么，当发送第一帧数据，然后用 USART_GetFlagStatus()检测状态时返回的是已经发送完毕（因为 TC 位是置 1 的），所以程序会马上发送下一帧数据，那么这样，第一帧数据就被第二帧数据给覆盖了，所以看不到第一帧数据的发送。

```
void USART_Config(void)
{
 USART_InitTypeDef USART_InitStructure; //定义串口初始化结构体
 USART_InitStructure.USART_BaudRate = 9600;//波特率 9600
 USART_InitStructure.USART_WordLength = USART_WordLength_8b;//8 位数据
 USART_InitStructure.USART_StopBits = USART_StopBits_1;//1 个停止位
 USART_InitStructure.USART_Parity = USART_Parity_No ; //无校验位
 USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
 //禁用 RTSCTS 硬件流控制
 USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;//使能发送接收
 USART_InitStructure.USART_Clock = USART_Clock_Disable; //串口时钟禁止
 USART_InitStructure.USART_CPOL = USART_CPOL_Low; //时钟下降沿有效
 USART_InitStructure.USART_CPHA = USART_CPHA_2Edge;//数据在第二个时钟沿捕捉
 USART_InitStructure.USART_LastBit = USART_LastBit_Disable;
 //最后数据位的时钟脉冲不输出到 SCLK 引脚
 USART_Init(USART2, &USART_InitStructure);//初始化串口 2
 USART_Cmd(USART2, ENABLE);//串口 2 使能
}
```

四、定时器

1、定时器介绍

STM32F103RBT6 中一共有 8 个定时器（由于功能差别较大，此处不包括两个看门狗定时器和一个系统嘀嗒定时器（SysTick）。），其中 TIM1 和 TIM8 两个高级控制定时器，TIM2-TIM5 四个通用定时器和 TIM6 和 TIM7 两个基本定时器。它们完全独立，不共享任何资源，但是可以一起同步操作。所有 TIMx 定时器在内部相连，用于定时器同步或链接。当一个定时器处于主模式时，它可以对另一个处于从模式的定时器的计数器进行复位、启动、停止或提供时钟等操作。

定时器	计数器分辨率	计数器类型	预分频系数	产生 DMA 请求	捕获/比较通道	互补输出
TIM1 TIM8	16 位	向上，向下，向上/向下	1-65536 之间的任意数	可以	4	有
TIM2 TIM3 TIM4 TIM5	16 位	向上，向下，向上/向下	1-65536 之间的任意数	可以	4	没有
TIM6 TIM7	16 位	向上	1-65536 之间的任意数	可以	0	没有

其中 TIM1 和 TIM8 是能够产生 3 对 PWM 互补输出的高级定时器，常用于三相电机的驱动，时钟由 APB2 的输出产生。TIM2-TIM5 是通用定时器，TIM6 和 TIM7 是基本定时器，其时钟由 APB1 输出产生。由于 STM32 的 TIMER 功能非常复杂，这里只做下基本的介绍，详细功能需要参考资料手册。

计数器时钟可由下列时钟源提供：

（1）内部时钟(CK_INT)

（2）外部时钟模式 1：外部输入脚(TIx)

（3）外部时钟模式 2：外部触发输入(ETR)

（4）内部触发输入(ITRx)：使用一个定时器作为另一个定时器的预分频器，如可以配置一个定时器 Timer1 作为另一个定时器 Timer2 的预分频器。

这些时钟，具体选择哪个可以通过 TIMx_SMCR 寄存器的相关位来设置。这里的 CK_INT 时钟不是直接来自于 APB1，而是来自于输入为 APB1 的一个倍频器。这个倍频器的作用是：当 APB1 的预分频系数为 1 时，这个倍频器不起作用，定时器的时钟频率等于 APB1 的频率；当 APB1 的预分频系数为其他数值时（即预分频系数为 2、4、8 或 16），这个倍频器起作用，定时器的时钟频率等于 APB1 的频率的 2 倍。设置这个倍频器可以保证在其他外设使用较低时钟频率时，TIM2-TIM5 仍然可以得到较高的时钟频率。

2、高级定时器 TIM1 和 TIM8 主要特性：

- 16 位向上、向下、向上/下自动装载计数器

● 16 位可编程(可以实时修改)预分频器，计数器时钟频率的分频系数为 1～65535 之间的任意数值
● 多达 4 个独立通道： ― 输入捕获 ― 输出比较 ―PWM 生成(边缘或中间对齐模式)― 单脉冲模式输出
● 死区时间可编程的互补输出
● 使用外部信号控制定时器和定时器互联的同步电路
● 允许在指定数目的计数器周期之后更新定时器寄存器的重复计数器
● 刹车输入信号可以将定时器输出信号置于复位状态或者一个已知状态
● 如下事件发生时产生中断/DMA： ― 更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)― 触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)― 输入捕获 ― 输出比较 ― 刹车信号输入
● 支持针对定位的增量(正交)编码器和霍尔传感器电路
● 触发输入作为外部时钟或者按周期的电流管理

3、通用定时器 TIM2、TIM3、TIM4 和 TIM5 主要特性：

● 16 位向上、向下、向上/向下自动装载计数器
● 16 位可编程(可以实时修改)预分频器，计数器时钟频率的分频系数为 1～65536 之间的任意数值
● 4 个独立通道： ― 输入捕获 ― 输出比较 ―PWM 生成(边缘或中间对齐模式)― 单脉冲模式输出
● 使用外部信号控制定时器和定时器互连的同步电路
● 如下事件发生时产生中断/DMA： ― 更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)― 触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)― 输入捕获 ― 输出比较
● 支持针对定位的增量(正交)编码器和霍尔传感器电路
● 触发输入作为外部时钟或者按周期的电流管理

4、基本定时器 TIM6 和 TIM7 的主要特性：

● 16 位自动重装载累加计数器
● 16 位可编程(可实时修改)预分频器，用于对输入的时钟按系数为 1～65536 之间的任意数值分频
● 触发 DAC 的同步电路　注:此项是 TIM6/7 独有功能
● 在更新事件(计数器溢出)时产生中断/DMA 请求

5、计数器模式

计数器模式分为向上计数，向下计数和中央对齐模式(向上/向下计数)。在中央对齐模式中，计数器从 0 开始计数到自动加载的值(TIMx_ARR 寄存器)−1，产生一个计数器溢出事件，

然后向下计数到 1 并且产生一个计数器下溢事件，然后再从 0 开始计数。

6、输入捕获模式

在输入捕获模式下，当检测到 ICx 信号上相应的边沿后，计数器的当前值被锁存到捕获/比较寄存器(TIMx_CCRx)中。当捕获事件发生时，相应的 CCxIF 标志(TIMx_SR 寄存器)被置'1'，如果使能了中断或者 DMA 操作，则将产生中断或者 DMA 操作。在捕获模式下，捕获发生在影子寄存器上，然后再复制到预装载寄存器中。

7、PWM 输入捕获模式

PWM 输入捕获模式是输入捕获模式的特例。

（1）每个定时器有四个输入捕获通道 IC1、IC2、IC3、IC4。且 IC1 IC2 一组，IC3 IC4 一组。并且可是设置管脚和寄存器的对应关系。

（2）同一个 TIx 输入映射了两个 ICx 信号。

（3）这两个 ICx 信号分别在相反的极性边沿有效。

（4）两个边沿信号中的一个被选为触发信号，并且从模式控制器被设置成复位模式。

（5）当触发信号来临时，被设置成触发输入信号的捕获寄存器，捕获"一个 PWM 周期（即连续的两个上升沿或下降沿）"，它等于包含 TIM 时钟周期的个数（即捕获寄存器中捕获的为 TIM 的计数个数 n）。

（6）同样另一个捕获通道捕获触发信号和下一个相反极性的边沿信号的计数个数 m，即（即高电平的周期或低电平的周期）

（7）由此可以计算出 PWM 的时钟周期和占空比了

frequency=f（TIM 时钟频率）/n。

duty cycle=（高电平计数个数/n），

若 m 为高电平计数个数，则 duty cycle=m/n

若 m 为高电平计数个数，则 duty cycle=（n-m）/n

注：因为计数器为 16 位，所以一个周期最多技术 65535 个，所以测得的 最小频率= TIM 时钟频率/65535。

8、输出比较——翻转模式

（1）输出比较：打开一个 TIMx 计数器，再打开 TIMx 的一路或几路输出比较器（共 4 路），都配置好以后，计数器开始计数，当计数器里的值和比较寄存器里的值相等时，产生输出比较中断，在中断中将计数器中的值读出，与翻转周期相加再写道比较寄存器中，使得和下一个事件有相同的翻转周期。

（2）举例说明：例如 TIM 时钟频率设置为 12MHZ，输出比较寄存器中的自装载值为 600（高电平或低电平计数值），则输出的 PWM 频率为 frequency = 12MHZ/（600*2）=10KHZ。

9、PWM 模式（PWMOut）

（1）PWM 模式由 TIM_ARR 寄存器确定频率，由 TIM_CCR 寄存器确定占空比的信号。
（2）举例说明：例如 TIM 时钟频率设置为 36MHZ，输出比较寄存器中的自装载值为 3599 即 ARR Register = 3599，则输出的 PWM 频率为 frequency = 36MHZ/（ARR+1）=10KHZ。设置捕获寄存器的值 CCR_Value（即高电平计数值）= 1800.，则占空比
duty cycle = 1800//3600=50%。

五、ADC

1、基础知识

（1）18 个通道，可测 16 个外部和 2 个内部信号源，可设置成单次、连续、扫描、间断模式执行
（2）12 位精度
（3）扫描模式，通道 0 到通道 n 的自动转化
（4）自校准
（5）按通道配置采样时间
（6）间断模式
（7）双 ADC 模式
（8）供电要求 2.4~3.6V
（9）输入范围 0~3.6V
（10）通道管脚如图 12.1
（11）外部触发源如图 12.2

	ADC1	ADC2	ADC3
通道0	PA0	PA0	PA0
通道1	PA1	PA1	PA1
通道2	PA2	PA2	PA2
通道3	PA3	PA3	PA3
通道4	PA4	PA4	PF6
通道5	PA5	PA5	PF7
通道6	PA6	PA6	PF8
通道7	PA7	PA7	PF9
通道8	PB0	PB0	PF10
通道9	PB1	PB1	
通道10	PC0	PC0	PC0
通道11	PC1	PC1	PC1
通道12	PC2	PC2	PC2
通道13	PC3	PC3	PC3
通道14	PC4	PC4	
通道15	PC5	PC5	
通道16	温度传感器		
通道17	内部参照电压		

图 12.1 ADC 通道对应管脚

表64　ADC1和ADC2用于规则通道的外部触发

触发源	类型	EXTSEL[2:0]
TIM1_CC1事件	来自片上定时器的内部信号	000
TIM1_CC2事件		001
TIM1_CC3事件		010
TIM2_CC2事件		011
TIM3_TRGO事件		100
TIM4_CC4事件		101
EXTI线11/TIM8_TRGO事件(1)(2)	外部引脚/来自片上定时器的内部信号	110
SWSTART	软件控制位	111

1．TIM8_TRGO事件只存在于大容量产品

2．对于规则通道，选中EXTI线路11或TIM8_TRGO作为外部触发事件，可以分别通过设置ADC1和ADC2的ADC1_ETRGREG_REMAP位和ADC2_ETRGREG_REMAP位实现。

表65　ADC1和ADC2用于注入通道的外部触发

触发源	连接类型	JEXTSEL[2:0]
TIM1_TRGO事件	来自片上定时器的内部信号	000
TIM1_CC4事件		001
TIM2_TRGO事件		010
TIM2_CC1事件		011
TIM3_CC4事件		100
TIM4_TRGO事件		101
EXTI线15/TIM8_CC4事件(1)(2)	外部引脚/来自片上定时器的内部信号	110
JSWSTART	软件控制位	111

