BS32F030XX

入门开发指南

前言

BS32F030 拥有非常多的寄存器，对于新手来说，直接操作寄存器有很大的难度，所以我们提供了一套固件库函数，大家不需要再直接操作繁琐的寄存器，而是直接调用 LL 库函数即可实现操作寄存器的目的。当然，我们要了解一些外设的原理，必须对寄存器有一定的了解，这对以后开发和调试也是非常有帮助的，所以在我们手册中会保留一些重要寄存器的讲解，但是实例代码基本都是调用 LL 库函数来实现的。有关寄存器操作的实例，大家可以参考寄存器的手册及函数底层代码。

BS32F030 系列微控制器采用高性能 32 位 Cortex-M0+内核，内嵌 128KbFlash 和 8KbSRAM，最高工作频率 48MHz。

BS32F030XX 至多 48 个引脚，常用封装类型为 LQFP48。产品提供 7 个 16 位定时器，其中包括 1 个高级控制定时器。产品提供多种标准通讯接口：2 路 UART、2 路 IIC、2 路 SPI。

BS32F030 系列微控制器工作电压为宽电压 2.0V~5.5V，一般工作温度-40℃~+85℃，产品提供多种不同的工作模式，以应对不同情况下的功耗需求。BS32F030 系列微控制器应用于多种场景：医疗和手持设备、PC 游戏外设和 GPS 平台、警报系统、视频对讲和暖气通风空调系统等。

本手册应结合 BS32F030XX 系列芯片数据手册、参考手册使用。

目录

前言 I
第一章 BS32F030 MCU 开发环境搭建 1
1. 1 使用 Keil 开发 BS32F030 1
1.1.1 在 Keil5 中添加 BS32F030 MCU Device 2
1.1.2 新建基于固件库的 keil5 工程模板 3
1.1.3 魔术棒选项卡配置 6
1.2 使用 IAR 开发 BS32F030 7
第二章 BS32F030 MCU 烧录说明 10
2.1 JTAG/SWD 调试接口烧录 10
2.1.1 连接 BS32 11
2.1.2 Keil uVision 5 配置 12
2.1.3 Keil uVision 5 使用 14
2.2 IAP 代码烧录 15
2.2.1 应用代码地址配置 15
2.2.2 .axf 文件转.bin 文件 15
第三章 BS32F030 开发基础知识 17
3.1 固件库使用方法与启动流程介绍 17
3.2 C 语言基础知识 19
3.2.1 位操作 19
3.2.2 define 宏定义 20
3.2.3 ifdef 条件编译 20
3.2.4 extern 变量声明 20
3.2.5 typedef 类型别名 21
3.2.6 结构体 22
3.2.7 断言 23
3.2.8 Doxygen 注释规范 24
3.2.9 防止头文件重复包含 24
3.3 启动文件详解 26
3.3.1 ARM 汇编指令 26
3.3.2 启动文件代码 27
3.4 存储器映射 33
3.5 寄存器映射 35
3.5.1 BS32 的外设地址映射 35
3.5.2 C 语言对寄存器的封装 37
3.6 通信概念 40
3.6.1 串行通讯与并行通讯 40
3.6.2 全双工、半双工及单工通讯 40
3.6.3 同步通讯与异步通讯 40
3.6.4 通讯速率 41
第四章 BS32F030 MCU 常见外设介绍 42
4.1 BS32F0xx 中断应用 42
4.1.1 中断类型 42

4.1.2 NVIC 介绍 43
4.1.3 中断编程 45
4.3.4 EXTI 中断 45
4.2 RCC 时钟应用 48
4.1.1 时钟结构 48
4.1.2 RCC 库函数说明 50
4.3 GPIO 模块介绍 54
4.3.1 GPIO 框图分析 54
4.3.2 GPIO 工作模式 57
4.3.3 GPIO 输出编程要点（点亮 LED） 58
4.3.4 GPIO 输入编程要点（按键检测） 58
4.4 UART 模块介绍 59
4.4.1 串口通信 59
4.4.2 BS32 的 UART 60
4.4.3 硬件流控制功能（RTS、CTS 和 RS485） 65
4.4.4 UART 初始化结构体详解 66
4.4.5 UART 编程要点（UART 接发通信） 67
4.5 DMA 模块介绍 68
4.5.1 DMA 结构说明 68
4.5.2 DMA 数据配置 70
4.5.3 DMA 初始化结构体详解 71
4.5.4 DMA 传输模式 73
4.5.5 DMA 应用流程与注意事项 83
4.6 IIC 模块介绍 85
4.6.1 IIC 协议概述 85
4.6.2 BS32 的 IIC 89
4.6.3 BS32 IIC 通讯过程 95
4.6.4 IIC 初始化结构体详解 96
4.6.5 IIC 编程要点（读写 EEPROM） 97
4.6.6 IIC 通信说明 100
4.6.7 IIC 通讯配置与注意事项 106
4.7 SPI 模块介绍 110
4.7.1 SPI 协议概述 110
4.7.2 BS32 的 SPI 114
4.7.3 BS32 SPI 通讯过程 115
4.7.4 SPI 初始化结构体详解 119
4.7.5 SPI 编程要点（读写串行 FLASH） 121
4.7.6 SPI 应用流程举例 121
4.8 ADC 模块介绍 125
4.8.1 ADC 功能解析 125
4.8.2 ADC 初始化结构体详解 127
4.8.3 BS32 ADC 功能概述 129
4.8.4 ADC 采集实验（单通道） 132

4.9 TIMER 模块介绍 134
4.9.1 时钟/触发控制器 135
4.9.2 时基单元 139
4.9.3 输入捕获 142
4.9.4 输出比较 144
4.9.5 刹车功能 147
4.9.6 输入捕获应用 150
4.6.7 输出比较应用 151
4.6.8 电机功能应用 155
4.6.9 发生外部事件时清除 OCxREF 信号 158
4.6.10 定时器输入异或功能 159
4.6.11 DMA 连续传送模式 159
4.6.12 定时器初始化结构体详解 160
4.10 IWDG 模块介绍 164
4.10.1 IWDG 功能详解 164
4.10.2 IWDG 的使用 165
4.11 WWDG 模块介绍 166
4.11.1 WWDG 功能详解 166
4.11.2 WWDG 的使用 166
4.12 PMU 模块介绍 168
4.12.1 PMU 模块功能 168
4.12.2 电源监控器 168
4.12.3 BS32 的电源系统 169
4.12.4 不同电源模式中断唤醒 171
4.12.5 PMU 应用流程 172
4.13 FMC 模块介绍 173
4.13.1 内部 FLASH 的构成 173
4.13.2 对内部 FLASH 的功能操作 174
4.13.3 System block（NVR 第一页）说明 179
4.14 RTC 模块介绍 181
4.14.1 RTC 功能说明 181
4.14.2 RTC 应用流程 184
4.14.3 RTC 使用注意事项 185
4.15 CRC 模块介绍 186
4.15.1 CRC 功能说明 186
4.15.2 CRC 应用建议 187
附件 188
帮助文件目录 188
版本历史 189

第一章 BS32F030 MCU 开发环境搭建

BS32F030 MCU 为通用型 32 位 MCU，所以开发环境也可以使用通用型的 IDE，目前使用较多的是 KEIL，IAR 等，客户可以根据个人喜好来选择相应的开发环境，但是更推荐使用 keil，其有如下优势：

- BS32F030XX 架构为 Arm® 32-bit Cortex®-M0+，keil 的编译器对 ARM 内核支持相对较好
- 可以向导化生成启动代码，方便上手
- 资源丰富便于学习交流等

下面介绍这两种开发环境的搭建：

1. 1 使用 Keil 开发 BS32F030

MDK 源自德国的 KEIL 公司，目前最新版本为：MDK5.36，该版本使用 uVision5 IDE 集成开发环境，是目前针对 ARM 处理器，尤其是 Cortex M 内核处理器的最佳开发工具。

MDK5 由两个部分组成：MDK Core 和 Software Packs：

MDK Core 又分成四个部分：uVision IDE with Editor（编辑器），ARM C/C++ Compiler（编译器），Pack Installer（包安装器），uVision Debugger with Trace（调试跟踪器）。uVision IDE 从 MDK4.7 版本开始就加入了代码提示功能和语法动态检测等实用功能，相对于以往的 IDE 改进很大。

Software Packs（包安装器）又分为：Device（芯片支持），CMSIS（ARM Cortex 微控制器软件接口标准）和 Mdidleware（中间库）三个小部分，通过包安装器，我们可以安装最新的组件，从而支持新的器件、提供新的设备驱动库以及最新例程等，加速产品开发进度。

MDK5 安装包可以在：http://www.keil.com/demo/eval/arm.htm 下载到。而器件支持、设备驱动、CMSIS 等组件，则可以点击 MDK5 的 Build Toolbar 的最后一个图标调出 Pack Installer，来进行各种组件的安装。也可以在 http://www.keil.com/dd2/pack 这个地址下载，然后进行安装。

在 MDK5 安装完成后，要让 MDK5 支持 BS32F030 的开发，我们还需要安装 BS32F0 的器件支持包：Keil.BS32F0xx_DFP.1.0.5.pack（BS32F0 的器件包）。这个包以及 MDK5.36 安装软件，可以在我们的软件安装文件目录下找到。

Keil 安装流程：

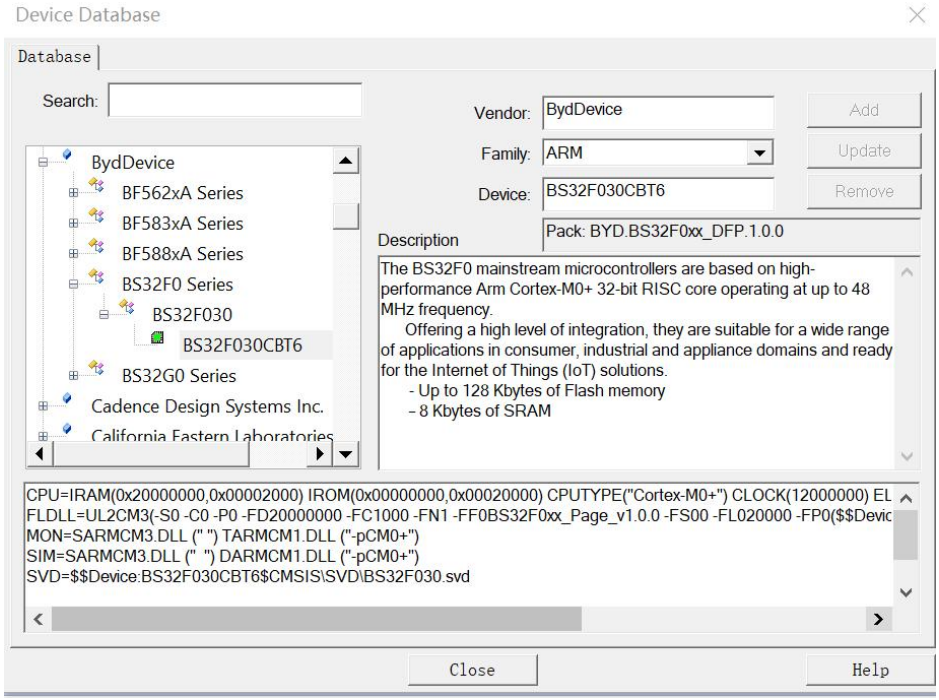
1. 下载 keil5 的解压包，点击运行 mdk514.exe 文件
2. 在弹出的界面，点击 Next（下一步）
3. 选中同意软件使用条约，点击 Next（下一步）
4. 选择安装路径（需在英文目录下），点击点击 Next（下一步）

5. 任意填写用户名与邮箱，点击点击 Next（下一步）

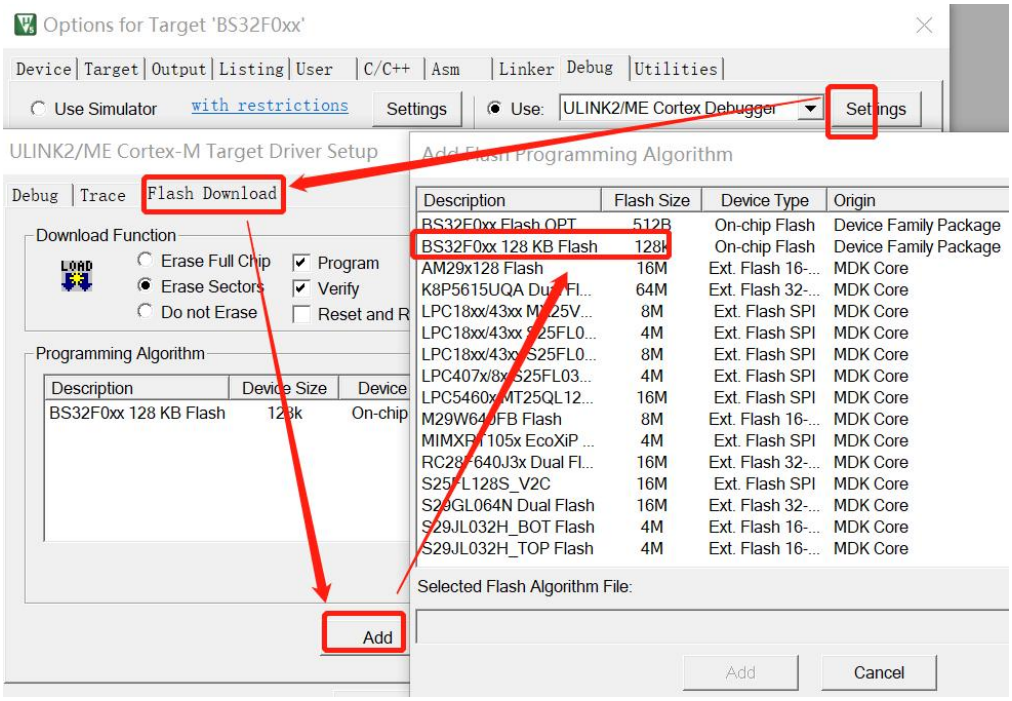
6. 等待安装进度条完成，点击 finish 完成安装

1.1.1 在 Keil5 中添加 BS32F030 MCU Device

双击安装 BS32F0 的器件支持包：BYD.BS32F0xx_DFP.1.0.0，安装完成后可以在 File->Device Database 中出现 BydDevice 的下拉选项，点击可以查看到相应的型号。



在 Options for Target->Debug->Settings->Flash Download 中添加 flash 算法，会出现 BS32F0XX 的算法，即说明安装成功。根据相应的芯片选择合适的算法，即可下载仿真。



1.1.2 新建基于固件库的 keil5 工程模板

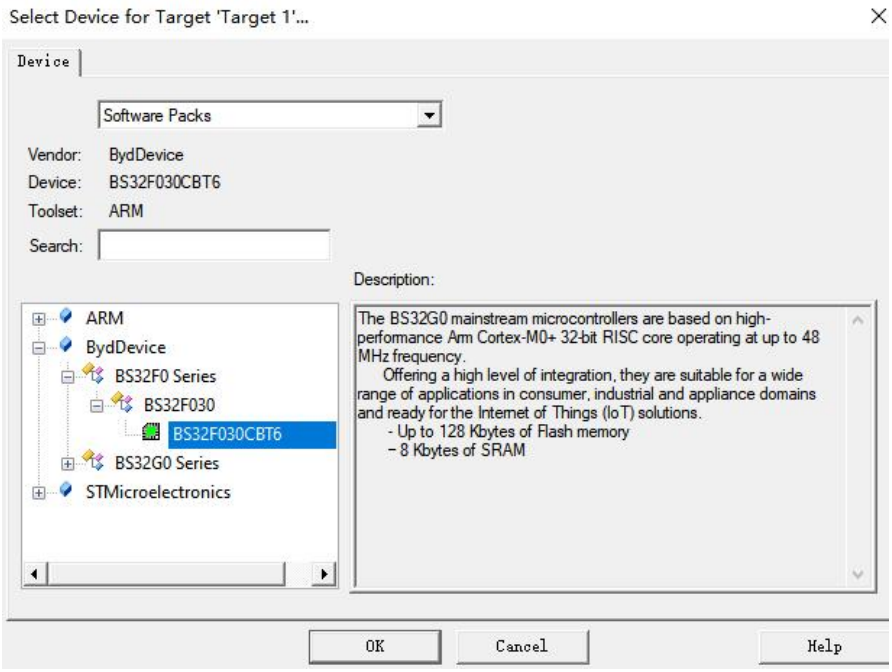
在建立工程模板时，要新建几个文件夹，其中文件夹的名称和含义是：（这里是推荐示例）

- USER 用来存放工程文件和用户代码，包括主函数 main.c，和用户自己写的一些.c 文件
- BSlib 用来存放 BS32 库里面的 inc 和 src 这两个文件，这两个文件包含了芯片上的所有驱动。
- CMSIS 用来存放库为我们自带的启动文件和一些 M0+系列通用的文件。这里就包含了 Core 文件中的内容，即核内外设访问层。
- Output 用来保存软件编译后输出的文件。
- Listing 用来存放一些编译过程中产生的文件。

下面开始讲解使用 keil 新建 BS32 工程的流程：

我们可以通过新建一个名为 Template 的工程来作为之后开发应用的模板。点击 MDK 的菜单：Project–>New Uvision Project，然后将目录定位到刚才建立的文件夹 Template 之下，在这个目录下面建立子文件夹 USER(我们的代码工程文件都是放在 USER 目录，很多人喜欢新建“Project”目录放在下面，这个就看个人喜好了),然后定位到 USER 目录下面，我们的工程文件就都保存到 USER 文件夹下面。工程命名为 Template，点击保存。

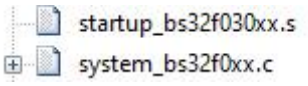
接下来会出现一个选择 MCU 的界面，就是选择使用的芯片型号：BS32F030XX。特别注意：一定要安装对应的器件 pack 后才会显示这些内容。



点击 OK，MDK 会弹出 Manage Run-Time Environment 对话框，这是 MDK5 新增的一个功能，在这个界面，我们可以添加自己需要的组件，从而方便构建开发环境，这里我们不多做介绍，直接点击 Cancel 即可。

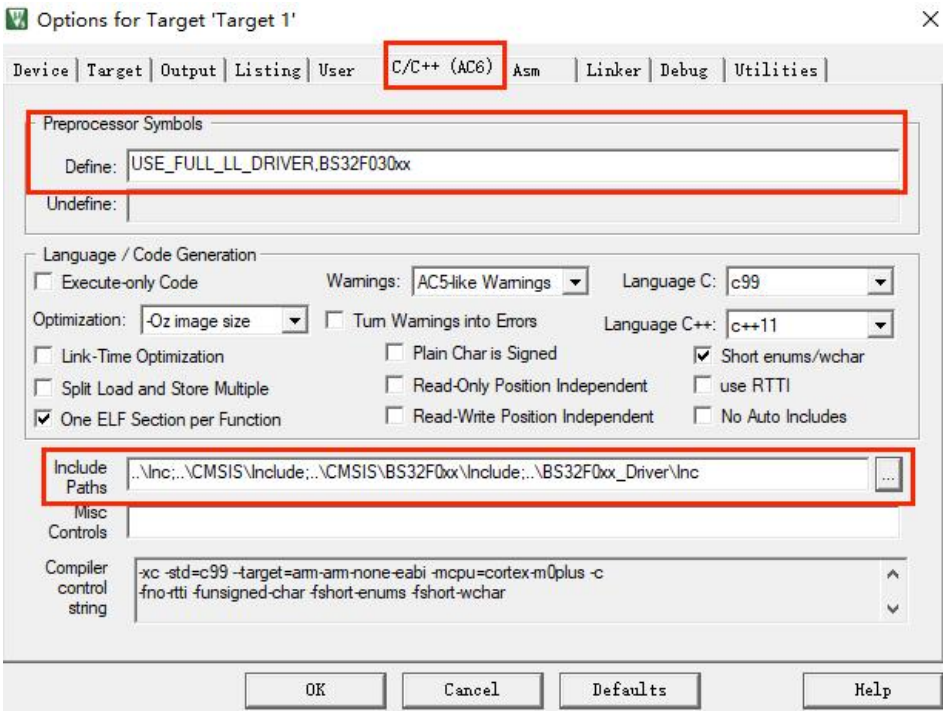
到这里，我们还只是建了一个框架，还需要添加启动文件、系统文件以及各类外设的驱动.c 文件等，用户可以在工程模板文件夹中找到并将这些文件添加到自己的工程中节约开发时间。详细步骤如下：

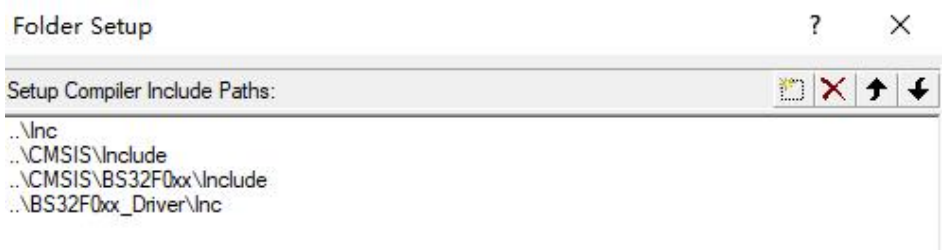
首先在软件支持包中 Template 模板文件夹中找到 startup_bs32f030xx.s 与 system_bs32f0xx.c 两个文件拷贝到新建的工程文件夹中，并通过右键添加文件到对应的 Group 下。



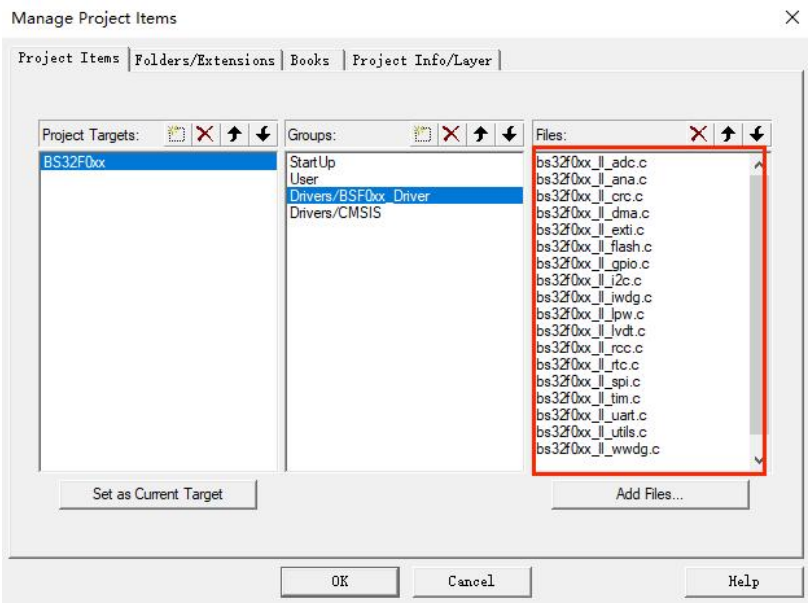
接下来复制软件支持包中 Drivers 文件夹下 BS32F0xx_Driver 中的 Inc、CMSIS 中的 Include、BS32F0xx 中的 Include 以及 Template 中 Inc 文件夹到新建工程中，并在 keil 的 include Paths 选择这些新建工程中的文件夹。

然后我们在 Define 栏中输入对应的预定义内容：USE_FULL_LL_DRIVER,BS32F030xx





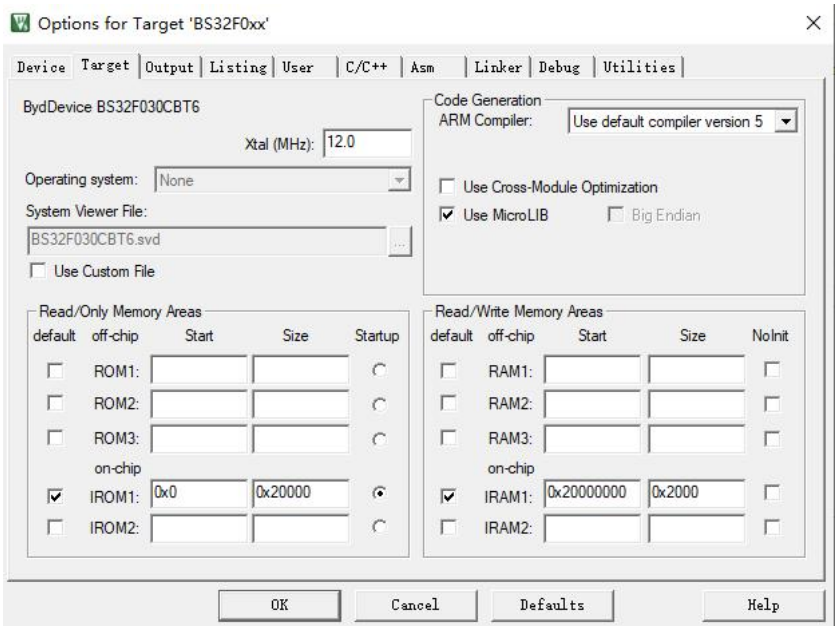
最后，根据我们的开发需要添加需要使用的对应外设.c 文件即可。



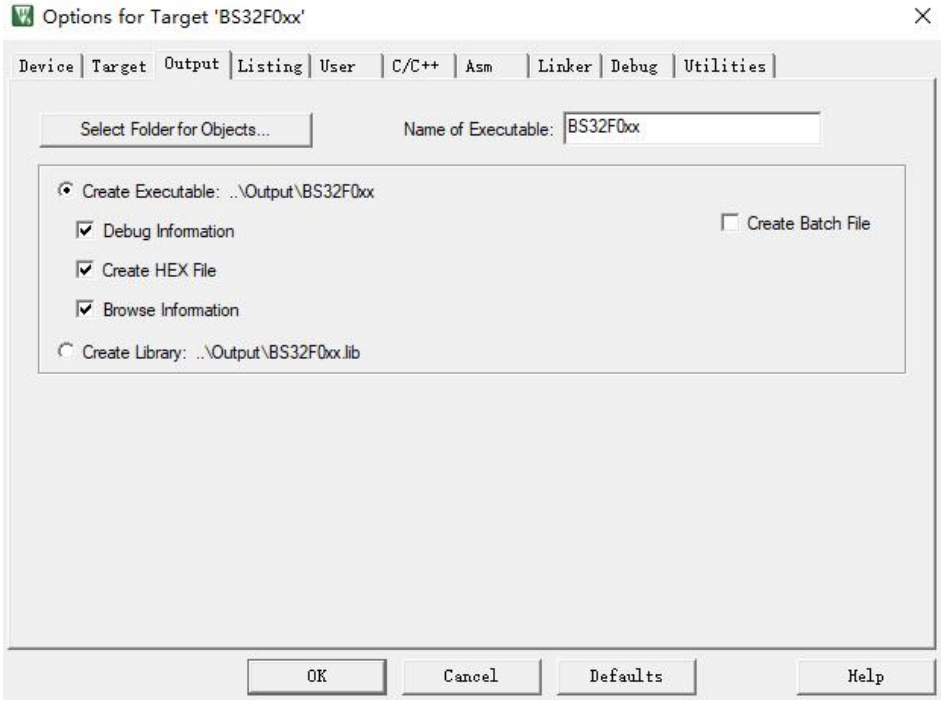
1.1.3 魔术棒选项卡配置

这一步的配置工作很重要，很多人串口用不了 printf 函数，编译有问题，下载有问题，都是这个步骤的配置出了错。

1.Target 中选中微库“Use MicroLib”,为的是之后编写串口驱动的时候可以使用 printf 函数。

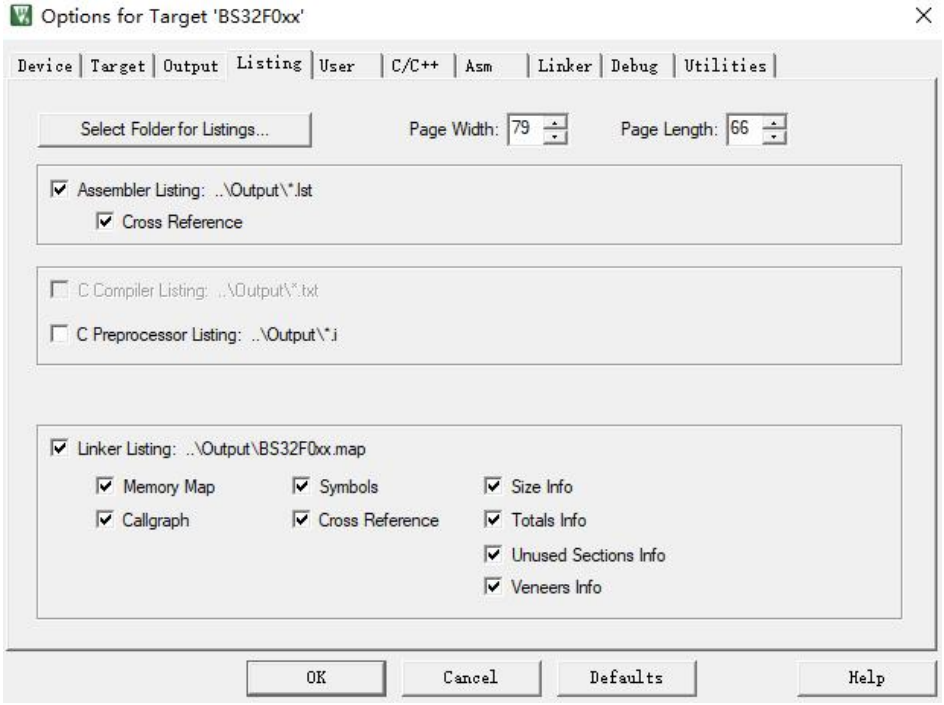


Output 选项卡中把输出文件夹定位到我们工程目录下的 output 文件夹，如果想在编译的过程中生成 hex 文件，那么把 Create HEX File 选项勾上。使用 keil 烧录代码时用到的是.axf 文件，而 hex 文件可以在后面可以转成 bin 文件用于 IAP 烧录。



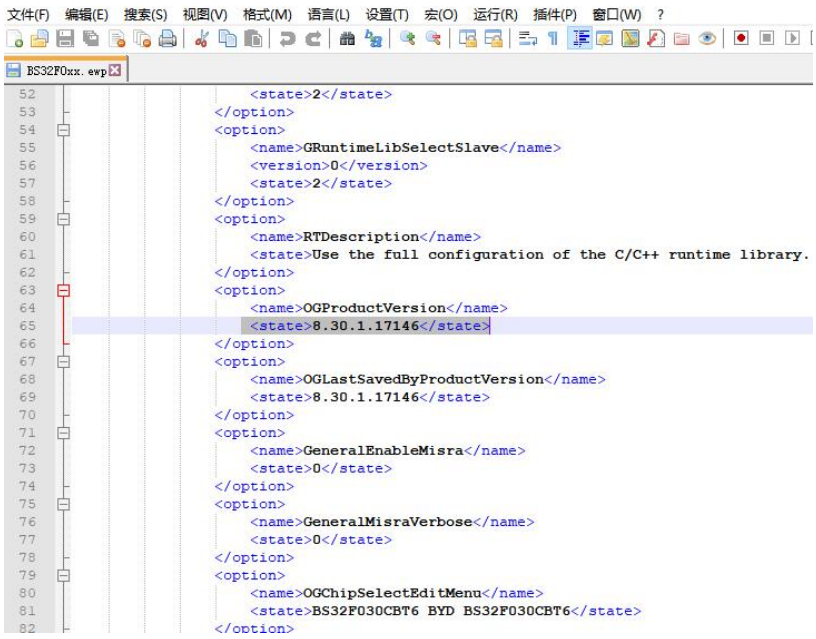
在 Listing 选项卡中把输出文件夹定位到我们工程目录下的 Listing 文件夹，里面主要存放一些中

间过程的映射文件，这一步可以解决 keil 双击工程名无法打开.map 文件的问题。



1.2 使用 IAR 开发 BS32F030

IAR 版本众多，版本之间的兼容性并不好，低版本 IAR 打开高版本 IAR 工程，工程配置会错乱，导致编译报错问题，如果初次使用建议安装最高版本，一般都可以向下兼容，也即是说：高版本的 IDE 可以打开低版本的软件工程。但是出现跨越大版本的情况，也会出现一些细节问题，此时可以通过 EWP 文件查询 IDE 版本信息，下载安装对应版本即可。

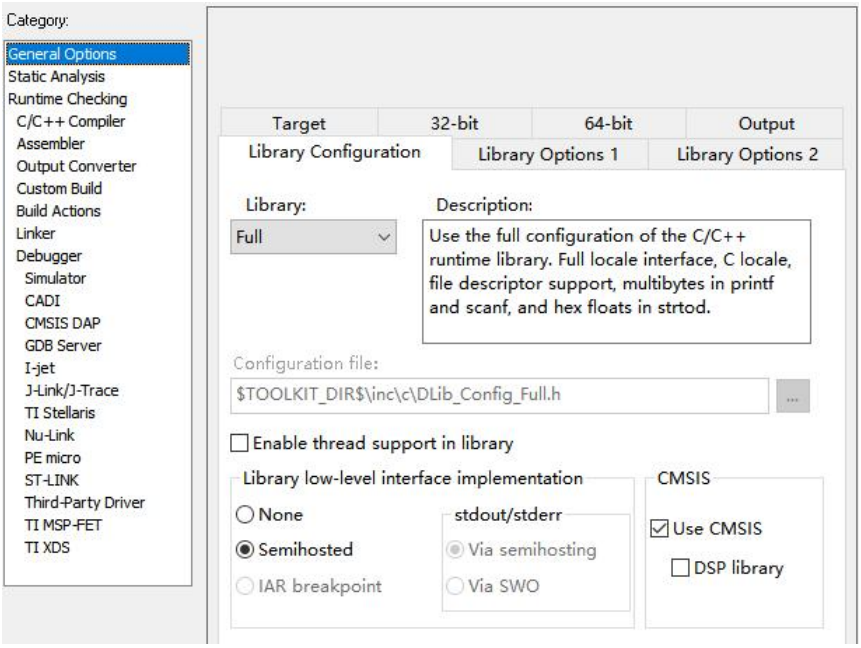


可登录官网地址进行下载或在软件安装支持包中安装：https://www.iar.com/products/architectures/arm/，安装完成后开始下面的步骤：

安装 BS32F030 的 IAR 器件支持包：BS32F0xx_v1.0.0.exe，需安装在正确的 IAR 路径之下，例如：F:\Program Files\IAR Systems\arm\config。

使用 IAR 编译 BS 的 MCU，有两个办法，一种是使用现有的工程进行修改，还有就是重新建立工程，这里就不细说具体工程应该如何建立，BS32F030 的工程建立和别的平台都一致，建立工程时选择 BS32 的相应型号。

以后的 IAR 不需要添加 CMSIS 文件（core_cm4.c 和 core_cm4.h），但是需要勾选 Project->options for node->General Options->Library Configuration 的 Use CMSIS，如果软件代码有使用到 printf 函数，还需要修改 Library 为 FULL。

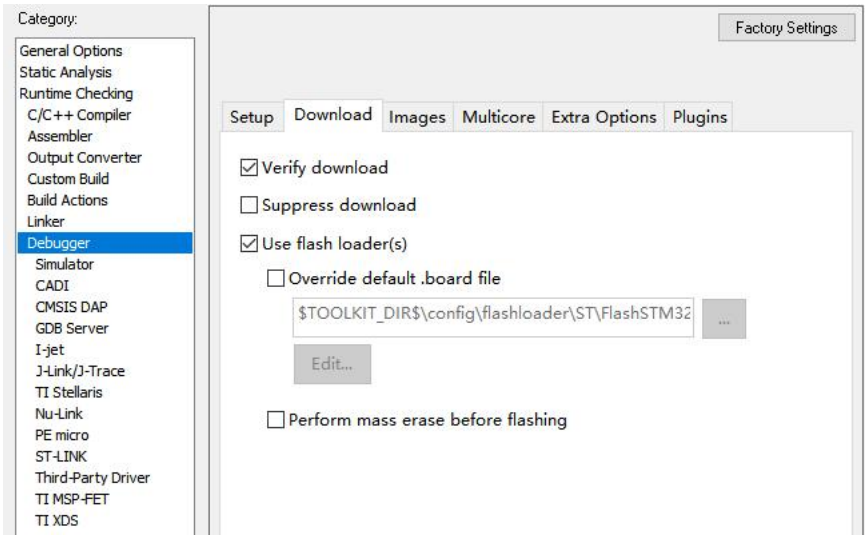


芯片的 Link 文件建立工程时会默认根据型号选定，但是编译前还是要有检查的习惯，检查一下 ICF 文件是否有配置，是否正确。

配置 Debugger->Setup 选项，新建立的工程默认是 Simulator 模拟，如果需要调试那么需要根据实际情况来选择：

- 使用 J-Link 选择 J-Link/J-Trace；
- 使用 ULink2 选择 CMSIS DAP。

配置 Debugger->Download 选项，新建的工程有可能没有配置 download 选项，如果我们需要调试代码那么务必要勾选 User flash loader 选项，且保证 board file 准确，否则程序无法正常下载至芯片内部。



如果选择了 Debugger 选项，那么还需要根据 Debugger 选项设定对应的调试选项；如果选择的是 BS 的型号，在 IAR 下面已经固定将所有的调试接口都配置为 SWD 接口，可以忽略该选项配置，直接进行相关的代码 debug 工作。

第二章 BS32F030 MCU 烧录说明

程序烧录的方式有很多种，根据不同的烧录需求可以分为以下四种：

1.编程器烧写：大规模烧写，芯片先烧写完成再焊接在目标板上。

2.ISP：通过 MCU 片内 BootROM 完成在板 Flash 烧写操作。要求芯片出厂时带有 ISP 固件，一般通过串行方式烧写。

3.IAP：支持用户程序运行时烧写 Flash，但要求 BootLoader 程序已经烧写至 MCU 片内 Flash。

4.调试接口烧写：借助仿真器通过 MCU 的调试接口(如 JTAG/SWD 等)控制 MCU 完成片内 Flash 甚至外扩的 Flash 烧写。

本指南手册中着重介绍 BS32F030 比较常用的使用调试接口 JTAG/SWD 烧录与 IAP 烧录方法。其中使用调试接口 JTAG/SWD 烧录主要用于项目的开发调试过程中，而 IAP 烧录则主要是用于后期用户应用程序升级的需求。

2.1 JTAG/SWD 调试接口烧录

下面我们以常用的含有 JTAG/SWD 调试接口的烧录工具 ULINK 作为该节的烧录示例，相比于只能下载代码，不能实时跟踪调试的串口烧录，ULINK 就可以实时跟踪程序，从而找到你程序中的 bug，使你的开发事半功倍。

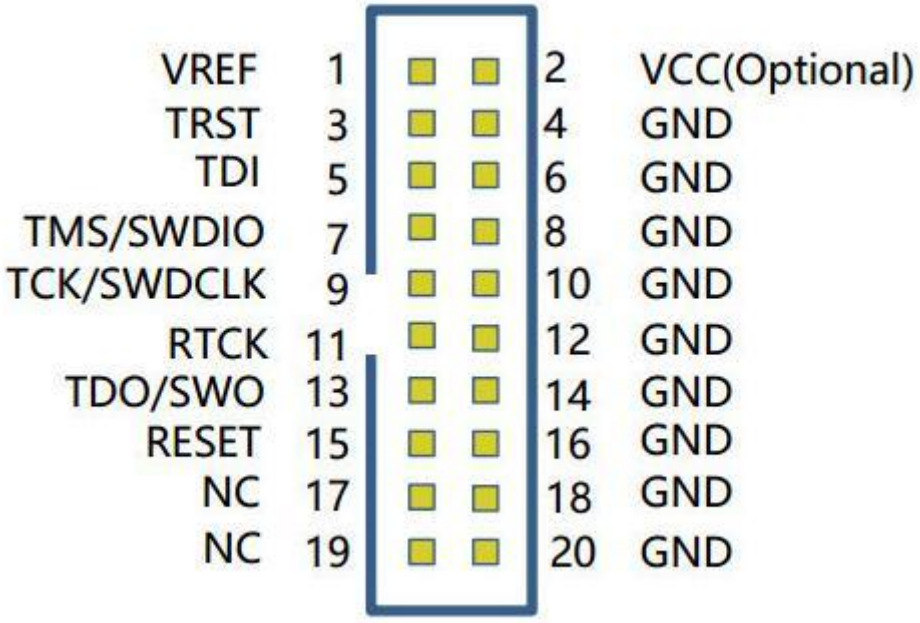
ULink 是用于 32 位微控制器的在线调试器和编程器，也是大家口中的下载器。