图 12.2 ADC 外部触发源

2、功能描述

（1）通道选择

STM32 的每个 ADC 模块通过内部的模拟多路开关，可以切换到不同的输入通道并进行转换。在任意多个通道上以任意顺序进行的一系列转换构成成组转换。例如，可以如下顺序完成转换：通道 3、通道 8、通道 2、通道 2、通道 0、通道 2、通道 2、通道 15。

有 2 种划分转换组的方式：规则通道组和注入通道组。通常规则通道组中可以安排最多 16 个通道，而注入通道组可以安排最多 4 个通道。

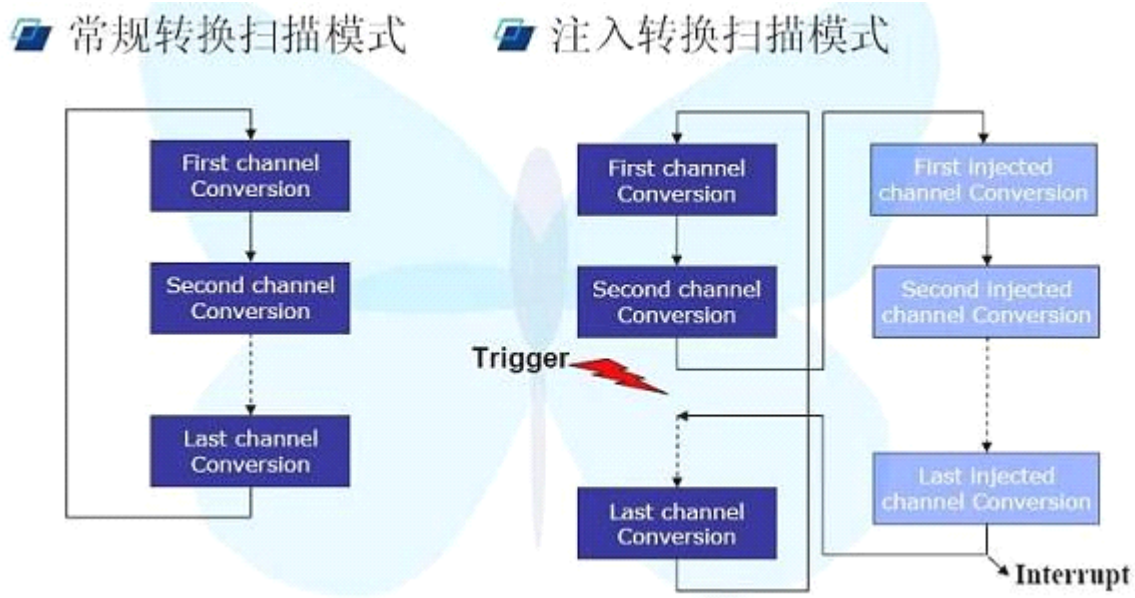
在执行规则通道组扫描转换时，如有例外处理则可启用注入通道组的转换。规则转换和注入转换均有外部触发选项，规则通道转换期间有 DMA 请求产生，而注入转换则无 DMA 请求，需要用查询或中断的方式保存转换的数据。

一个不太恰当的比喻是：规则通道组的转换好比是程序的正常执行，而注入通道组的转换则好比是程序正常执行之外的一个中断处理程序。规则组由多达 16 个转换组成，规则通道和它们的转换顺序在 ADC_SQRx 寄存器中选择。规则组中转换的总数写入 ADC_SQR1 寄存器的 L[3:0]位中。

注入组由多达 4 个转换组成。注入通道和它们的转换顺序在 ADC_JSQR 寄存器中选择。注入组里的转换总数目写入 ADC_JSQR 寄存器的 L[1:0]位中。

如果规则转换已经在运行，为了在注入转换后确保同步，所有的 ADC(主和从)的规则转

换被停止，并在注入转换结束时同步恢复，见下图所示。



（2）单次转换模式

单次转换模式里，ADC 只执行一次转换。

（3）连续转换模式

在连续转换模式中，当前面 ADC 转换一结束马上就启动另一次转换。

（4）扫描模式：

此模式用来扫描一组模拟通道。

（5）注入模式管理：

- 触发注入，详见参考手册
- 自动注入，如果设置了 JAUTO 位，在规则组通道之后，注入组通道被自动转换。这可以用来转换在 ADC_SQRx 和 ADC_JSQR 寄存器中设置的多至 20 个转换序列，在此模式里，必须禁止注入通道的外部触发。

（6）间断模式：

（1）规则组

此模式通过设置 ADC_CR1 寄存器上的 DISCEN 位激活。它可以用来执行一个短序列的 n 次转换(n<=8)，此转换是 ADC_SQRx 寄存器所选择的转换序列的一部分。N 由 ADC_CR1 寄存器的 DISCNUM[2:0]位给出。一个外部触发信号可以启动 ADC_SQRx 寄存器中描述的下一轮 n 次转换，直到此序列所有的转换完成为止。总的序列长度由 ADC_SQR1 寄存器的 L[3:0]定义。

举例：

n=3，被转换的通道 =0, 1, 2, 3, 6, 7, 9, 10

第一次触发：转换的序列为 0，1，2

第二次触发：转换的序列为 3，6，7

第三次触发：转换的序列为 9，10，并产生 EOC 事件

第四次触发：转换的序列 0，1，2 注意：当一规则组以间断模式转换时，转换序列结束后不自动从头开始。当所有子组被转换完成，下一次触发启动第一个子组的转换。在上面的例子中，第四次触发重新转换第一子组的通道 0，1 和 2。

（2）注入组

此模式通过设置 ADC_CR1 寄存器的 JDISCEN 位激活。在一个外部触发事件后，给模式按序转换 ADC_JSQR 寄存器中选择的序列。

一个外部触发信号可以启动 ADC_JSQR 寄存器选择的下一个通道序列的转换，直到序列中所有的转换完成为止。总的序列长度 ADC_JSQR 寄存器的 JL[1:0]位定义。

例子：

n=1，被转换的通道 =1，2，3

第一次触发：通道 1 被转换

第二次触发：通道 2 被转换

第三次触发：通道 3 被转换，并且产生 EOC 和 JEOC 事件

第四次触发：通道 1 被转换

注意：

- 当完成所有注入通道转换，下个触发启动第 1 个注入通道的转换。在上述例子中，第四个触发重新转换第 1 个注入通道。
- 不能同时使用自动注入和间断模式。
- 必须避免同时为规则和注入组设置间断模式，间断模式只能作用于一组转换。

（7）双 ADC 模式

（8）数据对齐

ADC_CR2 寄存器中的 ALIGN 位选择转换后数据储存的对齐方式。数据可以左对齐或右对齐。

注入组通道转换的数据值已经减去了在 ADC_JOFRx 寄存器中定义的偏移量，因此结果可以是一个负值，SEXT 位是扩展的符号值。

对于规则组通道，不需减去偏移值，因此只有 12 个位有效。

数据右对齐：

注入组

SEXT SEXT SEXT SEXT D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0

规则组

0 0 0 0 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0

数据左对齐：

注入组

SEXT D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0 0 0 0

规则组

D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0 0 0 0 0

3、 程序设计与软件配置

```
void ADC_Config(void)
{
    ADC_InitTypeDef ADC_InitStructure;//定义 ADC 初始化结构体变量
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;//ADC1 和 ADC2 工作在独立模式
    ADC_InitStructure.ADC_ScanConvMode = ENABLE; //使能扫描
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;//ADC 转换工作在连续模式
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;//由软件控制转换
```

```
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;//转换数据右对齐
    ADC_InitStructure.ADC_NbrOfChannel = 1;//转换通道数目为 1
    ADC_Init(ADC1, &ADC_InitStructure); //初始化 ADC
    ADC_RegularChannelConfig(ADC1, ADC_Channel_14, 1, ADC_SampleTime_28Cycles5);
    //ADC1 选择通道 14,规则组采样顺序（1~16）,采样时间 239.5 个周期
    ADC_DMACmd(ADC1, ENABLE);//使能 ADC1 模块 DMA
    ADC_Cmd(ADC1, ENABLE);//使能 ADC1
    ADC_ResetCalibration(ADC1); //重置 ADC1 校准寄存器
    while(ADC_GetResetCalibrationStatus(ADC1));//等待 ADC1 校准重置完成
    ADC_StartCalibration(ADC1);//开始 ADC1 校准
    while(ADC_GetCalibrationStatus(ADC1));//等待 ADC1 校准完成
    ADC_SoftwareStartConvCmd(ADC1, ENABLE); //使能 ADC1 软件开始转换
}
```

六、DMA 控制器使用

1、DMA 基础

DMA（直接存储器访问），提供外设与存储器、存储器与存储器之间的高速数据传输，不需要 CPU 的参与，共有七个通道。

（1）DMA 传输有三个操作完成

- 从外设数据寄存器或者从 DMA_CMARx 寄存器指定地址的存储器单元执行加载操作。
- 存数据到外设数据寄存器或者存数据到 DMA_CMARx 寄存器指定地址的存储器单元。
- 执行一次 DMA_CNDTRx 寄存器的递减操作。该寄存器包含未完成的操作数目。

（2）仲裁器（优先级）

软件：优先级由 DMA_CCRX 配置，4 个优先级别：最高、高、中等、低。

硬件：若软件优先级相同，则通道号小的优先级更高。

（3）属性

- 可编程的数据大小
- 指针增量：传输地址加 1，2 或者 4。
- 通道配置过程：DMA_CPARX 设置外设地址——DMA_CMARX 设置数据存储地址——设置数据量——设置优先级别——设置传输防线、循环模式等——启动

（4）循环模式

可以处理循环缓存和连续数据流，当床书数目变为 0 是，将会自动的被恢复成配置通道是设置的初值，DMA 操作将会继续进行。

（5）DMA 中断

可以在 DMA 传输过半、传输完成和传输错误时产生中断。STM32 中 DMA 的不同中断（传输完成、半传输、传输完成）通过“线或”方式连接至 NVIC，需要在中断例程中进行判断。

进行 DMA 配置前，不要忘了在 RCC 设置中使能 DMA 时钟。STM32 的 DMA 控制器挂在 AHB 总线上。

2、程序分析

```
DMA_DeInit(DMA1_Channel1);                    //将 DMA 通道 1 所有寄存器重设为缺省值
DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;  //DMA 外设基地址（外设存储器映像值+地址偏移）
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADCConvertedValue;  //该参数用以定义 DMA 内存基地址
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;   //外设作为数据传输的来源
DMA_InitStructure.DMA_BufferSize = 1;//指定 DMA 通道的 DMA 缓存的大小，也是就传几个数据
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设地址寄存器不变
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;//内存地址寄存器不变
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;// 外 设 数
```

据宽度为 16 位
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;//存储器的数据宽度为 16 位
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;//工作在循环缓存模式
DMA_InitStructure.DMA_Priority = DMA_Priority_High;//DMA 通道 1 拥有高优先级
DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;//没有设置成内存到内存
DMA_Init(DMA1_Channel1, &DMA_InitStructure);//根据 DMA_InitStruct 中指定的参数初始化 DMA 通道 1 寄存器
/* Enable DMA1 channel1 */
DMA_Cmd(DMA1_Channel1, ENABLE);//DMA 使能

七、SPI 接口

1、SPI 管脚

● MISO：主设备输入/从设备输出引脚。该引脚在从模式下发送数据，在主模式下接收数据。
● MOSI：主设备输出/从设备输入引脚。该引脚在主模式下发送数据，在从模式下接收数据。
● SCK：串口时钟，作为主设备的输出，从设备的输入
●NSS：从设备选择。这是一个可选的引脚，用来选择主/从设备。它的功能是用来作为“片选引脚”，让主设备可以单独地与特定从设备通讯，避免数据线上的冲突。从设备的 NSS 引脚可以由主设备的一个标准 I/O 引脚来驱动。一旦被使能(SSOE 位)，NSS 引脚也可以作为输出引脚，并在 SPI 处于主模式时拉低；此时，所有的 SPI 设备，如果它们的 NSS 引脚连接到主设备的 NSS 引脚，则会检测到低电平，如果它们被设置为 NSS 硬件模式，就会自动进入从设备状态。当配置为主设备、NSS 配置为输入引脚(MSTR=1，SSOE=0)时，如果 NSS 被拉低，则这个 SPI 设备进入主模式失败状态：即 MSTR 位被自动清除，此设备进入从模式。

2、时钟信号的相位和极性

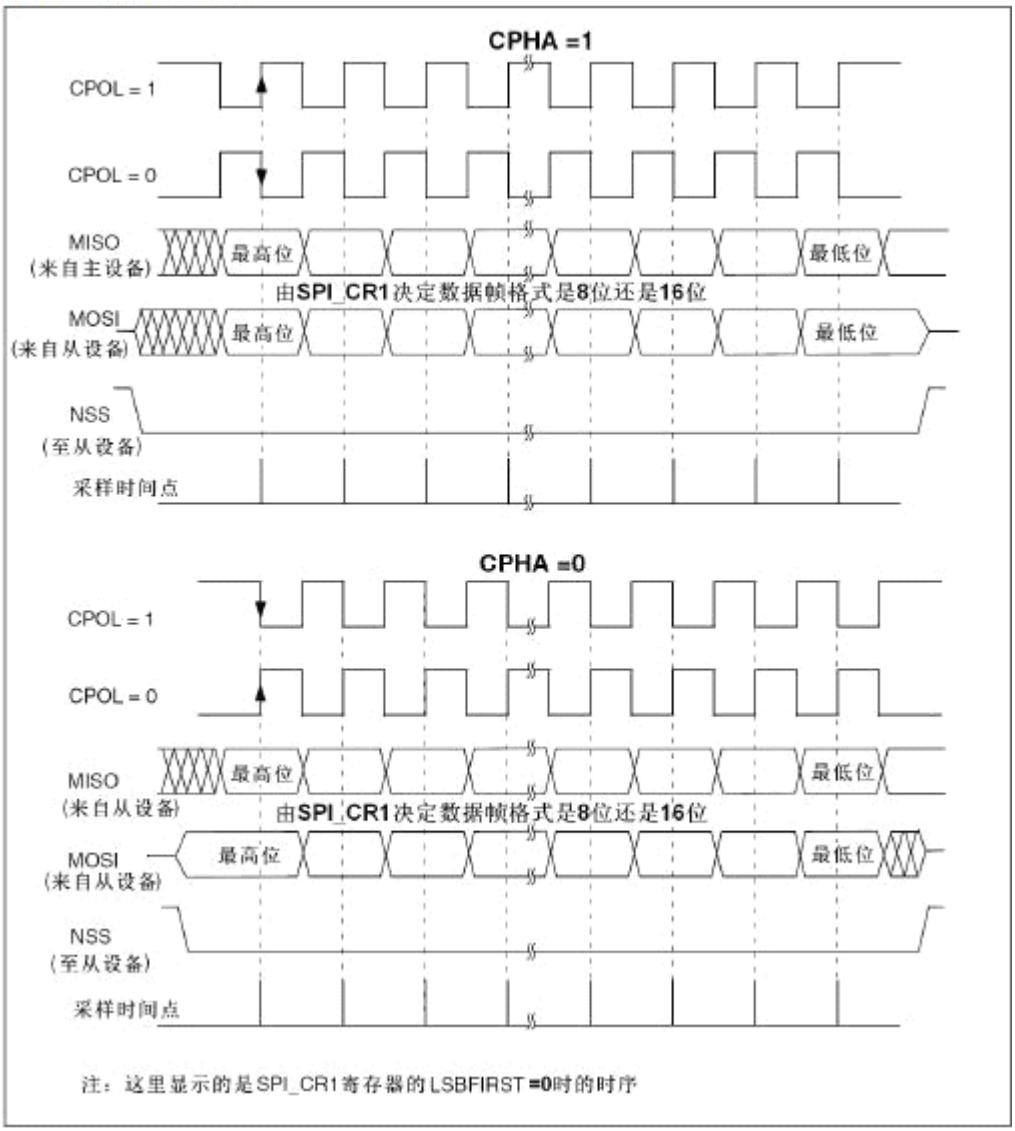
SPI_CR 寄存器的 CPOL 和 CPHA 位，能够组合成四种可能的时序关系。CPOL(时钟极性)位控制在没有数据传输时时钟的空闲状态电平，此位对主模式和从模式下的设备都有效。如果 CPOL 被清’0’，SCK 引脚在空闲状态保持低电平；如果 CPOL 被置’1’，SCK 引脚在空闲状态保持高电平。

如果 CPHA(时钟相位)位被置’1’，SCK 时钟的第二个边沿(CPOL 位为 0 时就是下降沿，CPOL 位为’1’时就是上升沿)进行数据位的采样，数据在第二个时钟边沿被锁存。如果 CPHA 位被清’0’，SCK 时钟的第一边沿(CPOL 位为’0’时就是下降沿，CPOL 位为’1’时就是上升沿)进行数据位采样，数据在第一个时钟边沿被锁存。

CPOL 时钟极性和 CPHA 时钟相位的组合选择数据捕捉的时钟边沿。

下图显示了 SPI 传输的 4 种 CPHA 和 CPOL 位组合。此图可以解释为主设备和从设备的 SCK 脚、MISO 脚、MOSI 脚直接连接的主或从时序图。

图212　数据时钟时序图



CPOL 时钟极性和 CPHA 时钟相位的组合选择数据捕捉的时钟边沿。

上图显示了 SPI 传输的 4 种 CPHA 和 CPOL 位组合。此图可以解释为主设备和从设备的 SCK 脚、MISO 脚、MOSI 脚直接连接的主或从时序图。

注意：

（1）在改变 CPOL/CPHA 位之前，必须清除 SPE 位将 SPI 禁止。

（2）主和从必须配置成相同的时序模式。

（3）SCK 的空闲状态必须和 SPI_CR1 寄存器指定的极性一致(CPOL 为'1'时，空闲时应上拉 SCK 为高电平；CPOL 为'0'时，空闲时应下拉 SCK 为低电平)。

（4）数据帧格式(8 位或 16 位)由 SPI_CR1 寄存器的 DFF 位选择，并且决定发送/接收的数据长度。

3、NSS 模式

（1）软件 NSS 模式：可以通过设置 SPI_CR1 寄存器的 SSM 位来使能这种模式。在这种模式下 NSS 引脚可以用作它用，而内部 NSS 信号电平可以通过写 SPI_CR1 的 SSI 位来驱动

（2）硬件 NSS 模式，分两种情况：

- NSS 输出被使能：当 STM32F10xxx 工作为主 SPI，并且 NSS 输出已经通过 SPI_CR2 寄存器的 SSOE 位使能，这时 NSS 引脚被拉低，所有 NSS 引脚与这个主 SPI 的 NSS 引脚相连并配置为硬件 NSS 的 SPI 设备，将自动变成从 SPI 设备。当一个 SPI 设备需要发送广播数据，它必须拉低 NSS 信号，以通知所有其它的设备它是主设备；如果它不能拉低 NSS，这意味着总线上有另外一个主设备在通信，这时将产生一个硬件失败错误(HardFault)。
- NSS 输出被关闭：允许操作于多主环境。

4、数据帧格式

根据 SPI_CR1 寄存器中的 LSBFIRST 位，输出数据位时可以 MSB 在先也可以 LSB 在先。根据 SPI_CR1 寄存器的 DFF 位，每个数据帧可以是 8 位或是 16 位。所选择的数据帧格式对发送和/或接收都有效。

5、SPI 初始化

```
SPI_InitTypeDef  SPI_InitStructure;
RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2,ENABLE);
SPI_Cmd(SPI2, DISABLE);                                  //必须先禁能,才能改变 MODE
SPI_InitStructure.SPI_Direction =SPI_Direction_2Lines_FullDuplex;   //两线全双工
SPI_InitStructure.SPI_Mode =SPI_Mode_Master;       //主
SPI_InitStructure.SPI_DataSize =SPI_DataSize_8b;     //8 位
SPI_InitStructure.SPI_CPOL =SPI_CPOL_High;         //CPOL=1 时钟悬空高
SPI_InitStructure.SPI_CPHA =SPI_CPHA_1Edge;       //CPHA=1  数据捕获第 2 个
SPI_InitStructure.SPI_NSS =SPI_NSS_Soft;             //软件 NSS
SPI_InitStructure.SPI_BaudRatePrescaler =SPI_BaudRatePrescaler_2;   //2 分频
SPI_InitStructure.SPI_FirstBit =SPI_FirstBit_MSB;     //高位在前
SPI_InitStructure.SPI_CRCPolynomial =7;               //CRC7
SPI_Init(SPI2,&SPI_InitStructure);                    //SPI 配置函数
SPI_Cmd(SPI2, ENABLE);                                //SPI 使能
```

第六章 基础实验

一、编程架构

对于 STM32 外设进行编程，可以按照下面的架构来初始化，这样思路会比较清楚。当出现问题的时候，可以较容易地判断出错的位置。

通用初始化

1、 RCC 初始化

全局与外设的时钟配置，要先开时钟再用外设。

2、 NVIC 初始化

中断通道使能及优先级设定。

3、 SysTick 初始化

嘀嗒定时器，提供时钟基准，精确延时需要。

4、 GPIO 初始化

GPIO 方式要与外设对应。

外设初始化

5、 定时器、ADC、USART、LCD。。。

主函数

6、根据实际情况，调用相关算法，实现所需功能。

中断

7、编写中断处理函数

二、实验说明

1、Keil 环境使用可参考其它资料；

2、实验二之后不在讲通用初始化部分，具体可参考实验源码；

3、下边实验中，LCD 部分与 GPIO 关于位带操作部分是参考正点原子代码，详细解释可看其 STM32 不完全手册；

4、所有例子在正点原子实验板上验证通过，芯片为 STM32F103RBT6。

实验一 点亮 LED

一、实验目的

了解开发流程，具体如下。

1、 安装 Keil（Keil v4.10）编程环境；
2、 安装程序下载软件 FlyMCU（www.mcuisp.com），此处版本为 2.2;
3、 编译跑马灯项目；
4、 下载.Hex 文件到开发板中运行。

二、实验步骤

1、用 keil 打开跑马灯工程文件后，在 Project 目录下打开 options for target 对话框如图 1.1 所示，勾选 Creat HEX File 与 Browse Information（方便我们定位源函数等），在 Name of Executable 处写入 STM32。这样我们就可以生成 STM32.hex 文件以供我们下载程序用。