注意：开发板如果不供电的话，仅靠仿真器电源供应 VCC 会是 2.9V 左右，在使用 ADC 等外设的时候可能出现采样误差很大的问题。

ULink 具有 JTAG / SWD 通信接口，用于 32 位微控制器进行通信。

两种接口说明：

- JTAG： Joint Test Action Group，联合测试工作组，是一种国际标准测试协议。
- SWD： Serial Wire Debugging，串行调试接口。



通常我们用 20pin 的排线连接的是图中标准的 JTAG 接口，但是 ULink 还有 SWD 接口，SWD 只需要两根线（SWCLK 与 SWDIO）就可以下载并调试代码了（当然最好还是同时连接上 VCC、GND、RESET），这同我们使用串口下载代码差不多，所以 SWD 是一种十分节约资源的调试口，

ULink 上的 SWD 接口与 JTAG 是共用的，只要接上 JTAG，你就可以使用 SWD 模式了（其实并不需要 JTAG 这么多线），当然，你的调试器必须支持 SWD 模式，ULINK2/ME 即支持 SWD 调试。特别提醒，JTAG 有几个信号线用来接其他外设了，但是 SWD 是完全没有接任何其他外设的，所以在使用的时候，推荐使用 SWD 模式。

SWD 和传统的调试方式区别：

1.SWD 模式比 JTAG 在高速模式下面更加可靠。在大数据量的情况下 JTAG 下载程序会失败，但是 SWD 发生的几率会小很多。基本使用 JTAG 仿真模式的情况下是可以直接使用 SWD 模式的，只要你的仿真器支持，所以推荐大家使用这个模式。

2.当 CPU 的 GPIO 不够用的时候，可以使用 SWD 仿真，这种模式支持更少的引脚。

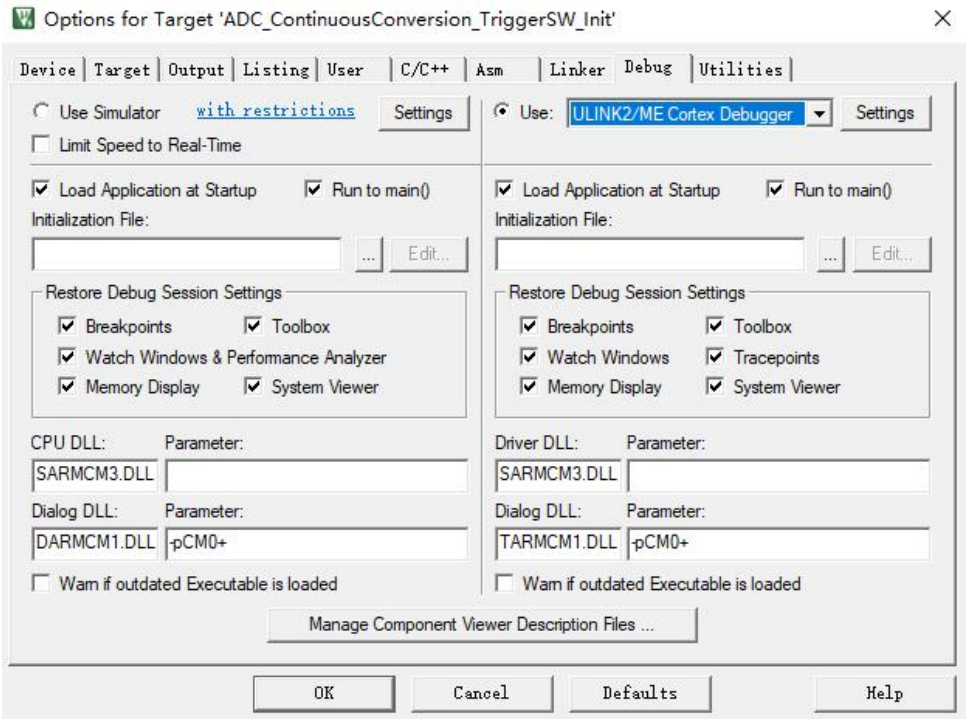
3.在硬件 PCB 的体积有限的时候推荐使用 SWD 模式，它需要的引脚少，当然需要的 PCB 空间就小，可以选择一个很小的 2.54 间距的 5 芯端子做仿真接口。

2.1.1 连接 BS32

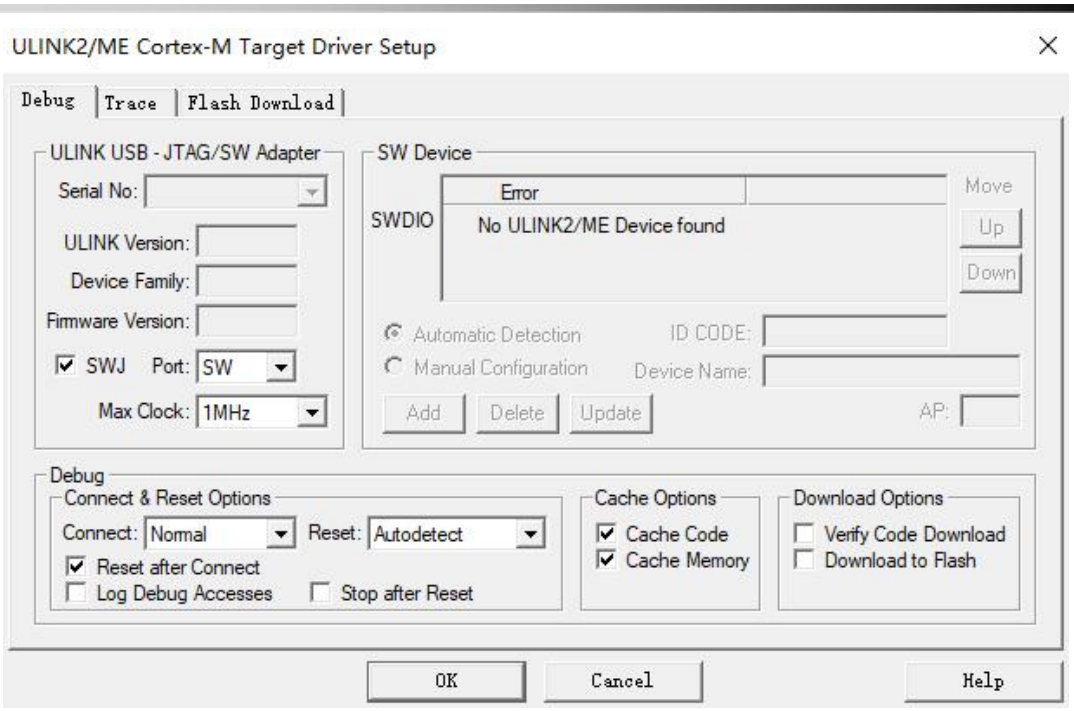
开发 BS32 微控制器应用程序时，需要使用 20 针的 JTAG 排线或使用对应的 SWD 引脚将 ULink 连接至开发板。

2.1.2 Keil uVision 5 配置

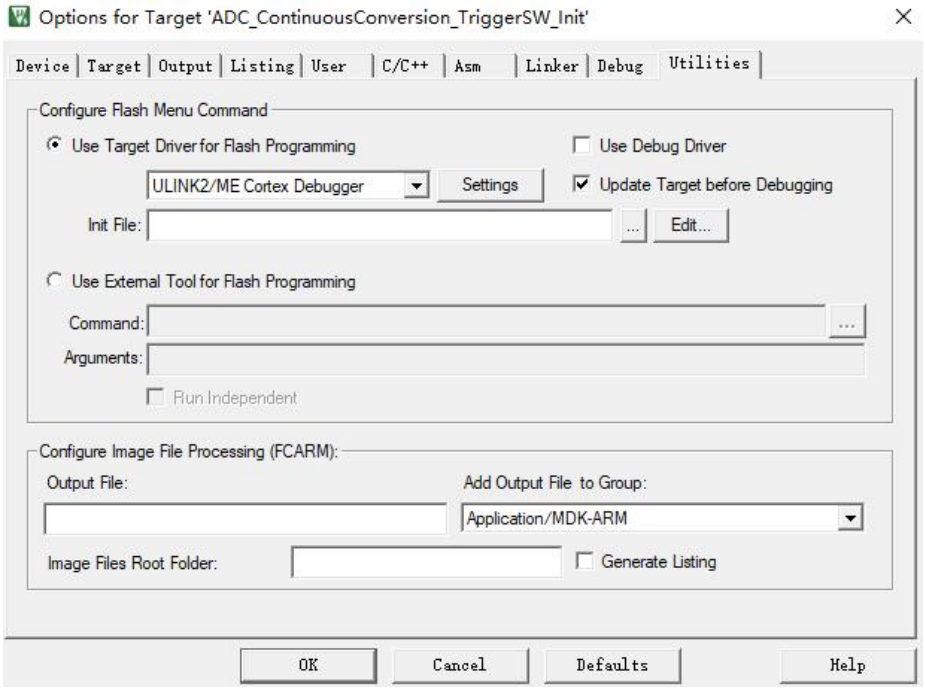
JTAG（联合测试行动小组，Joint Test Action Group）是一种国际标准测试协议，标准 JTAG 接口采用 4 枚针脚，但是目前较为常用的是 14、20 个针脚，当前使用的 ULINK2/ME 也同样采用了该种 20 针接口，这里直接将其插入 BS32 实验电路中的调试插座，然后打开 Keil uVision 5 界面上的【Options for Target】对编译目标进行设置，选择【Debug】选项卡并选中【Use ULink2/ME Cortex Debugger】选项。



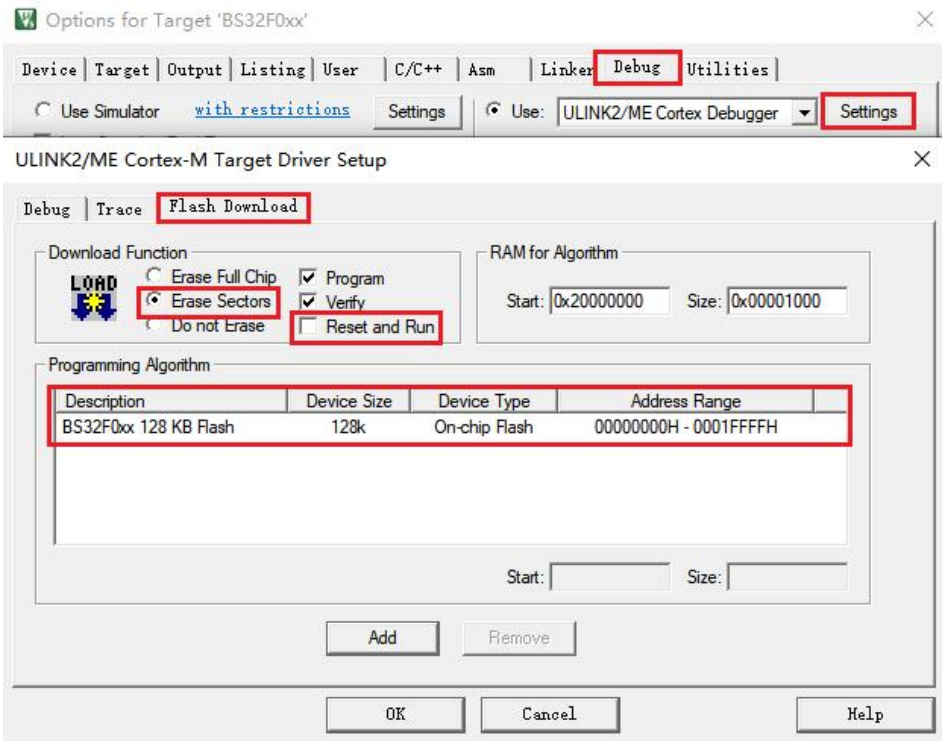
接下来再点击上图界面右上的【Settings】选项，在弹出的【ULINK2/ME Cortex-M Target Driver Setup】设置界面当中，将【Debug】标签下的【Port】选择为 SW，点击【确定】按钮以后退出(此处选择 JTAG 也是可以的，这里选择 SW 主要考虑的是其占用的引脚资源较少，因为 ULINK 上 SWD 与 JTAG 接口是共用的，所以实际开发过程中开发者可根据个人需求自由配置)。



接下来再打开【Options for Target】界面上的【Utilities】选项卡，将【Use Target Driver for Flash Programming】选中为 ULINK2/ME Cortex Debugger ，或者直接勾选【Use Debug Driver】也可实现同样的目标。



最后点击选项菜单后面的【Settings】按钮，重新进入【Cortex-M Target Driver Setup】设置界面，点击【Add】添加合适的 Flash 编程算法，确定后保存退出。



- Erase Sectors：选择 Sectors 擦除，如果选择 Full Chip 擦除则会非常慢。
- Reset and Run：勾选 Reset and Run，则下载完成后程序会自动运行，不用手动复位。
- Programming Algorithm：选择芯片，这个根据实际的型号来选择，如果这里没有选，则下载会提示 Algorithm 错误。

配置好 ULINK 之后，使用 ULINK 下载代码就非常简单，大家只需要点击 LOAD 按钮 就可以进行程序下载。下载完成之后程序就可以直接在开发板执行。

2.1.3 Keil uVision 5 使用

点击 Keil uVision 5 菜单栏上的【Download】按钮，可以通过 ULink 将程序烧写至微控制器的 Flash 当中并且运行。完成下载之后，程序可以脱离 ULink 独立运行 FLASH 当中。

点击 Keil uVision 5 菜单栏上的【Start/Stop Debug】按钮图标，就可以进入仿真模式。

接下来，我们就可以和软件仿真一样的开始操作了，不过这是真正的在硬件上的运行，而不是软件仿真，其结果更可信。

2.2 IAP 代码烧录

IAP 烧录简单来说就是通过调用特定的 bootloader 程序，对程序存储器(一般是片内 flash)的指定段进行读/写操作，从而实现对目标板的程序的修改。其工作原理如下：

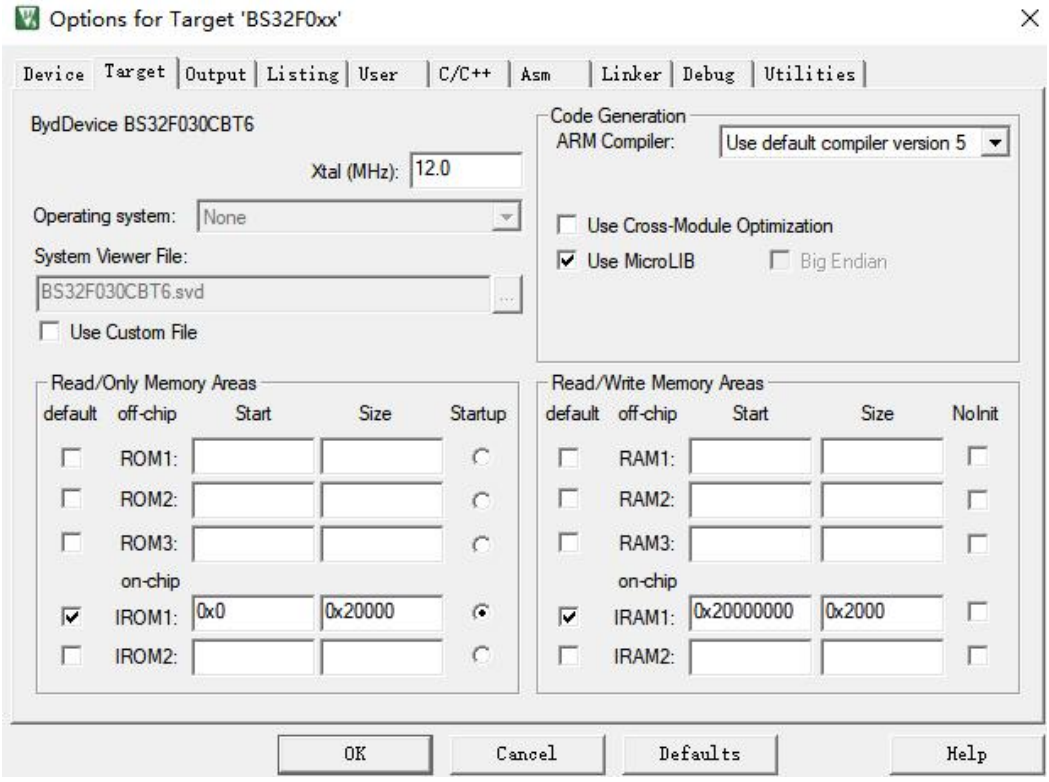
从软件层面上将芯片的程序存储器划分为两个区域，其中区域①使用 ISP 的方式（芯片出厂前已完成）烧录特定的 bootloader 程序，该程序只具备两种与用户应用程序无关的功能:

- 通过某种通信方式(IIC、UART 等)接收程序或数据的功能；
- 对存储器的擦写功能；

区域②则存放真正的项目应用代码，而该区域的代码则是通过区域①进行更改。IAP 烧录方式可以不用拆卸 MCU 模块而通过预留的接口直接进行应用程序烧录更新，多用于后期维护升级。

2.2.1 应用代码地址配置

用户在烧录应用代码时，应注意 flash 的部分地址/区域①已经出厂烧录好 bootloader 程序，应将用户代码烧录配置至区域②的地址：



2.2.2 .axf 文件转.bin 文件

编译好应用代码后 keil 生成.axf 文件，我们需要将.axf 文件转成.bin 文件才能通过烧录工具将代码离线烧录进应用设备之中，使用 BS-link 脱机烧录时无需连接到 PC 端口。

转换可以使用 Keil MDK 里面的 fromelf.exe 工具，位于 keil\ARM\ARMCC\bin 文件夹下，转换

主要分为四个步骤（以下地址为示例）：

- 选择 fromelf.exe 所在的目录：D:\keil\ARM\ARMCC\bin\fromelf.exe;
- 配置 fromelf.exe 的语法选项：--bin -o；
- 选择生成的文件路径及文件名：..\OBJ\ele.bin;
- 选择可执行文件路径及文件名：..\MDK-ARM\ele\ele.axf;

所以整体的语句为：D:\keil\ARM\ARMCC\bin\fromelf.exe --bin -o ..\OBJ\ele.bin ..\MDK-ARM\ele\ele.axf

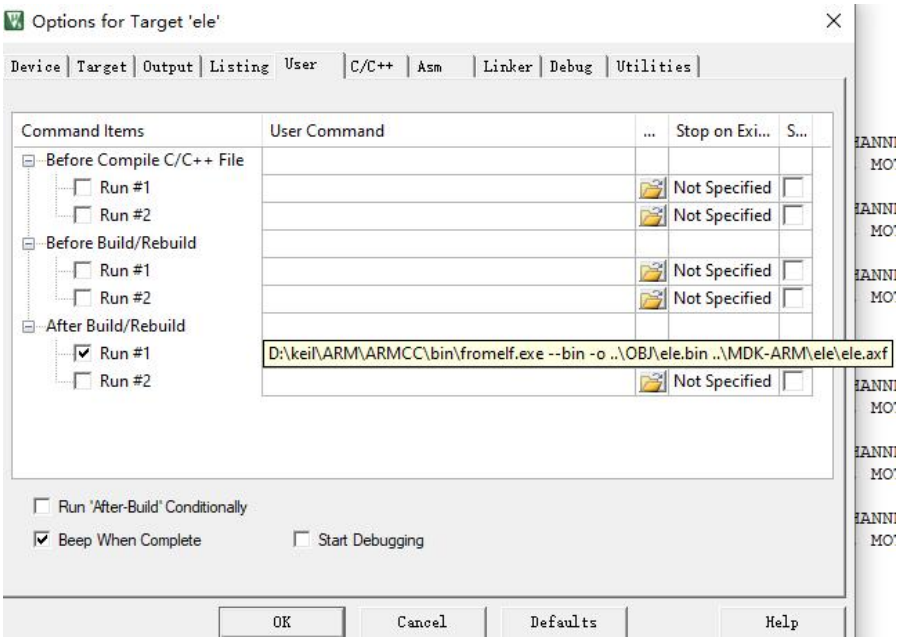
注意：

1.整体语句是填写在 keil 的 Options for target -> User -> After Build/Rebuild -> Run #1 里面，并勾选后点击 OK 重新编译即可在生成文件路径中找到所需.bin 文件;

2.文件的路径：如果是..\表示项目文件所在目录的上一级目录下查找，也可输入完整路径以固定路径形式查找；

3.文件的名字：可执行文件.axf 的名字和路径与项目编译所生成文件的名字与路径一致，即 MDK 在 Options for target -> Output 的设置项-> Name of Executable 与 -> Select Folder for Objects。生成的文件.bin 路径及文件名不作要求；

4.生成的文件与可执行文件两者的语法位置顺序是根据 fromelf.exe 的语法来的，上面的整体语句也可写成：F:\tool\Keil_v5\ARM\ARMCC\bin\fromelf.exe --bin ..\MDK-ARM\ele\ele.axf -o ..\OBJ\ele.bin



第三章 BS32F030 开发基础知识

这一章我们主要介绍一些 BS32F030 开发的一些基础知识，让您对 BS32F030 开发有一个初步的了解。

3.1 固件库使用方法与启动流程介绍

BS32F030 MCU 标准固件库是一个固件函数包，它由程序、数据结构和宏组成，包括了 BS32 MCU 所有外设的性能特征。固件库还包括每一个外设的驱动描述和基于板级的固件库使用例程。通过使用固件库，用户无需深入掌握细节，也可以轻松应用每一个外设。使用固件库可以大大减少用户的编程时间，从而降低开发成本。每个外设驱动都由一组函数组成，这组函数覆盖了该外设所有功能。可以通过调用一组通用 API(application programming interface 应用编程界面)来实现对外设的驱动，这些 API 的结构、函数名称和参数名称都进行了标准化规范。而不同的固件库都要遵循固定的 CMSIS 软件接口标准。即我们在使用 BS32 芯片的时候首先要进行系统初始化，CMSIS 规范就规定，系统初始化函数名字必须为 SystemInit。CMSIS 还对各个外设驱动文件的文件名字规范化，以及函数名字规范化等等一系列规定。

这个时候你不需要再直接去操作寄存器了，你只需要知道怎么使用对应的函数就可以了。在你对外设的工作原理有一定的了解之后，你再去看对应的库函数，基本上函数名字能告诉你这个函数的功能是什么，该怎么使用，这样开发会方便很多。在此以 BS32F0xx 系列固件库为例进行说明固件库架构及使用方法，其他系列固件库可类比参考。

文件夹 Examples，对应每一个 BS32 外设均包含一个子文件夹。 每个子文件夹包含了关于本外设的一个或多个例程，来示范如何使用对应外设。

CMSIS 子文件夹包含有 Cortex M0+内核的支持文件、 基于 Cortex M0+内核处理器的启动代码和库引导文件以及基于 BS32F0xx 的全局头文件和系统配置文件。

Template 文件夹包含一个关于使用 LED、 USART 打印、 按键控制的简单例程，（IAR_project 用于 IAR 编译环境，Keil_project 用于 Keil5 编译环境）。用户可以使用该工程模板进行固件库例程的移植编译，具体使用方法见下：

选择文件

打开“Examples”文件夹，选择需要测试的模块，如 SPI，打开”SPI”文件夹，选择 SPI 的一个例程，如”SPI_master_transmit_slave_receive_interrupt”

拷贝文件

打开“Template”文件夹，将”IAR_project”和”Keil_project”两个文件夹保留，其他文件都删除，然后将“SPI_master_transmit_slave_receive_interrupt”文件夹中的所有文件拷到“Template”文件夹子目录下

打开工程

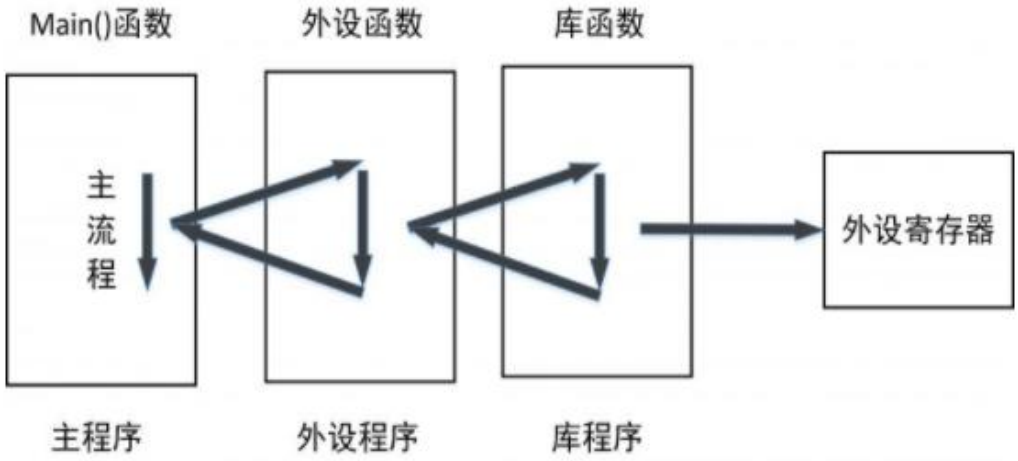
BS 提供 Keil 和 IAR 两种版本的工程，根据客户所安装的软件，打开不同的 project，如”Keil_project”，打开\Template\Keil_project\Project.uvproj

编译调试下载

首先编译整个工程，如果无错误，按照 readme 中的介绍，选择正确的跳线及连线，然后再将程序下载到目标板上，则会有如 readme 中描述的现象。IDE 的具体使用，请参考相应的本手册章节与软件使用说明。

任何处理器，不管它有多么的高级，归根结底都是要对处理器的寄存器进行操作。但是固件库不是万能的，您如果想要把 BS32F030 学透，光读 BS32F030 固件库是远远不够的。你还是要了解一下 BS32F030 的原理，了解 BS32F030 各个外设的运行机制。所以在学习库函数的同时，要了解一下寄存器大致配置过程。

文件名	描述
main.c	主函数体示例。
BS32F030_it.h	头文件，包含所有中断处理函数原形。
BS32F030_it.c	外设中断函数文件。用户可以加入自己的中断程序代码。对于指向同一个中断向量的多个不同中断请求，可以利用函数通过判断外设的中断标志位来确定准确的中断源。固件库提供了这些函数的名称。
BS32F030_xxx.h	外设 xxx 的头文件。包含外设 xxx 函数的定义，以及这些函数使用的变量。
BS32F030_xxx.c	由 C 语言编写的外设 xxx 的驱动源程序文件。
readme.txt	固件库例程使用及配置说明文档。



应用程序的调用过程示意

3.2 C 语言基础知识

本节将介绍一些常用的 C 语言基础知识，引导基础不是很扎实的用户快速熟悉开发 BS32F030 的程序。

3.2.1 位操作

位操作就是对基本类型变量在位级别进行操作，C 语言支持如下 6 种位操作：

运算符	含义	运算符	含义
&	按位与	~	取反
|	按位或	<<	左移
^	按位异或	>>	右移

这些是 MCU 开发的常用操作符，下面将介绍这些位操作在单片机开发中的实用技巧。

1) 不改变其他位的值的情况下，对某几个位进行设值。

这个场景 MCU 开发中经常使用，方法就是先对需要设置的位用&操作符进行清零操作， 然后用|操作符设值。比如我要改变 GPIOA 的状态,可以先对寄存器对应位的值进行&清零操作

```
GPIOA->CRL&=0XFFFFFF0F;     //将第 4-7 位清 0
```

然后再对需要设置的位进行|或运算即可置值对应位

```
GPIOA->CRL|=0X00000040;     //设置第六位的值，不改变其他位的值
```

2) 移位操作提高代码的可读性

移位操作在 MCU 开发中也非常重要，下面让我们看看固件库的 GPIO 初始化的函数里面的一行代码

```
GPIOx->BSRR = (((uint32_t)0x01) << pinpos);
```

这个操作就是将 BSRR 寄存器的第 pinpos 位设置为 1，为什么要通过左移而不是直接设置一个固定的值，是为了提高代码的可读性以及可重用性。这行代码可以很直观明了的知道，是将第 pinpos 位设置为 1。如果写成

```
GPIOx->BSRR =0x0030;
```

这样的格式可读性差也不好重用，类似的代码有很多：

```
GPIOA->ODR|=1<<5;  //PA.5 输出高,不改变其他位
```

这样我们一目了然，5 告诉我们是第 5 位也就是第 6 个端口，1 告诉我们是设置为 1 了。

3) 取反操作使用技巧

SR 寄存器的每一位都代表一个状态，某个时刻我们希望去设置某一位的值为 0，同时其他位都保留为 1，简单的作法是直接给寄存器设置一个值：

```
TIMx->SR=0xFFF7;
```

这样的作法设置第 3 位为 0，但是这样的作法同样不好看，并且可读性很差。看看库函数代码中怎样使用的：

```
TIMx->SR = (uint16_t)~TIM_FLAG;
```

而 TIM_FLAG 是通过宏定义定义的值：

```
#define TIM_FLAG_Update         ((uint16_t)0x0001)
#define TIM_FLAG_CC1            ((uint16_t)0x0002)
```

看这个应该很容易明白，可以直接从宏定义中看出 TIM_FLAG_Update 就是设置的第 0 位了，可读性非常强。

3.2.2 define 宏定义

define 是 C 语言中的预处理命令，它用于宏定义，可以提高源代码的可读性，为编程提供方便。常见的格式：

```
#define 标识符 字符串
```

“标识符”为所定义的宏名。“字符串”可以是常数、表达式、格式串等。例如：

```
#define SYSCLK_FREQ_32MHz 32000000
```

定义标识符 SYSCLK_FREQ_32MHz 的值为 32000000。

3.2.3 ifdef 条件编译

单片机程序开发过程中，经常会遇到一种情况，当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。条件编译命令最常见的形式为：

```
#ifdef 标识符
程序段 1
#else
程序段 2
#endif
```

它的作用是：当标识符已经被定义过(一般是用#define 命令定义)，则对程序段 1 进行编译，否则编译程序段 2。其中#else 部分也可以没有，即：

```
#ifdef
程序段 1
#endif
```

这个条件编译在 MDK 里面是用得很多的，在 bs32g0xx.h 这个头文件中经常会看到这样的语句。

3.2.4 extern 变量声明

C 语言中 extern 可以置于变量或者函数前，以表示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。这里面要注意，对于 extern 申明变量可以多次，但定义只有一次。在编程过程中会经常看到这样的语句：

```
extern u16 USART_RX_STA;
```

这个语句是申明 USART_RX_STA 变量在其他文件中已经定义了，在这里要使用到。所以，你

肯定可以找到在某个地方有变量定义的语句：

```
u16 USART_RX_STA;
```

下面通过一个例子说明一下使用方法。在 main.c 定义的全局变量 id，id 的初始化都是在 main.c 里面进行的。

main.c 文件

```
u8 id;          //定义只允许一次
main()
{
id=1;
printf("%d",id);//id=1
test();
printf("%d",id);//id=2
}
```

但是我们希望在 test.c 的 changeId(void)函数中使用变量 id,这个时候我们就需要在 test.c 里面去申明变量 id 是外部定义的了，因为如果不申明，变量 id 的作用域是到不了 test.c 文件中。看下面 test.c 中的代码：

```
extern u8 id;     //申明变量 id 是在外部定义的，申明可以在很多个文件中进行
void test(void){
id=2;
}
```

在 test.c 中申明变量 id 在外部定义，然后在 test.c 中就可以使用变量 id 了。

3.2.5 typedef 类型别名

typedef 用于为现有类型创建一个新的名字，或称为类型别名，用来简化变量的定义。typedef 在 MDK 用得最多的就是定义结构体的类型别名和枚举类型了。我们可以为结构体定义一个别名 GPIO_TypeDef，这样我们就可以在其他地方通过别名 GPIO_TypeDef 来定义结构体变量了。方法如下：

```
typedef struct
{
 __IO uint32_t MODER;
 __IO uint32_t OTYPER;
 …
} GPIO_TypeDef;
```

Typedef 为结构体定义一个别名 GPIO_TypeDef，这样我们可以通过 GPIO_TypeDef 来定义结构体变量：

```
GPIO_TypeDef _GPIOA,_GPIOB;
```

3.2.6 结构体

MDK 中用到结构体与结构体指针的地方很多。声明结构体类型：

```
Struct 结构体名{
成员列表;
}变量名列表;
```

例如：

```
Struct Uart_TYPE {
Int BaudRate;
Int DataWidth;
}uart1,uart2;
```

在结构体申明的时候可以定义变量，也可以申明之后定义，方法是：

```
Struct 结构体名字 结构体变量列表;
```

例如：

```
struct Uart_TYPE uart1,uart2;
```

结构体成员变量的引用方法是：

```
结构体变量名.成员名
```

比如要引用 uart1 的成员 BaudRate，方法是：uart1.BaudRate;

结构体指针变量定义也是一样的，跟其他变量没有区别。

例如：struct Uart_TYPE *usart1；//定义结构体指针变量 uart1;

结构体指针成员变量引用方法是通过“->”符号实现，比如要访问 uart1 结构体指针指向的结构体的成员变量 BaudRate,方法是：

```
Uart1->BaudRate;
```

上面讲解了结构体和结构体指针的一些知识，下面我们通过实例来演示结构体的作用。

在我们单片机程序开发过程中，经常会遇到要初始化一个外设比如串口，它的初始化状态是由几个属性来决定的，比如串口号，波特率，极性，以及模式。对于这种情况，在我们没有学习结构体的时候，我们一般的方法是：

```
void UART_Init(u8 usartx,u32 BaudRate,u8 parity,u8 mode);
```

这种方式在一定场合是可取的。但是如果有一天，我们希望往这个函数里面再传入一个参数，那么势必我们需要修改这个函数的定义，重新加入字长这个入口参数。于是我们的定义被修改为：

```
void UART_Init (u8 usartx,u32 BaudRate, u8 parity,u8 mode,u8 DataWidth);
```

但是如果这个函数的入口参数随着开发不断的增多，那么我们就要不断的修改函数的定义，这给我们开发带来很多的麻烦，这样我们使用结构体就能解决这个问题了。我们可以在不改变入口参数的情况下，只需要改变结构体的成员变量，就可以达到上面改变入口参数的目的。

结构体就是将多个变量组合为一个有机的整体。上面的函数里，BaudRate,DataWidth,Parity 这些参数，他们对于串口而言，是一个有机整体，都是来设置串口参数的，所以我们可以将他们通

过定义一个结构体来组合在一个。MDK 中是这样定义的：

```
typedef struct
{
    uint32_t BaudRate;
    uint16_t DataWidth;
    uint16_t StopBits;
    uint16_t Parity;
    uint16_t TransferDirection;
    uint16_t HardwareFlowControl;
} LL_UART_InitTypeDef;
```

于是，我们在初始化串口的时候入口参数就可以是 UART_InitTypeDef 类型的变量或者指针变量了，MDK 中是这样做的：

```
void UART_Init(UART_TypeDef* UARTx, UART_InitTypeDef* UART_InitStruct);
```

这样，我们只需要修改结构体成员变量，往结构体中间加入新的成员变量，而不需要修改函数定义就可以达到修改入口参数同样的目的了。这样的好处是不用修改任何函数定义就可以达到增加变量的目的。

使用结构体组合参数，可以提高代码的可读性，不会觉得变量定义混乱。当然，MDK 中用结构体来定义外设也不仅仅只是这个作用，后面在开发过程中逐步接触。

3.2.7 断言

```
assert_param(IS_LL_DMA_MODE(DMA_InitStruct->Mode));
```

基本上每个库函数的开头都会有这样类似的内容，此处的“assert_param”实际是一个宏，在库函数中它用于检查输入参数是否符合要求，若不符合要求则执行某个函数输出警告。

```
#ifdef USE_FULL_ASSERT
/**
  * @brief   assert_param 宏用于函数的输入参数检查
  * @param   expr: 若 expr 值为假，则调用 assert_failed 函数报告文件名及错误行号
  *                若 expr 值为真，则不执行操作
  */
#define assert_param(expr) ((expr) ? (void)0 : assert_failed((uint8_t
*)_,→_FILE__, __LINE__))
/* 输出错误函数*/
void assert_failed(uint8_t* file, uint32_t line);
#else
#define assert_param(expr) ((void)0)
#endif
```

这段代码的意思是，假如我们不定义“USE_FULL_ASSERT”宏，那么“assert_param”就是一个空的宏(#else 与#endif 之间的语句生效)，没有任何操作。从而所有库函数中的 assert_param 实际上都无意义，我们就当看不见好了。

假如我们定义了“USE_FULL_ASSERT”宏，那么“assert_param”就是一个有操作的语句(#if 与#else 之间的语句生效)，该宏对参数 expr 使用 C 语言中的问号表达式进行判断，若 expr 值为真，则无操作(void0)，若表达式的值为假，则调用“assert_failed”函数，且该函数的输入参数为“__FILE__”及“__LINE__”，这两个参数分别代表“assert_param”宏被调用时所在的“文件名”及“行号”。

但以上只对“assert_failed”写了函数声明，没有写函数定义，实际用时需要用户来定义，我们一般可以通过 printf 函数来输出这些信息。

```
void assert_failed(uint8_t * file, uint32_t line)
{
  printf(“\r\n 输入参数错误，错误文件名 =%s, 行号 =%s”,file,line);
}
```

那么为什么函数输入参数不对的时候，assert_param 宏中的 expr 参数值会是假呢？以 GPIO_Init 函数为例，看它对 assert_param 宏的调用，它被调用时分别以“IS_GPIO_ALL_PERIPH(GPIOx)”、“IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin)”等作为输入参数，也就是说被调用时，expr 实际上是一条针对输入参数的判断表达式。例如“IS_GPIO_PIN”的宏定义：

```
#define IS_GPIO_PIN(PIN) ((PIN) != (uint32_t)0x00)
```

若它的输入参数 PIN 值为 0，则表达式的值为假，PIN 非 0 时表达式的值为真。我们知道用于选择 GPIO 引脚号的宏“GPIO_Pin_x”的值至少有一个数据位为 1，这样的输入参数才有意义，若 GPIO_InitStruct->GPIO_Pin 的值为 0，输入参数就无效了。配合“IS_GPIO_PIN”这句表达式，“assert_param”就实现了检查输入参数的功能。对 assert_param 宏的其它调用方式类似，大家可以自己看库源码来研究一下。

3.2.8 Doxygen 注释规范

在编程过程中建议使用如下注释规范：

```
/**
  * @brief   函数说明
  * @param   无
  * @retval  无
  */
```

这是一种名为“Doxygen”的注释规范，如果在工程文件中按照这种规范去注释，可以使用 Doxygen 软件自动根据注释生成帮助文档。我们提供的库帮助文档《BS32f0xx_lib_um.chm》，就是由该软件根据库文件的注释生成的。

3.2.9 防止头文件重复包含

在 LL 库的所有头文件以及用户自己编写的头文件中，可以看到类似下面这样的宏定义。它的功能是防止头文件被重复包含，避免发生编译错误。

```
#ifndef __user_H
#define __user_H

  /* 此处省略头文件的具体内容 */
```

```
#endif
```

在头文件的开头，使用“#ifndef”关键字，判断标号“__user_H”是否被定义，若没有被定义，则从“#ifndef”至“#endif”关键字之间的内容都有效，也就是说，这个头文件若被其它文件“#include”，它就会被包含到其该文件中了，且头文件中紧接着使用“#define”关键字定义上面判断的标号“__user_H”。当这个头文件被同一个文件第二次“#include”包含的时候，由于有了第一次包含中的“#define__user_H”定义，这时再判断“#ifndef__user_H”，判断的结果就是假了，从“#ifndef”至“#endif”之间的内容都无效，从而防止了同一个头文件被包含多次，编译时就不会出现“redefine（重复定义）”的错误了。

一般来说，我们不会直接在 C 的源文件写两个“#include”来包含同一个头文件，但可能因为头文件内部的包含导致重复，这种代码主要是避免这样的问题。如“bsp_user.h”文件中使用了“#include“BS32F030XXx.h”” 语句，按习惯，可能我们写主程序的时候会在 main 文件写“#include“bsp_user.h”及#include“BS32F030XXx.h””，这个时候“BS32F030XXx.h”文件就被包含两次了，如果没有这种机制，就会出错。

至于为什么要用两个下划线来定义“__user_H”标号，其实这只是防止它与其它普通宏定义重复了，如我们用“GPIO_PIN_0”来代替这个判断标号，就会因为 BS32F030XXx.h 已经定义了 GPIO_PIN_0，结果导致“bsp_user.h”文件无效了，“bsp_user.h”文件一次都没被包含。

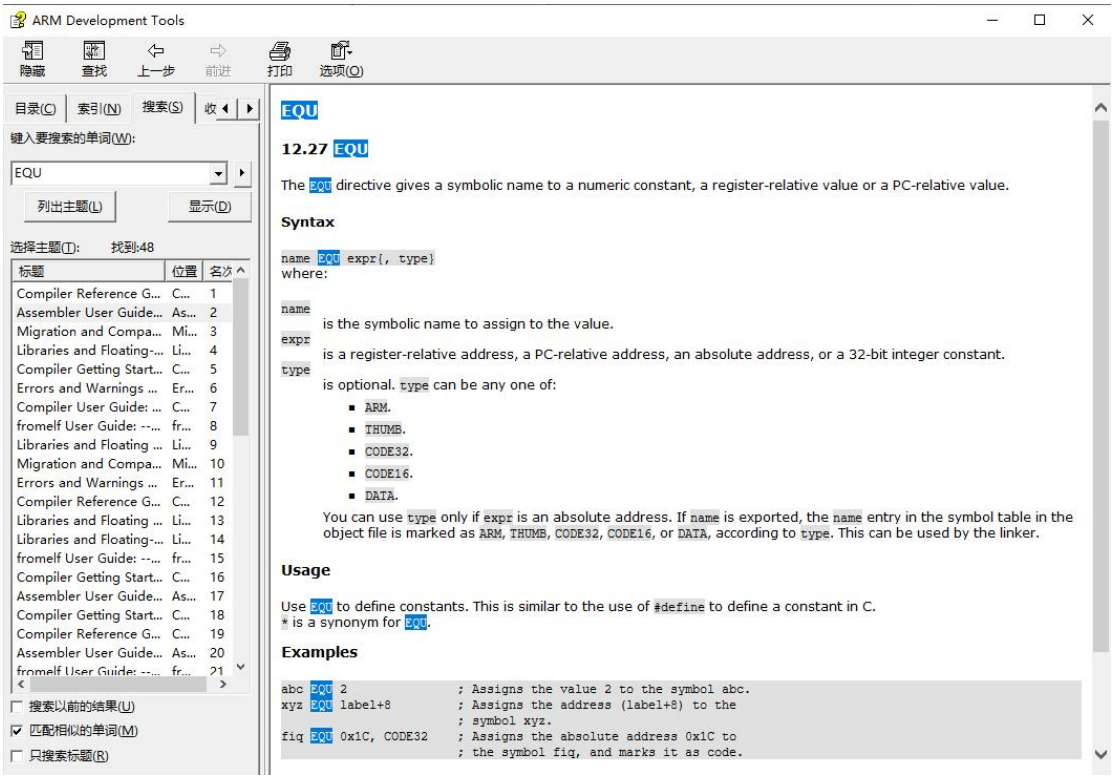
3.3 启动文件详解

启动文件由汇编编写，是系统上电复位后第一个执行的程序。主要做了以下工作：

- 初始化堆栈指针 SP=_initial_sp
- 初始化 PC 指针=Reset_Handler
- 初始化中断向量表
- 配置系统时钟
- 调用 C 库函数_main 初始化用户堆栈，从而最终调用 main 函数去到 C 的世界

3.3.1 ARM 汇编指令

在讲解启动代码的时候，会涉及到 ARM 的汇编指令和 Cortex 内核的指令，有关 Cortex 内核的指令我们可以参考《CM0 权威指南》中的指令集章节。剩下的 ARM 的汇编指令我们可以在 MDK->Help->UvisionHelp 中搜索到，以 EQU 为例，检索如下：



检索出来的结果会有很多，我们只需要看 Assembler User Guide 这部分即可。下面列出了启动文件中使用到的 ARM 汇编指令，该列表的指令全部从 ARM Development Tools 这个帮助文档里面检索而来。其中编译器相关的指令 WEAK 和 ALIGN 为了方便也放在同一个表格了。

指令名称	作用
EQU	给数字常量取一个符号名，相当于 C 语言中的 define
AREA	汇编一个新的代码段或者数据段
SPACE	分配内存空间
PRESERVE8	当前文件堆栈需按照 8 字节对齐
EXPORT	声明一个标号具有全局属性，可被外部的文件使用
DCD	以字为单位分配内存，要求 4 字节对齐，并要求初始化这些内存
PROC	定义子程序，与 ENDP 成对使用，表示子程序结束
WEAK	弱定义，如果外部文件声明了一个标号，则优先使用外部文件定义的标号，如果外部文件没有定义也不出错。要注意的是：这个不是 ARM 的指令，是编译器的，这里放在一起只是为了方便。
IMPORT	声明标号来自外部文件，跟 C 语言中的 EXTERN 关键字类似
B	跳转到一个标号
ALIGN	编译器对指令或者数据的存放地址进行对齐，一般需要跟一个立即数，缺省表示 4 字节对齐。要注意的是：这个不是 ARM 的指令，是编译器的，这里放在一起只是为了方便。
END	到达文件的末尾，文件结束
IF,ELSE,ENDIF	汇编条件分支语句，跟 C 语言的 if else 类似

3.3.2 启动文件代码

3.3.2.1 Stack 栈

```
Stack_Size      EQU     0x00000400

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp
```

开辟栈的大小为 0X00000400（1KB），名字为 STACK，NOINIT 即不初始化，可读可写，8（2^3）字节对齐。

栈的作用是用于局部变量，函数调用，函数形参等的开销，栈的大小不能超过内部 SRAM 的大小。如果编写的程序比较大，定义的局部变量很多，那么就需要修改栈的大小。如果某一天，你写的程序出现了莫名奇怪的错误，并进入了硬 fault 的时候，这时你就要考虑下是不是栈不够大，溢出了。

EQU：宏定义的伪指令，相当于等于，类似与 C 中的 define。

AREA：告诉汇编器汇编一个新的代码段或者数据段。STACK 表示段名，这个可以任意命名；NOINIT 表示不初始化；READWRITE 表示可读可写，ALIGN=3，表示按照 2^3 对齐，即 8 字节对齐。

SPACE：用于分配一定大小的内存空间，单位为字节。这里指定大小等于 Stack_Size。

标号__initial_sp 紧挨着 SPACE 语句放置，表示栈的结束地址，即栈顶地址，栈是由高向低生长的。

3.3.2.2 Heap 堆

```
Heap_Size       EQU     0x00000200

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

开辟堆的大小为 0X00000200（512 字节），名字为 HEAP，NOINIT 即不初始化，可读可写，8（2^3）字节对齐。__heap_base 表示堆的起始地址，__heap_limit 表示堆的结束地址。堆是由低向高生长的，跟栈的生长方向相反。

堆主要用来动态内存的分配，像 malloc()函数申请的内存就在堆上面。这个在 BS32 里面用的比较少。

```
                PRESERVE8
                THUMB
```

PRESERVE8：指定当前文件的堆栈按照 8 字节对齐。

THUMB：表示后面指令兼容 THUMB 指令。THUBM 是 ARM 以前的指令集，16bit，现在 Cortex-M 系列的都使用 THUMB-2 指令集，THUMB-2 是 32 位的，兼容 16 位和 32 位的指令，是 THUMB 的超集。

3.3.2.3 向量表

```
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size
```

定义一个数据段，名字为 RESET，可读。并声明__Vectors、__Vectors_End 和__Vectors_Size 这三个标号具有全局属性，可供外部的文件调用。

EXPORT：声明一个标号可被外部的文件使用，使标号具有全局属性。如果是 IAR 编译器，则使用的是 GLOBAL 这个指令。

当内核响应了一个发生的异常后，对应的异常服务例程(ESR)就会执行。为了决定 ESR 的入口地址，内核使用了“向量表查表机制”。这里使用一张向量表。向量表其实是一个 WORD（32 位整数）数组，每个下标对应一种异常，该下标元素的值则是该 ESR 的入口地址。向量表在地址空间中的位置是可以设置的，通过 NVIC 中的一个重定位寄存器来指出向量表的地址。在复位后，该寄存器的值为 0。因此，在地址 0（即 FLASH 地址 0）处必须包含一张向量表，用于初始时

的异常分配。要注意的是这里有个另类：0 号类型并不是什么入口地址，而是给出了复位后 MSP 的初值。

```
__Vectors       DCD     __initial_sp                ; 栈顶地址
                DCD     Reset_Handler               ; 复位程序地址
                DCD     NMI_Handler                 ; 非屏蔽地址
                DCD     HardFault_Handler           ; 硬件错误地址
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     SVC_Handler                 ; SVCall Handler
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     PendSV_Handler              ; PendSV Handler
                DCD     SysTick_Handler             ; SysTick Handler

                ; External Interrupts
                DCD     WWDG_IRQHandler             ; Window Watchdog
                DCD     LVDT_IRQHandler             ; LVDT
                DCD     RTC_TAMP_IRQHandler         ; RTC/TAMP/LSE_ERR
                DCD     FLASH_IRQHandler            ; FLASH
                DCD     RCC_IRQHandler              ; RCC/HSE_ERR
                DCD     EXTI0_1_IRQHandler          ; EXTI Line 0 and 1
                DCD     EXTI2_3_IRQHandler          ; EXTI Line 2 and 3
                DCD     EXTI4_15_IRQHandler         ; EXTI Line 4 to 15
                ......
__Vectors_End

__Vectors_Size  EQU  __Vectors_End - __Vectors
```

__Vectors 为向量表起始地址，__Vectors_End 为向量表结束地址，两个相减即可算出向量表大小。

向量表从 FLASH 的 0 地址开始放置，以 4 个字节为一个单位，地址 0 存放的是栈顶地址，0X04 存放的是复位程序的地址，以此类推。从代码上看，向量表中存放的都是中断服务函数的函数名，可我们知道 C 语言中的函数名就是一个地址。

DCD：分配一个或者多个以字为单位的内存，以四字节对齐，并要求初始化这些内存。在向量表中，DCD 分配了一堆内存，并且以 ESR 的入口地址初始化它们。

3.3.2.4 复位程序

```
AREA    |.text|, CODE, READONLY
```

定义一个名称为.text 的代码段，可读。

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

复位子程序是系统上电后第一个执行的程序，调用 SystemInit 函数初始化系统时钟，然后调用 C 库函数_mian，最终调用 main 函数去到 C 的世界。

WEAK：表示弱定义，如果外部文件优先定义了该标号则首先引用该标号，如果外部文件没有声明也不会出错。这里表示复位子程序可以由用户在其他文件重新实现，这里并不是唯一的。

IMPORT：表示该标号来自外部文件，跟 C 语言中的 EXTERN 关键字类似。这里表示 SystemInit 和__main 这两个函数均来自外部的文件。

SystemInit()是一个标准的库函数，在 system_bs32g0xx.c 这个库文件总定义。主要作用是配置中断向量表位置添加偏移地址。

__main 是一个标准的 C 库函数，主要作用是初始化用户堆栈，并在函数的最后调用 main 函数去到 C 的世界。这就是为什么我们写的程序都有一个 main 函数的原因。

LDR、BLX、BX 是 CM0+内核的指令，可在《CM0 权威指南》指令集章节里面查询到，具体作用见下表：

指令名称	作用
LDR	从存储器中加载字到一个寄存器中
BL	跳转到由寄存器/标号给出的地址，并把跳转前的下条指令地址保存到 LR
BLX	跳转到由寄存器给出的地址，并根据寄存器的 LSE 确定处理器的状态，还要把跳转前的下条指令地址保存到 LR
BX	跳转到由寄存器/标号给出的地址，不用返回

3.3.2.5 中断服务程序

在启动文件里面已经帮我们写好所有中断的中断服务函数，跟我们平时写的中断服务函数不一样的就是这些函数都是空的，真正的中断服务程序需要我们在外部的 C 文件里面重新实现，这里只是提前占了一个位置而已。

如果我们在使用某个外设的时候，开启了某个中断，但是又忘记编写配套的中断服务程序或者函

数名写错，那当中断来临的时，程序就会跳转到启动文件预先写好的空的中断服务程序中，并且在这个空函数中无限循环，即程序就“死”在这里。

```
NMI_Handler     PROC
                EXPORT  NMI_Handler                     [WEAK]
                B       .
                ENDP
HardFault_Handler\
                PROC
                EXPORT  HardFault_Handler               [WEAK]
                B       .
                ENDP
SVC_Handler     PROC
                EXPORT  SVC_Handler                     [WEAK]
                B       .
                ENDP
PendSV_Handler  PROC
                EXPORT  PendSV_Handler                  [WEAK]
                B       .
                ENDP
SysTick_Handler PROC
                EXPORT  SysTick_Handler                 [WEAK]
                B       .
                ENDP

Default_Handler PROC

                EXPORT  WWDG_IRQHandler                 [WEAK]
                EXPORT  LVDT_IRQHandler                 [WEAK]
                EXPORT  RTC_TAMP_IRQHandler             [WEAK]
                ……
WWDG_IRQHandler
LVDT_IRQHandler
RTC_TAMP_IRQHandler
……
                B       .

                ENDP
                ALIGN
```

B：跳转到一个标号。这里跳转到一个‘.’，即表示无限循环。

3.3.2.6 用户堆栈初始化

ALIGN：对指令或者数据存放的地址进行对齐，后面会跟一个立即数。缺省表示 4 字节对齐。

```
                IF      :DEF:__MICROLIB

                EXPORT  __initial_sp
                EXPORT  __heap_base
                EXPORT  __heap_limit

                ELSE

                IMPORT  __use_two_region_memory
                EXPORT  __user_initial_stackheap

__user_initial_stackheap

                LDR     R0, =  Heap_Mem
                LDR     R1, =(Stack_Mem + Stack_Size)
                LDR     R2, = (Heap_Mem +  Heap_Size)
                LDR     R3, = Stack_Mem
                BX      LR

                ALIGN

                ENDIF

                END
```

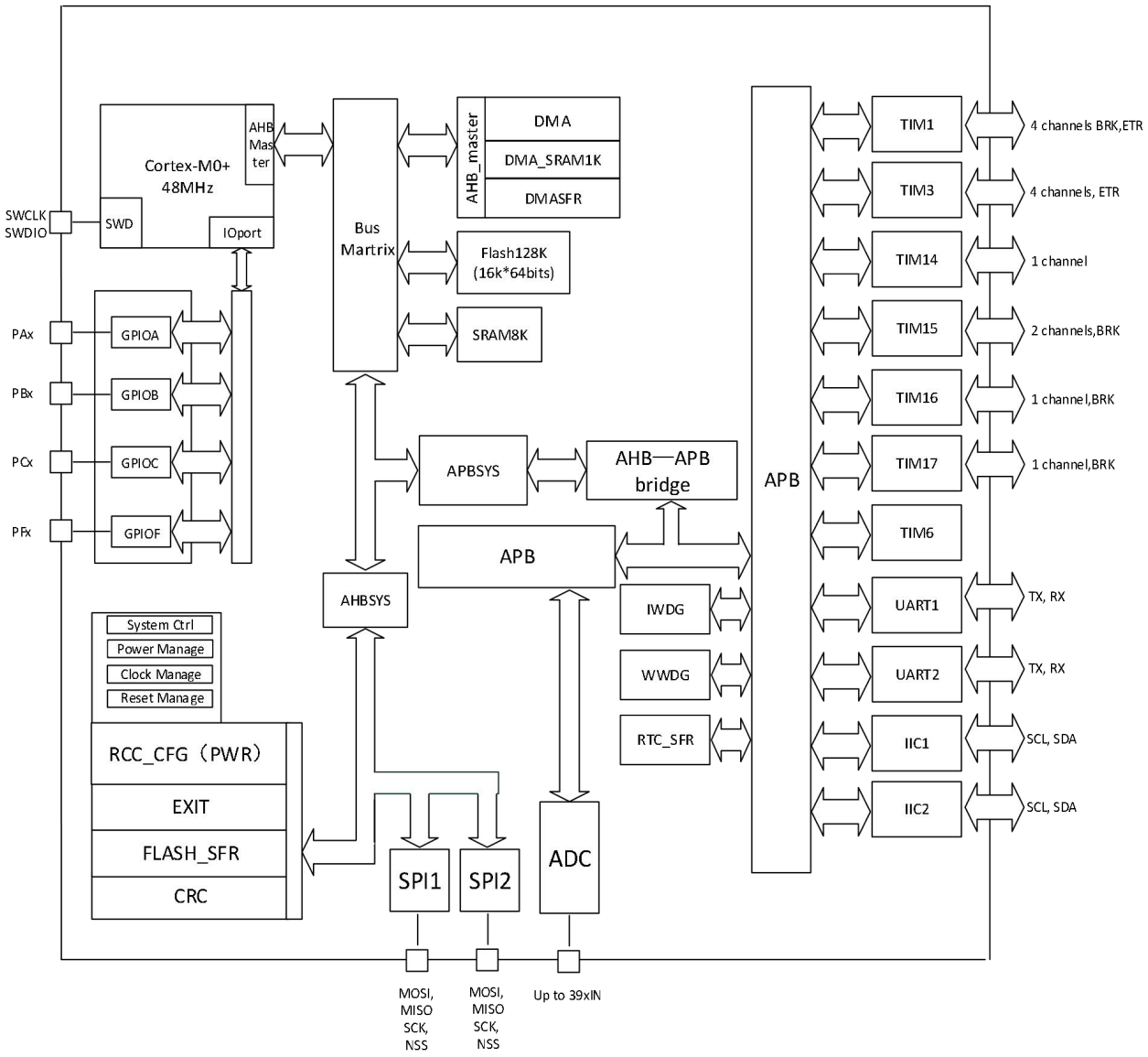
如果没有定义__MICROLIB，则才用双段存储器模式，且声明标号__user_initial_stackheap 具有全局属性，让用户自己来初始化堆栈。

IF,ELSE,ENDIF：汇编的条件分支语句，跟 C 语言的 if,else 类似

END：文件结束

3.4 存储器映射

在 BS32F030 系统框图中，被控单元 FLASH、SRAM 以及各片上外设都被排列在一个 4GB 的地址空间内。我们在编程的时候，可以通过他们的地址找到他们，然后通过 C 语言来对他们的数据进行读写。



在这 4GB 的地址空间中，ARM 均分为了 8 小块，每块 512MB，每个块规定了各自的用途，芯片产商一般只用其中的一部分地址，在这 8 个 Block 里面，有 3 个块是与我们编程息息相关的。

存储器本身不具有地址信息，它的地址由芯片厂商或用户分配，给存储器分配地址的过程就叫存储器映射。具体地址分配见下表：

类型	模块	基地址	结束地址
IOPORT	GPIOF	0x5000_1400	0x5000_17FF
	预留	0x5000_1000	0x5000_13FF
	GPIOC	0x5000_0800	0x5000_0BFF
	GPIOB	0x5000_0400	0x5000_07FF
	GPIOA	0x5000_0000	0x5000_03FF
APB	TIM17	0x4001_4800	0x4001_4BFF
	TIM16	0x4001_4400	0x4001_47FF
	TIM15	0x4001_4000	0x4001_43FF
	UART1	0x4001_3800	0x4001_3BFF
	TIM1	0x4001_2C00	0x4001_2FFF
	ADC	0x4001_2400	0x4001_27FF
	I2C2	0x4000_5800	0x4000_5BFF
	I2C1	0x4000_5400	0x4000_57FF
	UART2	0x4000_4400	0x4000_47FF
	IWDG	0x4000_3000	0x4000_33FF
	WWDG	0x4000_2C00	0x4000_2FFF
	RTC	0x4000_2800	0x4000_2BFF
	TIM14	0x4000_2000	0x4000_23FF
	TIM6	0x4000_1000	0x4000_13FF
	TIM3	0x4000_0400	0x4000_07FF
AHB	DMA_SRAM	0x4003_0000	0x4003_03FF
	DMASFR	0x4003_0400	0x4003_07FF
	AHB2APB	0x4000_0000	0x4001_63FF
	SPI2	0x4002_A000	0x4002_A3FF
	SPI1	0x4002_A400	0x4002_A7FF
	CRC	0x4002_3000	0x4002_33FF
	系统设置	0x4002_2000	0x4002_23FF
	EXTI	0x4002_1800	0x4002_1BFF
	RCC	0x4002_1000	0x4002_13FF
SRAM	SRAM	0x2000_0000	0x2000_1FFF
CODE	系统信息块	0x0002_0200	0x0002_03FF
	选项字节块	0x0002_0000	0x0002_01FF
	FLASH	0x0000_0000	0x0001_FFFF

其中 FLASH 属于存储器 Block0 区域范围，SRAM 属于 Block1 区域范围，片上外设属于 Block 2 区域范围。

3.5 寄存器映射

在存储器 Block2 这片区域，设计的是片上外设，以四个字节为一个单元，共 32bit，每一个单元对应不同的功能，当我们控制这些单元时就可以驱动外设工作。我们可以找到每个单元的起始地址，然后通过 C 语言指针的操作方式来访问这些单元，如果每次都是通过这种地址的方式来访问，不仅不好记忆还容易出错，这时我们可以根据每个单元功能的不同，以功能为名给这个内存单元取一个别名，这个别名就是我们经常说的寄存器，这个给已经分配好地址的有特定功能的内存单元取别名的过程就叫寄存器映射。

比如，我们找到 GPIOB 端口的输出数据寄存器 ODR 的地址是 0x50000414，ODR 寄存器是 32bit，低 16bit 有效，对应着 16 个外部 IO，写 0/1 对应的的 IO 则输出低/高电平。现在我们通过 C 语言指针的操作方式，让 GPIOB 的 16 个 IO 都输出高电平，具体代码（通过绝对地址访问内存单元）如下:

```
// GPIOB 端口全部输出高电平
*(unsigned int*)(0x5000 0414) = 0xFFFF;
```

0x50000414 在我们看来是 GPIOB 端口 ODR 的地址，但是在编译器看来，这只是一个普通的变量，是一个立即数，要想让编译器也认为是指针，我们得进行强制类型转换，把它转换成指针，即(unsigned int*)0x50000414，然后再对这个指针进行*操作。但是通过绝对地址访问内存单元不好记忆且容易出错，我们可以通过寄存器的方式来操作，具体代码（通过寄存器别名方式访问内存单元）如下:

```
// GPIOB 端口全部输出 高电平
#define GPIOB_ODR          (unsigned int*)(GPIOB_BASE+0x14)
* GPIOB_ODR = 0xFFFF;
```

为了方便操作，我们可以把指针操作“*”也定义到寄存器别名中，代码如下：

```
// GPIOB 端口全部输出 高电平
#define GPIOB_ODR          *(unsigned int*)(GPIOB_BASE+0x14)
GPIOB_ODR = 0xFFFF;
```

3.5.1 BS32 的外设地址映射

片上外设分为两条总线，根据外设速度的不同，不同总线挂载着不同的外设，APB 挂载低速外设，AHB 挂载高速外设。相应总线的最低地址我们称为该总线的基地址，总线基地址也是挂载在该总线上的首个外设的地址。

总线上挂载着各种外设，这些外设也有自己的地址范围，特定外设的首个地址称为“XX 外设基地址”，也叫 XX 外设的结束地址。具体有关 BS32F030 外设的边界地址请参考 BS32F0xx 参考手册的存储器映射表。这里我们以 GPIO 这个外设来讲解外设的基地址，GPIO 属于高速的外设。

外设寄存器

在 XX 外设的地址范围内，分布着的就是该外设的寄存器。以 GPIO 外设为例，GPIO 是通用输入输出端口的简称，简单来说就是 BS32 可控制的引脚，基本功能是控制引脚输出高电平或者低电平。最简单的应用就是把 GPIO 的引脚连接到 LED 灯的阴极，LED 灯的阳极接电源，然后通

过 BS32 控制该引脚的电平，从而实现控制 LED 灯的亮灭。

GPIO 有很多个寄存器，每一个都有特定的功能。每个寄存器为 32bit，占四个字节，在该外设的基地址上按照顺序排列，寄存器的位置都以相对该外设基地址的偏移地址来描述。如下所示不同端口的地址范围不同，各端口的寄存器在对应的地址范围内按偏移地址排列。

GPIOA(0x5000_0000~0x5000_03FF)
GPIOB(0x5000_0400~0x5000_07FF)
GPIOC(0x5000_0800~0x5000_0BFF)
GPIOD(0x5000_0C00~0x5000_0FFF)
GPIOF(0x5000_1400~0x5000_17FF)

有关外设寄存器详细说明可参考《BS32F0xx 参考手册》中具体章节的寄存器描述部分，在编程的过程中我们需要反复查阅外设的寄存器说明。

这里我们以“GPIO 端口置位/复位寄存器”为例，说明一下如何理解寄存器的说明：

GPIO 端口位置位/复位寄存器（GPIOx_BSRR）(x=A..F) 偏移地址：0x18 复位值：0x0000 0000	
BR <31:16>	BR[15:0]：端口 x 复位 I/O 引脚 y(PortxresetI/Opiny)（y=15 到 0） 这些位为只写。读取这些位可返回值 0x0000。 0：不会对相应的 ODRx 位执行任何操作 1：复位相应的 ODRx 位 注：如果同时对 BSx 和 BRx 置位，则 BSx 的优先级更高
BS <15:0>	BS[15:0]：端口 x 置位 I/O 引脚 y(PortxsetI/Opiny)（y=15 到 0） 这些位为只写。读取这些位可返回值 0x0000。 0：不会对相应的 ODRx 位执行任何操作 1：置位相应的 ODRx 位

名称

寄存器说明中首先列出了该寄存器中的名称，“(GPIOx_BSRR)(x=A…F)”这段的意思是该寄存器名为“GPIOx_BSRR”其中的“x”可以为 A-F，也就是说这个寄存器说明适用于 GPIOA、GPIOB 至 GPIOF，这些 GPIO 端口都有这样的一个寄存器。

偏移地址

偏移地址是指本寄存器相对于这个外设的基地址的偏移。本寄存器的偏移地址是 0x18，从参考手册中我们可以查到 GPIOA 外设的基地址为 0x5000 0000，我们就可以算出 GPIOA 的这个 GPIOA_BSRR 寄存器的地址为：0x5000 0000+0x18；同理，由于 GPIOB 的外设基地址为 0x5000 0400，可算出 GPIOB_BSRR 寄存器的地址为：0x5000 0400+0x18。其他 GPIO 端口以此类推即可。

寄存器位

位的参数包含：名称、权限与编号，其中读写权限又分为 w 表示只写，r 表示只读，r/w 表示可读写。本寄存器中的位权限都是 w，所以只能写，如果读本寄存器，是无法保证读取到它真正内容的。而有的寄存器位只读，一般是用于表示 BS32 外设的某种工作状态的，由 BS32 硬件自动

更改，程序通过读取那些寄存器位来判断外设的工作状态。

位功能说明

位功能是寄存器说明中最重要的部分，它详细介绍了寄存器每一个位的功能。例如本寄存器中有两种寄存器位，分别为 BRy 及 BSy，其中的 y 数值可以是 0-15，这里的 0-15 表示端口的引脚号，如 BR0、BS0 用于控制 GPIOx 的第 0 个引脚，若 x 表示 GPIOA，那就是控制 GPIOA 的第 0 引脚，而 BR1、BS1 就是控制 GPIOA 第 1 个引脚。

其中 BRy 引脚的说明是“0：不会对相应的 ODRx 位执行任何操作；1：对相应 ODRx 位进行复位”。这里的“复位”是将该位设置为 0 的意思，而“置位”表示将该位设置为 1；说明中的 ODRx 是另一个寄存器的寄存器位，我们只需要知道 ODRx 位为 1 的时候，对应的引脚 x 输出高电平，为 0 的时候对应的引脚输出低电平即可。所以，如果对 BR0 写入“1”的话，那么 GPIOx 的第 0 个引脚就会输出“低电平”，但是对 BR0 写入“0”的话，却不会影响 ODR0 位，所以引脚电平不会改变。要想该引脚输出“高电平”，就需要对“BS0”位写入“1”，寄存器位 BSy 与 BRy 是相反的操作。

3.5.2 C 语言对寄存器的封装

以上所有的关于存储器映射的内容，最终都是为大家更好地理解如何用 C 语言控制读写外设寄存器做准备，此处是本章的重点内容。

封装总线与外设基地址

在编程上为了方便理解和记忆，我们把总线基地址和外设基地址都以相应的宏定义起来，总线或者外设都以他们的名字作为宏名，具体可见 SDK 文件：BS32F030XXx.h 中定义。

```
#define FLASH_BASE            (0x00000000UL)  /*!< FLASH 基地址 */
#define OB_BASE               (0x000201e8UL)  /*!< option byte(选项字节基地址) */
#define SRAM_BASE             (0x20000000UL)  /*!< SRAM 基地址 */
#define PERIPH_BASE           (0x40000000UL)  /*!< 外设基地址 */
#define IOPORT_BASE           (0x50000000UL)  /*!< IO 口基地址 */
#define SRAM_SIZE_MAX         (0x00002000UL)  /*!< 最大 SRAM 容量 (8 KBytes) */

#define APBPERIPH_BASE        (PERIPH_BASE)
#define AHBPERIPH_BASE        (PERIPH_BASE + 0x00020000UL)

#define GPIOB_BASE            (IOPORT_BASE + 0x0400)
#define GPIOB_BSRR            (GPIOB_BASE+0x18)
```

首先定义了“片上外设”基地址 PERIPH_BASE，接着在 PERIPH_BASE 上加入各个总线的地址偏移，得到 APB、AHB 总线的地址 APBPERIPH_BASE、AHBPERIPH_BASE，在其之上加入外设地址的偏移，得到外设地址，最后在外设地址上加入各寄存器的地址偏移，得到特定寄存器的地址。一旦有了具体地址，就可以用指针读写。

```
/* 控制 GPIOB 引脚 0 输出低电平(BSRR 寄存器的 BR0 置 1) */
*(unsigned int *)GPIOB_BSRR = (0x01<<(16+0));
```

```
/* 控制 GPIOB 引脚 0 输出高电平(BSRR 寄存器的 BS0 置 1) */
*(unsigned int *)GPIOB_BSRR = 0x01<<0;

unsigned int temp;
/* 读取 GPIOB 端口所有引脚的电平(读 IDR 寄存器) */
temp = *(unsigned int *)GPIOB_IDR;
```

该代码使用(unsigned int *)把 GPIOB_BSRR 宏的数值强制转换成了地址，然后再用“*”号做取指针操作，对该地址的赋值，从而实现了写寄存器的功能。同样，读寄存器也是用取指针操作，把寄存器中的数据取到变量里，从而获取 BS32 外设的状态。

封装寄存器列表

用上面的方法去定义地址，还是稍显繁琐，例如 GPIOA-GPIOF 都各有一组功能相同的寄存器，如 GPIOA_ODR/GPIOB_ODR/GPIOC_ODR 等等，它们只是地址不一样，但却要为每个寄存器都定义它的地址。为了更方便地访问寄存器，我们引入 C 语言中的结构体语法对寄存器进行封装：

```
/* 相对 GPIOx 基址的偏移 */
typedef struct
{
  __IO uint32_t MODER;       /*!< GPIO 端口模式寄存器,         偏移地址: 0x00      */
  __IO uint32_t OTYPER;      /*!< GPIO 端口输出类型寄存器,     偏移地址: 0x04      */
  __IO uint32_t OSPEEDR;     /*!< GPIO 端口输出速度寄存器,     偏移地址: 0x08      */
  __IO uint32_t PUPDR;       /*!< GPIO 端口上拉/下拉寄存器,    偏移地址: 0x0C      */
  __IO uint32_t IDR;         /*!< GPIO 端口输入数据寄存器,     偏移地址: 0x10      */
  __IO uint32_t ODR;         /*!< GPIO 端口输出数据寄存器,     偏移地址: 0x14      */
  __IO uint32_t BSRR;        /*!< GPIO 端口位置位/复位寄存器,  偏移地址: 0x18      */
  __IO uint32_t LCKR;        /*!< GPIO 端口配置锁定寄存器,     偏移地址: 0x1C      */
  __IO uint32_t AFR[2];      /*!< GPIO 复用功能低位/高位寄存器,偏移地址: 0x20-0x24 */
  __IO uint32_t BRR;         /*!< GPIO 端口位复位寄存器,       偏移地址: 0x28      */
} GPIO_TypeDef;
```

这段代码用 typedef 关键字声明了名为 GPIO_TypeDef 的结构体类型，结构体内有 10 个成员变量，变量名正好对应寄存器的名字。C 语言的语法规定，结构体内变量的存储空间是连续的，其中 32 位的变量占用 4 个字节，16 位的变量占用 2 个字节。

也就是说，我们定义的这个 GPIO_TypeDef，假如这个结构体的首地址为 0x5000 0400（这也是第一个成员变量 MODER 的地址），那么结构体中第二个成员变量 OTYPER 的地址即为 0x5000 0400+0x04，加上的这个 0x04，正是代表 MODER 所占用的 4 个字节地址的偏移量，其它成员变量相对于结构体首地址的偏移，在上述代码右侧注释已给。

这样的地址偏移与 BS32 GPIO 外设定义的寄存器地址偏移一一对应，只要给结构体设置好首地址，就能把结构体内成员的地址确定下来，然后就能以结构体的形式访问寄存器。

```
GPIO_TypeDef * GPIOx;    //定义一个 GPIO_TypeDef 型结构体指针 GPIOx
GPIOx = GPIOB_BASE;      //把指针地址设置为宏 GPIOB_BASE 地址

GPIOx->ODR = 0xFFFF;
```

```
uint32_t temp;
temp = GPIOx->IDR;       //读取 GPIOB_IDR 寄存器的值到变量 temp 中
```

这段代码先用 GPIO_TypeDef 类型定义一个结构体指针 GPIOx，并让指针指向地址 GPIOB_BASE(0x50000400)，使用地址确定下来，然后根据 C 语言访问结构体的语法，用 GPIOx->ODR 及 GPIOx->IDR 等方式读写寄存器。

最后，我们更进一步，直接使用宏定义好 GPIO_TypeDef 类型的指针，而且指针指向各个 GPIO 端口的首地址，使用时我们直接用该宏访问寄存器即可：

```
/* 使用 GPIO_TypeDef 把地址强制转换成指针 */
#define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB                ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC                ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD                ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOF                ((GPIO_TypeDef *) GPIOF_BASE)