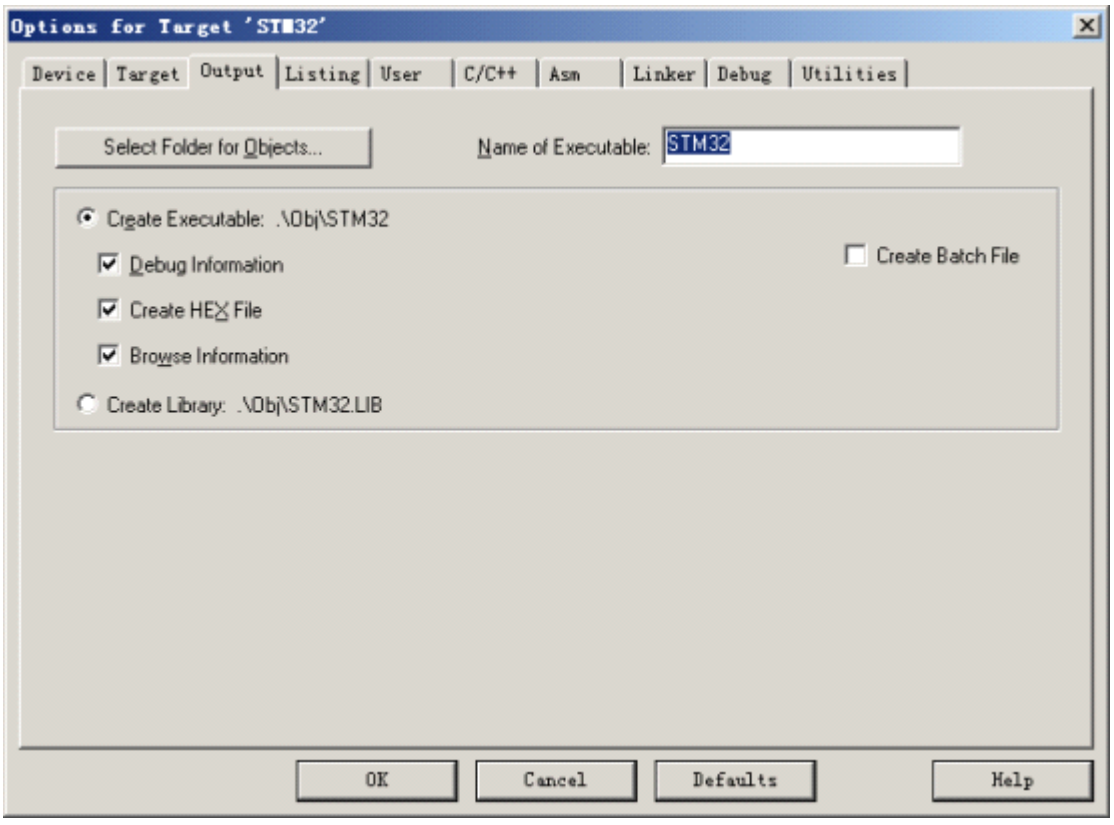


图 1.1 Options for Target 'STM32'

2、用 USB 线连接好开发板，保证开发板 BOOT0 与 BOOT1 引脚与 GND 短接，PA9 与 PA10 分别于 RXD 和 TXD 短接，具体可参照开发板硬件原理图。

3、打开 FlyMCU 程序如图 1.2 所示，根据实际路径进行选择，打开 实验一 跑马灯

\Project\Template\RVMDK\STM32.hex 文件。按照图 1.2 进行复选框的设置，在最下边的复选框中，选择 DTR 的低电平复位，RTS 高电平进 BootLoader，这是为了解决自动下载程序（免手动复位）而采取的手段，当然需要硬件上的电路来配合（可参照开发板原理图）。

图 1.2 程序下载界面

4、点击搜索串口，选择开发板对应的串口号，点读器件信息按钮，结果如图 1.2 右边所示，点击开始编程按钮把跑马灯的 STM32.hex 文件下载到开发板中，我们就可以观察到 DS0，DS1 在交替闪烁。

三、实验总结

第一次下载时，需要耐心，重点查看硬件是不是连接好，FlyMcu 中是不是找到了对应的串口。

FlyMcu 设置时，严格按照图 1.2 来做做，否则很有可能下载不成功，或出现其它问题。

实验一与实验二源码是相同的，此实验只为熟悉开发流程。

实验二 跑马灯

一、实验目的

1、 了解 RCC、NVIC、SysTick 初始化过程；
2、 了解 GPIO 端口初始化过程；
3、 了解延时函数编写
4、 了解 IO 口操作方式

二、实验步骤

1、打开实验例程；
2、编译代码；
3、下载代码到开发板，运行看代码是否正确；
4、按照下面实验代码逐一理解并尝试修改。

三、实验代码

1、主函数

```
/**************************************************************************
        LED0: PORTA.8
        LED1: PORTD.2
**************************************************************************/
int main(void)
{
  RCC_Configuration();     /* 时钟配置 */
  NVIC_Configuration();   /*中断向量配置 */
  SysTick_Configuration();/*SysTick 配置 */
  GPIO_Configuration();   /* GPIO 配置 */


  while(1)
  {
      置位端口;
      延时 1s;
      复位端口;
      延时 1s;
  }
```

}

2、RCC_Configuration()时钟初始化函数

SystemInit();//初始化时钟配置，指定时钟频率等（涉及到寄存器，可参考手册，此处不做详细介绍，库函数操作在前面已经有过介绍）

RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO | RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOD , ENABLE);

注意 AFIO 时钟，此处打开，用到复用功能时需要。

3、NVIC_Configuration()初始化函数

```
NVIC_InitTypeDef NVIC_InitStructure;//定义 NVIC 结构体
NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);//设置向量表基址为 FLASH 开始处

NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0); //设置中断优先级组，根据情况来改

/* Enable the TIM1 Interrupt */
NVIC_InitStructure.NVIC_IRQChannel = TIM1_TRG_COM_IRQn;// 指定外中断通道号
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;// 指定抢占式优先级
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;// 指定响应优先级
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;//使能
NVIC_Init(&NVIC_InitStructure); //NVIC 初始化

  /* Enable the TIM2 global Interrupt */ //设置同上
NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQChannel;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 4; //
NVIC_InitStructure.NVIC_IRQChannelCmd =ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

• • •

4、SysTick_Configuration()初始化函数

```
if (SysTick_Config((SystemFrequency)/1000))
  {
    /* Capture error */
    while (1);
  }
```

//NVIC_SetPriority(SysTick_IRQn, n);//n 取 0-15，对应中断优先级高四位的设置，针对不同的优先级分组，可以得到其抢占优先级与响应优先级。默认情况下，为最低优先级。

5、GPIO_Configuration()初始化函数

```
GPIO_InitTypeDef GPIO_InitStructure;//结构体定义
//LED0 PA8
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;//选择管脚 8
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;//定义该管脚为推挽输出方式
```

```
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //设置 IO 口翻转速度为 50MHz
GPIO_Init(GPIOA, &GPIO_InitStructure);//调用初始化函数
```

//LED1 PD2 解释同上

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

6、Delay 延时函数

//延时函数（nTime ms），此处单位为 SysTick 定时设定时间。

```
void Delayms(__IO uint32_t nTime)
{
  TimingDelay = nTime;

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

//SysTick 中断调用函数

```
void TimingDelay_Decrement(void)
{
  if (TimingDelay != 0x00)
  {
    TimingDelay--;
  }
}
```

8、 IO 口操作代码

//例子一 直接寄存器操作 PA.8（GPIOA.8）

```
GPIOA->ODR = 1<<8;
Delayms(1000);
GPIOA->ODR = 0<<8;
Delayms(1000);
```

//例子二 宏定义操作 PA.8（GPIOA.8） PD.2（GPIOD.2）

```
LED0 = !LED0; //GPIOA->ODR ^= (1<<8);
Delayms(1000);
LED1 = !LED1; //GPIOD->ODR ^= (1<<2);
```

//例子三 调用库函数 PA.8，涉及的库函数可参照 固件库手册

```
GPIO_WriteBit(GPIOA,GPIO_Pin_8,Bit_SET);
Delayms(1000);
```

```
GPIO_WriteBit(GPIOA,GPIO_Pin_8,Bit_RESET);
Delayms(1000);

//例子四 调用库函数 PD.2
GPIO_SetBits(GPIOD,GPIO_Pin_2);
Delayms(1000);
GPIO_ResetBits(GPIOD,GPIO_Pin_2);
Delayms(1000);
```

四、GPIO 相关库函数

GPIO_DeInit 重新初始化外围设备 GPIOx 相关寄存器到它的默认复位值
GPIO_AFIODeInit 初始化交错功能(remap, event control 和 EXTI 配置) 寄存器
GPIO_Init 根据 GPIO_初始化结构指定的元素初始化外围设备 GPIOx
GPIO_StructInit 填充 GPIO_初始化结构（GPIO_InitStruct）内的元素为复位值
GPIO_ReadInputDataBit 读指定端口引脚输入数据
GPIO_ReadInputData 读指定端口输入数据
GPIO_ReadOtputDataBit 读指定端口引脚输出数据
GPIO_ReadOtputData 读指定端口输出数据
GPIO_SetBits 置 1 指定的端口引脚
GPIO_ResetBits 清 0 指定的端口引脚
GPIO_WriteBit 设置或清除选择的数据端口引脚
GPIO_Write 写指定数据到 GPIOx 端口寄存器
GPIO_PinLockConfig 锁定 GPIO 引脚寄存器
GPIO_EventOutputConfig 选择 GPIO 引脚作为事件输出
GPIO_EventOutputCmd 允许或禁止事件输出
GPIO_PinRemapConfig 改变指定引脚的映射
GPIO_EXTILineConfig 将 IO 口注册到中断线(将 IO 口映射到中断线)

五、实验总结

跑马灯本资料的第一个实验，涉及面多，花时间理解是有必要的。用宏定义的操作涉及到位段的概念，详见 Cortex-M3 权威指南 87 页。

GPIO 是最常用的，需要经常查阅对应管脚及其复用功能，所以打印出管脚对应表格是有必要的，对应表格见附录。

实验三 1S 定时

一、实验目的

1、 了解定时器 TIM2 初始化流程；
2、 了解定时器中断编写过程；

二、实验步骤

1、打开实验例程；
2、下载并运行代码；
3、了解定时器初始化函数（RCC、NVIC 与 SysTick 配置参照实验二 跑马灯）；
4、编写定时器中断函数，实现 LED 以 1s 为周期闪烁（stm32f10x_it.c）；
5、编写空的主函数。

三、实验代码

1、定时器初始化

时钟预分频

//APB1, TIM2,36MHz 时钟，分频系数 36000，定时器时钟 36000 000/36000 = 1000Hz;

TIM_TimeBaseStructure.TIM_Prescaler = (SystemFrequency)/1000/2-1;

计数次数

//TIM2 定时时间 1ms * 1000 = 1s;

TIM_TimeBaseStructure.TIM_Period =1000 ; //1s

定时器时钟分割：定义在定时器时钟(CK_INT)频率与数字滤波器(ETR,TIx)使用的采样频率之间的分频比例

TIM_TimeBaseStructure.TIM_ClockDivision = 0;

计数模式，向上

TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

定时器初始化函数

TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

定时器使能

TIM_Cmd(TIM2, ENABLE);

定时器中断使能

TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

2、定时器中断函数

```
void TIM2_IRQHandler(void)// TIM2_IRQHandler 与中断向量表中名称一致
{

    TIM_ClearITPendingBit(TIM2,TIM_IT_Update);//清除中断标志
    LED0 = !LED0;//翻转
}
```

3、主函数

空循环

四、实验总结

实验进行应该顺利，难点在没有参考例程的情况下，如何用库函数来实现定时功能，就需要了解定时器初始化相关步骤，详细理解需参考 STM32 技术手册。

此实验也是第一个有点难度的实验，也是 STM32 中比较重要的，花大力气搞清楚是值得的。

实验四 串口通信

一、实验目的

1、 了解串口通信初始化步骤；
2、 了解串口通信中断函数编写；

二、实验步骤

1、打开实验例程；
2、编译代码，并下载测试；
3、了解 GPIO 中串口的初始化；
4、了解串口初始化步骤；
5、了解串口中断函数；
6、编写主程序，实现发送功能（尝试修改）；