/* 使用定义好的宏直接访问 */
/* 访问 GPIOB 端口的寄存器 */
GPIOB->BSRR = 0xFFFF;      //通过指针访问并修改 GPIOB_BSRR 寄存器
GPIOB->MODER = 0xFFFF;     //修改 GPIOB_MODER 寄存器
GPIOB->ODR =0xFFFF;        //修改 GPIOB_ODR 寄存器

uint32_t temp;
temp = GPIOB->IDR;         //读取 GPIOB_IDR 寄存器的值到变量 temp 中
```

这里我们仅是以 GPIO 这个外设为例，给大家讲解了 C 语言对寄存器的封装。以此类推，其他外设也同样可以用这种方法来封装。这部分工作都已经通过我们的支持固件库完成了，这里我们只是分析了这个封装的过程，让大家有所了解。

3.6 通信概念

MCU 与片上外设之间，外设与外设之间都需要进行通信，所以这里统一了解一下通信相关内容。

3.6.1 串行通讯与并行通讯

按数据传送的方式，通讯可分为串行通讯与并行通讯，串行通信是按数据位形式一位一位地传输数据的通讯方式，并行通讯可以同时传输多个数据位的数据。

因为一次可传输多个数据位的数据，在数据传输速率相同的情况下，并行通讯传输的数据量要大得多，而串行通讯则可以节省数据线的硬件成本(特别是远距离时)以及 PCB 的布线面积。

不过由于并行传输对同步要求较高，且随着通讯速率的提高，信号干扰的问题会显著影响通讯性能，现在随着技术的发展，越来越多的应用场合采用高速率的串行差分传输。

3.6.2 全双工、半双工及单工通讯

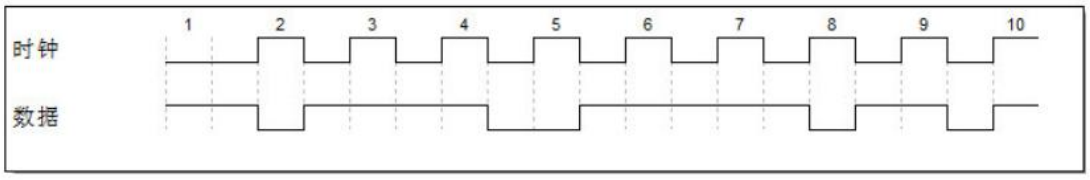
根据数据通讯的方向，通讯又分为全双工、半双工及单工通讯，它们主要以信道的方向来区分。

- 全双工：在同一时刻，两个设备之间可以同时收发数据
- 半双工：两个设备之间可以收发数据，但不能在同一时刻进行
- 单工：在任何时刻都只能进行一个方向的通讯，即一个固定为发送设备，另一个固定为接收设备

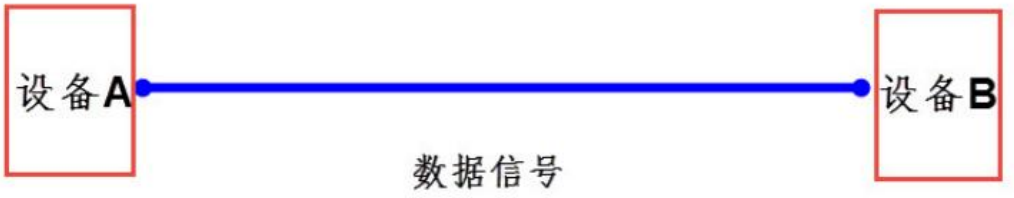
3.6.3 同步通讯与异步通讯

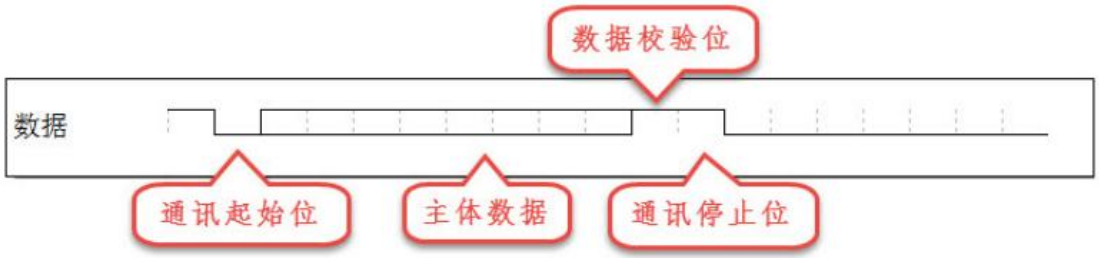
根据通讯的数据同步方式，又分为同步和异步两种，可以根据通讯过程中是否有使用到时钟信号进行简单的区分。

在同步通讯中，收发设备双方会使用一根信号线表示时钟信号，在时钟信号的驱动下双方进行协调，同步数据。通讯中通常双方会统一规定在时钟信号的上升沿或下降沿对数据线进行采样。



在异步通讯中不使用时钟信号进行数据同步，它们直接在数据信号中穿插一些同步用的信号位，或者把主体数据进行打包，以数据帧的格式传输数据，某些通讯中还需要双方约定数据的传输速率，以便更好地同步。





在同步通讯中，数据信号所传输的内容绝大部分就是有效数据，而异步通讯中会包含有帧的各种标识符，所以同步通讯的效率更高，但是同步通讯双方的时钟允许误差较小，而异步通讯双方的时钟允许误差较大。

3.6.4 通讯速率

衡量通讯性能的一个非常重要的参数就是通讯速率，通常以比特率(Bitrate)来表示，即每秒钟传输的二进制位数，单位为比特每秒(bit/s)。容易与比特率混淆的概念是“波特率”(Baudrate)，它表示每秒钟传输了多少个码元。而码元是通讯信号调制的概念，通讯中常用时间间隔相同的符号来表示一个二进制数字，这样的信号称为码元。如常见的通讯传输中，用 0V 表示数字 0，5V 表示数字 1，那么一个码元可以表示两种状态 0 和 1，所以一个码元等于一个二进制比特位，此时波特率的大小与比特率一致；如果在通讯传输中，有 0V、2V、4V 以及 6V 分别表示二进制数 00、01、10、11，那么每个码元可以表示四种状态，即两个二进制比特位，所以码元数是二进制比特位数的一半，这个时候的波特率为比特率的一半。因为很多常见的通讯中一个码元都是表示两种状态，所以常常直接以波特率来表示比特率，虽然一般不会出现错误，但它们还是有区别的。

第四章 BS32F030 MCU 常见外设介绍

4.1 BS32F0xx 中断应用

BS32 中断是十分重要的章节，每个外设都可以产生中断，所以单独拿出来做一个系统性的梳理。

4.1.1 中断类型

F030 在内核水平上搭载了一个异常响应系统，支持为数众多的系统中断和外部中断。其中系统中断有 5 个，外部中断有 32 个。除了个别中断异常的优先级被定死外，其它中断的优先级都是可编程的。有关具体的系统异常和外部中断可在固件库文件 BS32F030XXx.h 这个头文件查询到，在 IRQn_Type 这个结构体里面包含了 F030 系列全部的中断声明。

F030 系统中断清单

编号	优先级	类型	Mcu 中断列表	地址	说明
--	--	--	--	0x0	保留
--	-3(最高)	固定	Reset	0x4	复位
-14	-2	固定	NMI	0x08	不可屏蔽中断。保留
-13	-1	固定	HardFault	0x0C	硬件错误
-5	3	可编程	SVCall	0x2C	通过 SVC 实现的请求管理调用
-2	5	可编程	PendSV	0x38	可挂起的系统服务请求
-1	6	可编程	SysTick	0x3C	系统节拍定时器

F030 外部中断清单

编号	优先级	类型	Mcu 中断列表	地址	说明
0	7	可编程	WWDG	0x40	窗口看门狗中断
1	8	可编程	PVD(LVDT)	0x44	电源电压检测器中断
2	9	可编程	RTC/TAMP	0x48	RTC /TAMP /LSE 中断
3	10	可编程	FLASH	0x4C	Flash 全局中断
4	11	可编程	RCC	0x50	RCC 全局中断(HSE_ERR)
5	12	可编程	EXTI0_1	0x54	EXTI 线 0 和 1 中断
6	13	可编程	EXTI2_3	0x58	EXTI 线 2 和 3 中断
7	14	可编程	EXTI4_15	0x5C	EXTI 线 4 到 15 中断
8	15	可编程	--	0x60	
9	16	可编程	DMA_INT0	0x64	DMA 通道口对应关系见 DMA 列表
10	17	可编程	DMA_INT1	0x68	DMA 通道口对应关系见 DMA 列表
11	18	可编程	DMA_INT2	0x6C	DMA 通道口对应关系见 DMA 列表
12	19	可编程	ADC_COMP	0x70	ADC 和 COMP 中断
13	20	可编程	TIM1_BRK_UP_TRG_COM	0x74	TIM1 刹车、更新、触发和换相中断
14	21	可编程	TIM1_CC	0x78	TIM1 捕获比较中断
15	22	可编程	--	0x7C	

16	23	可编程	TIM3	0x80	TIM3 全局中断
17	24	可编程	TIM6	0x84	TIM6 全局中断
18	25	可编程	--	0x88	--
19	26	可编程	TIM14	0x8C	TIM14 全局中断
20	27	可编程	TIM15	0x90	TIM15 全局中断
21	28	可编程	TIM16	0x94	TIM16 全局中断
22	29	可编程	TIM17	0x98	TIM17 全局中断
23	30	可编程	I2C1	0x9C	I2C1 全局中断
24	31	可编程	I2C2	0xA0	I2C2 全局中断
25	32	可编程	SPI1	0xA4	SPI1 全局中断
26	33	可编程	SPI2	0xA8	SPI2 全局中断
27	34	可编程	UART1	0xAC	UART1 全局中断
28	35	可编程	UART2	0xB0	UART2 全局中断
29	36	可编程	DMA_INT3	0xB4	DMA 通道口对应关系见 DMA 列表
30	37	可编程	DMA_INT4	0xB8	DMA 通道口对应关系见 DMA 列表
31	38	可编程	DMA_INT5	0xBC	DMA 通道口对应关系见 DMA 列表

4.1.2 NVIC 介绍

M0+内核中的可配置嵌套的向量式中断控制器(NVIC)与 M0+内核紧密耦合，它负责处理物理层线路与不可屏蔽中断(NMI)、可屏蔽中断相关事件和 Cortex-M0+中断。可灵活管理优先级。

处理器内核与 NVIC 的紧密耦合显著的降低了触发中断和进入中断服务函数(ISR)的时间。ISR 向量在向量列表中，存储在 NVIC 的基地址中。一个 ISR 的向量地址执行是由硬件上向量表的基地址和 ISR 编号的偏移量决定的。

如果高优先级的中断事件发生之前在低优先级的中断事件已经发生且准备执行，后来的高优先级中断事件先执行。还有一种优化叫做尾链，当从高优先级 ISR 返回然后开始一个挂起的低优先级的 ISR，非必需的处理器环境堆栈和挂起会被跳过。这减少延时且有助于节省功耗。

本节将介绍中断的优先级设置、如何定义中断函数名称、中断向量如何偏移。有关 NVIC 的更多知识，请见《ARM Cortex-M0 权威指南》。

注意：

- NVIC 负责 MCU 所有的外设中断，内部的异常中断则是由 SCB 系统控制块来管理。
- NVIC 属于内核外设，因此不同芯片公司同一 M0+内核的 NVIC 配置基本一致
- M0+内核没有优先级分组，即无抢占优先级与子优先级的区分，共支持 48 个中断：16 个内核+32 个中断
- 系统中断（比如：PendSV，SVC，SysTick）不一定比外部中断（比如 SPI,UART）要高，它们是在同一个 NVIC 下面设置的
- Cortex-M0+采用 Armv6-M 架构，优先级寄存器配置位有 8 位，但是有效位只有最高 2 位，Cortex-M3 则是最高 3 或 4 位有效位，在 arm 官方资料中有对比两个版本的差别。因此 Co

rtex-M0+可编程优先级有 4 个，加上 3 个固定的优先级(复位、NMI、HardFault)，Cortex-M0+总共有 7 个中断优先级。Cortex-M0+内核的中断优先级寄存器是以最高位（MSB）对齐的，并且只支持字传输，每次访问都会同时涉及 4 个中断优先级寄存器。见下图：

n	n	x	x	x	x	x	x
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
已使用		未使用，读出全为 0					

- 中断优先级寄存器的编程应该在中断使能之前，其通常是在程序开始时完成的。arm 官方资料提示应该避免在中断使能之后改变中断优先级，因为这种情况的结果在 ARMv6-M 系统结构是不可预知的，并且不被 Cortex-M0+处理器支持。Cortex-M3/M4 处理器的情况又有所不同，他们都支持中断优先级的动态切换。Cortex-M3 处理器和 Cortex-M0+处理器的另外一个区别是，Cortex-M3 访问中断优先级寄存器时支持字节或半字传输，因此可以每次只设置一个寄存器。如果需要改变优先级，程序中需要关闭中断后再重新设置中断优先级寄存器。

- Cortex-M0+处理器对中断嵌套的支持无需任何软件干预，如果 MCU 已经在运行一个中断，而有了新的更高优先级的中断请求，正在运行的中断将会被暂停，转而执行更高优先级的中断，高优先级中断执行完成后又回到原来的低优先级中断。如果出现软件优先级相同的情况，则比较硬件优先级。

NVIC 结构体定义，来自固件库头文件：core_cm0plus.h

```
typedef struct
{
  __IOM uint32_t ISER[1U];           /*!< 偏移地址: 0x000 (R/W)  中断设置使能寄存器 */
        uint32_t RESERVED0[31U];
  __IOM uint32_t ICER[1U];           /*!< 偏移地址: 0x080 (R/W)  中断清除使能寄存器 */
        uint32_t RSERVED1[31U];
  __IOM uint32_t ISPR[1U];           /*!< 偏移地址: 0x100 (R/W)  中断设置挂起寄存器 */
        uint32_t RESERVED2[31U];
  __IOM uint32_t ICPR[1U];           /*!< 偏移地址: 0x180 (R/W)  中断设置清除寄存器 */
        uint32_t RESERVED3[31U];
        uint32_t RESERVED4[64U];
  __IOM uint32_t IP[8U];             /*!< 偏移地址: 0x300 (R/W)  中断优先级寄存器 */
}  NVIC_Type;
```

在配置中断的时候我们一般只用 ISER、ICER 和 IP 这三个寄存器，ISER 用来使能中断，ICER 用来关闭中断，IP 用来设置中断优先级。

NVIC 中断配置固件库，来自固件库头文件：core_cm0plus.h

NVIC 库函数	描述
void __NVIC_EnableIRQ(IRQn_Type IRQn)	使能中断
void __NVIC_DisableIRQ(IRQn_Type IRQn)	关闭中断
void __NVIC_SetPendingIRQ(IRQn_Type IRQn)	使能中断挂起位
void __NVIC_ClearPendingIRQ(IRQn_Type IRQn)	清除中断挂起位

uint32_t __NVIC_GetPendingIRQ(IRQn_Type IRQn)	取挂起中断编号
void __NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)	设置中断优先级
uint32_t __NVIC_GetPriority(IRQn_Type IRQn)	获取中断优先级
void __NVIC_SystemReset(void)	系统复位

这些函数遵循 CMSIS 规则，只要是 Cortex-M0+的处理器都可以使用。

4.1.3 中断编程

在配置每个中断的时候一般有 3 个编程要点：

1.使能外设某个中断，这个具体由每个外设的相关中断使能位控制。比如串口有发送完成中断，接收完成中断，这两个中断都由串口控制寄存器的相关中断使能位控制。

2.配置 NVIC 相关函数，这里主要是指：配置中断优先级分组，使能中断请求。如下代码所示：

```
NVIC_SetPriority(TIM3, 3);
//中断源为 TIM3，具体成员配置见 IRQn_Type 这个结构体；优先级为 3
NVIC_EnableIRQ(TIM3);
//中断使能，操作的是 NVIC_ISER 这个寄存器；配置中断关闭时，操作 NVIC_ICER
```

3.编写中断服务函数

在启动文件 startup_BS32F030XXx.s 中我们预先为每个中断都写了一个中断服务函数，只是这些中断函数都是为空，为的只是初始化中断向量表。实际的中断服务函数都需要我们重新编写，为了方便管理我们把中断服务函数统一写在 BS32F030XXx_it.c 这个库文件中。

关于中断服务函数的函数名必须跟启动文件里面预先设置的一样，如果写错，系统就在中断向量表中找不到中断服务函数的入口，直接跳转到启动文件里面预先写好的空函数，并且在里面无限循环，实现不了中断。

4.3.4 EXTI 中断

EXTI（External interrupt controller）：外部中断控制器，管理了控制器的 16 个中断线。每个中断线都对应有一个边沿检测器，可以实现输入信号的上升沿检测和下降沿的检测。EXTI 可以实现对每个中断线进行单独配置。

产生中断线路目的是把输入信号输入到 NVIC，进一步会运行中断服务函数，实现功能，这样是软件级的。

中断/事件线	输入源
EXTI0-15	PAX/PBX/PCX/PDX/PFX(X 可为 0-15)

EXTI0 至 EXTI15 用于 GPIO，通过编程控制可以实现任意一个 GPIO 作为 EXTI 的输入源。EXTI0 可以通过 EXTI 的外部中断选择寄存器 1(EXTI_EXTICR1)的 EXTI0[7:0]位选择配置为 PA0、PB0、PC0、PD0 或者 PF0，详见 BS32F030 寄存器列表中的 EXTI 部分。其他 EXTI 线(EXTI 中断线)使用配置都是类似的。

4.1.4.1 EXTI 初始化结构体详解

固件库函数对每个外设都建立了一个初始化结构体，比如 LL_EXTI_InitTypeDef，结构体成员用于设置外设工作参数，并由外设初始化配置函数，比如 LL_EXTI_StructInit()调用，这些设定参数将会设置外设相应的寄存器，达到配置外设工作环境的目的。

初始化结构体和初始化库函数配合使用是标准库精髓所在，理解了初始化结构体每个成员意义基本上就可以对该外设运用自如了。初始化结构体定义在 bs32g0xx_ll_exti.h 文件中，初始化库函数定义在 bs32g0xx_ll_exti.c 文件中，编程时我们可以结合这两个文件内注释使用。

EXTI 初始化结构体

```
typedef struct
{
  uint32_t Line;              //中断线
  uint32_t LineCommand;       //EXTI 使能
  uint32_t Mode;              //EXTI 模式
  uint32_t Trigger;           //触发类型
} LL_EXTI_InitTypeDef;
```

EXTI 初始化库函数

```
void LL_EXTI_StructInit(LL_EXTI_InitTypeDef *EXTI_InitStruct)
{
  EXTI_InitStruct->Line         = LL_EXTI_LINE_NONE;      //无中断线
  EXTI_InitStruct->LineCommand  = DISABLE;                //关闭使能
  EXTI_InitStruct->Mode         = LL_EXTI_MODE_IT;        //中断模式
  EXTI_InitStruct->Trigger      = LL_EXTI_TRIGGER_FALLING;//下降沿模式触发
}
```

Line：EXTI 中断线选择，可选 EXTI0 至 EXTI15。

Mode：EXTI 模式选择，可选为产生中断(LL_EXTI_MODE_IT)。

Trigger：EXTI 边沿触发事件，可选无触发（LL_EXTI_TRIGGER_NONE）、上升沿触发(LL_EXTI_TRIGGER_RISING)、下降沿触发(LL_EXTI_TRIGGER_FALLING)或者上升沿和下降沿都触发(LL_EXTI_TRIGGER_RISING_FALLING)。

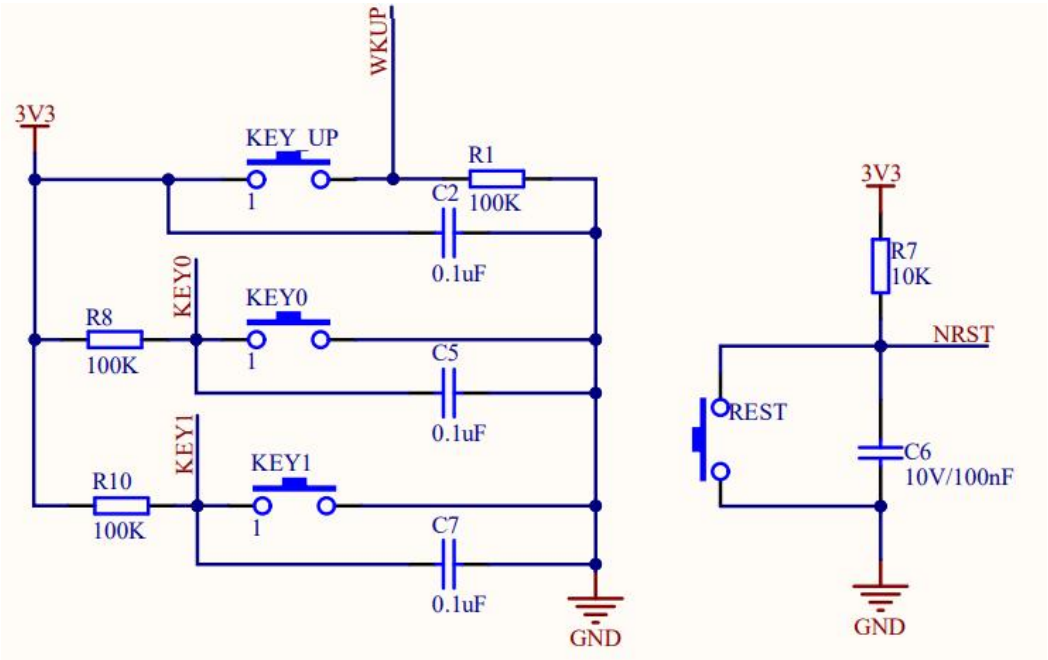
LineCommand：控制是否使能 EXTI 线，可选使能 EXTI 线(ENABLE)或禁用(DISABLE)。

4.1.4.2 外部中断控制实验

中断在嵌入式应用中占有非常重要的地位，几乎每个控制器都有中断功能。中断对保证紧急事件得到第一时间处理是非常重要的。我们设计使用外接的按键来作为触发源，使得控制器产生中断，并在中断服务函数中实现控制任务。

硬件设计

按键按下的时候引脚接通，同时电平产生变化。



软件设计

此处仅介绍核心代码，完整代码请参考配套工程文件，编程要点：

- 初始化用来产生中断的 GPIO；
- 初始化 EXTI;
- 配置 NVIC；
- 编写中断服务函数；

4.2 RCC 时钟应用

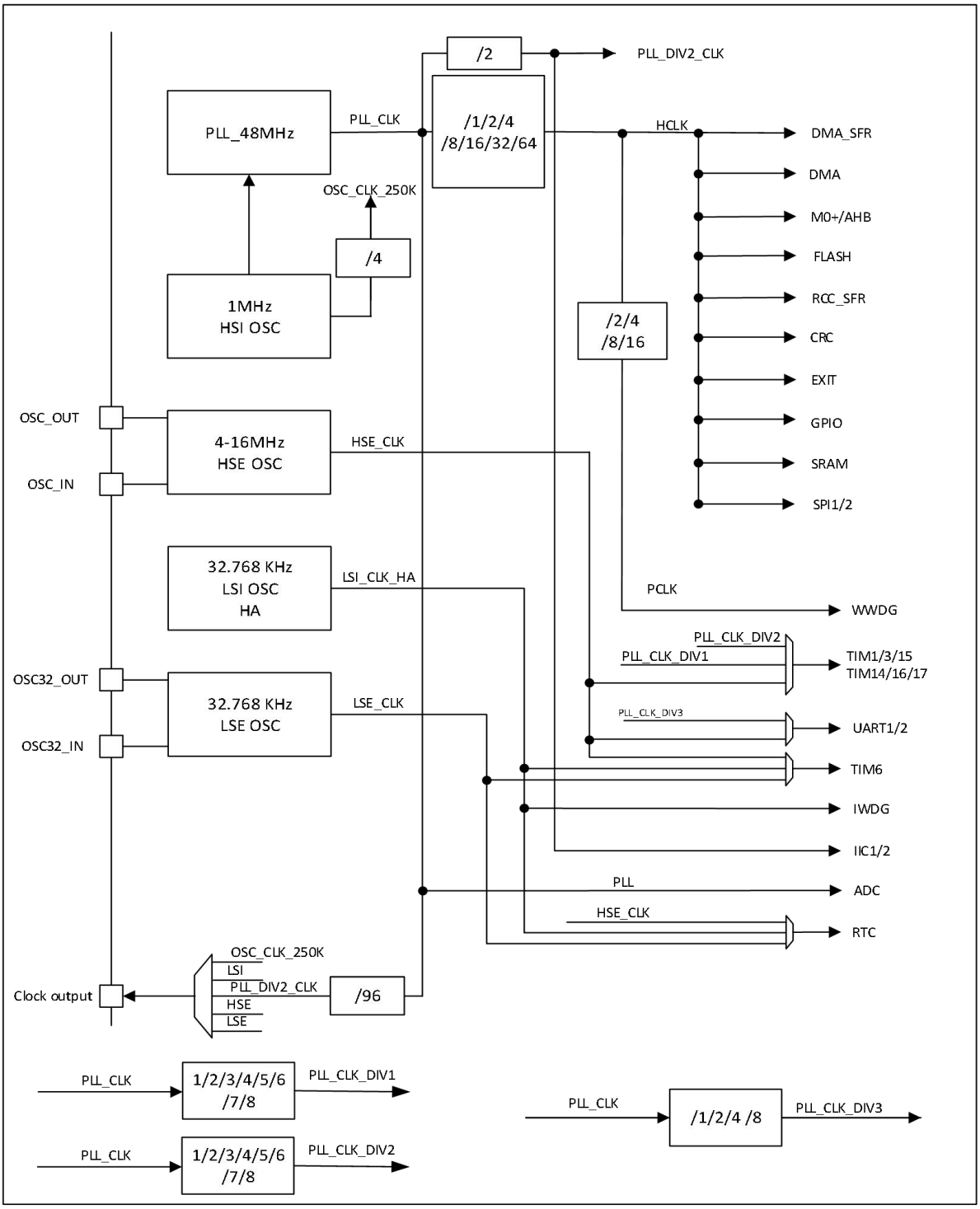
RCC：reset clock control 复位和时钟控制器。本章我们主要讲解时钟部分，特别是要着重理解时钟树，理解了时钟树，BS32 的一切时钟的来龙去脉都会了如指掌。

系统时钟源为 1MHz 的 HSI、经过 PLL 倍频后设置 HPRE 分频因子（决定 HCLK 等于多少）、设置 PPRE 分频因子（决定 PPRE 等于多少）。MCU 的时钟源有四种，分别是：高速内部时钟 HSI、高速外部时钟 HSE、低速高精度内部时钟 LSI、低速外部时钟 LSE。

4.1.1 时钟结构

接下来我们结合代码来讲解时钟树。这里选取库函数时钟系统时钟函数：**voidSystemClock_Config(void)**以这个函数的编写流程来讲解时钟树，这个函数也是我们用库的时候默认的系统时钟设置函数。该函数的功能是利用 HSI 经过 PLL 倍频后把时钟设置为：HCLK=SYSCLK=48M，PCLK=HCLK/2=24M（这里的系统时钟指的是 M0+内核的时钟频率，也是嘀嗒时钟的时钟源；具体 AHB 与 APB 总线分频因子选项在对应寄存器说明或程序注释中可以查看）。下面我们就以这个代码的流程为主线，来分析时钟树。

BS32F030XX 系统时钟树



HSE 高速外部时钟信号

HSE 是高速的外部时钟信号，可以由有源晶振或者无源晶振提供，频率从 4-16MHZ 不等。当使用有源晶振时，时钟从 OSC_IN 引脚进入，OSC_OUT 引脚悬空，当选用无源晶振时，时钟从 OSC_IN 和 OSC_OUT 进入，并且要配谐振电容。HSE 最常使用的就是 8M 的无源晶振。HSE 提供可选时钟给 TIM1/3/15、TIM14/16/17、UART1/2 与 TIM6

PLL 时钟源

PLL 的时钟源为 HSI，频率为 1M，根据温度和环境的情况频率会有一定的漂移。PLL 默认固定倍频因子为 48，因此 PLL 时钟：PLLCLK=1M*48=48M。

其他时钟

其他外设的时钟，如 ADC、IIC1/2、RTC 等可以对照时钟树得知。

4.1.2 RCC 库函数说明

以下函数取自于固件库文件 bs32f0xx_ll_rcc.c,完成的是 BS32 系统时钟的配置。

```
void SystemClock_Config(void)
{
    LL_RCC_SetSystemClockFreq(LL_RCC_SYSCLK_DIV_1);
    //由 PLL 分频得到系统时钟 HCLK
    LL_RCC_SetPCLKClockFreq(LL_RCC_APB1_DIV_1);
    //由 HCLK 分频得到 PCLK
    LL_RCC_SetHSEClockFreq(8000000UL);
    //根据外接晶振配置
    LL_Init1msTick(LL_RCC_GetSystemClockFreq());
    //初始化滴答定时器
}
```

4.1.2.1 系统时钟/HCLK 配置

该函数用于主要配置 HCLK，包含 DMA_SFR/SRAM、DMA、M0+/AHB、FLASH_CTRL/FLASH IP、RST_CTRL、INT_CTRL、RCC_SFR、CRC、EXIT、GPIO、SPI1/2，同时也作为 PCLK 的时钟源。其中 HPRE 分频因子由 RCC 的 CR 寄存器低三位<2:0>组成，其值可以是[1，2，4，8，16，32，64]中的一个，其他值选择默认配置 12MHz。

注意：时钟选择电路逻辑切换，分频与不分频间时序有等待时间。

```
void LL_RCC_SetSystemClockFreq(uint32_t Prescaler)
{
    MODIFY_REG(RCC->CR, RCC_CR_HPRE, Prescaler);
    //操作 CR 寄存器修改 HPRE 位
    SystemCoreClock =
    LL_RCC_GetPLLClockFreq()>>(AHBPrescTable[Prescaler]);
    //赋值系统核心时钟，用于嘀嗒时钟
}
```

4.1.2.2 PCLK 配置/HSE 配置

该函数用于配置 PCLK，提供时钟给 WWDG、RTC_APB、TIM1/3/15 与 TIM14/16/17，其中分频因子 PPRE 由 RCC 的 CR 寄存器三位<5:3>组成，其值可以是[2，4，8，16]中的一个，其他值选择默认配置 2 分频。

注意：考虑到时钟选择电路逻辑切换情况，现用时钟门控代替时钟切换，此方案时钟输出占空比不是 50%，高电平时间固定为 1/128 ns 时间。

```
void LL_RCC_SetPCLKClockFreq(uint32_t Prescaler)
{
    MODIFY_REG(RCC->CR, RCC_CR_PPRE, Prescaler);
    //操作 CR 寄存器修改偏 PPRE 位
}
```

根据外部晶振的频率来配置 HSE：

```
LL_RCC_SetHSEClockFreq(8000000UL);
```

4.1.2.3 Systick 定时器配置

SysTick:系统定时器是属于 CM0+内核中的一个外设，内嵌在 NVIC 中。计数器每计数一次的时间为 1/SYSCLK,有关寄存器的定义和部分库函数都在 core_cm0plus.h 这个头文件中实现。

SysTick 定时器除了能服务于操作系统之外，还能用于其它目的：如作为一个闹铃，用于测量时间等。下面的嘀嗒初始化函数给 MCU 配置了 1ms 的时基。

编程要点：

- 设置重装载寄存器的值
- 清除当前数值寄存器的值
- 配置控制与状态寄存器

```
__STATIC_INLINE void LL_InitTick(uint32_t HCLKFrequency, uint32_t Ticks)
{
  /* Configure the SysTick to have interrupt in 1ms time base */
  SysTick->LOAD  = (uint32_t)((HCLKFrequency / Ticks) - 1UL);
  /* 配置重装载寄存器，ticks=1000 */
  SysTick->VAL   = 0UL;
  /* 装载系统嘀嗒计数器值 */
  SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                   SysTick_CTRL_ENABLE_Msk;
  /* 使能系统嘀嗒定时器 */
}
```

配置 SysTick 中断优先级

用户可通过调用库函数 NVIC_SetPriority()来配置系统定时器的中断优先级，该库函数在 core_cm0plus.h 中定义，原型如下：

```
__STATIC_INLINE void __NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
{
  if ((int32_t)(IRQn) >= 0)
  {
```

```
    NVIC->IP[_IP_IDX(IRQn)]  = ((uint32_t)(NVIC->IP[_IP_IDX(IRQn)]  & ~(0xFFUL <<
_BIT_SHIFT(IRQn))) |
       (((priority << (8U - __NVIC_PRIO_BITS)) & (uint32_t)0xFFUL) <<
_BIT_SHIFT(IRQn)));
  }
  else
  {
    SCB->SHP[_SHP_IDX(IRQn)] = ((uint32_t)(SCB->SHP[_SHP_IDX(IRQn)] & ~(0xFFUL <<
_BIT_SHIFT(IRQn))) |
       (((priority << (8U - __NVIC_PRIO_BITS)) & (uint32_t)0xFFUL) <<
_BIT_SHIFT(IRQn)));
  }
}
```

函数首先先判断形参 IRQn 的大小，如果是小于 0，则表示这个是系统异常，系统异常的优先级由内核外设 SCB 的寄存器 SHP 控制，如果大于 0 则是外部中断，外部中断的优先级由内核外设 NVIC 中的 IP 寄存器控制。

```
// 设置系统定时器中断优先级
NVIC_SetPriority(SysTick_IRQn, 3);
```

片上外设与 systick 同时设置了中断的时候，根据优先级来分先后，优先级设置一样的时候，比较他们在中断向量表中的硬件编号，编号越小，优先级越高。

软件延时

Systick 的寄存器 CTRL 可以用于软件延时之中，调用下面的函数可以自定义以 ms 为单位的延时，代码如下所示：

```
void LL_mDelay(uint32_t Delay)
{
    /* 首先清 COUNTFLAG 位 */
    __IO uint32_t  tmp = SysTick->CTRL;
    uint32_t tmpDelay;
    /* 添加此代码声明局部变量没有被使用 */
    ((void)tmp);
    tmpDelay  = Delay;
    /* 添加周期以保证最小等待 */
    if (tmpDelay  < LL_MAX_DELAY)
    {
        tmpDelay ++;
    }
    /* 判断 Systick 是否为 0 */
    while (tmpDelay  != 0U)
    {
        if ((SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk) != 0U)
        {
```

```
            tmpDelay --;
        }
    }
}
```

4.1.2.4 MCO 输出

在 BS32F030 系列中，PA8 与 PA9 可以复用为 MCO 引脚，对外提供时钟输出，我们可以通过示波器监测该引脚的输出来判断我们的系统时钟是否正确设置。步骤如下：

1. MCO GPIO 的初始化
2. MCO 输出时钟选择

这里仅介绍 MCO 时钟选择库函数，代码如下：

```
__STATIC_INLINE void LL_RCC_ConfigMCO(uint32_t MCOSource)
{
   MODIFY_REG(RCC->TESTCR, RCC_TESTCR_CLKSEL, MCOSource);
  //LL_RCC_MCOSOURCE_PLL48M_DIV96/PLL64M_DIV128/HSI1M_DIV4/LSI32KHA/LSE/HSE
}
```

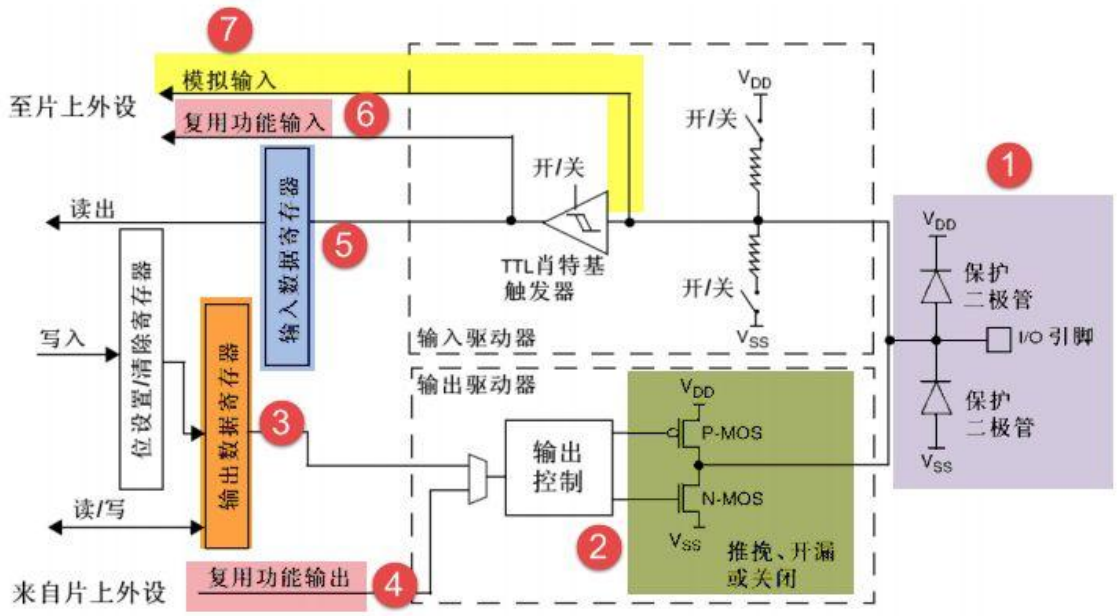
4.3 GPIO 模块介绍

GPIO 是通用输入输出端口的简称，简单来说就是 BS32 可控制的引脚，BS32 芯片的 GPIO 引脚与外部设备连接起来，从而实现与外部通讯、控制以及数据采集的功能。BS32 芯片的 GPIO 被分成很多组，每组有 0-16 个引脚不等，如型号为 BS32F030XX 型号的芯片有 GPIOA、GPIOB、GPIOC 与 GPIOF 共 4 组 GPIO，芯片一共 48 个引脚，所有的 GPIO 引脚都有基本的输入输出功能。

最基本的输出功能是由 BS32 控制引脚输出高、低电平，实现开关控制，如把 GPIO 引脚接入到 LED 灯，那就可以控制 LED 灯的亮灭，引脚接入到继电器或三极管，那就可以通过继电器或三极管控制外部大功率电路的通断。

最基本的输入功能是检测外部输入电平，如把 GPIO 引脚连接到按键，通过电平高低区分按键是否被按下。

4.3.1 GPIO 框图分析



通过 GPIO 硬件结构框图，就可以从整体上深入了解 GPIO 外设及它的各种应用模式。该图从最右端看起，最右端就是代表 BS32 芯片引出的 GPIO 引脚，其余部件都位于芯片内部。

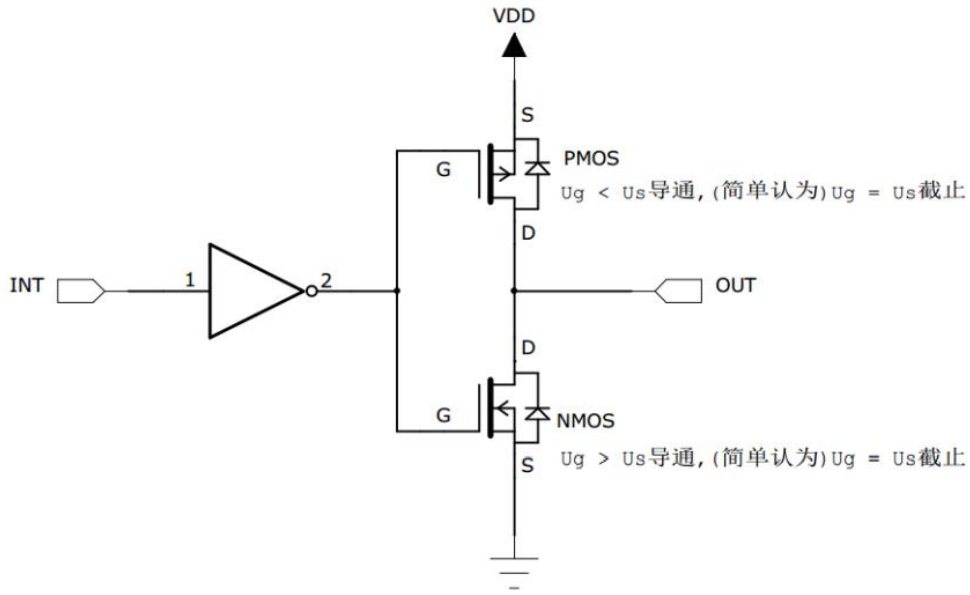
保护二极管及上、下拉电阻

引脚的两个保护二级管可以防止引脚外部过高或过低的电压输入，当引脚电压高于 VDD 时，上方的二极管导通，当引脚电压低于 VSS 时，下方的二极管导通，防止不正常电压引入芯片导致芯片烧毁。尽管有这样的保护，并不意味着 BS32 的引脚能直接外接大功率驱动器件，如直接驱动电机，强制驱动要么电机不转，要么导致芯片烧坏，必须要加大功率及隔离电路驱动。

P-MOS 管与 N-MOS 管

GPIO 引脚线路经过两个保护二极管后，向上流向“输入模式”结构，向下流向“输出模式”结构。先看输出模式部分，线路经过一个由 P-MOS 和 N-MOS 管组成的单元电路。这个结构使 GPIO 具有了“推挽输出”和“开漏输出”两种模式。

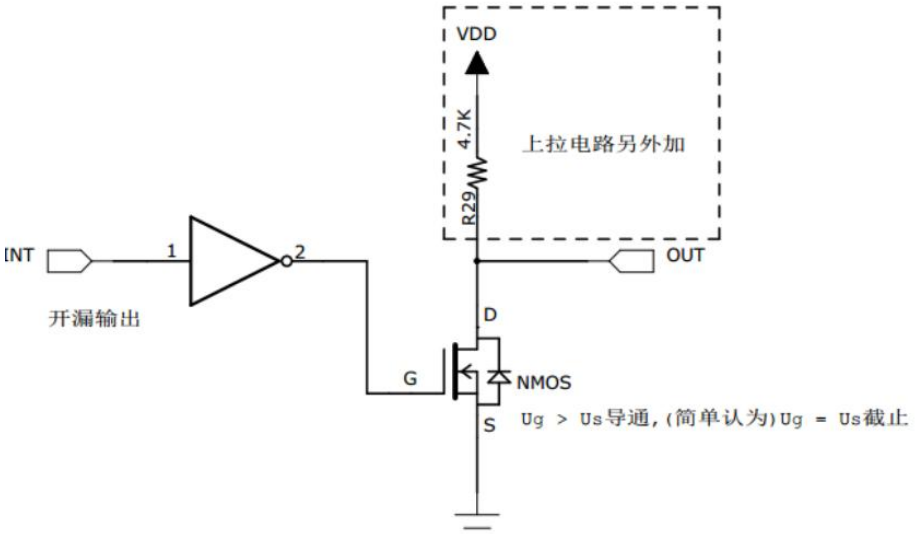
推挽输出模式：根据这两个 MOS 管的工作方式来命名的。在该结构中输入高电平时，经过反向后，上方的 P-MOS 导通，下方的 N-MOS 关闭，对外输出高电平；而在该结构中输入低电平时，经过反向后，N-MOS 管导通，P-MOS 关闭，对外输出低电平。当引脚高低电平切换时，两个管子轮流导通，P 管负责灌电流，N 管负责拉电流，使其负载能力和开关速度都比普通的方式有很大的提高。推挽输出的低电平为 0 伏，高电平为 3.3 伏，下图就是推挽输出模式时的等效电路。



而在开漏输出模式时，上方的 P-MOS 管完全不工作。如果我们控制输出为 0，低电平，则 P-MOS 管关闭，N-MOS 管导通，使输出接地，若控制输出为 1(它无法直接输出高电平)时，则 P-MOS 管和 N-MOS 管都关闭，所以引脚既不输出高电平，也不输出低电平，为高阻态。正常使用时必须外部接上拉电阻。它具有“线与”特性，也就是说，若有很多个开漏模式引脚连接到一起时，只有当所有引脚都输出高阻态，才由上拉电阻提供高电平，此高电平的电压为外部上拉电阻所接的电源的电压。若其中一个引脚为低电平，那线路就相当于短路接地，使得整条线路都为低电平 0 伏。

推挽输出模式一般应用在输出电平为 0 和 3.3 伏而且需要高速切换开关状态的场合。在 BS32 的应用中，除了必须用开漏模式的场合，我们都习惯使用推挽输出模式。

开漏输出一般应用在 I2C、SMBUS 通讯等需要“线与”功能的总线电路中。除此之外，还用在电平不匹配的场合，如需要输出 5 伏的高电平，就可以在外部接一个上拉电阻，上拉电源为 5 伏，并且把 GPIO 设置为开漏模式，当输出高阻态时，由上拉电阻和电源向外输出 5 伏的电平。



输出数据寄存器

前面提到的双 MOS 管结构电路的输入信号，是由 GPIO“输出数据寄存器 GPIOx_ODR”提供的，因此我们通过修改输出数据寄存器的值就可以修改 GPIO 引脚的输出电平。而“置位/复位寄存器 GPIOx_BSRR”可以通过修改输出数据寄存器的值从而影响电路的输出。

复用功能输出

“复用功能输出”中的“复用”是指 BS32 的其它片上外设对 GPIO 引脚进行控制，此时 GPIO 引脚用作该外设功能的一部分，算是第二用途。从其它外设引出来的“复用功能输出信号”与 GPIO 本身的数据据寄存器都连接到双 MOS 管结构的输入中，通过图中的梯形结构作为开关切换选择。例如我们使用 USART 串口通讯时，需要用到某个 GPIO 引脚作为通讯发送引脚，这个时候就可以把该 GPIO 引脚配置成 USART 串口复用功能，由串口外设控制该引脚，发送数据。

输入数据寄存器

看 GPIO 结构框图的上半部分，GPIO 引脚经过内部的上、下拉电阻，可以配置成上/下拉输入，然后再连接到施密特触发器，信号经过触发器后，模拟信号转化为 0、1 的数字信号，然后存储在“输入数据寄存器 GPIOx_IDR”中，通过读取该寄存器就可以了解 GPIO 引脚的电平状态。

复用功能输入

与“复用功能输出”模式类似，在“复用功能输入模式”时，GPIO 引脚的信号传输到 BS32 其它片上外设，由该外设读取引脚状态。

如我们使用 USART 串口通讯时，需要用到某个 GPIO 引脚作为通讯接收引脚，这个时候就可以把该 GPIO 引脚配置成 USART 串口复用功能，使 USART 可以通过该通讯引脚接收远端数据。

模拟输入

当 GPIO 引脚用于 ADC 采集电压的输入通道时，用作“模拟输入”功能，此时信号是不经过施密特触发器的，因为经过施密特触发器后信号只有 0、1 两种状态，所以 ADC 外设要采集到原始的模拟信号，信号源输入必须在施密特触发器之前。类似地，当 GPIO 引脚用于 DAC 作为模拟电压输出通道时，此时作为“模拟输出”功能，DAC 的模拟信号输出就不经过双 MOS 管结

构，模拟信号直接输出到引脚。但 BS32F0xx 系列暂时不支持 DAC 功能。

4.3.2 GPIO 工作模式

GPIO 的结构决定了可以配置成以下模式：

```
#define LL_GPIO_MODE_INPUT              (0x00000000U)       /*!< 选择输入模式 */
#define LL_GPIO_MODE_OUTPUT             GPIO_MODER_MODE0_0  /*!< 选择输出模式 */
#define LL_GPIO_MODE_ALTERNATE          GPIO_MODER_MODE0_1  /*!< 选择复用功能模式 */
#define LL_GPIO_MODE_ANALOG             GPIO_MODER_MODE0    /*!< 选择模拟模式 */

#define LL_GPIO_OUTPUT_PUSHPULL         (0x00000000U) /*!< 选择推挽作为输出类型 */
#define LL_GPIO_OUTPUT_OPENDRAIN        GPIO_OTYPER_OT0 /*!< 选择开漏作为输出类型 */

#define LL_GPIO_PULL_NO                 (0x00000000U) /*!< 选择 IO 浮空 */
#define LL_GPIO_PULL_UP                 GPIO_PUPDR_PUPD0_0 /*!< 选择 I/O 上拉 */
#define LL_GPIO_PULL_DOWN               GPIO_PUPDR_PUPD0_1 /*!< 选择 I/O 下拉 */

#define LL_GPIO_AF_0                    (0x0000000U) /*!< 选择复用功能 0 */
...
#define LL_GPIO_AF_7                    (0x0000007U) /*!< 选择复用功能 7 */
```

上述 GPIO 工作模式可以归类为如下：

输入模式（模拟/浮空/上拉/下拉）

在输入模式时，施密特触发器打开，输出被禁止，可通过输入数据寄存器 GPIOx_IDR 读取 I/O 状态。其中输入模式，可设置为上拉、下拉、浮空和模拟输入四种。上拉和下拉输入很好理解，默认的电平由上拉或者下拉决定。浮空输入的电平是不确定的，完全由外部的输入决定，一般接按键的时候用的是这个模式。模拟输入则用于 ADC 采集。

输出模式（推挽/开漏）

在输出模式中，推挽模式时双 MOS 管以轮流方式工作，输出数据寄存器 GPIOx_ODR 可控制 I/O 输出高低电平。开漏模式时，只有 N-MOS 管工作，输出数据寄存器可控制 I/O 输出高阻态或低电平。输出速度可配置，有 2M/10M/50M 的选项。此处的输出速度即 I/O 支持的高低电平状态最高切换频率，支持的频率越高，功耗越大，如果功耗要求不严格，把速度设置成最大即可。在输出模式时施密特触发器是打开的，即输入可用，通过输入数据寄存器 GPIOx_IDR 可读取 I/O 的实际状态。

复用功能（推挽/开漏）

复用功能模式中，输出使能，输出速度可配置，可工作在开漏及推挽模式，但是输出信号源于其它外设，输出数据寄存器 GPIOx_ODR 无效；输入可用，通过输入数据寄存器可获取 I/O 实际状态，但一般直接用外设的寄存器来获取该数据信号。

通过对 GPIO 寄存器写入不同的参数，就可以改变 GPIO 的工作模式，寄存器详细说明请翻阅《BS32F030XX 参考手册》中对应外设的寄存器内容。

4.3.3 GPIO 输出编程要点（点亮 LED）

这里编程的逻辑就是：直接向 BSRR、BRR 和 ODR 这三个寄存器写入控制指令来实现的，对 BSRR 写 1 输出高电平，对 BRR 写 1 输出低电平，对 ODR 寄存器某位进行异或操作可反转位的状态，完整的代码请参考本节配套的工程文件。

- 使能 GPIO 端口时钟
- 初始化 GPIO 目标引脚为推挽输出模式
- 编写简单测试程序，控制 GPIO 引脚输出高、低电平

4.3.4 GPIO 输入编程要点（按键检测）

在硬件电路设计中要注意通过硬件电容充放电或软件延时来消抖滤波。

- 使能 GPIO 端口时钟
- 初始化 GPIO 目标引脚为输入模式（浮空输入）
- 编写简单的测试程序，检测按键的状态，实现按键控制

4.4 UART 模块介绍

4.4.1 串口通信

串口通讯(Serial Communication)是一种设备间非常常用的串行通讯方式，因为它简单便捷，因此大部分电子设备都支持该通讯方式，电子工程师在调试设备时也经常使用该通讯方式输出调试信息。

在计算机科学里，大部分复杂的问题都可以通过分层来简化。如芯片被分为内核层和片上外设；BS32 LL 库则是在寄存器与用户代码之间的软件层。对于通讯协议，我们也以分层的方式来理解，最基本的是把它分为物理层和协议层。物理层规定通讯系统中具有机械、电子功能部分的特性，确保原始数据在物理媒体的传输。协议层主要规定通讯逻辑，统一收发双方的数据打包、解包标准。

下面我们分别对串口通讯协议的物理层及协议层进行讲解。

物理层

串口通讯的物理层有很多标准与变种，常用的有 RS-232 标准 RS-232 标准主要规定了信号的用途、通讯接口以及信号的电平标准。

通常两个通讯设备的“DB9 接口”之间通过串口信号线建立起连接，串口信号线中使用“RS-232 标准”传输数据信号。由于 RS-232 电平标准的信号不能直接被 MCU 直接识别，所以这些信号会经过一个“电平转换芯片”转换成控制器能识别的“TTL 标准”的电平信号，才能实现通讯。

我们常见的电子电路中常使用 TTL 的电平标准，理想状态下，使用 5V 表示二进制逻辑 1，使用 0V 表示逻辑 0；而为了增加串口通讯的远距离传输及抗干扰能力，RS-232 使用-15V 表示逻辑 1，+15V 表示逻辑 0。因为控制器一般使用 TTL 电平标准，所以常常会使用电平转换芯片对 TTL 及 RS-232 电平的信号进行互相转换。

在目前的工业控制使用的串口通讯中，一般只使用 RXD、TXD 以及 GND 三条信号线，直接传输数据信号，而 DB9 接口中 RTS、CTS、DSR、DTR 及 DCD 信号都被裁剪掉了。

协议层

串口通讯的数据包由发送设备通过自身的 TXD 接口传输到接收设备的 RXD 接口。在串口通讯的协议层中，规定了数据包的内容，它由启始位、主体数据、校验位以及停止位组成，通讯双方的数据包格式要约定一致才能正常收发数据。

- 波特率：本章中主要讲解的是串口异步通讯，异步通讯中由于没有时钟信号，所以两个通讯设备之间需要约定好波特率，即每个码元的长度，以便对信号进行解码，上图中串口数据包的基本组成中用虚线分开的每一格就是代表一个码元。常见的波特率为 4800、9600、

115200 等。

- 通讯的起始和停止信号：串口通讯的一个数据包从起始信号开始，直到停止信号结束。数据包的起始信号由一个逻辑 0 的数据位表示，而数据包的停止信号可由 0.5、1、1.5 或 2 个逻辑 1 的数据位表示，只要双方约定一致即可。
- 有效数据：在数据包的起始位之后紧接着的就是要传输的主体数据内容，也称为有效数据，有效数据的长度常被约定为 5、6、7 或 8 位长。
- 数据校验：在有效数据之后，有一个可选的数据校验位。由于数据通信相对更容易受到外部干扰导致传输数据出现偏差，可以在传输过程加上校验位来解决这个问题。校验方法有奇校验(odd)、偶校验(even)以及无校验(non)。

  奇校验要求有效数据和校验位中“1”的个数为奇数，比如一个 8 位长的有效数据为：01101001，此时总共有 4 个“1”，为达到奇校验效果，校验位为“1”，最后传输的数据将是 8 位的有效数据加上 1 位的校验位总共 9 位。

  偶校验与奇校验要求刚好相反，要求帧数据和校验位中“1”的个数为偶数，比如数据帧：11001010，此时数据帧“1”的个数为 4 个，所以偶校验位为“0”。

4.4.2 BS32 的 UART

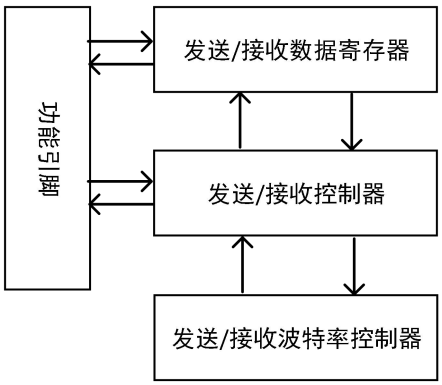
UART(Universal Asynchronous Receiverand Transmitter)，它是在 USART 基础上裁剪掉了同步通信功能，只有异步通信。简单区分同步和异步就是看通信时需不需要对外提供时钟输出，我们平时用的串口通信基本都是 UART。

UART 支持使用 DMA，可实现高速数据通信，有关 DMA 具体应用将在 DMA 章节作具体讲解。

UART 在 BS32 应用最多是“打印”程序信息，一般在硬件设计时都会预留一个 UART 通信接口连接电脑，用于在调试程序时可以把一些调试信息“打印”在电脑端的串口调试助手工具上，从而了解程序运行是否正确、如果出错具体哪里出错等等。

UART 功能框图

熟悉该外设的功能框图，编程思路会比较清晰。



功能引脚

TX：发送数据输出引脚

RX：接收数据输入引脚

SW_RX：数据接收引脚，只用于单线和智能卡模式，属于内部引脚，没有具体外部引脚。

nRTS：请求以发送(Request To Send)，n 表示低电平有效。如果使能 RTS 流控制，当 USART 接收器准备好接收新数据时就会将 nRTS 变成低电平；当接收寄存器已满时，nRTS 将被设置为高电平。该引脚只适用于硬件流控制。

nCTS：清除以发送(Clear To Send)，n 表示低电平有效。如果使能 CTS 流控制，发送器在发送下一帧数据之前会检测 nCTS 引脚，如果为低电平，表示可以发送数据，如果为高电平则在发送完当前数据帧之后停止发送。该引脚只适用于硬件流控制。

SCLK：发送器时钟输出引脚。这个引脚仅适用于同步模式，这里我们不涉及。

UART 引脚在 BS32F030XX 芯片具体分布如下：

	UART1	UART2
TX	PA9/PB6	PA2/PA14
RX	PA10/PB7	PA3/PA15
CTS	PA11/PB4	PA0
RTS	PA12/PB3	PA1

BS32F030XX 有两个 UART，其时钟来源可以是 sysclk（最大频率 48M）或 HSE_CLK（最大频率 16M）。

数据寄存器

UART 数据寄存器只有低 9 位有效，且 9 位奇偶校验模式下，bit[8]为奇偶校验位，数据字长为 8 位。数据寄存器实际为发送数据寄存器（UART_TBUF）与接收数据寄存器（UART_RBUF）分别用于发送与读写操作。二者都是介于系统总线和移位寄存器之间。串行通信是一个位一个位传输的，发送时把 UART_TBUF 内容转移到发送移位寄存器，然后把移位寄存器数据每一位发送出去，接收时把接收到的每一位顺序保存在接收移位寄存器内然后才转移到 UART_RBUF。

UART 支持 DMA 传输，可以实现高速数据传输。

控制器

UART 有专门控制发送的控制器、控制接收的接收器，还有唤醒单元、中断控制等。使用 UART 之前需要向 UART_CR1 寄存器的 UE 位置 1 使能 UART，UE 位用来开启供给串口的时钟。

发送或接收数据字长可选 8 位或 9 位，由 UART_CR1 的 DM 位控制。

发送器

整个发送过程必须在模块使能时（UART_CR1_UE =1）进行

在 UART_CR1 中置 TE 位为 1，发射器被使能。通过把数据写入 UART 发送数据寄存器（UART_TBUF），开启发送过程。UART 发射器的中心元件是长度为 10 或 11 或 12 位（取决于 UART_CR1 寄存器的 DM 位和 STOP 控制位中的配置值）的发送移位寄存器。假设 DM=0，选择正常的 8 位数据模式。在 8 位数据模式中，移位寄存器中有 1 个起始位、8 个数据位和 1 或 2 个停止位。完整的发送过程中，待发送内容先写入数据寄存器 UART_TBUF，从 UART_TBUF

发送出去后，则设置发送完成（UART_SR_TXEF）状态标记，显示新的数据可以再次写入UART_TBUF，重新开启新的发送。如果发送中断使能（UART_CR2_TXEIE=1），会产生发送中断。

配置 UART_CR2_MSBFIRST，发送接收为小端模式（LSB 先发）和大端模式（MSB 先发）可选。奇偶校验有效时，大端模式（MSB）下只有数据位进行交换，奇偶校验位仍是最高位 bit[8]。

UART 使用 DMA 进行发送时，DMA 发送使能信号（UART_DER_TXDMAEN）拉高，如果发送寄存器空标志置位（UART_SR_TXEF=1），则会生成 DMA 发送请求，发送完成状态标记未置位（UART_SR_TXEF=0）则拉低请求。为了正常工作，传输完成后需要关闭 DMA 发送使能并清除相关 UART 状态标记。

以 UART1 为例，使用 DMA 发送配置流程：（要先配 UART 发送使能，再配 DMA 发送使能，否则无法完成第一次 DMA 发送请求）

1) 配置 UART 波特率等相关寄存器
2) 配置 DMA 控制数据结构基地址 DMA_CCDBPR_CCDBP、DMA 控制器使能 DMA_CFGR_EN
3) 配置 UART1 对应的发送通道使能 DMA_CENSR_CENS=0x20000000、通道中断使能 DMA_CIER_CIE=0x20000000（UART1 为通道 29，UART2 为通道 31）
4) 配置 DMA 源数据结束地址 DMA_CSEARx_SDEA 为待发送数据所在 SRAM 地址、DMA 目的数据结束地址 DMA_CDEARx_DDEA 为 UART_TBUF 的地址
5) 配置 DMA_CCFGRx 寄存器选取传输数据大小、传输数据数量以及传输类型（传输数据大小：8 位数据模式可以选择 byte 和 halfword，9 位数据模式选择 halfword；UART 传输时仲裁率 ART 要保持为 0 不能配置）
6) 配置 UART 发送使能 UART_CR1_TE
7) 配置 UART 的 DMA 发送使能 UART_DER_TXDMAEN，UART 使用 DMA 发送
8) DMA 传输完成后（达到设置的传输数据数量）会产生 DMA 中断，为下一次的通信正常，需要在 DMA 中断关闭 DMA 发送使能
9) 等待 UART 发送完成，在 UART 中断中清除状态标记

接收器

整个接收过程必须在模块使能时（UART_CR1_UE =1）进行。

设置 UART_CR1 中的 RE 位，接收器被使能。数据字符由逻辑 0 的起始位、8 个（或 9 个）数据位和逻辑 1 的停止位组成。在把停止位接收到接收移位寄存器后，如果接收数据寄存器还未完成(UART_SR_RXCF=0)，数据字符就被传输到接收数据寄存器，并设置接收数据寄存器已完成（UART_SR_RXCF =1）状态标记；如果此时已经设置了接收数据寄存器已完成的UART_SR_RXCF=1，就设置溢出（UART_SR_ROFE）状态标记，新数据丢失。

当前接收的数据会有检测机制，可检测接收溢出、帧出错、奇偶校验出错 3 种错误，均需要软件清除标记。建议检测到接收中断后，读出状态标记，读 UART_RBUF，最后将接收数据状态标记均清除（UART_SCR[3:0]）。

闲置字符是以接收器活动一段时间（第一笔数据发送完停止位）后才开始检测的。一旦检测到 bit 为 0 则清除计数，检测到足够的 1（10/11/12 位），将产生闲置字符检测标记（UART_MUTE_IDLEIF）。

UART 使用 DMA 进行接收时，一个数据被 UART 成功接收之后便会自动生成 DMA 请求。接收满标记在每次 DMA 读取数据寄存器的数据时都会被自动清零。注意，为了后续通信，使用 DMA 传输完成后，需要关闭 DMA 接收使能并清除 UART 相关状态标记。

以 UART1 为例，使用 DMA 接收配置流程：

1) 配置 UART 波特率，接收使能和错误中断使能等相关寄存器
2) 配置 DMA 控制数据结构基地址 DMA_CCDBPR_CCDBP、DMA 控制器使能 DMA_CFGR_EN
3) 配置 UART1 对应的接收通道使能 DMA_CENSR_CENS=0x10000000、通道中断使能 DMA_CIER_CIE=0x10000000（UART1 为通道 28，UART2 为通道 30）
4) 配置 DMA 源数据结束地址 DMA_CDEARx_DDEA 为 UART_RBUF 地址，DMA 目的数据结束地址 DMA_CDEARx_DDEA 为 SRAM 的地址
5) 配置 DMA_CCFGRx 寄存器选取传输数据大小、传输数据数量以及传输类型（传输数据大小：8 位数据模式可以选择 byte 和 halfword，9 位数据模式选择 halfword；UART 传输时仲裁率 ART 要保持为 0 不能配置）
6) 配置 UART 接收使能 UART_CR1_RE
7) 配置 UART 的 DMA 接收使能 UART_DER_RXDMAEN，UART 使用 DMA 接收
8) 如果接收数据出现错误（帧错误和溢出错误），UART 会生成错误中断，可在中断进行错误处理
9) DMA 传输完成后（达到设置的传输数据数量）会产生 DMA 中断，为下一次的通信正常，需要在 DMA 中断关闭 DMA 接收使能，清除 UART 状态标记

波特率生成

波特率生成模数 Baudrate 通过 UART_BDR 寄存器的位 BRR[14:0]设置。

波特率计算: UART 波特率= UART_CLK / (16xBaudrate)。UART_CLK 是 UART 工作时钟，UART_CLK 可选择 PLL 分频后时钟或 HSE 时钟。

每次配置波特率寄存器均会清零内部计数器重新生成波特率信号。

通信要求发射器和接收器使用相同的波特率。

寄存器自身产生的计算误差:波特率越小则误差越小，系统时钟频率越小误差越大，系统时钟误差越大计算误差越大。

通信允许的波特率偏差范围:8/(11*16)=4.5%。

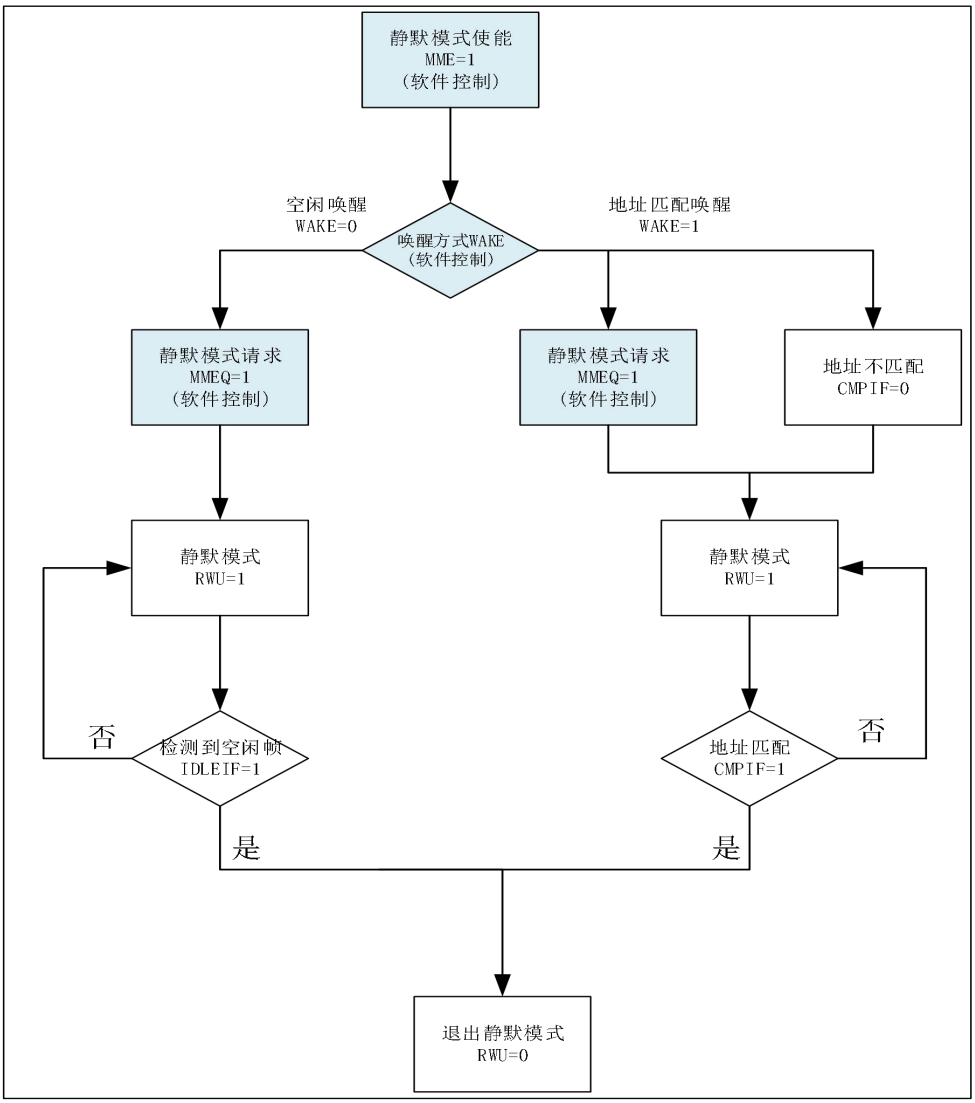
本系统支持自动波特率匹配，同步段字符为 0x55。首先配置自适应使能(UART_BDR 寄存器的位 ABREN=1)，波特率检测同步段 8 位，会在同步段字符通信完毕后自动更新 Baudrate，且可通过寄存器 UART_BDR 读出。这里注意，接收同步段并自动匹配波特率，该字符接收完毕后不

会出接收满中断。更新完 Baudrate 后，自适应使能位自动拉低。如需要再次同步，需要将使能位再置 1 (接收同步段不出接收中断，不保存数据)。

接收器多处理通信（静默模式）

接收器静默唤醒是一种硬件机制，此机制的作用是利用硬件检测消除了处理不重要信息字符的软件开销。允许 UART 接收器忽略用于不同 UART 接收器的信息中的字符。

进入与退出静默模式流程：



当 RWU 置位时，禁止接收数据，不会设置与接收器有关的状态标志。

空闲检测唤醒（WAKE=0）

静默模式下的空闲检测唤醒时，检测到空闲帧则 RWU 被自动清除，退出静默模式。但是注意，当前唤醒的空闲标志不拉高，不产生空闲中断。唤醒后接收器会在下一个字符接收时设置相应状态标志。

配置流程:配置接收使能，配置多处理器模式，配置空闲唤醒，没有检测到空闲帧 (足够的 1)则数据不被接收，检测到空闲帧后，退出静默模式，后续数据被接收并产生中断。

注意：当 MME 置 1 而 RWU 不为 1 时，空闲检测检测到空闲帧时，会正常出空闲标志以及空闲中断。

地址匹配唤醒（WAKE=1）

地址匹配唤醒是当接收器检测到已接收字符的最高位 (在 DM=0 模式中是第 8 位 bit[7],DM=1 模式中是第 9 位 bit[8])为逻辑 1 时，该数据被认为地址数据，然后硬件将地址数据 (除去最高位)与 UART_MUTE 寄存器的 ADDR[7:0]比较。如果地址数据与 ADDR[7:0]中的地址一致，RWU 标志被自动清除，同时地址匹配标志 (CMPIF)被拉高。唤醒后接收器相关状态标志及中断在当前字符即可以被设置。 (注:地址匹配唤醒后，匹配标志 CMPIF 要与 MME 一起清除。如果 MME 仍然需要保持为 1，则 CMPIF 不能清除，否则会马上进入静默模式 RWU=1，无法接收正常数据。)

配置流程:配置接收使能，配置多处理器模式，配置地址唤醒，接收到地址数据 (最高位=1)，如果地址数据与寄存器 ADDR[7:0]中的地址一致则配置关闭多处理器模式,数据接收并产生中断后续所有数据 (第 9 位=0)均能被接收并产生中断，直到下一次接收到地址数据，地址不匹配，则打开多处理器模式，则后续所有数据均不被接收，直到下一个地址数据，依次循环应用。

4.4.3 硬件流控制功能（RTS、CTS 和 RS485）

RTS 流控制

使能 RTS 只需要配置 UART_CR3 寄存器的位 RTSE=1。

使能 RTS 后，只要 UART 接收器准备好接收新数据，便会将 RTS 变为有效 (低电平)。当接收寄存器已满时，会将 RTS 变为无效 (高电平)。当读取寄存器 UART_RXDR 中的数据后，硬件自动拉低 RTS 使其有效，表示 UART 可以接收新的数据。

CTS 流控制

使能 CTS 只需要配置 UART_CR3 寄存器的位 CTSE=1。

使能 CTS 后，发送器会在发送下一帧前检查 CTS 信号。如果 CTS 有效 (连接到低电平)则发送下一帧数据，否则不会进行发送。如果数据传输过程中 CTS 变为无效 (高电平)，则当前数据传输完成后，发送器才停止。

RS485 驱动

RS485 驱动流程：

1) 使能 RS485 驱动信号 (UART_CR3 寄存器的位 DEM=1)。
2) 配置信号使能时间 (UART_CR3 寄存器的位 DEAT[4:0])和信号禁止时间(UART_CR3 寄存器的位 DEDT[4:0])，以波特率采样时钟为单位。
3) 配置 DE 信号极性 (UART_CR3 寄存器的位 DEP)。
4) 往发送数据寄存器 UART_TXDR 写入数据，DE 信号有效拉高。
5) 然后等待使能时间，UART 发送器才开始数据。
6) 当数据发送完后 (停止位结束)，等待到达禁止时间 DE 信号无效。

使能时间为 DE 信号有效与起始位开始(起始位下降沿)之间的时间；禁止时间为发送的消息中最后一个停止位结束与 DE 信号无效之间的时间。时间 (T)的计算公式为:

$$T=A/ (BAUDRATE*16)S$$

其中，A 为使能时间寄存器或禁止时间寄存器的值。

4.4.4 UART 初始化结构体详解

LL 库函数为每个外设都建立了一个初始化结构体，比如 LL_UART_InitTypeDef，结构体成员用于设置外设工作参数，并由外设初始化配置函数，比如 LL_UART_Init()调用，这些设定参数将会设置外设相应的寄存器，达到配置外设工作环境的目的。

初始化结构体和初始化库函数配合使用对用户编程十分有用，初始化结构体定义在 bs32f0xx_ll_uart.h 文件中，初始化函数定义在 bs32f0xx_ll_uart.c 文件中，编程时我们可以结合这两个文件内注释使用。

```
typedef struct
{
  uint32_t BaudRate;                  //波特率
  uint32_t DataWidth;                 // 数据位宽
  uint32_t StopBits;                  // 停止位宽
  uint32_t Parity;                    // 奇偶校验位
  uint32_t TransferDirection;         // 发送接收传输使能
  uint32_t HardwareFlowControl;       // 硬件控制流
} LL_UART_InitTypeDef;
```

BaudRate：波特率设置。一般设置为 9600、19200、115200。bs32f0xx_ll_uart.h 文件中可以找到可选的波特率典型值。

DataWidth：数据帧字长，可选 8 位或 9 位。通过设定 UART_CR1 寄存器的 DM 位的值实现。一般使用 8 位模式，8 位模式奇偶校验无效；9 位模式时，若奇偶校验使能，则最高位为奇偶校验位，否则最高位为数据。MSB 最高位在前有效时，奇偶校验位仍然在数据的最高位。

StopBits：停止位设置，通过设定 UART_CR1 寄存器的 STOP 位可选 2 位与 1 位，默认为 1 个停止位。

Parity：奇偶校验控制选择，通过设定 UART_CR1 寄存器的 PCE 位可选是否使能奇偶校验，默认不使能。

TransferDirection：UART 模式选择，通过设定 UART_CR1 寄存器的 TE 位与 RE 可选 UART 工作模式如下：

```
#define LL_UART_DIRECTION_NONE              0x00000000U
#define LL_UART_DIRECTION_RX                UART_CR1_RE
#define LL_UART_DIRECTION_TX                UART_CR1_TE
#define LL_UART_DIRECTION_TX_RX             (UART_CR1_TE |UART_CR1_RE)
```

HardwareFlowControl：硬件流控制选择，只有在硬件流控制模式才有效，通过操作 UART_CR3 寄存器的 RTSE 与 CTSE 位可选硬件流控制模式如下：

```
#define LL_UART_HWCONTROL_NONE                  0x00000000U
#define LL_UART_HWCONTROL_RTS                   UART_CR3_RTSE
#define LL_UART_HWCONTROL_CTS                   UART_CR3_CTSE
#define LL_UART_HWCONTROL_RTS_CTS               (UART_CR3_RTSE | UART_CR3_CTSE)
```

4.4.5 UART 编程要点（UART 接发通信）

UART 只需两根信号线即可完成双向通信，对硬件要求低，使得很多模块都预留 UART 接口来实现与其他模块或者控制器进行数据传输，在硬件设计时，注意还需要一根“共地线”。

我们经常使用 UART 来实现控制器与 PC 之间的数据传输。这使得我们调试程序非常方便，比如我们可以把一些变量的值、函数的返回值、寄存器标志位等等通过 UART 发送到串口调试助手，这样我们可以非常清楚程序的运行状态，当我们正式发布程序时再把这些调试信息去除即可。

我们不仅仅可以将数据发送到串口调试助手，我们还可以在串口调试助手发送数据给控制器，控制器程序根据接收到的数据进行下一步工作。

为利用 UART 实现开发板与电脑通信，需要用到一个 USB 转 UART 的 IC，我们可以选择 CH340 芯片来实现这个功能。

我们创建了两个文件：bs32f0xx_ll_uart.c 与 bs32f0xx_ll_uart.h 用以存放 UART 驱动程序及相关宏定义。

编程要点

- 使能 RX 和 TX 引脚 GPIO 时钟和 UART 时钟
- 初始化 GPIO，并将 GPIO 复用到 UART 上
- 配置 UART 参数
- 配置中断控制器并使能 UART 接收中断（接收由串口调试工具发送过来的数据）
- 使能 UART
- 在 UART 接收中断服务函数实现数据接收和发送

重定向 prinft 和 scanf 函数

```
//重定向 c 库函数 printf 到串口，重定向后可使用 printf 函数
int fputc(int ch, FILE *f)
{
  ...
}

//重定向 c 库函数 scanf 到串口，重写向后可使用 scanf、getchar 等函数
int fgetc(FILE *f)
{
  ...
}
```

在 C 语言标准库中，fputc 函数是 printf 函数内部的一个函数，功能是将字符 ch 写入到文件指针 f 所指向文件的当前写指针位置，简单理解就是把字符写入到特定文件中。我们使用 UART 函数重新修改 fputc 函数内容，达到类似“写入”的功能。

fgetc 函数与 fputc 函数非常相似，实现字符读取功能。在使用 scanf 函数时需要注意字符输入格式。

还有一点需要注意的，使用 fputc 和 fgetc 函数达到重定向 C 语言标准库输入输出函数必须在 MDK 的工程选项把“UseMicroLIB”勾选上，MicoroLIB 是缺省 C 库的备选库，它对标准 C 库进行了高度优化使代码更少，占用更少资源。

为使用 printf、scanf 函数需要在文件中包含 stdio.h 头文件。

4.5 DMA 模块介绍

DMA(Direct Memory Access)—直接存储器存取，是单片机的一个外设，它的主要功能是用来搬数据，但是不需要占用 CPU，即在传输数据的时候，CPU 可以干其他的事情，好像是多线程一样。数据传输支持从外设到存储器或者存储器到存储器，这里的存储器可以是 SRAM 或者是 FLASH。

BS32F030XX 的 DMA 寄存器分为两部分：功能相关寄存器与通道控制数据结构配置寄存器。BS32F030XX 拥有 32 个 DMA 通道。每个通道数据结构寄存器包含 4 个：通道控制寄存器、源数据结束地址寄存器、目的数据结束地址寄存器以及一个保留寄存器。因此其主数据与备用数据结构寄存器在 DMA_SRAM 中的所占空间分别为 0x4003_0000-0x4003_01FF 与 0x4003_0200-0x4003_03FF。

DMA 模块当任一通道传输完成或者发生总线错误均会产生中断，每个通道有单独的中断使能，传输完成中断标记和错误中断标记。

- 通道中断使能有效，当通道完成所配置的传输数量后，产生传输完成中断（一个脉宽 hclk 周期的高电平脉冲），同时对应通道传输完成中断标志位拉高。
- 通道中断使能有效，当 DMA 访问未定义地址空间时，产生错误中断（一个脉宽为 hclk 周期的高电平脉冲），同时对应通道传输错误中断标志位拉高。

4.5.1 DMA 结构说明

DMA 控制器独立于内核，属于一个单独的外设，从编程的角度，熟悉其三部分内容即可。

DMA 请求

若有外设想要通过 DMA 传输数据，必须先给 DMA 控制器发送 DMA 请求，DMA 收到请求信号之后，控制器会给外设一个应答信号，当外设应答后且 DMA 控制器收到应答信号之后，就会启动 DMA 的传输，直到传输完毕。

通道

BS32F030 具有 6 个 CPU 中断入口，每个入口可以接收多个外设的请求，但是同一时间根据优

先级只能接收一个，具体关系如下图所示：

外设	CPU 中断入口					
	DMA_INT0	DMA_INT1	DMA_INT2	DMA_INT3	DMA_INT4	DMA_INT5
ADC	ADC	--	--	--	--	--
SPI1	SPI1_RX	SPI1_TX	--	--	--	--
SPI2	--	--	SPI2_RX	SPI2_TX	--	--
IIC1	--	--	IIC1_RX	IIC1_TX	--	--
IIC2	--	--	IIC2_RX	IIC2_TX	--	--
UART1	--	--	USART1_TX	USART1_RX	--	--
UART2	USART2_TX	USART2_RX	--	--	--	--
TIM1	TIM1_CH1	TIM1_CH2	TIM1_CH4 TIM1_TRG TIM1_COM	TIM1_UPDATE	TIM1_CH3	--
TIM3	TIM3_UPDATE	--	--	TIM3_CH1	TIM3_CH3	TIM3_CH2 TIM3_CH4 TIM3_TRG
TIM15	TIM15_CH1	TIM15_CH2	--	--	TIM15_TRG TIM15_COM	TIM15_UPDATE
TIM16	--	--	TIM16_CH1	TIM16_COM	TIM16_UPDATE	--
TIM17	--	TIM17_COM	TIM17_CH1		TIM17_UPDATE	--

同时，DMA 请求入口/通道与 DMA 请求的优先级关系如下：

DMA 请求入口及中断优先级	优先级类型	DMA 请求	CPU 中断入口说明
0	可编程	ADC	DMA_INT0
1	可编程	SPI1_RX	DMA_INT0
2	可编程	SPI1_TX	DMA_INT1
3	可编程	SPI2_RX	DMA_INT2
4	可编程	SPI2_TX	DMA_INT3
5	可编程	IIC1_RX/IIC2_RX	DMA_INT2
6	可编程	IIC1_TX/IIC2_TX	DMA_INT3
7	可编程	TIM1_CH1	DMA_INT0
8	可编程	TIM1_CH2	DMA_INT1
9	可编程	TIM1_CH3	DMA_INT4
10	可编程	TIM1_CH4	DMA_INT2
11	可编程	TIM1_TRG/TIM1_COM	DMA_INT2
12	可编程	TIM1_UPDATE	DMA_INT3
13	可编程	TIM3_CH1	DMA_INT3
14	可编程	TIM3_CH2	DMA_INT5
15	可编程	TIM3_CH3	DMA_INT4
16	可编程	TIM3_CH4	DMA_INT5
17	可编程	TIM3_TRG/TIM3_COM	DMA_INT5

18	可编程	TIM3_UPDATE	DMA_INT0
19	可编程	TIM15_CH1	DMA_INT0
20	可编程	TIM15_CH2	DMA_INT1
21	可编程	TIM15_TRG/TIM15_COM	DMA_INT4
22	可编程	TIM15_UPDATE	DMA_INT5
23	可编程	TIM16_CH1	DMA_INT2
24	可编程	TIM16_COM	DMA_INT3
25	可编程	TIM16_UPDATE TIM17_UPDATE	DMA_INT4
26	可编程	TIM17_CH1	DMA_INT2
27	可编程	TIM17_COM	DMA_INT1
28	可编程	USART1_RX	DMA_INT3
29	可编程	USART1_TX	DMA_INT2
30	可编程	USART2_RX	DMA_INT1
31	可编程	USART2_TX	DMA_INT0

注意：

1. I2C1_RX/I2C2_RX 与 I2C1_TX/I2C2_TX 共用同一通道，不能同时配置 DMA 通道使能有效，同时配置 DMA 时只响应 I2C1 的请求。
2. TIM16_UPDATE 与 TIM17_UPDATE 共用同一通道，不能同时配置各自 DMA 通道使能有效，同时配置时 DMA 只响应 TIM16 的请求。
3. TIM1_TRG 与 TIM1_COM 共用同一通道，不能同时配置 TIM1 的 TRG,COM 触发 DMA 使能有效，未做控制，都响应。
4. TIM15_TRG 与 TIM15_COM 共用同一通道，不能同时配置 TIM15 的 TRG,COM 触发 DMA 使能有效，未做控制，都响应。

仲裁器

当发生多个 DMA 通道请求时，就意味着有先后相应处理的顺序问题，即使在同一中断入口下，DMA 通道请求分为两个阶段。第一阶段属于软件阶段，可以在 DMA_CPSR_CPS 与 DMA_CPCR_CPC：通道优先级设置寄存器中设置，分为高和低两个优先级。第二阶段属于硬件阶段，若两个或以上的 DMA 通道请求设置的优先级一样，则他们优先级取决于通道入口编号，编号越小优先权越高，比如通道 0 高于 1。

在以上的优先级设置前提下，针对新的更高优先级的事件发生，我们可以通过设置 DMA_CCFGRx 寄存器的 ART 位来设置仲裁率：每个 DMA 请求传输 N 个数据后重新根据最新优先级序列来传输数据（对于 ADC/UART/IIC/SPI 模块，ART 位必须配置为 0）。

4.5.2 DMA 数据配置

使用 DMA，最核心的是配置要传输的数据，包括数据从哪里来，要到哪里去，传输的数据的单位是什么，要传多少数据等等。

- DMA 数据传输类型有三种：外设与存储器间传输、外设与外设的传输、存储器间传输。我们通过配置对应的源地址和目的地址来实现不同类型选择。
- DMA 传输数据量和数据位宽通过通道 DMA_CCFGRx 寄存器配置。要想数据传输正确，源和目的地址存储的数据宽度必须一致；要完成数据传输，还需要正确设置两边数据指针的地址增量。
- DMA 数据传输进度可以通过查询标志位或通过中断的方式鉴别。传输完成与传输错误都会有相应的标志位，若使能该类型中断后，则会产生中断。

4.5.3 DMA 初始化结构体详解

我们的 LL 库函数为每个外设都建立了一个初始化结构体 LL_xxx_InitTypeDef(xxx 为外设名称)，结构体成员用于设置外设工作参数，并由 LL 库函数 LL_xxx_Init()调用这些设定参数进入设置外设相应的寄存器，达到配置外设工作环境的目的。

结构体 LL_xxx_InitTypeDef 与库函数 LL_xxx_Init()配合使用可以对该外设应用自如。结构体定义在该外设头文件中，库函数定义在该外设对应.c 文件中，编程时我们结合这两个文件内注释使用。

DMA_InitTypeDef 初始化结构体