三、实验代码

1、GPIO 初始化

```
//USART 1   PA10:RXD
/* Configure USART1 Rx (PA.10) as input floating */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;//输入
GPIO_Init(GPIOA, &GPIO_InitStructure);

//USART 1   PA9:TXD，复用功能管脚
/* Configure USART1 Tx (PA.09) as alternate function push-pull */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

2、USART_Configuration()串口初始化函数

```
USART_InitTypeDef USART_InitStructure;
USART_InitStructure.USART_BaudRate = 115200;//波特率设置
USART_InitStructure.USART_WordLength = USART_WordLength_8b;//8 位数据
USART_InitStructure.USART_StopBits = USART_StopBits_1;// 1 停止位
USART_InitStructure.USART_Parity = USART_Parity_No ;//无校验
```

```
//无硬件流控制
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;//收发模式
/* Configure USART1 *///初始化上面设置
USART_Init(USART1, &USART_InitStructure);

/* Enable USART1 Receive interrupts *///使能接收中断
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
/* Enable the USART1 *///打开串口
USART_Cmd(USART1, ENABLE);
```

3、串口中断函数

```
if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)   //确认是否接收到数据
    {
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);//清除中断标志位
        USART_SendData(USART1, USART1->DR);//发送数据
        while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET){}; //判断是否发送完成
    }
```

4、重定向 printf 函数

(1)#include "stdio.h" //printf 原型

(2)定义函数

```
int fputc(int ch, FILE *f)
{
  /* Write a character to the USART */
  USART_SendData(USART1, (uint8_t) ch);

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

(3)在 Options for Target\Target 中，勾选 Use MicroLIB

五、实验总结

注意 IO 口的初始化，发送端是复用推挽输出。重定向 printf 步骤要全，串口中断中的 USART_GetITStatus 获取中断标志状态的函数与 USART_GetFlagStatus 获取发送完成标志的函数是不同的，不能混淆。注意先换行后回车，否则的话，起不到我们想要的换行效果。

实验五 频率测量

一、实验目的

了解频率测量原理；

二、实验步骤

1、打开实验例程；
2、编译代码并运行；
3、了解 TIM3 初始化过程；
4、了解 TIM3 中断处理过程；
5、编写主函数，通过串口传输频率值。

三、实验代码

1、TIM3 初始化

```
/* Time base configuration */
//预分频 360，得到 TIM3 时钟为 100KHz
  TIM_TimeBaseStructure.TIM_Prescaler = 359;//SystemFrequency/100000/2-1;//359;
//计数周期
  TIM_TimeBaseStructure.TIM_Period = 65535; //1us
定时器时钟分割：定义在定时器时钟(CK_INT)频率与数字滤波器(ETR,TIx)使用的采样频率之间的分频比例
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
计数模式，向上
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
定时器初始化函数
    TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
选择通道 4 为频率输入脚
  TIM_ICInitStructure.TIM_Channel = TIM_Channel_4;
输入/捕获上升沿有效
  TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;
管脚与寄存器对应关系
  TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
分频器，每次检测到捕获输入就触发一次捕获
  TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
滤波设置
  TIM_ICInitStructure.TIM_ICFilter = 0x0;
初始化
  TIM_ICInit(TIM3, &TIM_ICInitStructure);
```

定时器使能

```
TIM_Cmd(TIM3 ENABLE);
```

定时器中断使能

```
TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE);
```

2、中断处理函数

```
 if(TIM_GetITStatus(TIM3, TIM_IT_CC4) == SET)
{
/* Clear TIM3 Capture compare interrupt pending bit */
      TIM_ClearITPendingBit(TIM3, TIM_IT_CC4);//清除中断标志
      if(CaptureNumber == 0)
      {
        /* Get the Input Capture value */
        IC3ReadValue1 = TIM_GetCapture4(TIM3);//得到第一次计数值
        CaptureNumber = 1;
      }
      else
            if(CaptureNumber == 1)
            {
              /* Get the Input Capture value */
              IC3ReadValue2 = TIM_GetCapture4(TIM3); //得到第二次计数值

              /* Capture computation *///计算个数
                    if (IC3ReadValue2 > IC3ReadValue1)
                    {
                      Capture = (IC3ReadValue2 - IC3ReadValue1);
                    }
                    else
                    {
                      Capture = ((0xFFFF - IC3ReadValue1) + IC3ReadValue2);
                    }
              /* Frequency computation */
              TIM3Freq = 100000/ Capture;      //得到给定信号的频率
              CaptureNumber = 0;
            }
  }
```

五、实验总结

输入捕获也是定时器中重要的一部分，需要的是耐心，往往出不来结果是由于某个环节忽略掉了，按照前面的架构重新梳理思路应该都可以解决。

实验六 6 步 PWM 输出

一、实验目的

1、 了解 6 步 PWM 产生配置（IO: PA8,PA9,PA10,PB13,PB14,PB15）；
2、 了解换向过程；

二、实验步骤

1、打开实验例程；
2、编译代码并运行；
3、了解 TIM1 的初始化过程；
4、了解 TIM1 中断过程；

三、实验代码

1、TIM1 初始化

```
  TIM_DeInit(TIM1);
/* Time Base configuration */
  TIM_TimeBaseStructure.TIM_Prescaler = 0; //预分频器
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_CenterAligned2;// 中央对齐模式
  TIM_TimeBaseStructure.TIM_Period = TIMPeriod-1;//自动重装载寄存器
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;    //采样时钟分频比例
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;//重复计数器 中心对齐模式下指半周期数目
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure); //初始化配置函数

/* Channel 1, 2,3 and 4 Configuration in PWM mode */
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Timing; //输出比较时间模式(输出引脚冻结无效)
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //CCER[0]输出使能
  TIM_OCInitStructure.TIM_OutputNState = TIM_OutputNState_Enable;//互补输出使能
  TIM_OCInitStructure.TIM_Pulse = TIMPeriod/2;    //捕获比较寄存器
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;// 关闭时默认输出极性
  TIM_OCInitStructure.TIM_OCNPolarity = TIM_OCPolarity_Low;//互补端关闭时默认极性
//  TIM_OCInitStructure.TIM_OCIdleState = TIM_OCIdleState_Set;    //死区后状态
//  TIM_OCInitStructure.TIM_OCNIdleState = TIM_OCNIdleState_Set;
```

```
//1 通道初始化
TIM_OC1Init(TIM1, &TIM_OCInitStructure);
//2 通道初始化
TIM_OCInitStructure.TIM_Pulse = TIMPeriod/2 ;
TIM_OC2Init(TIM1, &TIM_OCInitStructure);
//3 通道初始化
TIM_OCInitStructure.TIM_Pulse = TIMPeriod/2 ;
TIM_OC3Init(TIM1, &TIM_OCInitStructure);

TIM_OCInitStructure.TIM_Pulse = TIMPeriod - 50;
TIM_OC4Init(TIM1, &TIM_OCInitStructure);

TIM_CCPreloadControl(TIM1, ENABLE);//预装载控制 CCxE,CCxNE(CCER 0 2),CCxM
TIM_ITConfig(TIM1, TIM_IT_COM, ENABLE);   //DIER 中断使能寄存器      //优先级
/* TIM1 counter enable */
TIM_Cmd(TIM1, ENABLE);
/* Main Output Enable */
TIM_CtrlPWMOutputs(TIM1, ENABLE);
```

2、TIM1_TRG_COM_IRQHandler()中断处理函数

```
switch(step) //以 AB 相为例，step＝1
  {
    case 1:
            {
                /* Next step: Step 2 Configuration AB---------------------- */
                 /*   Channel3 configuration */
                 TIM_CCxCmd(TIM1, TIM_Channel_3, TIM_CCx_Disable);
                 TIM_CCxNCmd(TIM1, TIM_Channel_3, TIM_CCxN_Disable);

                 /*   Channel1 configuration */ PWM 输出
                 TIM_SelectOCxM(TIM1, TIM_Channel_1, TIM_OCMode_PWM1);
                 TIM_CCxCmd(TIM1, TIM_Channel_1, TIM_CCx_Enable);
                 TIM_CCxNCmd(TIM1, TIM_Channel_1, TIM_CCxN_Disable);

                 /*   Channel2 configuration */ 高电平输出
                 TIM_SelectOCxM(TIM1, TIM_Channel_2,   TIM_ForcedAction_Active );
                 TIM_CCxCmd(TIM1, TIM_Channel_2, TIM_CCx_Disable);
                 TIM_CCxNCmd(TIM1, TIM_Channel_2, TIM_CCxN_Enable);
                 step++;
            } break;
    case 2:
```

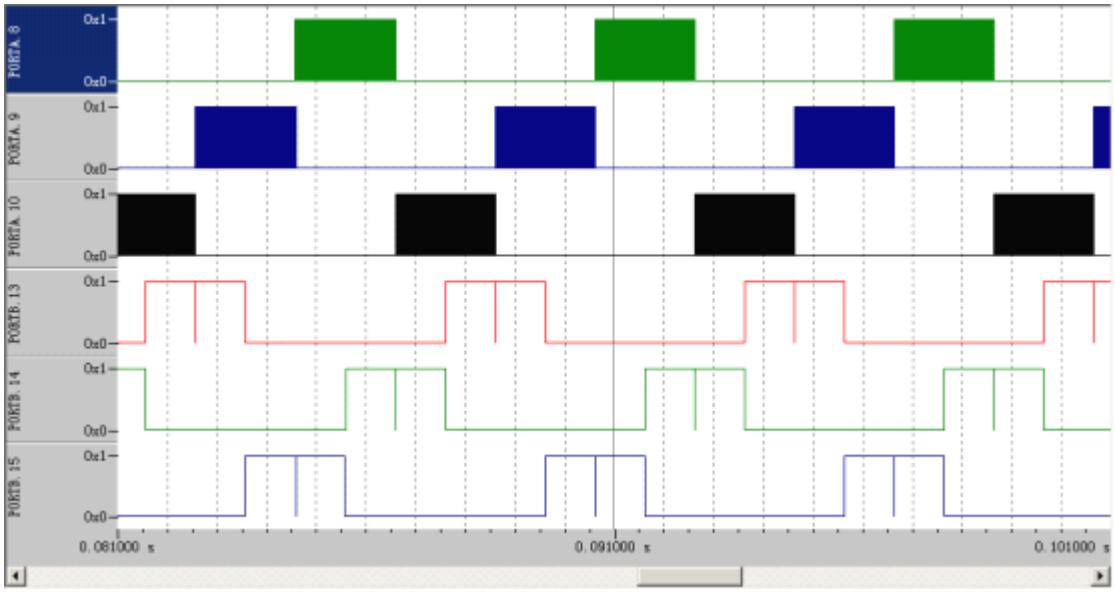
• • •

3、SysTick_Handler()中断函数

为了照顾延时函数，在 SysTick 以 1us 中断的情况下，此代码实现了 1ms 触发 COM 中断

```
while(systic_i>=1000)
{
    systic_i = 0;
    TIM_GenerateEvent(TIM1,TIM_EventSource_COM);//触发 COM 中断
};
```

四、仿真输出结果



五、实验总结

本实验为了说明 6 步 PWM 的产生原理，以周期为 1ms 时间触发 COM 中断。此实验是三项电机的基础，出来现象不难，有很多资料可参考，但仍需我们细看 STM32 技术参考手册。