```
typedef struct
{
  uint32_t SrcEndAddress;     //源数据结束地址,范围 0~0xffffffff
  uint32_t DstEndAddress;     //目的数据结束地址,范围 0~0xffffffff
  uint32_t Mode;              //工作模式选择
  uint32_t SrcIncMode;        //源地址增量
  uint32_t DstIncMode;        //目的地址增量
  uint32_t SrcDataSize;       //源数据位宽
  uint32_t DstDataSize;       //目的数据位宽,必须和源数据位宽一致
  uint32_t NbData;            //传输数据长度,以数据位宽为单位,范围 0~1023,代表 1~1024 传输
  uint32_t Priority;          //传输优先级,相同优先级的情况下 DMA 通道号小的优先级越高
  uint32_t ArbitrationRate;   //传输仲裁率,注意不要给低优先级通道设置太高的仲裁率,否则 DMA
控制器就不能及时响应高优先级的通道请求
  uint32_t ChannelDataStruct; //通道控制数据结构选择,主数据结构或备用数据结构
} LL_DMA_InitTypeDef;
```

- **SrcEndAddress：源数据结束地址**

控制器在执行一个 DMA 传输之前，必须配置源数据结束地址（最后一个数据的起始地址）。

- **DstEndAddress：目的数据结束地址**

控制器在执行一个 DMA 传输之前，必须配置目的数据结束地址（最后一个数据的起始地址）。

- **Mode：工作模式**

DMA 工作模式由控制配置寄存器 DMA_CCFGRx 的 MODE 位配置。

工作模式	模式特点	传输数据长度（单位字/半字/字节可选）	传输类型
INVALID	无效数据结构，不进行数据传输；		
BASIC	一次传输完成后硬件关闭通道使能；	最大为 1024	外设与存储间传输：发生仲裁后，需要外设再次发送请求信号，继续传输；
PING_PONG	交替使用主备用数据结构，直到读到无效数据结构或软件编程关闭通道使能； 产生传输完成中断后，需重新配置通道数据结构配置寄存器；	可大于 1024	
Peripheral-scatter-gather（Mode1/Mode2）	通道备用数据结构寄存器配置值存储于系统 SRAM 中；每次 DMA（N 个数据）传输完成后，控制器通过主数据结构传输自动配置备用数据结构寄存器；	可大于 1024	
AUTOREQ	一次传输完成后硬件关闭使能；	最大为 1024	存储器间传输：一个请求信号可完成一次 DMA 传输，发生仲裁后控制器自动产生请求信号；
Memory-scatter-gather（Mode1/Mode2）	通道备用数据结构寄存器配置值存储于系统 SRAM 中；每次 DMA（N 个数据）传输完成后，控制器通过主数据结构传输自动配置备用数据结构寄存器；	可大于 1024	

此部分内容较多，有关各模式的详细讲解见 4.5.4 小节。

- **SrcIncMode：源地址增量**

源地址增量由控制配置寄存器 DMA_CCFGRx 的 SRCINC 位配置。

	源数据位宽：byte	源数据位宽：halfword	源数据位宽：word
源地址增量	b00= byte b01= halfword b10= word b11=无增量	b00= halfword b01= halfword b10= word b11=无增量	b00=word b01= word b10= word b11=无增量

- **DstIncMode：目的地址增量**

源地址增量由控制配置寄存器 DMA_CCFGRx 的 DSTINC 位配置，与源地址增量的配置规则一致。

- **SrcDataSize：源数据位宽**

源数据位宽由控制配置寄存器 DMA_CCFGRx 的 SRCSIZE 位配置，源数据位宽可配置选项如下：

b00= byte
b01= halfword
b10= word
b11= word

- **DstDataSize：目的数据位宽**

目的数据位宽由控制配置寄存器 DMA_CCFGRx 的 DSTSIZE 位配置，目的数据位宽必须与源数据位宽设置相同的值，配置规则也与源数据位宽规则一致。

- **NbData：传输数据长度**

传输数据长度由控制配置寄存器 DMA_CCFGRx 的 LEN 位配置，其传输数据长度可配置范围为 1-1024，优先于控制器进入仲裁过程。

- **Priority：传输优先级**

传输优先级由通道优先级设置寄存器 DMA_CPSR_CPS 写入 1 来配置对应通道高优先级；通过 DMA_CPCR_CPC 置 1 来使对应通道为默认低优先级，同一优先级，通道号小的优先级更高。

- **ArbitrationRate：传输仲裁率**

传输仲裁率由控制配置寄存器 DMA_CCFGRx 的 ART 位配置，其每个 DMA 请求可配置传输数据为 1-1024 个，对于 ADC/UART/IIC/SPI 模块，ART 必须配置为 0，也就是一个数据。

- **ChannelDataStruct：通道控制数据结构选择**

通道控制数据结构选择由通道主备用设置寄存器 DMA_CPASR 配置，写 1 选择备用数据结构；通道主备用清除寄存器 DMA_CPACR 对应通道位写入 1 则选择该通道主数据结构。

在控制器完成以下情况时，硬件切换 DMA_CPACR[C]位的值：

- 对于 memory scatter-gather 或 peripheral scatter-gather DMA 周期类型，主数据结构传输完成后，切换为选择备用数据结构，备用数据结构指定的所有 DMA 传输完成后，再次切换为选择主数据结构。
- 对于一个 ping-pong DMA 周期类型，主数据结构或备用数据结构指定的所有 DMA 传输完成后，切换到另一数据结构。

4.5.4 DMA 传输模式

4.5.4.1 INVALID

无效数据结构，该模式不进行数据传输；当响应 DMA 传输请求，对应通道 DMA 工作模式为 invalid 时，硬件关闭通道使能。

4.5.4.2 BASIC

外设与存储间或外设与外设间的基础传输模式，一次 DMA 传输最大数据长度为 1024（单位字/半字/字节可选）。

在这种工作模式下，控制器接收到 DMA 传输请求（优先级最高）后：

A. 控制器执行仲裁率 ART 或传输长度 LEN（根据较小的值）个 DMA 传输；

B. 传输长度 LEN 较小：DMA 控制器执行 LEN 个数据传输，传输完成后控制器产生中断，并标记 DMA_CCISR 寄存器对应 TCF 位；

C. 仲裁率 ART 较小：DMA 控制器执行 ART 个数据传输后暂停传输：

- 如果一个更高优先级的通道正在请求相应，控制器响应那个通道；
- 该通道外设或者软件发出一个请求给控制器，控制器响应请求后继续之前的传输，重复步骤 C 直到 LEN 个数据传输完成；

一次 DMA 传输完成，产生 DMA 传输完成中断，硬件关闭通道使能，硬件清零该通道 DMA_CCFGRx 寄存器的 LEN 和 MODE 位。

4.5.4.3 AUTOREQ

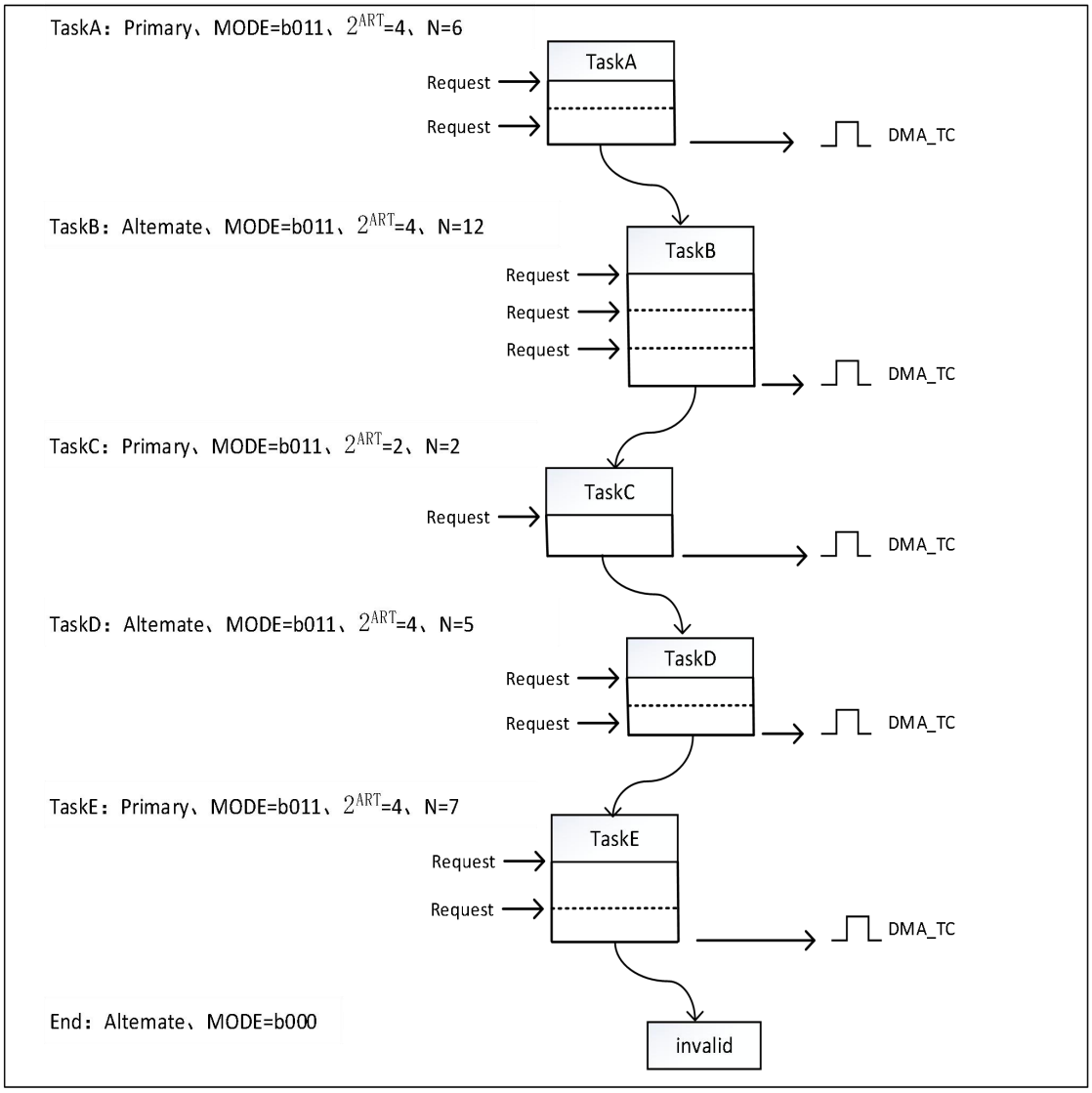
用于存储器间数据长度小于 1024（单位字/半字/字节可选）的传输，配置软件请求启动 DMA 传输。

一次 DMA 传输完成，产生 DMA 传输完成中断，硬件关闭通道使能，硬件清零该通道 DMA_CCFGRx 寄存器的 LEN 和 MODE 位。

4.5.4.4 PING_PONG

当外设与存储器间传输数据大于 1024（单位字/半字/字节可选）时，可选择使用该模式。在这种工作模式下，控制器交替使用主和备用数据结构；在执行完一次 DMA 传输后，产生传输完成中断，可重新软件编程 DMA_CCFGRx 控制通道数据结构寄存器；下次 DMA 传输硬件自动切换为另一数据结构；每次 DMA（LEN 个数据）传输完成后不会硬件关闭通道使能，直到读到一个无效的数据结构（硬件关闭通道使能），或者软件编程关闭通道使能。

传输示例



Task A

(1) 配置对应通道主数据/备用数据控制配置寄存器 DMA_CCFGRx 的 MODE=b011 即 PING_PONG 模式，仲裁率 ART，数据传输长度 LEN 用于 task A 与 task B。

(2) 当控制器接收到一个请求时（硬件），如果这个通道拥有最高优先级则流程继续。

(3) 控制器执行 DMA 传输 4 个数据。

(4) 控制器仲裁，当控制器再次接收到一个请求时，如果这个通道拥有最高优先级则流程继续。

(5) 控制器执行 DMA 传输剩余 2 个数据。

(6) 控制器设置对应通道传输完成中断信号为高持续一个 HCLK 周期，并进入到交替的流程。

在 task A 完成之后，处理器硬件置位（DMA_CPASR）选择备用数据结构并软件编程通道主数据结构的配置寄存器（DMA_CCFGRx）用于控制 task C。

在控制器接收到这个通道新的请求之后（优先级最高），task B 开始：

Task B

(7) 控制器执行 DMA 传输 4 个数据。

(8) 控制器仲裁，控制器在接收到这个通道的一个请求之后，如果这个通道拥有最高优先级则流程继续。

(9) 控制器接收一个请求并执行 DMA 传输 4 个数据。

(10) 控制器仲裁，控制器在接收到这个通道的一个请求之后，如果这个通道拥有最高优先级则流程继续。

(11) 控制器执行 DMA 传输剩余 4 个数据。

(12) 控制器设置对应通道传输完成中断信号为高持续一个 HCLK 周期，并进入到交替的流程。

在 task B 完成之后，处理器硬件置位（DMA_CPASR）选择主数据结构并软件编程通道备用数据结构的配置寄存器（DMA_CCFGRx），用于控制 task D。

在控制器接收到这个通道新的请求之后（优先级最高），task C 开始：

Task C

(13) 控制器执行 2 个 DMA 传输。

(14) 控制器设置对应通道传输完成中断信号为高持续一个 HCLK 周期，并进入到交替的流程。

在 task C 完成之后，处理器硬件置位（DMA_CPASR）选择备用数据结构并软件编程通道主数据结构的配置寄存器（DMA_CCFGRx），用于控制 task E。

在控制器接收到这个通道新的请求之后（优先级最高），然后 task D 开始：

Task D

(15) 控制器执行 DMA 传输 4 个数据

(16) 控制器仲裁，控制器在接收到这个通道的一个请求之后，如果这个通道拥有最高优先级则流程继续。

(17) 控制器执行 DMA 传输剩余的 1 个数据。

(18) 控制器设置对应通道传输完成中断信号为高持续一个 HCLK 周期，并进入到交替的流程。

在控制器接收到这个通道新的请求之后（优先级最高），然后 task E 开始：

Task E

(19) 控制器执行 DMA 传输 4 个数据

(20) 控制器仲裁，控制器在接收到这个通道的一个请求之后，如果这个通道拥有最高优先级则流程继续。

(21) 控制器执行剩余的 3 个 DMA 传输

(22) 控制器设置对应通道传输完成中断信号为高持续一个 HCLK 周期，并进入到交替的流程。

控制接收到这个通道一个新的请求之后（优先级最高），开启下一个 task：

End

因为处理器没有重新配置通道备用数据结构寄存器，在 task D 传输完成后，控制器配置通道备用数据结构寄存器 DMA_CCFGRx 的 MODE 位为 b000（无效数据结构），所以不进行数据传输，硬件关闭对应通道使能。

注：如果通道控制数据结构配置寄存器 DMA_CCFGRx 的 MODE 位配置为 b001，task E 是一个 basic DMA 周期类型，则 task E 传输完成后，设置传输完成中断信号为高持续一个 HCLK 周期，并且硬件关闭对应通道使能。

4.5.4.5 Memory-scatter-gather

用于存储器间数据长度大于 1024（单位字/半字/字节可选）的传输或分块传输，配置软件请求启动 DMA 传输。

在这种工作模式下，DMA 控制器使用通道备用数据结构进行存储器间的数据传输，通道备用数据结构寄存器配置值存储于系统 SRAM 中（主数据结构的 DMA_CSEARx 寄存器指向该地址），控制器通过主数据结构配置同一通道备用数据结构寄存器。

注意：

1. 通道主数据结构执行 DMA 传输来配置备用数据结构寄存器，因此主数据结构寄存器配置必须按照如下要求：

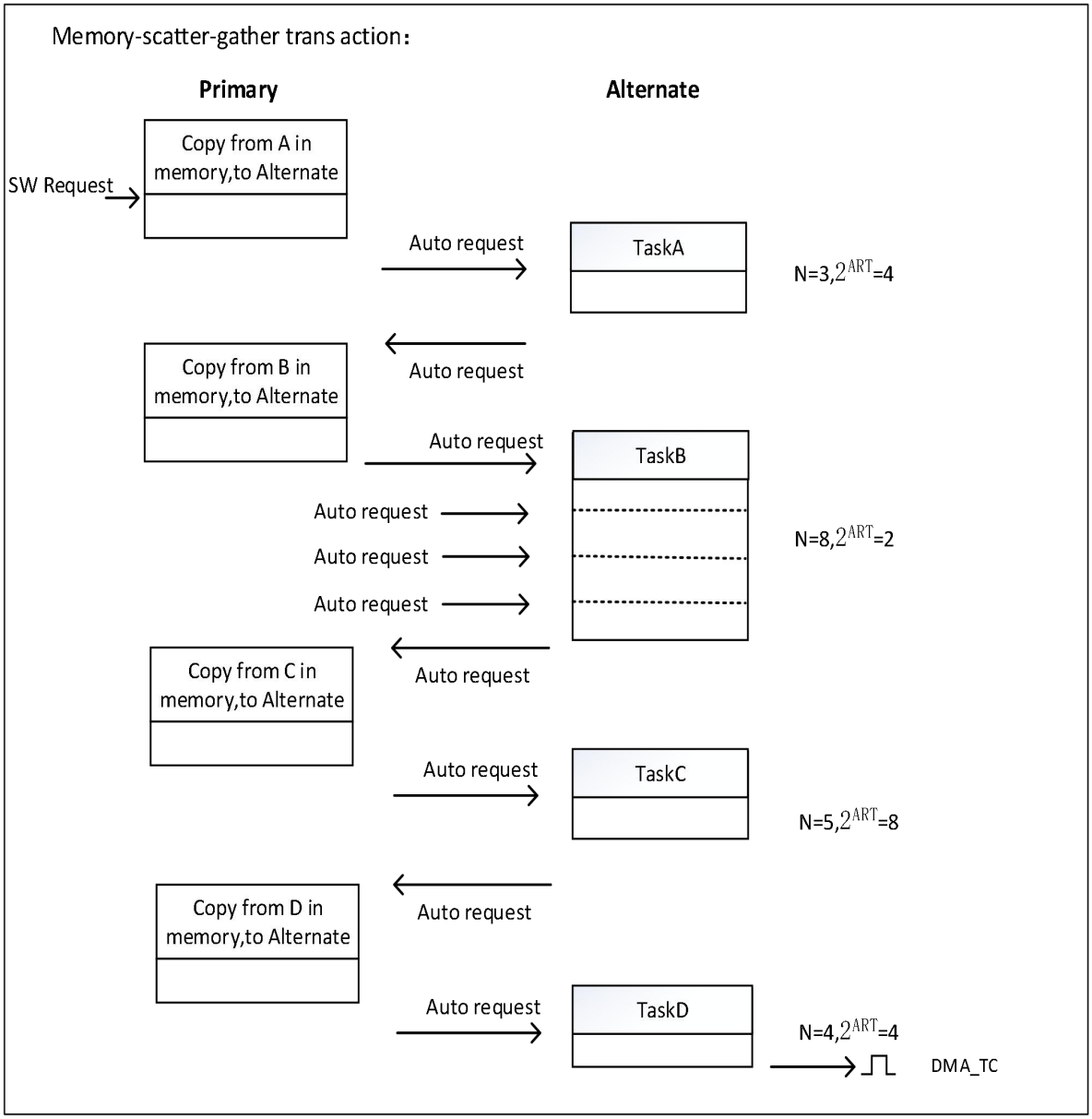
   主数据配置控制寄存器 DMA_CCFGRx 的 ART 配置为 4'b0010，N 为 4 的倍数；

   主数据结构的 DMA_CDEARx 寄存器低 4 位配置为 0xC (指向通道备用控制数据结构寄存器地址)；

2. 主数据结构执行 DMA 传输，按 0x0、0x4、0x8、0xC 的顺序写；
3. 最后一次备用数据结构必须配置为 Auto-Request 操作模式，整个 DMA 传输完成后，控制器产生 DMA 传输完成中断；
4. 如果 2ART=N,配置成 Basic 也可以正确传输，传输完成后控制器产生 DMA 传输完成中断。
5. 如果要再次开启传输，需先配置 DMA_CPACRx 相应位为 1，切换回主数据结构。

Memory-scatter-gather DMA 传输示例

配置主数据结构,使能 Task A,B,C 和 D 的拷贝操作: MODE= b100,2ART = 4,N=16。将备用数据结构的配置按照如下表所示的结构写入 Memory，该 Memory 的结束地址写入主数据结构的 DMA_CDEARx 寄存器



初始化

(1) 处理器配置通道主数据结构：

- 配置配置主数据结构寄存器 DMA_CCFGRx 的 MODE=b100 (Memory scatter/gather (使用主数据结构))，2^{ART} =4 (因为一个通道的数据结构寄存器是由 4Word 组成)，在这个例子中有 4 个 Task，所以数据长度 N 配置为 16，源数据和目标数据的位宽 Size=Word，地址增量 Inc=Word；
- 配置源数据结束地址寄存器为存储备用数据结构寄存器配置值的内存空间结束地址；
- 配置目的数据结束地址寄存器为备用通道结构寄存器结束地址：
  备用通道结构寄存器结束地址=备用数据控制寄存器基地址指针+{通道号，4'hc}

(2) 处理器写 task A,B,C,D 使用的数据结构配置值到通道主数据结构源数据结束地址寄存器配置的存储地址空间中。

(3) 处理器使能 DMA 及相应通道。

当控制器接收到一个 DMA 请求（优先级最高），memory scatter-gather 传输开始，传输流程如下所示：

Primary,copy A

1) 控制器执行 DMA 传输 4 个数据，这些传输对 task A 写备用数据结构寄存器。

2) 控制器产生通道 auto-request，然后仲裁。

如果当前通道优先级最高，则继续传输流程：

Task A, alternate

3) 控制器 DMA 传输 3 个数据，控制器产生通道 auto-request，然后仲裁。

如果当前通道优先级最高，则继续传输流程：

Primary,copy B

4) 控制器执行 DMA 传输 4 个数据，这些传输对 task B 写备用数据结构。

5) 控制器产生通道 auto-request，然后仲裁。

如果当前通道优先级最高，则继续传输流程：

Task B, alternate

6) 控制器执行 DMA 传输 2 个数据，产生通道 auto-request，然后仲裁。

7) 如果当前通道优先级最高，控制器继续执行 DMA 传输两个数据，产生通道 auto-request，然后仲裁。

8) 如果当前通道优先级最高，控制器继续执行 DMA 传输两个数据，产生通道 auto-request，然后仲裁。

9) 如果当前通道优先级最高，控制器继续执行 DMA 传输剩余两个数据，完成传输，产生通道 auto-request，然后仲裁。

如果当前通道优先级最高，则继续传输流程：

Primary,copy C

10) 控制器执行 DMA 传输 4 个数据，这些传输对 task C 写备用数据结构。

11) 控制器产生通道 auto-request，然后仲裁。

如果当前通道优先级最高，则继续传输流程：

Task C, alternate

12) 控制器执行 DMA 传输 5 个数据，完成传输，控制器产生通道 auto-request，然后仲裁。

如果当前通道优先级最高，则继续传输流程：

Primary,copy D

13) 控制器执行 DMA 传输 4 个数据，这些传输对 task D 写备用数据结构。

14) 控制器设置主数据结构的 MODE=b000（无效数据结构）。

15) 控制器产生通道 auto-request，然后仲裁。

如果当前通道优先级最高，则继续传输流程：

Task D, alternate

控制器采用一个 AUTOREQ 周期类型执行 task D：控制器执行 DMA 传输 4 个数据，完成传输后，设置通道传输完成中断信号为高持续一个 HCLK 周期，并且硬件关闭对应通道使能。

4.5.4.6 Peripheral scatter-gather

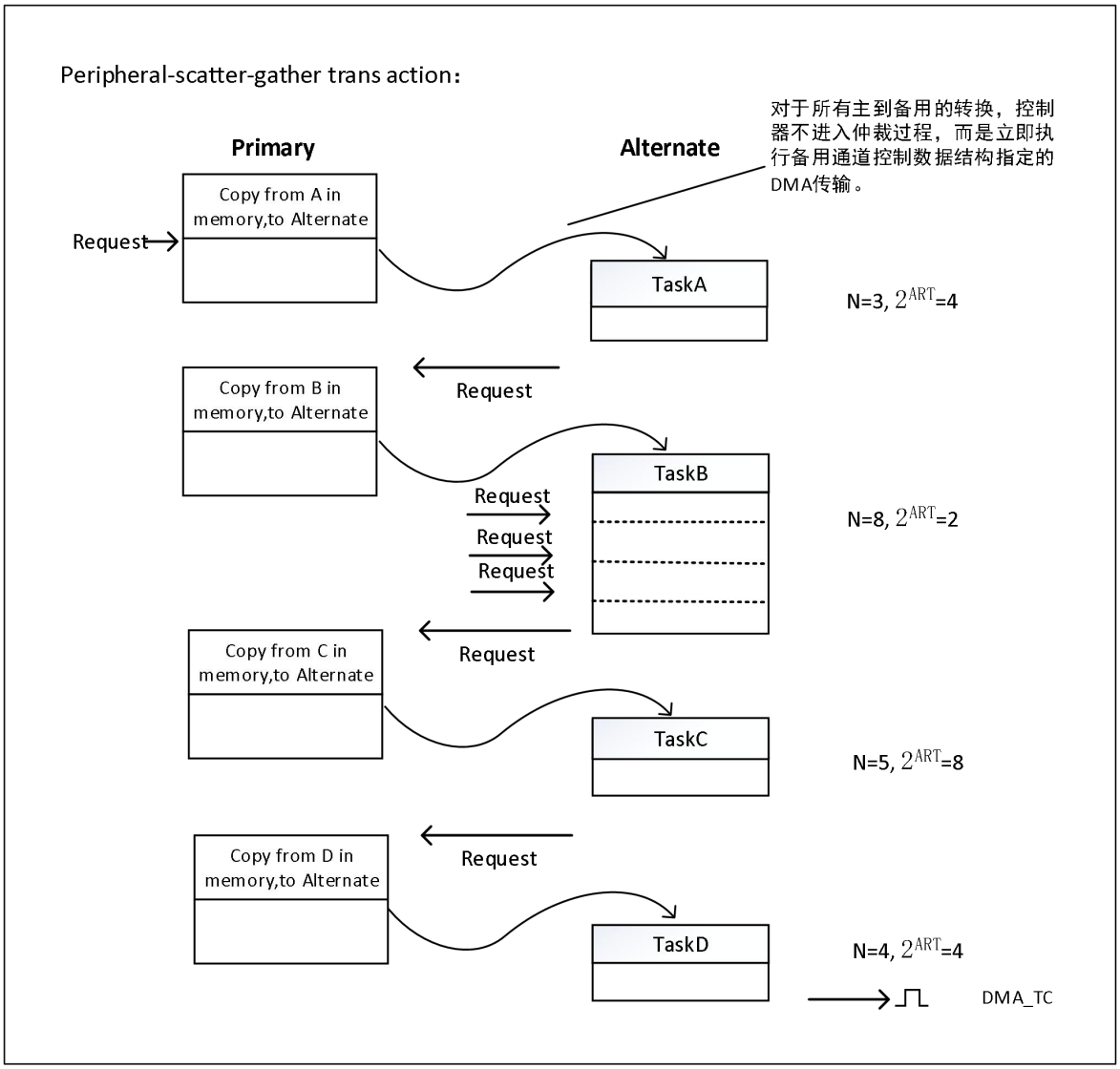
当外设与存储器间传输数据大于 1024（单位字/半字/字节可选）时，可选择使用该模式。在这种工作模式下，DMA 控制器使用通道备用数据结构进行外设与存储器间的数据传输，通道备用数据结构寄存器配置值存储于系统 SRAM 中（主数据结构的 DMA_CSEARx 寄存器指向该地址），控制器通过主数据结构配置同一通道备用数据结构寄存器。

注意：

1. Peripheral scatter-gather 工作模式，通道主数据结构执行 DMA 传输来配置备用数据结构寄存器，因此主数据结构寄存器配置必须按照以下请求：
   - 主数据结构的 DMA_CCFGRx 寄存器 ART 配置为 4'b0010，N 为 4 的倍数；
   - 主数据目的 DMA_CDEARx 寄存器低四位配置为 0xC（指向通道备用控制数据结构寄存器地址）；
2. 主数据结构执行 DMA 传输，按 0x0、0x4、0x8、0xC 的顺序写；
3. 最后一次备用数据结构必须配置为 basic 操作模式，整个 DMA 传输完成后，控制器产生 DMA 传输完成中断。
4. 如果要再次开启传输，需先配置 DMA_CPACRx 相应位为 1，切换回主数据结构。

Peripheral scatter-gather DMA 传输示例

配置主数据结构,使能 Task A,B,C 和 D 的拷贝操作: MODE= b110,2ART = 4,N=16。将备用数据结构的配置按照如下表所示的结构写入 Memory，该 Memory 的结束地址写入主数据结构的 DMA_CDEARx 寄存器



初始化

1. 处理器配置通道主数据结构：
   - 配置 DMA_CCFGRx 的 MODE=b110 (Peripheralscatter-Gather Mode)，2ART =4 (因为一个通道的数据结构寄存器是由 4word 组成)，在这个例子中有 4 个 Task，所以 N 配置为 16，源数据和目标数据的位宽 Size=Word，地址增量 Inc=Word；
   - 配置源数据结束地址寄存器为存储备用数据结构寄存器配置值的内存空间结束地址；
   - 配置目的数据结束地址寄存器为备用通道结构寄存器结束地址：
   - 备用通道结构寄存器结束地址=备用数据控制寄存器基地址指针+{通道 N*0x10+0x0c}
2. 处理器写 task A,B,C,D 使用的数据结构配置值到通道主数据结构源数据结束地址寄存器配置的存储地址空间中。
3. 处理器使能 DMA 及相应通道。

当控制器接收到一个 DMA 请求（优先级最高），外设 scatter-gather 传输开始，传输流程如下所示：

Primary,copy A

1. 控制器执行 DMA 传输 4 个数据，这些传输对 task A 写备用数据结构。

不发生仲裁，控制器继续执行 task A：

Task A, alternate

2. 控制器执行 DMA 传输 4 个数据，完成传输。

在外设发出一个新的请求之后（优先级最高），则继续传输流程：

Primary,copy B

3. 控制器执行 DMA 传输 4 个数据，这些传输对 task B 写备用数据结构。

不发生仲裁，控制器继续执行 task B：

Task B, alternate

4. 控制器执行 DMA 传输 2 个数据，然后仲裁。
5. 当控制器接收到一个请求时（优先级最高），控制器执行 DMA 传输两个数据，然后仲裁。
6. 当控制器接收到一个请求时（优先级最高），控制器执行 DMA 传输两个数据，然后仲裁。
7. 当控制器接收到一个请求时（优先级最高），控制器执行 DMA 传输剩余两个数据，完成传输。

注：为了使控制器完成这个 task，外设必须发出额外的三个请求。

在外设发出一个新的请求之后（优先级最高），则继续传输流程：

Primary,copy C

8. 控制器执行 DMA 传输 4 个数据，这些传输对 task C 写备用数据结构。

不发生仲裁，控制器继续执行 task C：

Task C, alternate

9. 控制器执行 DMA 传输 5 个数据，完成传输。

在外设发出一个新的请求之后（优先级最高），则继续传输流程：

Primary,copy D

10. 控制器执行 DMA 传输 4 个数据，这些传输对 task D 写备用数据结构。
11. 控制器设置主数据结构的控制配置寄存器 DMA_CCFGRx 的模式选择位 MODE=b000（无效数据结构）。

不发生仲裁，控制器继续执行 task D：

Task D, alternate

控制器采用一个 basic DMA 周期类型执行 task D：控制器执行 DMA 传输 4 个数据，完成传输后，设置设置通道传输完成中断信号为高持续一个 HCLK 周期，并且硬件关闭对应通道使能。

4.5.5 DMA 应用流程与注意事项

外设与 memory 间传输

一、配置 DMA 寄存器

1．DMA_CCDBPR 寄存器:配置指向通道数据结构基地址 (0x40030000)；

2．DMA_CFGR 寄存器:配置使能 DMA 控制器；

3．DMA_CENSR 寄存器:配置使能 DMA 通道；

4．DMA_CIER 寄存器:配置使能中断；

5．DMA_CPSR 寄存器:设置优先级，默认为低优先级；

6．在 basic 和 auto-request 模式下，可配置 DMA_CPASR 寄存器，选择通道数据结构，默认使用主数据结构；

二、配置通道控制数据结构寄存器（存放于 DMA SRAM 空间）：

1．DMA_CSEARx 寄存器:配置指向源数据结束地址；

2．DMA_CDEARx 寄存器：配置指向目的数据结束地址；

3．DMA_CCFGRx 寄存器：配置 DMA 工作模式，每次 DMA 传输长度，仲裁率，源数据/目的数据位宽，源地址/目的地址增量；

4．Unused：保留；

三、配置外设 DMA 使能；

四、DMA 传输完成后产生传输完成中断，可读通道完成标志位，判断传输完成通道；如需再次传输，需重新配置通道使能及通道数据结构寄存器（DMA 传输长度，工作模式）。

memory 到 memory 的传输：

一、配置 DMA 寄存器，同上；

二、配置通道控制数据结构寄存器（存放于 DMA SRAM 空间），同上：

MODE 位可配置为 AUTOREQ 或 Memory scatter/gather 周期类型（如果一次传输大 于 1024（单位字、半字、字节可选），可选择 Memory scatter/gather 周期类型）；

三、配置 DMA_CRGR 寄存器，产生 DMA 软件请求信号；

四、DMA 传输完成后产生传输完成中断，可读通道完成标志位，判断传输完成通道；如需再次传输，需重新配置通道使能及通道数据结构（DMA 传输长度，工作模式）。

注意：

1. 当访问数据位宽为字时，配置源数据结束地址/目的数据结束地址时必须按 4 字节对齐。
2. 当访问数据位宽为半字时，配置源数据结束地址/目的数据结束地址时按 2 字节对齐。
3. 数据从数据起始地址到结束地址正向传输。

4. 当 DMA 访问未定义地址空间时，产生错误中断，传输错误中断标志寄存器 DMA_CEISR 相应位置 1，硬件关闭通道使能。
5. 禁止使用 DMA 擦写 FLASH。
6. SPI/UART/I2C/ADC 外设的 DMA 数据传输，必须配置仲裁率 ART 为 0:

   如配置错误，通信未完成即产生中断，通道传输完成标志相应位置1，通道传输错误标志为0；

   如果未重新配置通道控制寄存器和通道使能，DMA对应通道将不再响应外设请求；外设为发送数据，将不再发送数据；外设为接收数据，接收溢出，相应外设溢出标志位置位。
7. TIM 模块的 DMA 传输：

   当一次请求只传输一笔数据时，ART 必须等于 0，数据地址可以直接指向目的寄存器，也可以通过 TIMx_DMAR 控制寄存器映射；

   当一次请求出传输多笔数据时，通过 TIM DMA 控制寄存器映射，注意 DBL 的配置与 R 保持一致 (ART 为 2 的指数次，TIMX_DCR 的位 DBL[4:0]只能配置成偶数)；
8. 在使用 ping-pong 模式和 scatter-gather 模式时：最后一次传输设置为 basic 模式时，传输完成后硬件不会自动切换通道主备用结构状态，开启下次传输时注意软件配置 DMA_CPASR、DMA_CPACR 寄存器；对于 ping-pong 模式，传输过程中不需要再次配置通道使能，按需配置通道数据结构寄存器 (DMA 传输长度，工作模式)。
9. 配置 DMA_CxxSR/DMA_CxxCR 寄存器时注意不能先回读再配置相应的位，直接按下述方式配置:

   例如：

   清除 BIT[0]通道使能:DMA_CENCR=0X1；

   配置BIT[0]通道使能:DMA_CENSR=0X1；

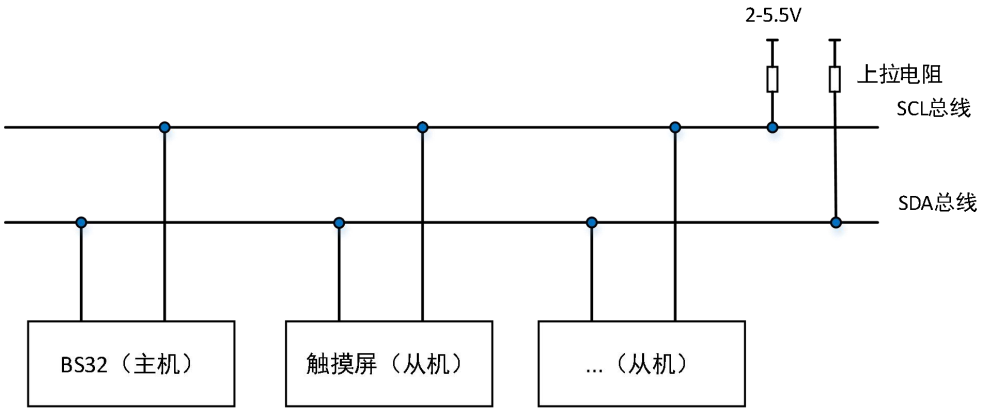
4.6 IIC 模块介绍

IIC（Inter-Integrated Circuit）：它是一种串行通信总线，使用多主从架构，具有引脚少，硬件实现简单等优点。

4.6.1 IIC 协议概述

物理层

IIC 通讯设备之间的常用连接方式如下：



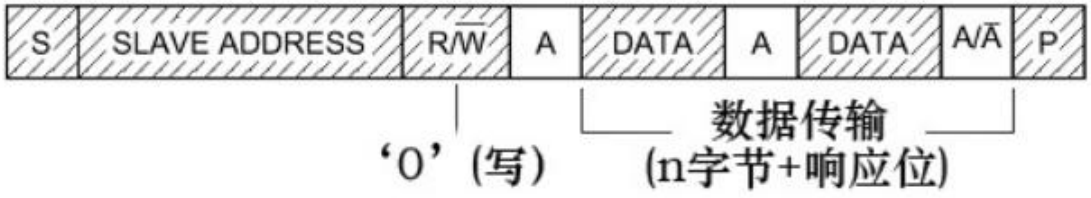
它的物理层有如下特点：

1. 它是一个支持设备的总线。“总线”指多个设备共用的信号线。在一个 IIC 通讯总线中，可连接多个 IIC 通讯设备，支持多个通讯主机及多个通讯从机。
2. 一个 IIC 总线只使用两条总线线路，一条双向串行数据线（SDA），一条串行时钟线（SCL）。数据线即用来表示数据，时钟线用于数据收发同步。
3. 每个连接到总线的设备都有一个独立的地址，主机可以利用这个地址进行不同设备之间的访问。
4. 总线通过上拉电阻接到电源。当 IIC 设备空闲时，会输出高阻态，而当所有设备都空闲，都输出高阻态时，由上拉电阻把总线拉成高电平。
5. 多个主机同时使用总线时，为了防止数据冲突，会利用仲裁方式来决定哪个设备占用总线。
6. 具有三种传输模式：标准模式传输速率为 100kbit/s，快速模式为 400kbit/s，超快速模式 1 Mbit/s（以 BS32F030 为例）。
7. 连接到相同总线的 IC 数量受到总线的最大电容限制。

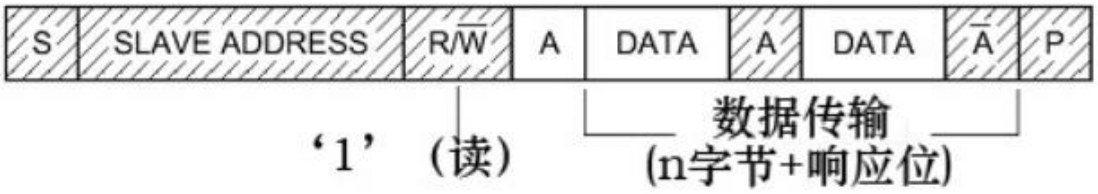
协议层

IIC 的协议定义了通讯的起始于停止信号、数据有效性、响应、仲裁、时钟同步和地址广播等环节。

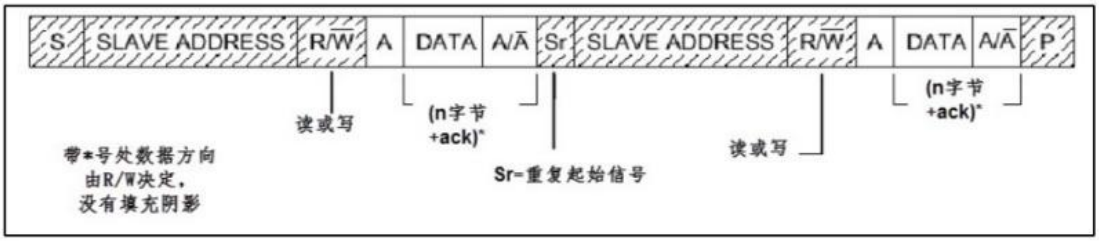
IIC 通讯过程：主机写数据到从机



IIC 通讯过程：主机读取从机数据



IIC 通讯过程：复合格式



图例：数据由主机传输至从机　S：传输开始信号

SLAVE_ADDRESS: 从机地址

数据由从机传输至主机　R/W：传输方向选择位，1 为读，0 为写

A/Ā：应答(ACK)或非应答(NACK)信号

P：停止传输信号

这些图表示的是主机和从机通讯时，SDA 线的数据包序列。

其中 S 表示由主机的 IIC 接口产生的传输起始信号（S），这时连接到 IIC 总线上的所有从机都会接收到这个信号。

起始信号产生后，所有从机就开始等待主机接下来广播的从机地址信号（SLAVE_ADDRESS）。在 IIC 总线上，每个设备的地址都是唯一的，当主机广播的地址与某个设备地址相同时，这个设备就被选中了，没被选中的设备将会忽略之后的数据信号。根据 IIC 协议，这个从机地址可以是 7 位或 10 位。

在地址位之后，是传输方向的选择位，该位为 0 时，表示后面的数据传输方向是由主机传输至

从机，即主机向从机写数据。该位为 1 时，则相反，即主机由从机读数据。

从机接收到匹配的地址后，主机或从机会返回一个应答（ACK）或非应答（NACK）信号，只有接收到应答信号后，主机才能继续发送或接收数据。

➢ **写数据**

若配置的方向传输位为“写数据”方向，广播完地址，接收到应答信号后，主机开始正式向从机传输数据（DATA），数据包的大小为 8 位，主机每发送完一个字节数据，都要等待从机的应答信号（ACK），重复这个过程，可以向从机传输 N 个数据，这个 N 没有大小限制。当传输结束时，主机向从机发送一个停止传输信号（P），表示不再传输数据。

➢ **读数据**

若配置的方向传输位为“读数据”方向，广播完地址，接收到应答信号后，从机开始向主机返回数据（DATA），数据包大小为 8 位，从机每发送完一个数据，都会等待主机的应答信号（ACK），重复这个过程，可以返回 N 个数据，这个 N 没有大小限制。当主机希望停止接收数据时，就向从机返回一个非应答信号（NACK），则从机自动停止数据传输。

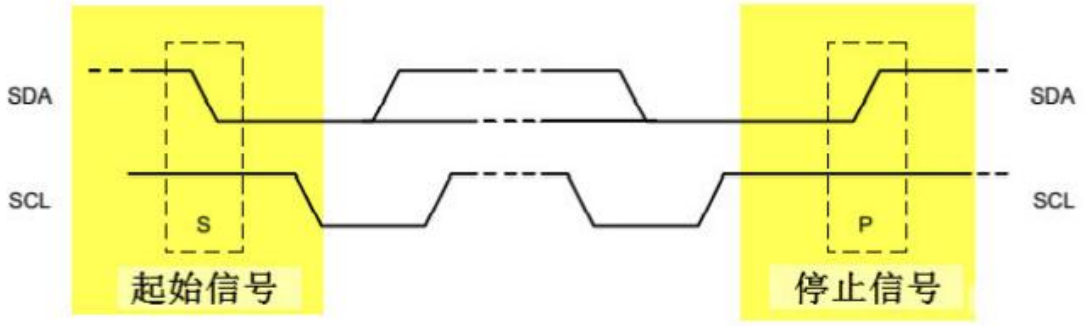
➢ **读和写数据**

除了基本的读写外，IIC 通讯更常用的是复合格式，该传输过程有两次起始信号（S）。一般在第一次传输中，主机通过 SLAVE_ADDRESS 寻找到从设备后，发送一段“数据”，这段数据通常用于表示从设备内部的寄存器或存储器地址（注意区分其与 SLAVE_ADDRESS 的区别）；在第二次传输中，对该地址的内容进行读或写。也即：第一次通讯告诉从机读写地址，第二次通讯是读写的实际内容。

以上通讯流程中包含的各个信号分解如下：

◆ **通讯的起始和停止信号**

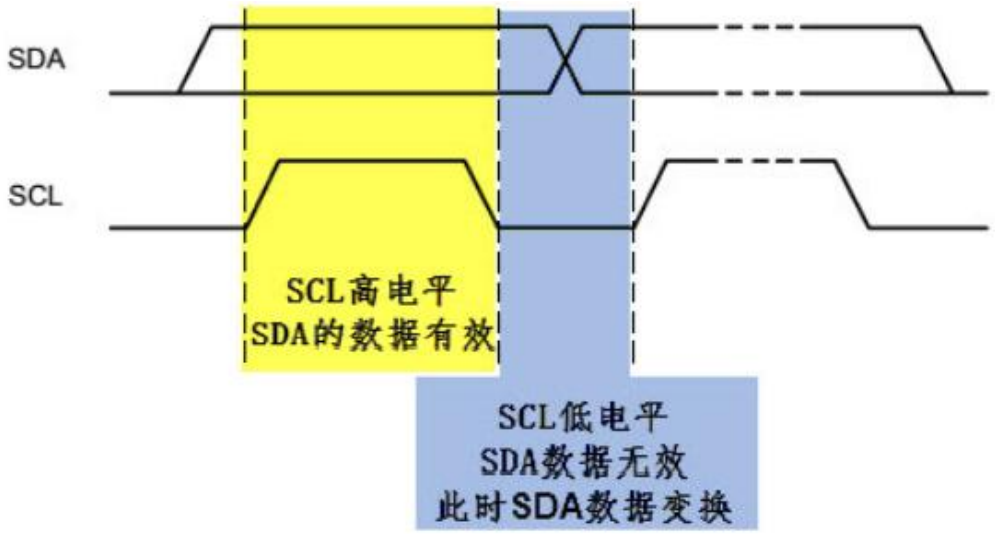
前面提到的起始（S）与停止（P）信号是两种特殊状态，如下图所示。当 SCL 线是高电平时 SDA 线从高电平向低电平切换，这个情况表示通讯的起始。当 SCL 是高电平时 SDA 线由低电平向高电平切换，表示通讯的停止。起始和停止信号一般由主机产生。



◆ **数据有效性**

IIC 使用 SDA 信号线来传输数据，使用 SCL 信号线来进行数据同步，如下图所示。SDA 数据线在 SCL 的每个时钟周期传输一位数据。传输时，SCL 为高电平的时候 SDA 表示的数据有效，即此时的 SDA 为高电平时表示数据“1”，为低电平时表示数据“0”。当 SCL 为低电平时，SD

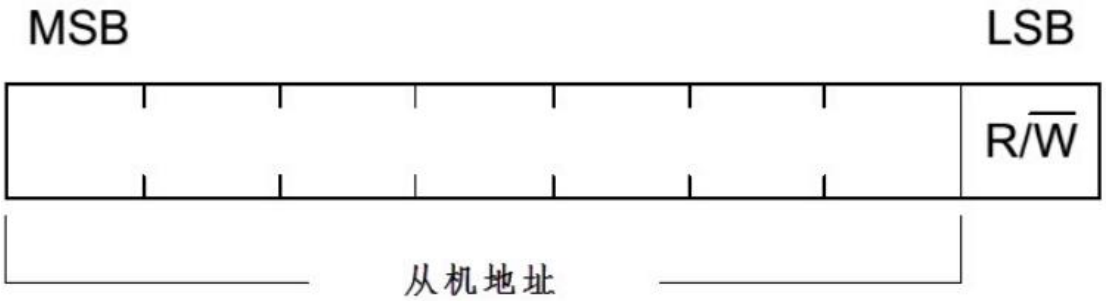
A 的数据无效，一般在这个时候 SDA 进行电平切换，为下一次表示数据做好准备。



每次数据传输都以字节为单位，每次传输的字节数不受限制。

◆ **地址与数据方向**

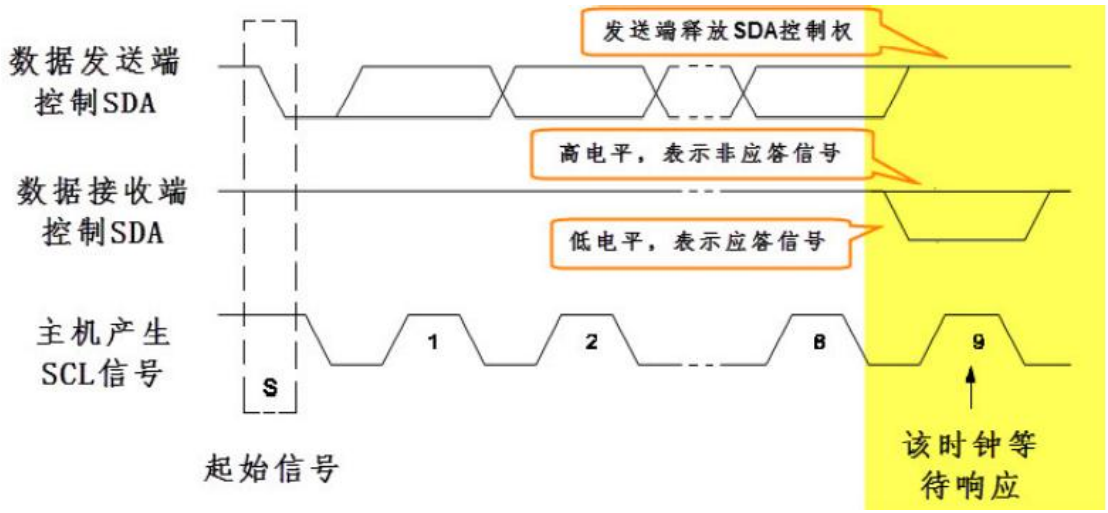
IIC 总线上的每个设备都有自己独立的地址，主机发起通讯时，通过 SDA 信号线发送设备地址（SLAVE_ADDRESS）来查找从机。IIC 协议规定设备地址可以是 7 位或 10 位，实际中 7 位的地址应用较多。紧跟设备地址的一个数据位用以表示数据传输方向，它是数据方向位（R/W），第 8 位或者第 11 位。数据方向位为“1”时表示主机读取从机数据，该位为“0”时表示主机向从机写数据，如下图所示。



读数据方向时，主机会释放对 SDA 信号线的控制，由从机控制 SDA 信号线，主机接收信号，写数据方向时，SDA 由主机控制，从机接收信号。

◆ **响应**

IIC 的数据和地址传输都带响应。响应包括“应答（ACK）”和“非应答（NACK）”两种信号。作为数据接收端时，当设备（无论主从机）接收到 IIC 传输的一个字节数据或地址后，若希望对方继续发送数据，则需要向对方发送“应答（ACK）”信号，发送方会继续发送下一个数据；若接收端希望结束数据传输，则向对方发送“非应答（NACK）”信号，发送方接收到该信号后会产生一个停止信号，结束信号传输，如下图所示。



传输时主机产生时钟，在第 9 个时钟时，数据发送端会释放 SDA 的控制权，由数据接收端控制 SDA，若 SDA 为高电平，表示非应答信号（NACK），低电平表示应答信号（ACK）。

4.6.2 BS32 的 IIC

若我们直接控制 BS32 的两个 GPIO 引脚，分别用作 SCL 与 SDA，按照上述信号的时序要求，直接像控制 LED 灯那样控制引脚的输出（若是接收数据时则读取 SDA 电平），就可以实现 IIC 通讯。同样，若我们按照 UART 的要求去控制引脚，亦可实现 UART 通讯。因此只要遵守协议，就是标准的通讯。

由于直接控制 GPIO 引脚电平产生通讯时序时，需要内核控制每个时刻的引脚状态，所以称之为“软件模拟协议”方式。

相对的，还有“硬件协议”方式，BS32 的 IIC 片上外设专门负责实现 IIC 通讯协议，只要配置好该外设，它就可以自动根据协议要求产生通讯信号，收发数据并缓存起来，内核只要检测该外设的状态和数据寄存器，就能完成数据收发。这种由硬件外设处理 IIC 协议的方式减轻了内核的工作。

4.6.2.1 BS32 功能特性

BS32F030 内置 2 个 IIC 外设，具有如下特性：

1. 主和从模式（与 SPI 不同，无寄存器配置主从，而是由设备发送时钟或起始信号后硬件确定为主机）。
2. 标准模式（100kbit/s）、快速模式（400kbit/s）、超快速模式（1Mbit/s）
3. 7 位寻址模式（BS32F030 目前仅支持 7 位寻址）
4. 广播呼叫
5. 多个 7 位从地址（两个从设备地址寄存器，1 个具有可配置的掩码位段）
6. 主机时序时间参数可配置

7. 可选时钟延长

8. 软件复位（寄存器关闭使能，复位内部状态机和相关时序）

9. 支持 DMA 传输

10. 地址匹配时从 Deepsleep Mode 唤醒

支持 7 位寻址（主从均可）与广播呼叫（从机独占）

该电路仅支持 7 位寻址方式，不兼容 10 位寻址。

主机在生成起始信号后，发送要寻址的从机地址码，如果接收到与之匹配的从机发出的应答信号，则进行通信。如果没有接收到来自从机发出的应答信号，主机将生成停止信号，并回到总线空闲状态。

从机支持广播呼叫，GCEN=1 时，对地址 7’b0000000 应答。与主机通讯的从机地址由软件配置。

延长时钟低电平（主从均可）

主机默认具有时钟低电平功能。

该电路是基于中断服务子程序来完成通信的。在 IIC 每发送或接收到一个字节后都会产生一个中断信号，告知 CPU 进行相应的处理，从而实现数据的发送或接收。当中断服务子程序的处理时间过长，导致不能正常的处理数据，必须延长低电平的时间达到使通信正常的目的。以及在通信完成时，拉低时钟线来决定后续的通信状态。

- 在第一个时钟之前没有准备好要发送的数据，会在第 9 个时钟下降沿和第一个时钟上升沿之间的低电平进行延长低电平。直到写入了 I2C_TXDR 为止。
- 在第八个时钟下降沿之前没有读取上一笔的数据，会在第 8 个时钟下降沿和第九个时钟上升沿之间的低电平进行延长低电平。直到读取了 I2C_RXDR 为止。
- 在非重载模式（reload=0）且软件结束模式（autoend=0）下，完成最后一笔数据传输时，会在第九个时钟下降沿和第一个时钟上升沿之间的低电平进行延长低电平。直到起始位或停止位置 1。
- 在重载模式下（reload=1），完成最后一笔数据传输时，会在第 9 个时钟下降沿和第一个时钟上升沿之间的低电平进行延长低电平。直到 NBYTES 写入一个非零值。

从机具有可选的延长时钟低电平功能。默认 NOSTRETCH=0，可延长时钟低电平。

该电路也是基于中断服务子程序来完成通信的，所以也可通过延长低电平的时间达到使通信正常的目的。

- ADDR 标志置 1 时：接收到的地址与其中一个使能的从地址匹配。会在第 8 个时钟下降沿和第 9 个时钟上升沿之间的低电平进行延长低电平。通过软件将 ADDRCF 位置 1 以清零 ADRR 标志时，将释放该时钟延展。
- 在第一个时钟之前没有准备好要发送的数据，会在第 9 个时钟下降沿和第一个时钟上升沿之间的低电平进行延长低电平。直到写入了 I2C_TXDR 为止。
- 在第八个时钟下降沿之前没有读取上一笔的数据，会在第 8 个时钟下降沿和第九个时钟上

升沿之间的低电平进行延长低电平。直到读取了 I2C_RXDR 为止。

- 在从器件 NACK/ACK 控制模式下，完成最后一笔数据传输时，会在第 8 个时钟下降沿和第 9 个时钟上升沿之间的低电平进行延长低电平。直到 NACK 位写入一个非零值。

检测从机时钟低电平延长功能（主机独占）

主机在刚发出 scl_out 时钟高电平时检测输入的 scl_in 状况，如果输入 scl_in 为低，说明从机时钟存在低电平延长现象，主机的时钟保持高不变，直到从机延长低电平结束，主机继续把高电平完成再拉低。

支持 NACK/ACK 控制模式（从机独占）

通常情况下，字节计数器为禁止状态。当从机希望控制是否对接收到的数据字节进行应答时，可以使用 NACK/ACK 控制功能。将 I2C_CR1 寄存器中的 SBC（从字节控制）位置 1 且将 I2C_CR2 寄存器的 RELOAD（重载模式）位置 1，使能 NACK/ACK 控制模式。此外 NACK/ACK 控制模式必须在允许延长时钟线时使用。

在该模式下，接受完 NBYTES 中所编程字节数的数据之后，从而延长 SCL 信号的第 8 个和第 9 个脉冲之间的低电平时间。TCR （传输完成等待重载）标志将置 1，并且 TCIE（传输完成中断使能）置 1 时将生成中断。然后通过配置 I2C_CR2 寄存器中的 NACK 位来决定是否应答。

不使用该模式时，从机接收时，一直发送 ACK。

不延长时钟低电平检测溢出异常（从机独占）

当 I2C_CR1 寄存器中的 NOSTRETCH = 1 时，I2C 从器件不会延长 SCL 信号。

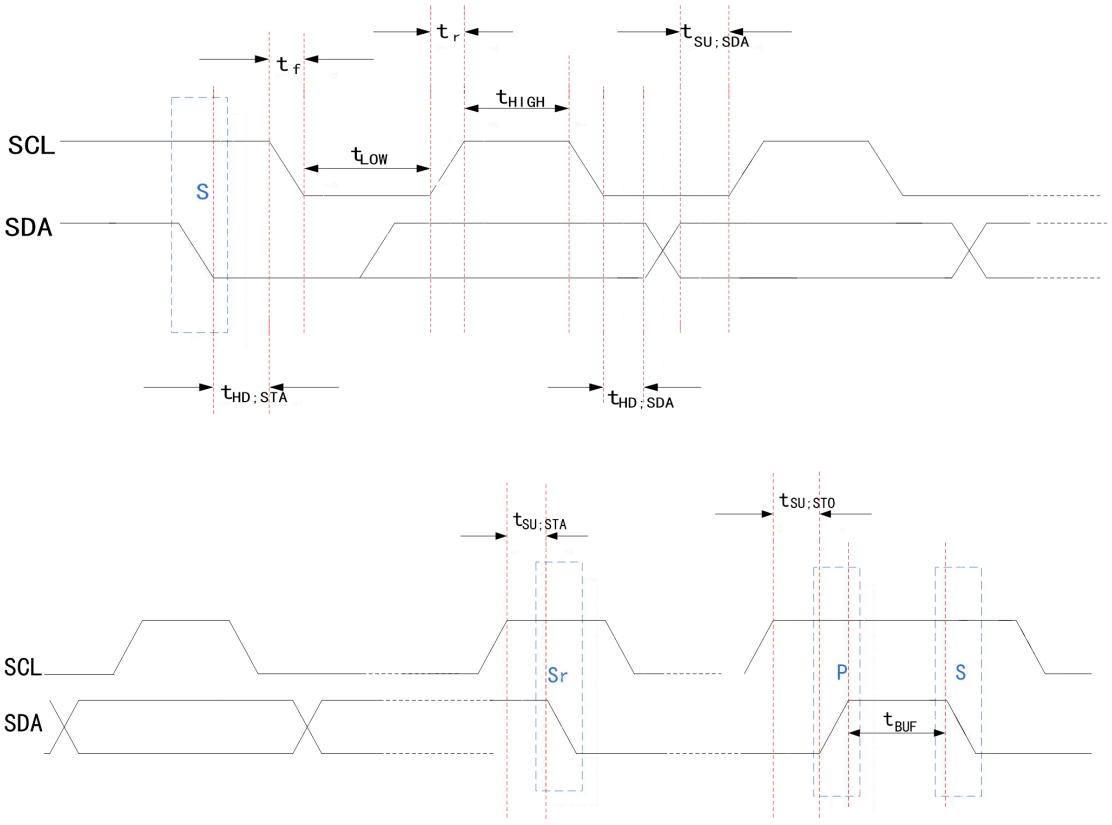
ADDR 标志置 1 时，不会延长 SCL 时钟。

发送时，必须在与发送数据对应的第一个 SCL 脉冲出现之前，向 I2C_TXDR 寄存器入数据。否则，会发生下溢，I2C_ISR 寄存器中的 OVR 标志将置 1，如果 I2C_CR1 寄存器中的 ERRIE 位置 1 ，还将生成中断。当第一次数据发送开始而 STOPF 位仍置 1（尚未清零）时，OVR 标志也将置 1 。因此，如果写入下一次传输要发送的第一个数据后才清零上一次传输的 STOPF 标志，则会出现 OVR 状态，甚至对于待发送的第一个数据也是如此。

接收时，必须在下一个数据字节的第 9 个 SCL 脉冲（ACK 脉冲）出现之前，从 I2C_RXDR 寄存器读取数据。否则，会发生上溢，I2C_ISR 寄存器中的 OVR 标志将置 1，如果 I2C_CR1 寄存器中的 ERRIE 位置一，还将生成中断。

通信时间可配置（主机独占）

为满足 IIC 通信要求，以下参数均可配置。具体可见 IIC_TIMINGR 的寄存器说明。



- SCL 时钟的高电平t_{HIGH}：由 SCLH 配置。
- SCL 时钟的低电平t_{LOW}：由 SCLL 配置。
- SDA 数据的保持时间$t_{HD;SDA}$： 由 SDADEL 配置。
- SDA 数据的建立时间 $t_{SU;SDA}$： SCLL 和 SDADEL 配置必须满足$t_{SU;SDA}$。
- 起始条件的保持时间$t_{HD;STA}$： 由 SCLH 配置。
- 重复起始条件的建立时间$t_{SU;STA}$： 由 SCLL 配置。
- 重复起始条件的建立时间$t_{SU;STA}$： 由 SCLL 配置。
- 停止和起始条件之间的总线空闲时间t_{BUF}： 由 SCLL 配置。

中断事件管理

为保证正常通信，电路中具有多个中断管理事件。打开相应使能配置才有效，具体配置参见寄存器说明。

主机模式

发送数据寄存器为空中断（txis）：第一个时钟第一个时钟时还没准备好当笔要发送的数据。产生在第 9 个 scl 下降沿后，发送数据寄存器为空（txe=1）时。非重载模式最后一笔数据发送完成后不产生此中断。

传输完成中断（tc）：传输完成(TC)或传输完成等待重载(TCR)。产生在非重载模式最后一笔数据完成或重载模式的 255 个数据完成的第 9 个 scl 下降沿。

接收数据寄存器不为空中断（rxne）：直到第八个时钟下降沿时还没有读取完上一笔接收到的数据。产生在第 8 个 scl 下降沿，接收数据寄存器为满（rxne=1）时。

接收到否定应答中断（nack）：接收到否定应答。产生在第 9 个 scl 下降沿。

停止位检测中断（stop）：主机已经发出停止位。

错误中断（errir）：仲裁丢失（ARLO）或总线错误检测（BERR）。传输过程中发生仲裁丢失或总线错误则产生该中断。

从机模式

从机除了包含主机的全部中断外，还独占地址匹配中断。

地址匹配中断（addr）：接收到的地址与使能的从机设备地址之一匹配时。产生在第 8 个 scl 下降沿（不含起始信号），地址匹配（addr=1）时。

仲裁模式可选

主机模式

总线 SCL 上升沿时，主机输出的 sda_out 与总线 SDA 输入的 sda_in 不同，主机丢失仲裁。主机一直在检测仲裁是否丢失，丢失仲裁将发生错误中断。如果将仲裁复位使能打开，内部状态机将复位。

从机模式

当 SDA 线上发送高电平但在 SCL 上升沿却采样到低电平时，会检测到仲裁丢失。从机将在数据阶段和数据应答阶段检测仲裁是否丢失，丢失仲裁将释放 SCL 和 SDA。

地址匹配时从系统停止模式唤醒（从机独占）

地址匹配中断 addr 在正常模式下由 pclk 产生，而在系统停止模式时并 I2C_CR1 寄存器中的 WUPEN 位置 1 时由 scl_in 时钟产生。被寻址时，IIC 能够从系统停止模式中唤醒 MCU。

地址匹配的情况下，MCU 唤醒时间内，IIC 延长 SCL 使其处于低电平。当软件清除 ADDR 标志时，此延长被释放，传输正常进行。因此必须在使能时钟低电平延长功能时才能确保该该唤醒功能。

支持软件复位（主机）软硬件复位（从机）

主机模式

可通过 I2C_CR1 寄存器中的 PE 位清零来执行软件复位。在这种情况下，IIC 线 SCL 和 SDA 被释放。内部状态机复位，通信控制位和状态位恢复为其复位值。配置寄存器不受影响。下面列出受影响的寄存器位：

1. I2C_CR2 寄存器：START 和 STOP。

2. I2C_ISR 寄存器：BUSY 、TXE 、TXIS、RXNE、ADDR 、NACKF、TCR 、TC、STOPF、BERR 和 ARLO。

发生仲裁丢失或总线错误时，可通过将 I2C_CR2 寄存器中的 ERR_RST_EN 位置 1 来复位内部状态机，释放 SCL 和 SDA，寄存器不受到影响。

系统复位会复位整个 IIC 模块。

从机模式

可通过 I2C_CR1 寄存器中的 PE 位清零来执行软件复位。在这种情况下，IIC 线 SCL 和 SDA 被释放。内部状态机复位，通信控制位和状态位恢复为其复位值。配置寄存器不受影响。下面列出受影响的寄存器位：

1. I2C_CR2 寄存器：START 和 STOP 和 NACK。

2. I2C_ISR 寄存器：BUSY、TXE、TXIS、RXNE、ADDR、NACKF、TCR、TC、STOPF、BERR、ARLO 和 OVR。

接收到开始信号、地址不匹配和仲裁丢失硬件会自动复位内部时序（其中开始信号复位不包括状态机），寄存器不受到影响。

系统复位会复位整个 IIC 模块。

支持 DMA 读写请求（主从均可）

支持在发送数据为空时，发出 DMA 写请求；接收数据为满时，发出 DMA 读请求。该功能需要使能延长时钟低电平功能。

4.6.2.2 BS32F030 IIC 结构分析

- **通讯引脚**

IIC 的所有硬件架构都是由 SCL 与 SDA 线展开的。BS32 芯片上有多个 IIC 外设，它们的 IIC 通讯信号引出到不同的 GPIO 引脚上，使用时必须配置到这些指定的引脚。关于 GPIO 引脚的复用，以规则书为准。

- **时钟控制逻辑**

SCL 线的时钟信号，由 IIC 接口根据时序寄存器（TIMINGR）控制，控制的参数主要为时钟时序。配置 IIC 的 TIMINGR 可以修改通讯速率相关的参数：

1. 主模式下的时序预分频（PRESC）：用于对 IIC 时钟进行预分频，以生成用于数据建立和保持计数器以及 SCL 高电平和低电平计数器的时钟周期 t_presc=（PRESC+1）*t_i2cclk。
2. 从模式数据最高位和 NACK 控制的建立时间（SCLDEL）：用于在 SDA 边沿和 SCL 上升沿之间生成延时 t_scldel。如果 NOSTRETCH=0（使能时钟延长），且在发送数据的最高位或控制 NACK 模式发送应答位时，SCL 线的低电平时间将在 t_scldel 期间延长。t_scldel=（SCLDEL+1）*t_presc。
3. 主模式数据保持时间（SDADEL）：用于在 SCL 下降沿和 SDA 边沿之间生成延时 t_sdadel。t_sdadel=（SDADEL+1）*t_presc。
4. 主模式 SCL 高电平周期：用于在主模式下生成 SCL 高电平周期 t_sclh=（SCLH+1）*t_presc。
5. 主模式 SCL 低电平周期：用于在主模式下生成 SCL 低电平周期 t_scll=（SCLL+1）*t_presc。SCLL 需保证同时满足数据保持时间和数据建立时间。

SCL 高电平周期与 SCL 低电平周期之和的倒数即可认为是 IIC 通讯速率。

➢ **数据控制逻辑**

IIC 的 SDA 信号主要连接到数据移位寄存器上，数据移位寄存器的数据来源及目标是数据寄存器（DR）、地址寄存器（OAR）以及 SDA 数据线。当向外发送数据的时候，数据移位寄存器以“数据寄存器”为数据源，把数据一位一位地通过 SDA 信号线发送出去；当从外部接收数据的时候，数据移位寄存器把 SDA 信号线采样到的数据一位一位地存储到“数据寄存器”中。当 BS32 的 IIC 工作在从机模式的时候，接收到设备地址信号时，数据移位寄存器会把接收到的地址与 BS32 的自身“IIC 地址寄存器”的值作比较，以便响应主机的寻址。BS32 的自身 IIC 地址可通过“自身地址寄存器”修改，支持同时使用两个 IIC 设备地址，两个地址分别存储于 OAR1 与 OAR2 中。

➢ **整体控制逻辑**

整体控制逻辑负责协调整个 IIC 外设，控制逻辑的工作模式根据我们配置的“控制寄存器（CR1/CR2）”的参数而改变。在外设工作时，控制逻辑会根据外设的工作状态修改“中断和状态寄存器 ISR”与“中断清零寄存器 ICR”，我们只需要读取这些寄存器相关的寄存器位，就可以了解 IIC 的工作状态。除此之外，控制逻辑还根据要求，负责控制产生 IIC 中断信号、DMA 请求及各种 IIC 的通讯信号（起始、停止、响应信号等）。

4.6.3 BS32 IIC 通讯过程

使用 IIC 外设通讯时，在通讯的不同阶段它会对“中断和状态寄存器 ISR”与“中断清零寄存器 ICR”的不同数据位写入参数，我们通过读取这些寄存器标志来了解通讯状态。

4.6.3.1 主发送器

主发送器通讯过程：IIC 作为通讯的主机端时，向外发送数据的过程。

1. 控制 CR2 的 START 位置 1 生成起始信号 S；
2. 接着发送设备地址并等待应答信号，若有从机应答，ISR 寄存器的“ADDR”位与“TXE”位被置 1，ADDR 为 1 表达地址完成匹配，TXE 为 1 表示数据寄存器为空；
3. 以上步骤正常执行并对 ADDR 位清零后，我们往 IIC 的“发送数据寄存器 TXDR”写入要发送的数据，这时 TXE 位会被重置为 0，表示数据寄存器非空，IIC 外设通过 SDA 信号线一位一位把数据发送过去后，发生事件：TXE 位被置 1，重复此过程，即可发送多字节数据；
4. 当我们数据发送完成后，控制 IIC 设备 CR2 寄存器的 STOP 位置 1 产生一个停止信号 P，此时 ISR 的 TXE 位与传输完成 TC 位置 1，表示通信结束。

若我们使能了 IIC 中断，以上事件产生时，都会产生 IIC 中断信号，进入同一个中断服务函数，到 IIC 中断服务函数后，再通过检查寄存器位来判断是哪一个事件。

4.6.3.2 主接收器

主接收器通讯过程：IIC 作为通讯的主机端时，从外部接收数据的过程。

1. 控制 CR2 的 START 位置 1 生成起始信号 S；
2. 接着发送设备地址并等待应答信号，若有从机应答，ISR 寄存器的“ADDR”位被置 1，表示地址完成匹配。
3. 从机端接收到地址后，开始向主机端发送数据。当主机接收到这些数据后，ISR 寄存器的 RXNE 位置 1，表示接收数据寄存器非空，我们读取该寄存器后，可对数据寄存器清空，以便接收下一次数据。此时我们可控制 IIC 发送应答信号（ACK）或非应答信号（NACK），若应答，则重复以上步骤接收数据，若非应答，则停止传输；
4. 发送非应答信号后，控制 IIC 设备 CR2 寄存器的 STOP 位置 1 产生停止信号（P），传输结束。

在发送和接收过程中，有的事件不只是标志了我们上面提到的状态位，还可能同时标志了主机状态之类的状态位，且读取之后还要清除标志位，我们可以使用技术人员编写好的 LL 库函数来直接检测这些事件的复合标志，降低编程难度。

4.6.4 IIC 初始化结构体详解

和其他外设一致，BS32 LL 库提供了 IIC 初始化结构体及初始化函数来配置 IIC 外设。初始化结构体及函数定义在库文件“bs32f0xx_ll_i2c.c”与“bs32f0xx_ll_i2c.h”中，编程时我们可以结合这两个文件内的注释使用或芯片的参数手册和数据手册。了解初始化结构体后对我们使用 IIC 十分有益。