实验七 LCD 显示

一、实验目的

1、 了解 LCD 代码移植流程；

二、实验步骤

1、打开实验例程；
2、编译代码并运行，观察效果；
3、在实验六重新按步骤 4-8 实现显示功能；
4、添加 ILI93xx.c(LCD 函数库)；
5、添加 lcd.h 头文件；
6、替换 LCD_Init()中原有的延时函数；
7、调用 LCD_Init()初始化函数；
8、设置字体颜色，并调用显示函数；

三、实验代码

1、LCD_Init()初始化函数

此处移植的是正点原子的代码，详见正点原子不完全手册 3.10 TFTLCD 显示实验。

五、实验总结

LCD 显示占用了很多管脚，当管脚被影响时，有可能输出不正常。PA8 是 LED0 管脚，如果与 6 步 PWM 同时用的话，PA8 会重复配置，此时只能有一项起作用。

实验八 ADC

一、实验目的

1、 了解规则转换与注入转换概念；
2、 了解软件触发与外部触发；
3、 了解 ADC 初始化与中断处理过程；
4、 了解 DMA 传输；

二、实验步骤

1、打开实验例程；
2、编译代码并运行；
3、理解 ADC 初始化流程；
4、理解 ADC 中断流程；
5、理解主函数原理；

三、实验代码

1、ADC 初始化

DMA 配置

```
DMA_DeInit(DMA1_Channel1);
DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;//外设地址
DMA_InitStructure.DMA_MemoryBaseAddr = (u32)ADC_RCVTab;//内存地址
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;//dma 传输方向单向
DMA_InitStructure.DMA_BufferSize = 15;//设置 DMA 在传输时缓冲区的长度
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//设置 DMA 的外设递增模式，一个外设
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;//设置 DMA 的内存递增模式，
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;// 外设数据字长
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;// 内存数据字长
DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;//设置 DMA 的传输模式：连续不断的循环模式
DMA_InitStructure.DMA_Priority = DMA_Priority_High;//设置 DMA 的优先级别
DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;// DMA 没有设置内存到内存传输
```

```
DMA_Init(DMA1_Channel1, &DMA_InitStructure);//初始化函数
/* Enable DMA1 channel1 */
DMA_Cmd(DMA1_Channel1, ENABLE);
```

ADC1 配置

```
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;//独立工作模式
ADC_InitStructure.ADC_ScanConvMode = ENABLE;//扫描方式，使能
ADC_InitStructure.ADC_ContinuousConvMode =DISABLE;//不连续转换
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1;
                        //ADC_ExternalTrigConv_None;//是否选择外部触发禁止
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;//数据右对齐
ADC_InitStructure.ADC_NbrOfChannel = 3;//用于转换的通道数
ADC_Init(ADC1, &ADC_InitStructure);


/* ADC1 regular channels configuration [规则模式通道配置]*/
ADC_RegularChannelConfig(ADC1, ADC_Channel_1 , 1, ADC_SampleTime_239Cycles5);
ADC_RegularChannelConfig(ADC1, ADC_Channel_2 , 2, ADC_SampleTime_239Cycles5);
ADC_RegularChannelConfig(ADC1, ADC_Channel_3 , 3, ADC_SampleTime_239Cycles5);


  /* Set injected sequencer length 设置注入转换长度*/
ADC_InjectedSequencerLengthConfig(ADC1, 3);
/* ADC1 injected channel Configuration 注入转换通道配置*/
ADC_InjectedChannelConfig(ADC1, ADC_Channel_4, 1, ADC_SampleTime_239Cycles5);
ADC_InjectedChannelConfig(ADC1, ADC_Channel_5, 2, ADC_SampleTime_239Cycles5);
ADC_InjectedChannelConfig(ADC1, ADC_Channel_6, 3, ADC_SampleTime_239Cycles5);
/* ADC1 injected external trigger configuration */
ADC_ExternalTrigInjectedConvConfig(ADC1, ADC_ExternalTrigInjecConv_None);// 是否允许外部触发

/* Enable automatic injected conversion start after regular one 当规则转换完毕后，自动启动注入转换*/
ADC_AutoInjectedConvCmd(ADC1, ENABLE);

/* Enable ADC1  DMA 使能 DMA*/
ADC_DMACmd(ADC1, ENABLE);

/* Enable ADC1 external trigger  使能外部触发转换*/
ADC_ExternalTrigConvCmd(ADC1, ENABLE);

/* Enable JEOC interrupt  使能注入转换中断*/
ADC_ITConfig(ADC1, ADC_IT_JEOC, ENABLE);
```

```
/* Enable ADC1 [使能 ADC1]*/
ADC_Cmd(ADC1, ENABLE);

/* Enable ADC1 reset calibaration register 复位校准寄存器*/
ADC_ResetCalibration(ADC1);
/* Check the end of ADC1 reset calibration register */
while(ADC_GetResetCalibrationStatus(ADC1));

/* Start ADC1 calibaration 启动 ADC1 校准*/
ADC_StartCalibration(ADC1);
/* Check the end of ADC1 calibration */
while(ADC_GetCalibrationStatus(ADC1));

/* Start ADC1 Software Conversion 软件触发启动函数*/
//ADC_SoftwareStartConvCmd(ADC1, ENABLE);
```

绿色的文字与触发方式（软件触发与外部触发）有关。

几个重要参数

(1) ADC_Mode，这里设置为独立模式：

ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;

在这个模式下，双 ADC 不能同步，每个 ADC 接口独立工作。所以如果不需要 ADC 同步或者只是用了一个 ADC 的时候，就应该设成独立模式了。

(2) ADC_ScanConvMode，这里设置为 DISABLE。

ADC_InitStructure.ADC_ScanConvMode = DISABLE;

如果只是用了一个通道的话，DISABLE 就可以了，如果使用了多个通道的话，则必须将其设置为 ENABLE。

(3) ADC_ContinuousConvMode，这里设置为 ENABLE，即连续转换。

如果设置为 DISABLE，则是单次转换。两者的区别在于连续转换直到所有的数据转换完成后才停止转换，而单次转换则只转换一次数据就停止，要再次触发转换才可以。所以如果需要一次性采集 1024 个数据或者更多，则采用连续转换。

(4) ADC_ExternalTrigConv，即选择外部触发模式。这里只讲三种：

a、第一种是最简单的软件触发，参数为 ADC_ExternalTrigConv_None。设置好后还要记得调用库函数：ADC_SoftwareStartConvCmd(ADC1, ENABLE);这样触发才会启动。

b、第二种是定时器通道输出触发。共有这几种：ADC_ExternalTrigConv_T1_CC1、ADC_ExternalTrigConv_T1_CC2、ADC_ExternalTrigConv_T2_CC2、ADC_ExternalTrigConv_T3_T 以及 ADC_ExternalTrigConv_T4_CC4。定时器输出需要设置相应的定时器。

c、第三种是外部引脚触发,对于规则通道，选择 EXTI 线 11 和 TIM8_TRGO 作为外部触发事件；而注入通道组则选择 EXTI 线 15 和 TIM8_CC4 作为外部触发事件。

(5) ADC_DataAlign,这里设置为 ADC_DataAlign_Right 右对齐方式。

建议采用右对齐方式，因为这样处理数据会比较方便。当然如果要从高位开始传输数据，那么采用左对齐优势就明显了。

(6) ADC_NbrOfChannel，顾名思义：通道的数量。

要是到多个通道采集数据的话就得设置一下这个参数。

2、ADC 中断

由于 ADC 只有一个中断入口，所以需要在中断里判断是规则转换中断还是注入转换中断

```
if(ADC_GetITStatus(ADC1,ADC_IT_JEOC)!=RESET)
{
ADC_ClearITPendingBit(ADC1,ADC_IT_JEOC);//清中断标志
ADC_ClearFlag(ADC1,ADC_FLAG_JSTRT);// 清启动标志
ADC_RCVTab[3]=ADC_GetInjectedConversionValue(ADC1, ADC_InjectedChannel_1);
ADC_RCVTab[4]=ADC_GetInjectedConversionValue(ADC1, ADC_InjectedChannel_2);
ADC_RCVTab[5]=ADC_GetInjectedConversionValue(ADC1, ADC_InjectedChannel_3);
}
```

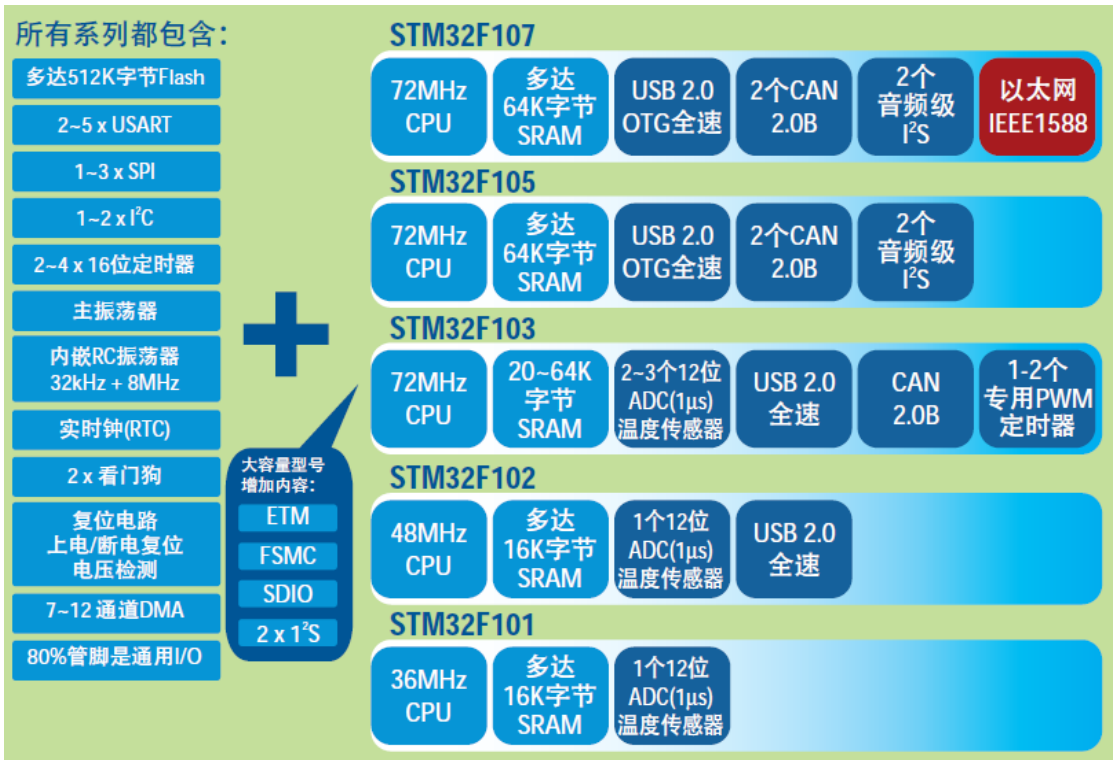
四、实验总结

ADC 管脚与 GPIO 的对应情况，外部触发方式对应的定时器与管脚等，用处较多，需要参考前面资料，或 STM32 技术手册等。

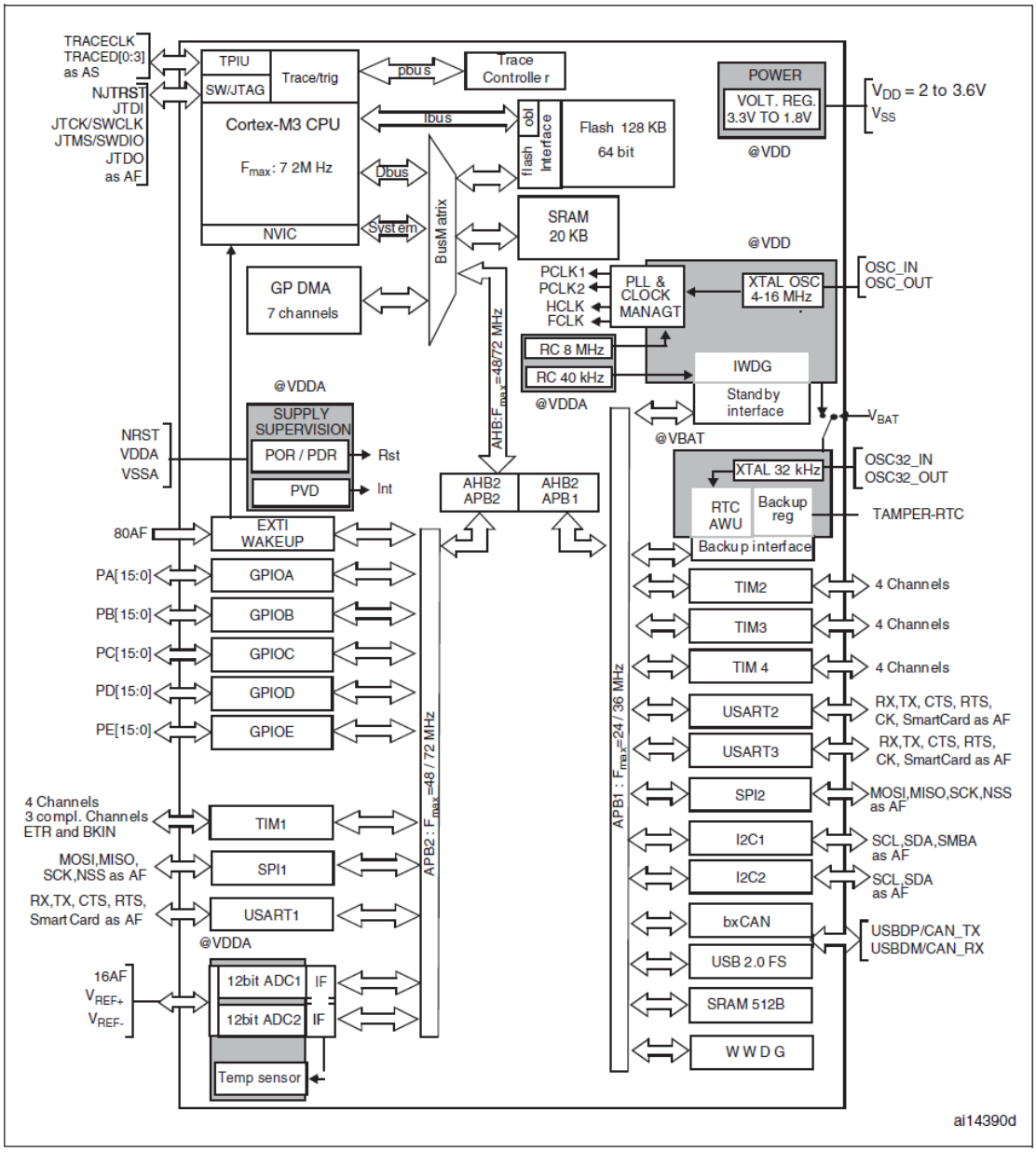
调试程序时，一般都是在现有基础上进行更改，所以更改触发模式软件或外部，增加 DMA 方式，改变通道数及修改中断处理函数等等，只需要按照所给例程修改相应的地方就可以了。难的主要就是入门时，所有的更改都得理解其原理，这就需要我们在用的时候参考 STM32 技术手册。

附录

STM32 系列处理器



STM32F103RB 内部结构



IO 管脚

Pins						Pin name	Type(1)	I / O Level(2)	Main function(3) (after reset)	Alternate functions(4)	
LFBGA100	LQFP48/VFQFPN48	TFBGA64	LQFP64	LQFP100	VFQFPN36					Default	Remap
A3	-		-	1	-	PE2	I/O	FT	PE2	TRACECK	
B3	-		-	2	-	PE3	I/O	FT	PE3	TRACED0	
C3	-		-	3	-	PE4	I/O	FT	PE4	TRACED1	
D3	-		-	4	-	PE5	I/O	FT	PE5	TRACED2	
E3	-		-	5	-	PE6	I/O	FT	PE6	TRACED3	
B2	1	B2	1	6	-	V_{BAT}	S		V_{BAT}		
A2	2	A2	2	7	-	PC13-TAMPER-RTC(5)	I/O		PC13(6)	TAMPER-RTC	
A1	3	A1	3	8	-	PC14-OSC32_IN(5)	I/O		PC14(6)	OSC32_IN	
B1	4	B1	4	9	-	PC15-OSC32_OUT(5)	I/O		PC15(6)	OSC32_OUT	
C2	-	-	-	10	-	V_{SS_5}	S		V_{SS_5}		
D2	-	-	-	11	-	V_{DD_5}	S		V_{DD_5}		
C1	5	C1	5	12	2	OSC_IN	I		OSC_IN		
D1	6	D1	6	13	3	OSC_OUT	O		OSC_OUT		
E1	7	E1	7	14	4	NRST	I/O		NRST		
F1	-	E3	8	15	-	PC0	I/O		PC0	ADC12_IN10	
F2	-	E2	9	16	-	PC1	I/O		PC1	ADC12_IN11	
E2	-	F2	10	17	-	PC2	I/O		PC2	ADC12_IN12	
F3	-	-(7)	11	18	-	PC3	I/O		PC3	ADC12_IN13	
G1	8	F1	12	19	5	V_{SSA}	S		V_{SSA}		
H1	-	-	-	20	-	V_{REF-}	S		V_{REF-}		
J1	-	G1(7)	-	21	-	V_{REF+}	S		V_{REF+}		
K1	9	H1	13	22	6	V_{DDA}	S		V_{DDA}		
G2	10	G2	14	23	7	PA0-WKUP	I/O		PA0	WKUP/ USART2_CTS(8)/ ADC12_IN0/ TIM2_CH1_ETR(8)	
H2	11	H2	15	24	8	PA1	I/O		PA1	USART2_RTS(8)/ ADC12_IN1/ TIM2_CH2(8)	

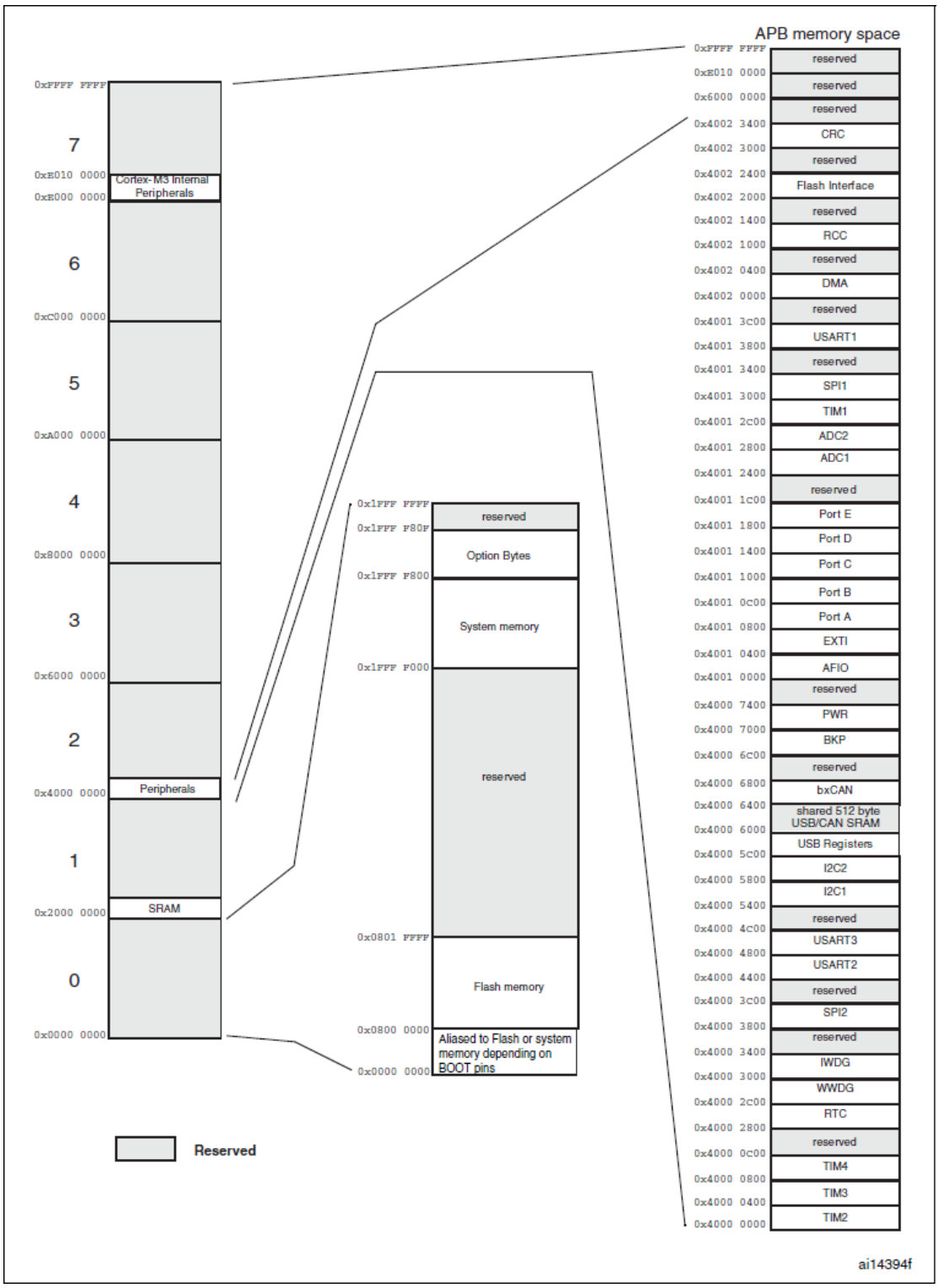
Pins						Pin name	Type[1]	I / O Level[2]	Main function[3] (after reset)	Alternate functions[4]	
LFBGA100	LQFP48/VFQFPN48	TFBGA64	LQFP64	LQFP100	VFQFPN36					Default	Remap
J2	12	F3	16	25	9	PA2	I/O		PA2	USART2_TX[8]/ ADC12_IN2/ TIM2_CH3[8]	
K2	13	G3	17	26	10	PA3	I/O		PA3	USART2_RX[8]/ ADC12_IN3/ TIM2_CH4[8]	
E4	-	C2	18	27	-	V_{SS_4}	S		V_{SS_4}		
F4	-	D2	19	28	-	V_{DD_4}	S		V_{DD_4}		
G3	14	H3	20	29	11	PA4	I/O		PA4	SPI1_NSS[8]/ USART2_CK[8]/ ADC12_IN4	
H3	15	F4	21	30	12	PA5	I/O		PA5	SPI1_SCK[8]/ ADC12_IN5	
J3	16	G4	22	31	13	PA6	I/O		PA6	SPI1_MISO[8]/ ADC12_IN6/ TIM3_CH1[8]	TIM1_BKIN
K3	17	H4	23	32	14	PA7	I/O		PA7	SPI1_MOSI[8]/ ADC12_IN7/ TIM3_CH2[8]	TIM1_CH1N
G4	-	H5	24	33		PC4	I/O		PC4	ADC12_IN14	
H4	-	H6	25	34		PC5	I/O		PC5	ADC12_IN15	
J4	18	F5	26	35	15	PB0	I/O		PB0	ADC12_IN8/ TIM3_CH3[8]	TIM1_CH2N
K4	19	G5	27	36	16	PB1	I/O		PB1	ADC12_IN9/ TIM3_CH4[8]	TIM1_CH3N
G5	20	G6	28	37	17	PB2	I/O	FT	PB2/BOOT1		
H5	-	-	-	38	-	PE7	I/O	FT	PE7		TIM1_ETR
J5	-	-	-	39	-	PE8	I/O	FT	PE8		TIM1_CH1N
K5	-	-	-	40	-	PE9	I/O	FT	PE9		TIM1_CH1
G6	-	-	-	41	-	PE10	I/O	FT	PE10		TIM1_CH2N
H6	-	-	-	42	-	PE11	I/O	FT	PE11		TIM1_CH2
J6	-	-	-	43	-	PE12	I/O	FT	PE12		TIM1_CH3N
K6	-	-	-	44	-	PE13	I/O	FT	PE13		TIM1_CH3
G7	-	-	-	45	-	PE14	I/O	FT	PE14		TIM1_CH4

Pins						Pin name	Type[1]	I / O Level[2]	Main function[3] (after reset)	Alternate functions[4]	
LFBGA100	LQFP48/VFQFPN48	TFBGA64	LQFP64	LQFP100	VFQFPN36					Default	Remap
H7	-	-	-	46	-	PE15	I/O	FT	PE15		TIM1_BKIN
J7	21	G7	29	47	-	PB10	I/O	FT	PB10	I2C2_SCL/ USART3_TX[8]	TIM2_CH3
K7	22	H7	30	48	-	PB11	I/O	FT	PB11	I2C2_SDA/ USART3_RX[8]	TIM2_CH4
E7	23	D6	31	49	18	V_{SS_1}	S		V_{SS_1}		
F7	24	E6	32	50	19	V_{DD_1}	S		V_{DD_1}		
K8	25	H8	33	51	-	PB12	I/O	FT	PB12	SPI2_NSS/ I2C2_SMBAI/ USART3_CK[8]/ TIM1_BKIN[8]	
J8	26	G8	34	52	-	PB13	I/O	FT	PB13	SPI2_SCK/ USART3_CTS[8]/ TIM1_CH1N [8]	
H8	27	F8	35	53	-	PB14	I/O	FT	PB14	SPI2_MISO/ USART3_RTS[8] TIM1_CH2N [8]	
G8	28	F7	36	54	-	PB15	I/O	FT	PB15	SPI2_MOSI/ TIM1_CH3N[8]	
K9	-	-	-	55	-	PD8	I/O	FT	PD8		USART3_TX
J9	-	-	-	56	-	PD9	I/O	FT	PD9		USART3_RX
H9	-	-	-	57	-	PD10	I/O	FT	PD10		USART3_CK
G9	-	-	-	58	-	PD11	I/O	FT	PD11		USART3_CTS
K10	-	-	-	59	-	PD12	I/O	FT	PD12		TIM4_CH1 / USART3_RTS
J10	-	-	-	60	-	PD13	I/O	FT	PD13		TIM4_CH2
H10	-	-	-	61	-	PD14	I/O	FT	PD14		TIM4_CH3
G10	-	-	-	62	-	PD15	I/O	FT	PD15		TIM4_CH4
F10	-	F6	37	63	-	PC6	I/O	FT	PC6		TIM3_CH1
E10		E7	38	64	-	PC7	I/O	FT	PC7		TIM3_CH2
F9		E8	39	65	-	PC8	I/O	FT	PC8		TIM3_CH3
E9	-	D8	40	66	-	PC9	I/O	FT	PC9		TIM3_CH4
D9	29	D7	41	67	20	PA8	I/O	FT	PA8	USART1_CK/ TIM1_CH1[8]/MCO	

Pins						Pin name	Type[1]	I / O Level[2]	Main function[3] (after reset)	Alternate functions[4]	
LFBGA100	LQFP48/VFQFPN48	TFBGA64	LQFP64	LQFP100	VFQFPN36					Default	Remap
C9	30	C7	42	68	21	PA9	I/O	FT	PA9	USART1_TX[8]/ TIM1_CH2[8]	
D10	31	C6	43	69	22	PA10	I/O	FT	PA10	USART1_RX[8]/ TIM1_CH3[8]	
C10	32	C8	44	70	23	PA11	I/O	FT	PA11	USART1_CTS/ CANRX[8]/ USBDM TIM1_CH4[8]	
B10	33	B8	45	71	24	PA12	I/O	FT	PA12	USART1_RTS/ CANTX[8] //USBDP TIM1_ETR[8]	
A10	34	A8	46	72	25	PA13	I/O	FT	JTMS/SWDIO		PA13
F8	-	-	-	73	-	Not connected					
E6	35	D5	47	74	26	V_{SS_2}	S		V_{SS_2}		
F6	36	E5	48	75	27	V_{DD_2}	S		V_{DD_2}		
A9	37	A7	49	76	28	PA14	I/O	FT	JTCK/SWCLK		PA14
A8	38	A6	50	77	29	PA15	I/O	FT	JTDI		TIM2_CH1_ETR/ PA15 /SPI1_NSS
B9	-	B7	51	78		PC10	I/O	FT	PC10		USART3_TX
B8	-	B6	52	79		PC11	I/O	FT	PC11		USART3_RX
C8	-	C5	53	80		PC12	I/O	FT	PC12		USART3_CK
D8	5	C1	5	81	2	PD0	I/O	FT	OSC_IN[9]		CANRX
E8	6	D1	6	82	3	PD1	I/O	FT	OSC_OUT[9]		CANTX
B7		B5	54	83	-	PD2	I/O	FT	PD2	TIM3_ETR	
C7	-	-	-	84	-	PD3	I/O	FT	PD3		USART2_CTS
D7	-	-	-	85	-	PD4	I/O	FT	PD4		USART2_RTS
B6	-	-	-	86	-	PD5	I/O	FT	PD5		USART2_TX
C6	-	-	-	87	-	PD6	I/O	FT	PD6		USART2_RX
D6	-	-	-	88	-	PD7	I/O	FT	PD7		USART2_CK
A7	39	A5	55	89	30	PB3	I/O	FT	JTDO		TIM2_CH2 / PB3 TRACESWO SPI1_SCK
A6	40	A4	56	90	31	PB4	I/O	FT	JNTRST		TIM3_CH1/ PB4/ SPI1_MISO

Pins						Pin name	Type[1]	I / O Level[2]	Main function[3] (after reset)	Alternate functions[4]	
LFBGA100	LQFP48/VFQFPN48	TFBGA64	LQFP64	LQFP100	VFQFPN36					Default	Remap
C5	41	C4	57	91	32	PB5	I/O		PB5	I2C1_SMBAI	TIM3_CH2 / SPI1_MOSI
B5	42	D3	58	92	33	PB6	I/O	FT	PB6	I2C1_SCL[8]/ TIM4_CH1[8]	USART1_TX
A5	43	C3	59	93	34	PB7	I/O	FT	PB7	I2C1_SDA[8]/ TIM4_CH2[8]	USART1_RX
D5	44	B4	60	94	35	BOOT0	I		BOOT0		
B4	45	B3	61	95	-	PB8	I/O	FT	PB8	TIM4_CH3[8]	I2C1_SCL / CANRX
A4	46	A3	62	96	-	PB9	I/O	FT	PB9	TIM4_CH4[8]	I2C1_SDA/ CANTX
D4	-	-	-	97	-	PE0	I/O	FT	PE0	TIM4_ETR	
C4	-	-	-	98	-	PE1	I/O	FT	PE1		
E5	47	D4	63	99	36	V_{SS_3}	S		V_{SS_3}		
F5	48	E4	64	100	1	V_{DD_3}	S		V_{DD_3}		

存储器映射


APB memory space
0xFFFF FFFF
0xE010 0000
0x6000 0000
0x4002 3400
0x4002 3000
0x4002 2400
0x4002 2000
0x4002 1400
0x4002 1000
0x4002 0400
0x4002 0000
0x4001 3C00
0x4001 3800
0x4001 3400
0x4001 3000
0x4001 2C00
0x4001 2800
0x4001 2400
0x4001 1C00
0x4001 1800
0x4001 1400
0x4001 1000
0x4001 0C00
0x4001 0800
0x4001 0400
0x4001 0000
0x4000 7400
0x4000 7000
0x4000 6C00
0x4000 6800
0x4000 6400
0x4000 6000
0x4000 5C00
0x4000 5800
0x4000 5400
0x4000 4C00
0x4000 4800
0x4000 4400
0x4000 3C00
0x4000 3800
0x4000 3400
0x4000 3000
0x4000 2C00
0x4000 2800
0x4000 0C00
0x4000 0800
0x4000 0400
0x4000 0000
reserved
CRC
Flash Interface
RCC
DMA
USART1
SPI1
TIM1
ADC2
ADC1
Port E
Port D
Port C
Port B
Port A
EXTI
AFIO
PWR
BKP
bxCAN
shared 512 byte USB/CAN SRAM
USB Registers
I2C2
I2C1
USART3
USART2
SPI2
IWDG
WWDG
RTC
TIM4
TIM3
TIM2
0xFFFF FFFF
0xE010 0000
0xE000 0000
0xC000 0000
0xA000 0000
0x8000 0000
0x6000 0000
0x4000 0000
0x2000 0000
0x0000 0000
7
6
5
4
3
2
1
0
Cortex- M3 Internal Peripherals
Peripherals
SRAM
0x1FFF FFFF
0x1FFF F80F
0x1FFF F800
0x1FFF F000
0x0801 FFFF
0x0800 0000
0x0000 0000
Option Bytes
System memory
Flash memory
Aliased to Flash or system memory depending on BOOT pins
Reserved
ai14394f

功耗与外设

Symbol	Parameter	Conditions	f_{HCLK}	Max(1)		Unit
				T_A = 85 °C	T_A = 105 °C	
I_{DD}	Supply current in Run mode	External clock(2), all peripherals enabled	72 MHz	50	50.3	mA
			48 MHz	36.1	36.2	
			36 MHz	28.6	28.7	
			24 MHz	19.9	20.1	
			16 MHz	14.7	14.9	
			8 MHz	8.6	8.9	
		External clock(2), all peripherals disabled	72 MHz	32.8	32.9	
			48 MHz	24.4	24.5	
			36 MHz	19.8	19.9	
			24 MHz	13.9	14.2	
			16 MHz	10.7	11	
			8 MHz	6.8	7.1	

Table 19. Peripheral current consumption(1)

Peripheral		Typical consumption at 25 °C	Unit
APB1	TIM2	1.2	mA
	TIM3	1.2	
	TIM4	0.9	
	SPI2	0.2	
	USART2	0.35	
	USART3	0.35	
	I2C1	0.39	
	I2C2	0.39	
	USB	0.65	
	CAN	0.72	
APB2	GPIO A	0.47	mA
	GPIO B	0.47	
	GPIO C	0.47	
	GPIO D	0.47	
	GPIO E	0.47	
	ADC1(2)	1.81	
	ADC2	1.78	
	TIM1	1.6	
	SPI1	0.43	
	USART1	0.85	

1. f_{HCLK} = 72 MHz, $f_{APB1} = f_{HCLK}/2$, $f_{APB2} = f_{HCLK}$, default prescaler value for each peripheral.
2. Specific conditions for ADC: f_{HCLK} = 56 MHz, $f_{APB1} = f_{HCLK}/2$, $f_{APB2} = f_{HCLK}$, $f_{ADCCLK} = f_{APB2/4}$, ADON bit in the ADC_CR2 register is set to 1.