```
typedef struct
{
  uint32_t Timing;            //通讯时序
  uint32_t AnalogFilter;      //模拟滤波
  uint32_t DigitalFilter;     //数字滤波
  uint32_t OwnAddress1;       //自身设备地址 1
  uint32_t TypeAcknowledge;   //ACK/NACK 生成（从模式）
} LL_I2C_InitTypeDef;
```

这些结构体的说明如下（括号内为对应参数在 LL 库中的宏）：

Timing

指定 SDA 起始信号时间与数据信号保持时间以及 SCL 高电平与低电平周期时间的值，可通过 LL_I2C_CONVERT_TIMINGS（）配置参数：PRESCALER、DATA_SETUP_TIME、DATA_HOLD_TIME、CLOCK_HIGH_PERIOD、CLOCK_LOW_PERIOD。

AnalogFilter

模拟滤波，使能或关闭 IIC 的模拟滤波。

DigitalFilter

数字滤波，使能或关闭 IIC 的数字滤波。

OwnAddress1

本成员配置的是 BS32 的 IIC 设备自己的地址，每个连接到 IIC 总线上的设备都要有一个自己的地址，作为主机也不例外。地址配置为 7 位且每个设备独占一个地址。

BS32 的 IIC 外设可同时使用两个地址，即同时对两个地址做出响应，这个结构成员默认配置的是 OAR1 寄存器存储的地址，若需配置第二个地址寄存器 OAR2，可以使用 LL 库函数中的 LL_I2C_EnableOwnAddress2（）与 LL_I2C_SetOwnAddress2（）自行配置。

TypeAcknowledge

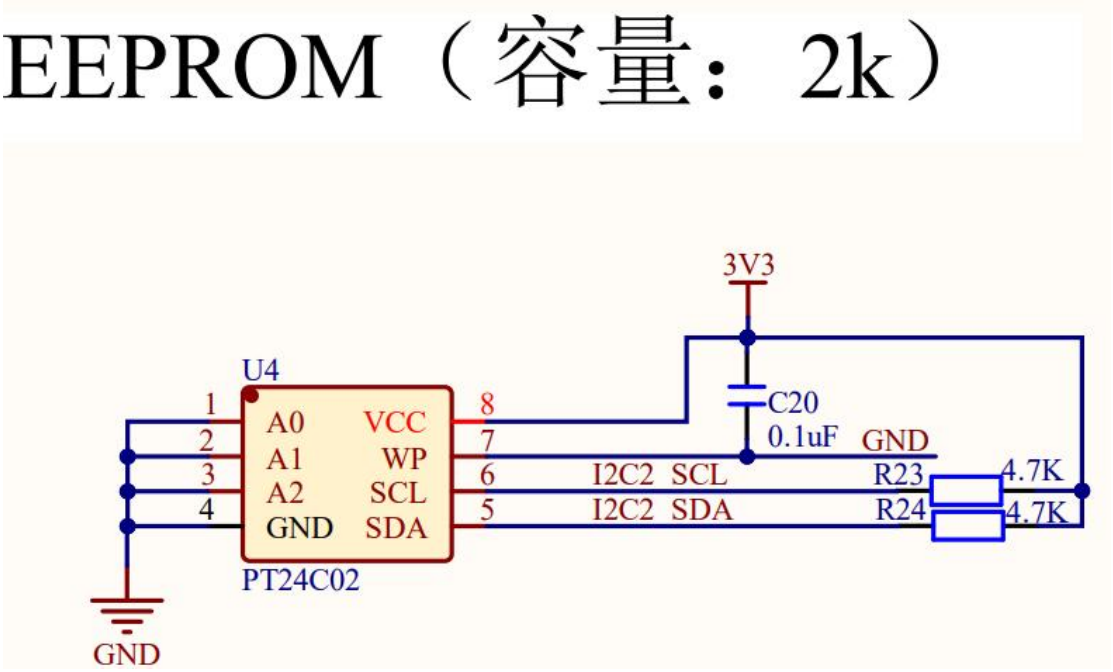
本成员对应的寄存器位 NACK 操作方式为 RS：软件可以读也可以设置此位，写 0 对此位无影响。NACK 由软件置 1（在当前接收的字节后发送 NACK），并可在发送 NACK 时、接收到停止位或匹配地址、或 PE=0 时由硬件清零（在当前接收的字节后发送 ACK）。

该位仅在从模式下使用：在主接收器模式下，无论 NACK 的值是什么，最后一个字节（后跟停止位或重复起始位）后都将自动生成 NACK。当从接收器 NOSTRETCH 模式下发生上溢时，无论 NACK 位的值是什么，都将自动生成 NACK。

4.6.5 IIC 编程要点（读写 EEPROM）

EEPROM 是一种掉电后数据不丢失的存储器，常用来存储一些配置信息，以便系统重新上电的时候加载。EEPROM 芯片最常用的通讯方式就是 IIC 协议，下面是以 BS32 的 IIC 外设为主模式进行读写 EEPROM 的说明，分别用作主发送器和主接收器，通过查询事件的方式来确保正常通讯。

4.6.5.1 硬件设计



我们的开发板使用的 EEPROM 芯片（型号：PT24C02）的 SCL 与 SDA 引脚连接到 BS32 对应的 IIC 引脚中，结合上拉电阻，构成了 IIC 通讯总线，它们通过 IIC 总线交互。EEPROM 芯片的设备地址一共有 7 位，其中高 4 位固定为：1010b，低 3 位则由 A0/A1/A2 信号线的电平决定，如下表所示是 EEPROM 的设备地址，表中的 R/W 是读写方向位，与地址无关。

1（MSB）	0	1	0	A_2	A_1	A_0	R/W(LSB)

按照我们此处的连接，A0/A1/A2 均为 0，所以 EEPROM 的 7 位设备地址是：1010000b，即 0x50。由于 IIC 通讯时常常是地址跟读写方向连在一起构成一个 8 位数，且当 R/W 位为 0 时，表示写方向，所以加上 7 位地址，其值为 0xA0，常称该值为 IIC 设备的“写地址”；当 R/W 位为 1 时，表示读方向，加上 7 位地址，其值为 0xA1，常称该值为“读地址”。

EEPROM 芯片中还有一个 WP 引脚，具有写保护功能，当该引脚电平为高时，禁止写入数据，当引脚为低电平时，可写入数据，我们直接接地，不使用写保护功能。电容 C20 的作用为滤波。

4.6.5.2 软件设计

为使工程更有条理，我们把读写 EEPROM 相关的代码独立存储，方便以后移植。在工程模板中新建板级 IIC 与 EEPROM 通信的.c 与.h 文件进行开发。

编程要点

(1) 配置通讯使用的目标引脚为开漏模式（此时若需要释放 SDA 总线，将 SDA 设置输出为 1 即可）；

(2) 使能 IIC 外设的时钟；

(3) 配置 IIC 外设的模式、地址、速率等参数并使能 IIC 外设；

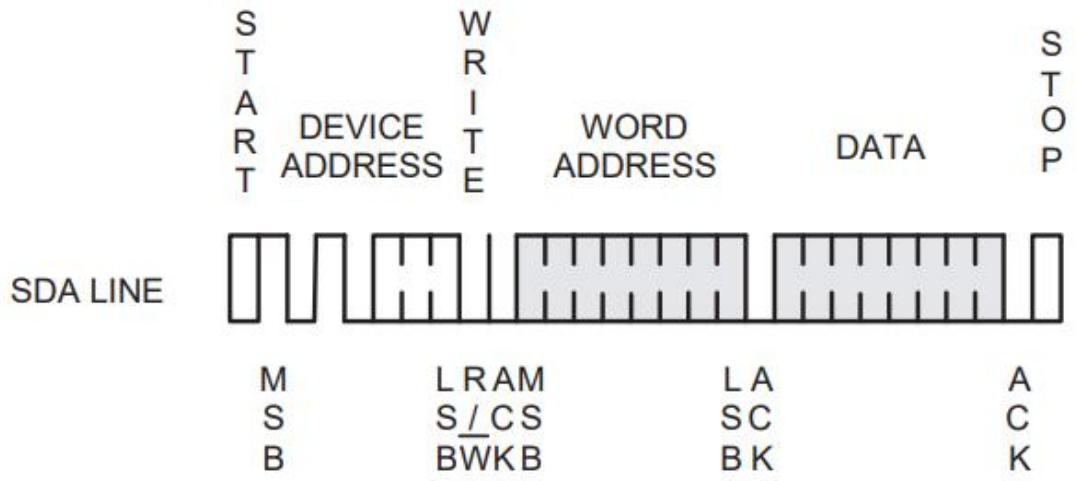
(4) 编写基本 IIC 按字节收发的函数；

(5) 编写读写 EEPROM 存储内容的函数；

(6) 编写测试程序，对读写数据进行校验。

向 EEPROM 写入一个字节的数据

BS32 实际上通过 IIC 向 EEPROM 发送了两个数据，如下图所示，由于 EEPROM 自己定义的单字节写入时序，第一个数据会被解释为 EEPROM 的内存地址。



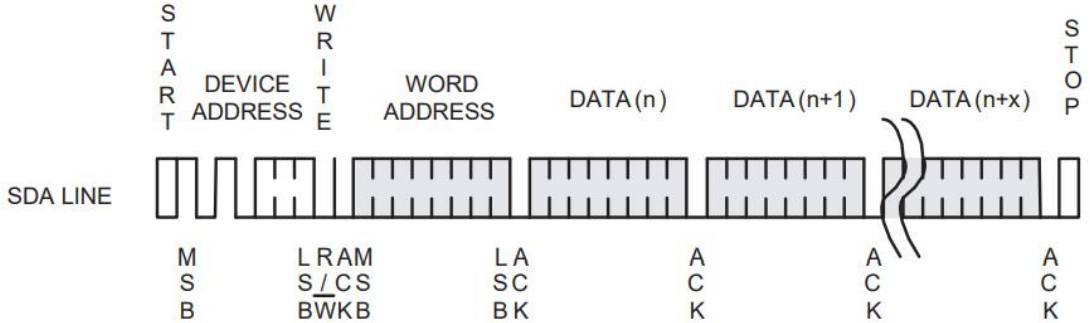
EEPROM 的单字节时序规定，向它写入数据的时候，第一个字节为内存地址，第二个字节是要写入的数据内容。

多字节写入及等待状态

我们在多次写入数据时，要先等待 EEPROM 内部擦写完毕。

EEPROM 的页写入

在上述的数据通讯中，每写入一个数据都要向 EEPROM 发送写入的地址，我们希望向连续地址写入多个数据的时候，只要告诉 EEPROM 第一个内存地址 WORD address，后面的数据按次序写入，这样可以节省通讯时间，加快速度。为应对这种需求，EEPROM 定义了一种页写入时序，如下图所示：



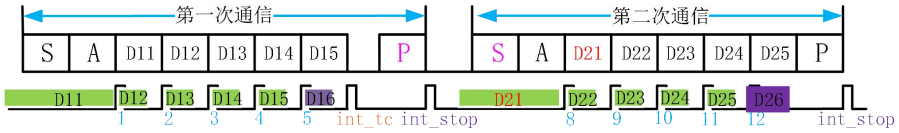
根据页写入时序，第一个数据被解释为要写入的内存地址 WORD address，后续可连续发送 N 个数据，这些数据会依次写入到内存中。

4.6.6 IIC 通信说明

4.6.6.1 主机发送接收

4.6.6.1.1 发送过程中的中断信号示意图

停止后开始信号说明



例：第一次通信发送 5 笔数据，软件结束。第二次通信发送 5 笔数据，自动结束。

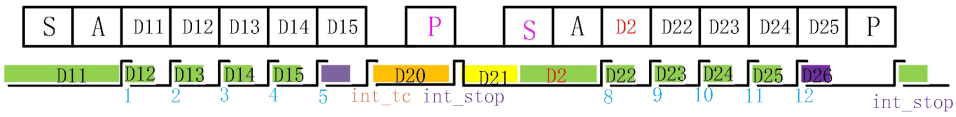
第一次通信，在初始化时提前准备好第一笔数据，即在地址应答完成之前准备好 D11，接收到地址应答后立即发送 D11，在第一个发送数据寄存器空中断 txis 里准备第二笔数据 D12，以此类推。在第四个发送数据寄存器空中断 txis 里准备好最后一笔数据 D15。

如果在第 5 个发送数据寄存器空中断 txis 里写入数据 D16，在 D15 发送完成后硬件置 TXE 为 1，D16 被丢失，为无效数据（这里也可不必写入无效数据）。

最后一笔 D15 发送完成时，不产生发送数据寄存器空中断 txis。产生发送完成中断 tc，等待写入 STOP。发出停止后，产生停止位检测中断 stop。

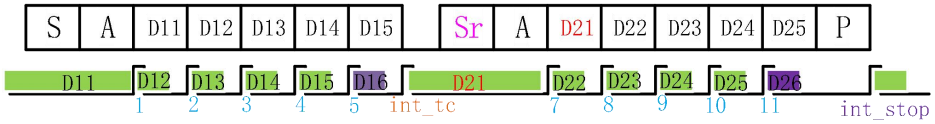
写 start 重新开始通信。在传输完成中断 tc 到地址应答完成之前可以提前准备第二次通信的第一笔数据 D21。同理 D26 为无效数据。发送最后一笔数据 D25 后,自动发出停止位，不产生任何中断。发出停止后，产生停止位检测中断 stop。

停止后开始并改变第一笔数据信号说明



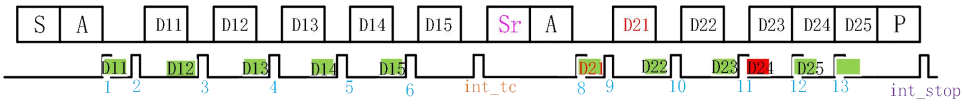
在传输完成中断 tc 到地址应答完成之前可以提前准备第二次通信的第一笔数据，可以通过软件置发送数据寄存器为空位 TXE 为 1，更新第一笔数据。地址应答完成之前最后一笔写入的数据 D2 作为第二次通信的第一笔数据。同理 D26 为无效数据。

重启动信号说明



软件结束模式，第一次通信结束后，产生传输完成中断 tc，等待写入 START，重新开始通信。

延长时钟低电平信号说明

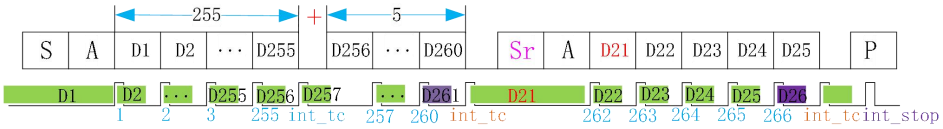


例：每次通信发送五笔数据。

第一次通信阶段，在初始化时（地址匹配中断 addr）没有提前准备好第一笔数据，延长时钟低电平，在第一个发送数据寄存器空中断 txis 里准备第一笔数据 D11，此时总线空闲随即发出 D11，同时产生第 2 个发送数据寄存器空中断 txis，用于准备第二个数据。第一个数据发送完成后，第二个数据还未准备好，再次延长时钟低电平，写入第二笔数据后，发出 D12，同时产生第 3 个发送数据寄存器空中断 txis。以此类推，在第五个发送数据寄存器空中断 txis 里准备好最后一笔数据 D15。

第二次通信阶段，在初始化时没有提前准备好第一笔数据。D23 发送完成之前，第 11 个中断里准备好第三个数据 D24，随后发送 D24 不需要延长时钟低电平。

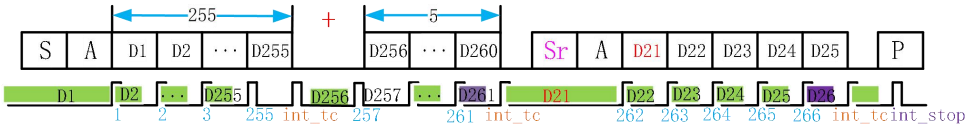
发送数据＞255 信号说明



第一次通信需要发送 260 个数据，分为 255+5 发送。先选择重载模式，发送完 255 个数据，产生传输完成中断 tc，延长时钟低电平，选择非重载模式，并写入待发送个数 5。

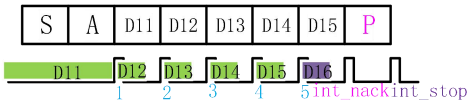
产生传输完成中断 tc 的同时会产生发送寄存器空中断 txis，但实际上 CPU 只进入一次中断。

发送数据＞255 沿长时钟线信号说明



第 255 个发送寄存器空中断 txis 里没有准备好第 256 个数据，发送完 255 个数据时，只产生传输完成中断 tc，不产生发送寄存器空中断 txis，延长低电平直到写入第 256 个数据，发送 D256，并产生第 257 个发送寄存器为空中断 txis。

接收到 NACK 信号说明



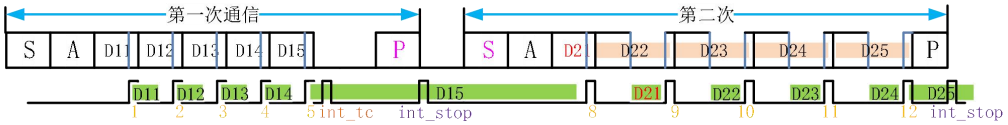
例：需要发送十笔数据。

发送完 D15 时接收到 nack，产生接收到否定应答中断 nack；且 nackend=1 时，自动发送停止位。

若 nackend=0，继续发送剩下的 5 笔数据。

4.6.6.1.2 接收数据过程中中断示意图

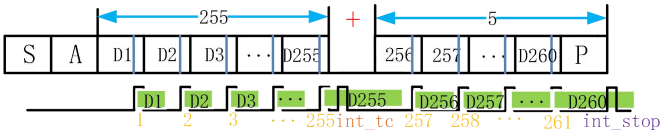
不延长+延长时钟线信号说明



第一次通信接收 5 笔数据，发送（到移位寄存器）完 8 位数据后产生接收数据寄存器不为空中断 rxne。在第 1 个 rxne 中断里读出 D11，在第 2 个中断里读出 D12，以此类推。在第 5 个中断后到下一次通信第一笔数据发送完之前读出最后一笔数据 D15。并在最后一笔数据应答位后产生传输完成中断 tc，等待写入 STOP 或 START。

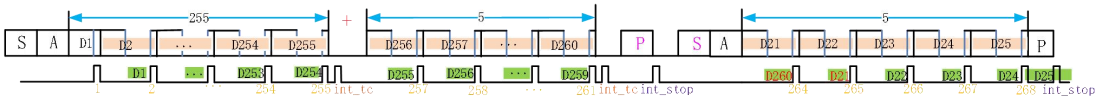
第二次通信接收 5 笔数据。进入第 8 个中断 rxne 后，没有在 D22 接收完成时读出 D21，需要延长时钟低电平，直到把 D21 读出为止。然后产生第 9 个中断 rxne，读取 D22。在第 12 个中断 rxne 后，读取最后一笔数据 D25。最后一笔数据接收完成后发送 NACK，随后发送停止位。

接收数据＞255 不延长时钟信号说明



接收 260 笔数据，分为 255+5，自动发送停止位。

接收数据＞255 延长时钟信号说明



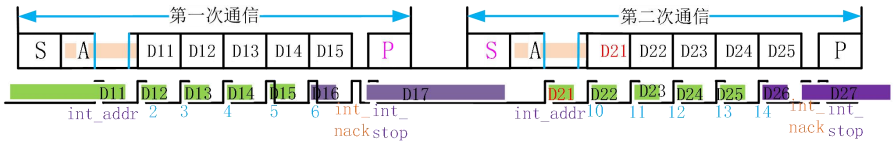
第一次通信接收 260 笔数据，分为 255+5，最后一笔数据应答位后产生传输完成中断 tc，延长时钟低电平，等待写入 STOP，发送停止位。

第二次通信的第一笔数据 D21 接收之前还未读出上次通信的最后一笔数据 D260，延长时钟线直到读出 D260。随后产生第 264 个中断 rxne，读取 D21。

4.6.6.2 从机发送接收

4.6.6.2.1 发送过程中的中断信号示意图

停止后开始信号说明



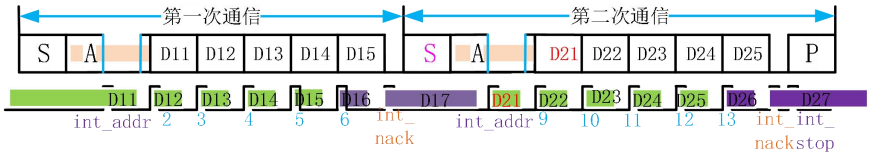
接收地址后进入地址中断，在地址中断里判断写读标志位 DIR 和匹配的地址 ADDCODE[7:0]。若 DIR=1，从机发送数据，则可在地址中断里准备第一笔数据，也可以提前准备。最后 ADDRCF 位置 1 清除地址匹配标志，以释放延长时钟低电平。

在第 2 个中断 txis 中准备第二笔要发送的数据，以此类推。主机发出 nack 时，可检测到接收到否定应答标志 NACKF，写 NACKCF 清零。同时会产生 nack 中断。此时无论数据是否为空，不产生 txis 中断。

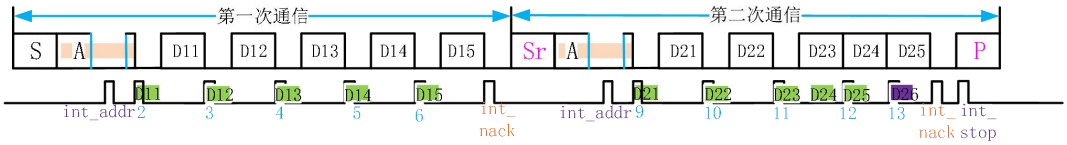
主机发出停止信号时，可检测到 STOPF 标志，写 STOPCF 清零。同时会产生停止位检测中断 stop。

在第 6 个中断 txis 里写入的 D16，可作为下一笔通信的第一笔数据。也可在后续中断里通过将 TXE 置 1，更新第一笔数据。则 D16，D17 为无效数据，第一笔数据为 D21。

重启动信号说明



延长时钟低电平信号说明

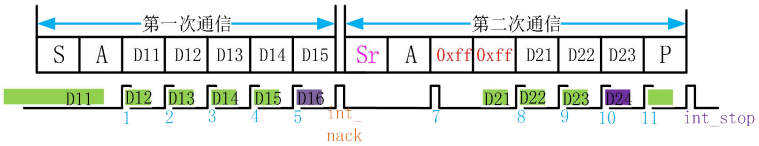


第一次通信阶段，没有准备好第一笔数据，延长时钟低电平，在第 1 个 txis 中断里准备第一笔数据 D11，随后发出 D11（发送至移位寄存器后到空闲的总线上），同时产生第 2 个 txis 中断，用于准备第二个数据。第一个数据发送完成后，第二个数据还未准备好，再次延长时钟低电平，写入第二笔数据后，发出 D12,同时产生第 3 个 txis 中断。以此类推，在第 5 个 txis 中断里准备好最后一笔数据 D15。

第二次通信阶段，在初始化时没有提前准备好第一笔数据。第 11 个中断里首先准备好 D23 发送到移位寄存器后，继续准备好第三个数据 D24，随后发送 D24 不需要延长时钟低电平。

这里是为了演示时钟延长的功能，在数据未准备好的时候，使能时钟低电平延长的功能可以使硬件自动实现时钟延长来完成传输。

不延长时钟低电平溢出信号说明



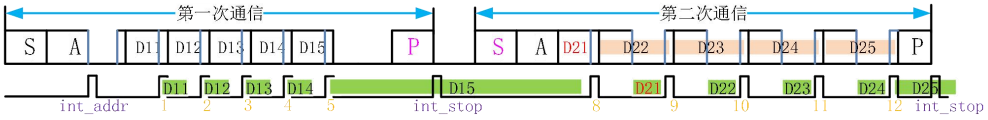
不使能延长时钟低电平功能且不使能地址中断时。

第一次通信：提前准备好第一笔数据，并且每个中断都可以提前准备下一笔待发送的数据，实现正常通信。

第二次通信：没有提前准备好第一笔数据，发生溢出，第一笔发送的数据为全 1。第一个 txis 中断里来不及准备第二笔数据，第二笔发送的数据全为 1。直到第二个 txis 中断里准备好后续的数据开始正常通信。

4.6.6.2.2 接收数据过程中中断示意图

不延长+延长时钟低电平应答信号说明

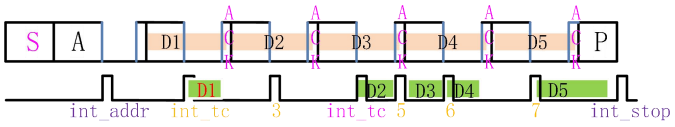


接收地址后进入地址中断，在地址中断里判断写读标志位 DIR 和匹配的地址 ADDCODE[7:0]。若 DIR=0，从机接收数据。最后 ADDRCF 位置 1 清除地址匹配标志，以释放延长时钟低电平。

第一次通信接收 5 笔数据，发送完 8 位数据后产生 rxne 中断。在第一个 rxne 中断里读出 D11，在第 2 个中断里读出 D12，以此类推。在第 5 个中断后到下一次通信第一笔数据发送完之前读出最后一笔数据 D15。检测到停止信号后产生停止位检测中断 stop。

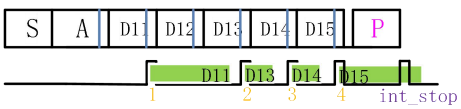
第二次通信接收 5 笔数据。进入第 8 个中断 rxne 后，没有在 D22 接收完成时读出 D21，需要延长时钟低电平，直到把 D21 读出为止。然后产生第 9 个中断 rxne，读取 D22。在第 12 个中断 rxne 里，读取最后一笔数据 D25。

字节控制模式信号说明



从机在接收完每笔数据后都控制 ACK 位，每笔数据的第八个时钟下降沿产生传输完成中断 tc。第一笔数据的第八个时钟下降沿同时产生 rxne 中断和 tc 中断，延长时钟低电平，写 NACK 信号后释放时钟。第二笔数据 D2 在接收完 D3 之前没读走，延长时钟低电平用于读取 D3。第三笔数据的第八个时钟下降沿后有传输完成中断 tc 和 rxne 两个不同的中断，并有两个延长时钟低电平，写入 NACK 信号和读取 D3 都完成才释放时钟。

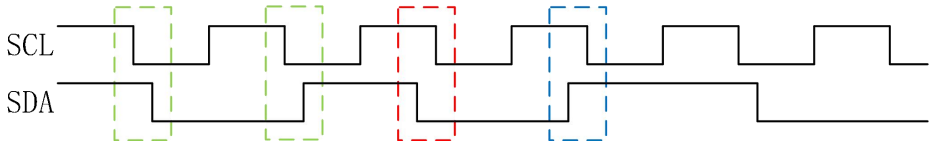
不延长时钟低电平溢出信号说明



第一笔数据 D11 在接收完第二笔数据 D12 之前没读走，会丢失 D12，且在 D12 的第八个时钟下降沿后不产生 rxne 中断。在接收完第三笔数据 D13 之前读走 D11，不会丢失 D13。

4.6.6.3 仲裁丢失与错误开始停止

错误的开始或停止示意图



绿色部分：正确的数据传输过程，SDA 在 SCL 的低电平变化。

红色部分：错误的开始信号。在地址和数据传输的过程中，在 SCL 高电平检测到 SDA 的下降沿。

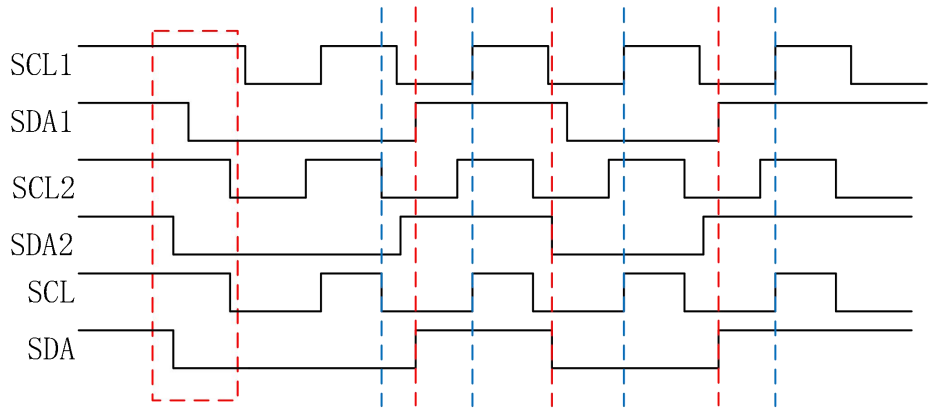
蓝色部分：错误的停止信号。在地址和数据传输的过程中，在 SCL 的高电平检测到 SDA 的上升沿。

主机在非空闲时检测除自身发送的开始信号之外的开始信号，并认为是错误开始。

主机在非空闲时检测除自身发送的停止信号之外的停止信号，并认为是错误停止。

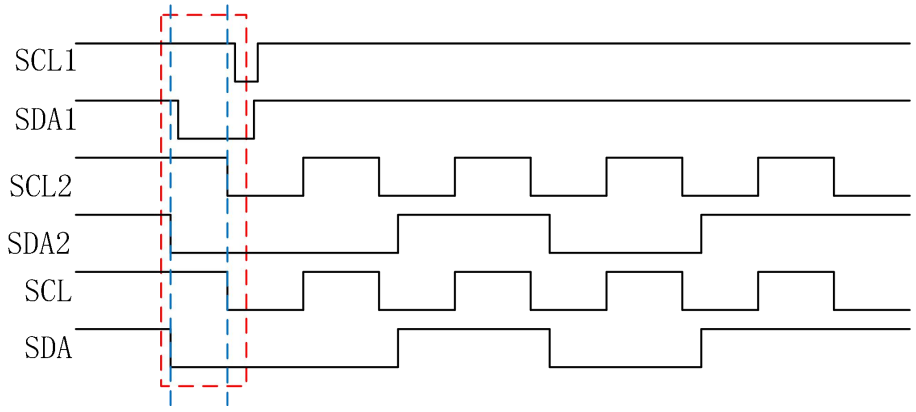
仲裁丢失示意图（多主机下）

- **时钟同步**



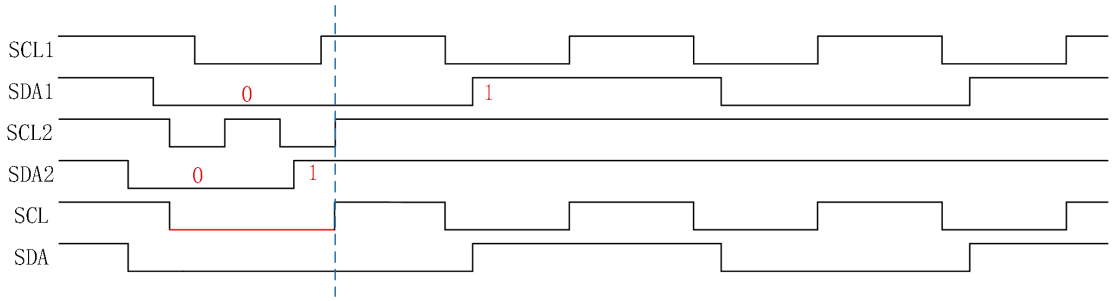
两个主机分别在 SCL 线上产生各自的 SCL 1 和 SCL2 时钟，来传输各自的数据。只要有一个 SCLn 为低，SCL 线上为低。SCL 时钟的低电平周期由低电平时钟周期最长的器件决定而高电平周期由高电平时钟周期最短的器件决定。

- **同频时钟丢失仲裁**



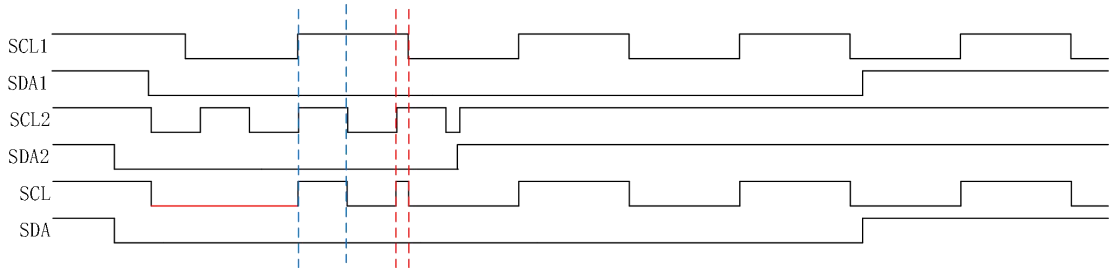
主机 1 和主机 2 时钟频率基本一致。主机 1 在地址阶段先发送了 1，而主机 2 发送 0，因为总线时逻辑与，因此主机 1 丢失仲裁。后续传输由主机 2 控制。

- **不同频时钟丢失仲裁**



主机 1 的时钟频率远小于主机 2 的时钟频率。它们需要发送的数据相同，但由于主机 2 时钟频率快，会先发出 1，丢失仲裁。

- **不同频率主机造成总线出错**



不同频时钟可能造成总线时钟混乱。

4.6.7 IIC 通讯配置与注意事项

4.6.7.1 主机配置流程建议

非重载模式配置（NBYTES[7:0]<=255）

1. 将 IIC_CR1 中的 PE（外设使能）清零。
2. 配置数字滤波和模拟滤波。

3. 配置 IIC_TIMINGR 中的 SDADEL[3:0] 、SCLH[7:0]和 SCLL[7:0]。

4. 将 IIC_CR1 中的 PE（外设使能）置 1。

5. 待发送的从地址：SADD[7:1]；

   传输方向：RD_WRN；

   待传输的字节数：NBYTES[7:0] ；

   必须配置 RELOAD=0；

   若配置 AUTOEND=1，NBYTES 数据传输结束后自动生成停止位；

   若配置 AUTOEND=0， NBYTES 数据传输结束后检测到 TC 标志，配置重新开始(START）或停止（STOP）。

   配置中断使能；

   配置重新开始之前可以改变上述配置，在传输过程中不能改变。

6. 检测到 busy 标志为 0，将 I2C_CR2 寄存器中的 START 位置 1。START 位置 1 时，不允许更改步骤 5 中的所有位。txdata[7:0]可以在 5 中配置，也可以在 START 位置 1 后配置。当检测到总线空闲时，它会在经过 t_BUF 的延时后自动发送起始位，随后发出从器件地址。

重载模式配置（NBYTES[7:0]>255）

1. 将 IIC_CR1 中的 PE（外设使能）清零。

2. 配置数字滤波和模拟滤波。

3. 配置 IIC_TIMINGR 中的 SDADEL[3:0] 、SCLH[7:0]和 SCLL[7:0]。

4. 将 IIC_CR1 中的 PE（外设使能）置 1。

5. 待发送的从地址：SADD[7:1]；

   传输方向：RD_WRN；

   待传输的字节数：NBYTES[7:0] 。

   必须配置 RELOAD=1；初始化时必须将 NBYTES[7:0]填充为 0xFF。传输完 255 数据后，检测到 TCR，如果剩下待传输个数依然大于 255，必须再次将 NBYTES[7:0]填充为 0xFF。直到剩下待传输个数小于等于 255，检测到 TCR 时，配置 RELOAD=0，以及配置 AUTOEND 和 NBYTES[7:0]。

   若配置 AUTOEND=1，NBYTES 数据传输结束后自动生成停止位；

   若配置 AUTOEND=0， NBYTES 数据传输结束后检测到 TC 标志，配置重新开始(START)或停止(STOP)。

   配置中断使能。

   配置重新开始之前可以改变上述配置，在传输过程中不能改变。

6. 检测到 busy 标志为 0，将 I2C_CR2 寄存器中的 START 位置 1。START 位置 1 时，不允

许更改步骤 5 中的所有位。txdata[7:0]可以在 5 中配置，也可以在 START 位置 1 后配置。当检测到总线空闲时，它会在经过 t_BUF 的延时后自动发送起始位，随后发出从器件地址。

NBYTES[7:0]大于 255 时，必须用重载模式和非重载模式的组合。

4.6.7.2 从机配置流程建议

普通模式配置

1. 将 IIC_CR1 中的 PE（外设使能）清零。
2. 使能配置一个自身地址 I2C_OAR1 / I2C_OAR1。
3. 延长时钟线功能时配置数据建立时间 SCLDEL。
4. 根据需要打开相应的中断使能：

使能延长时钟线功能时，可打开所有中断使能（使用轮询功能可以不打开地址匹配中断使能，否则必打开 ）。

不使能延长时钟线功能时，可打开除地址中断使能之外的中断使能。（地址中断和下一个发送数据中断很近，如果地址中断处理太久会影响下一个数据准备然后溢出，在停止之前，清掉地址匹配标志即可）

5. 根据需要配置 IIC_CR1 和 IIC_CR2 寄存器。还可以提前准备第一笔数据。
6. 将 IIC_CR1 中的 PE（外设使能）置 1。

NACK/ACK 控制模式配置

1. 将 IIC_CR1 中的 PE（外设使能）清零。
2. 使能配置一个自身地址 I2C_OAR1 / I2C_OAR1。
3. 延长时钟线功能时配置数据建立时间 SCLDEL。
4. 根据需要打开相应的中断使能：

使能延长时钟线功能时，可打开所有中断使能（使用轮询功能可以不打开地址匹配中断使能，否则必打开）。

不使能延长时钟线功能时，可打开除地址中断使能之外的中断使能。

5. 根据需要配置 IIC_CR1 和 IIC_CR2 寄存器，必须配置 NOSTRETCH =0、SBC=1、NBYTES!=0、RELOAD=1。完成 NBYTES 数据传输时，会延长低电平，需要检测到 TCR 标志，写 NACK 位且 NBYTES 不为零释放时钟线（实际上只要写上 I2C_CR2 寄存器且 NBYTES 不为零都有能释放）。还可以提前准备第一笔数据。
6. 将 IIC_CR1 中的 PE（外设使能）置 1。

4.6.7.3 DMA 模式配置

- 配置 DMA_CENSR 来选择 IIC 通道使能，DMA_CENSR[5] 为 IIC 读通道使能，DMA_CENSR[6]为 IIC 写通道使能。
- 配置 DMA_CIER 打开 DMA 的 IIC 通道中断使能，DMA_CIER[5] 为 IIC 读通道中断使能，DMA_CIER [6]为 IIC 写通道中断使能。
- 配置 DMA_CCDBPR 来指向通道数据地址(DMA_SRAM_BASE)。
- DMA 读请求时：
  - 配置 DMA_CSEARx 为 IIC_RXDR（读）地址(源地址)；
  - 配置 DMA_CDEARxR 为 SRAM 地址(目的地址)；
- DMA 写请求时：
  - 配置 DMA_CSEARx 为 SRAM 地址地址(源地址)；
  - 配置 DMA_CDEARxR 为 IIC_TXDR（写） (目的地址)；
- 配置 DMA_CCFGRx 决定 DMA 工作模式、传输长度、源数据位宽、源地址地址增量、目的数据位宽、目的数据地址增量。（具体参见 DMA 寄存器列表）。
- 配置 IIC_CR1 中 RXDMAEN （DMA 接收请求使能） 、TXDMAEN（DMA 接收请求使能）。
- 除了中断使能，IIC 相关配置与普通模式相同。不使能 TXIE、RXIE、TCIE、NACKIE 和 STOPIE。（如果使用轮询模式清除 STOPF 标志，可以不使能 STOPIE，否则需要使能 STOPIE 进中断清除标志。）

IIC 发送数据为空（TXE），并且 DMA 处理完上次 DMA 写请求时，发出 DMA 写请求；IIC 接收数据为满（RXNE），并且 DMA 处理完上次 DMA 读请求时，发出 DMA 读请求。

在非重载软件结束模式下，需要检测 tc 标志，并将起始位或停止位置 1 释放时钟线才能进行后续传输。

在重载模式下，需要检测 TCR 标志，并写 I2C_CR2 寄存器且 NBYTES 不为零 (建议写 NBYTES 非零值)释放时钟线才能进行后续传输。

IIC 从机需要检测 addr 标志，并将其清零。

使用 DMA 模式必须使用延长时钟线功能，否则从机发送数据时第一笔数据来不及准备（这里是考虑到 DMA 的启动时间）。

4.6.7.4 注意事项

主机模块

1. 初始化时将 START 位置 1 之前，先检测 busy 标志是否为 1，如果为 1，表示 IIC 总线不同时为零（不是空闲状态），此时如果将 START 位置 1，可能不能成功发送开始位。

2. 配置 RELOAD=1，即重载模式时，必须将 NBYTES[7:0]填充为 0xFF，并且发送 255 个数据。否则会延长时钟等待新写入数据，直到完成 255 个数据发送。TCR 标志置 1，再执行后续操作。

3. 如果 NBYTES[7:0]填充为 0，发送完地址后将自动发送停止位。

4. 发送停止后再次开始的模式，需要检测到 STOPF 标志后再写 START 位。否则可能不能成功发送停止位。

5. 不延长时钟低电平情况下，从已准备好的 txdata[7:0]将数据放入移位寄存器 shift_tx[7:0]中，并跨时域处理到 iic_clk 时域下，需要固定的 3*pclk+3*iic_clk。从第九个时钟下降沿到数据的最高位发出的时间为固定的 3*t_pclk+4*t_iic_clk。所以应答信号的保持时间固定为 3*pclk+4*iic_clk。数据最高位的建立时间为t_{LOW}-(3*pclk+4*iic_clk)。

6. 5 中的极限情况：IIC 通信频率为 1M，系统时钟为 12M 时，(3*pclk+4*iic_clk)=3*83.3+4*41.6=416.3ns，可以适当将 SCLL 配置放大，SCLH 配置缩小。

7. 数据的保持时间$t_{HD;SDA}$=(sdadel+1)*(presc+1)*t_iic_clk。极限情况：SDADEL 配置为零，且 PRECS 配置为 0 时，最小的数据保持时间为一个 iic_clk 周期（41.6ns）。SDADEL 不能配置为零。

8. 因为延长低电平的信号的控制，t_{LOW}必须大于 3*pclk+2*iic_clk。根据 4 可知t_{LOW}最小值为 3*pclk+4*iic_clk+ $t_{SU;SDA}$

9. 因为总线输入的 scl_in 和 sda_in 需要跨时域处理，t_{HIGH}至少大于 3* iic_clk。

10. 打开外设使能之前配置滤波寄存器。打开使能之后不允许更改滤波配置。

11. 打开外设使能之前配置时序寄存器。打开使能之后不允许更改时序配置。

12. 所有延长时钟低电平情况，应该把其他信号配置好，最后配置释放时钟线的操作。

13. 从第 9 个时钟下降沿到数据的最高位发出的时间为固定的 4*t_pclk，所以配置系统时钟时，应该满足 4*t_pclk<t_scll+（模拟滤波/数字滤波）。以模拟滤波 150ns 为例：

    - IIC 通信频率为 1M 时，pclk 的最低频率为 12M。
    - IIC 通信频率为 400K 时，pclk 的最低频率为 3M。
    - IIC 通信频率为 100K 时，pclk 的最低频率为 1M。

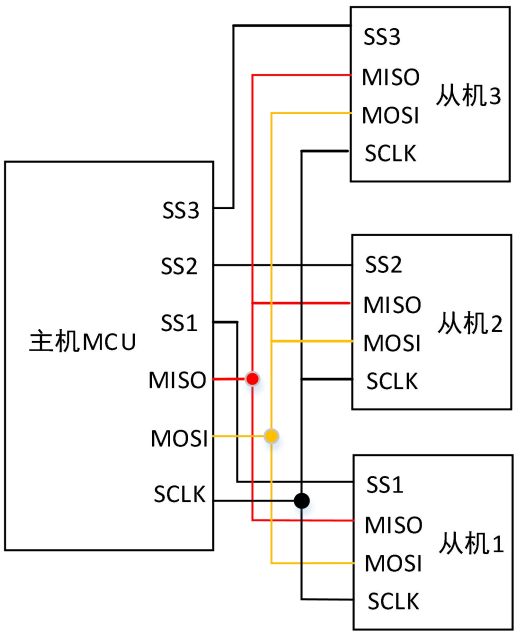
4.7 SPI 模块介绍

SPI（Serial Peripheral Interface）即串行外围设备接口，是一种高速全双工的通信总线。广泛使用在 ADC、LCD 等设备与 MCU 间，要求通信速率较高的场合。

4.7.1 SPI 协议概述

物理层

SPI 通讯设备之间的常用连接方式如下：



SPI 通讯使用 3 条总线及片选线，3 条总线分别为 SCLK、MOSI、MISO,片选线为 NSS，他们的作用介绍如下：

NSS（Slave Select）：从设备选择信号线，又称片选信号线、NSS、CS。当有多个 SPI 从设备与 SPI 主机相连时，设备的其他信号线 SCLK、MOSI 与 MISO 同时并联到相同的 SPI 总线上，即无论有多少从设备，都共同使用这三条总线；而每个从设备都有独立的 NSS 信号线，该信号线独占主机的一个引脚，即有多少个从设备，就有多少条片选信号线。IIC 协议中通过设备地址来寻址，选中总线上的某个设备并与其进行通讯；而 SPI 协议中没有设备地址，它使用 NSS 信号线来寻址，当主机要选择从设备时，把该设备的 NSS 信号线设置为低电平，该设备即被选中，即片选有效，接着主机开始与被选中的从设备进行 SPI 通讯。所以 SPI 通讯以 NSS 线置低电平为开始信号，以 NSS 线被拉高为结束信号。

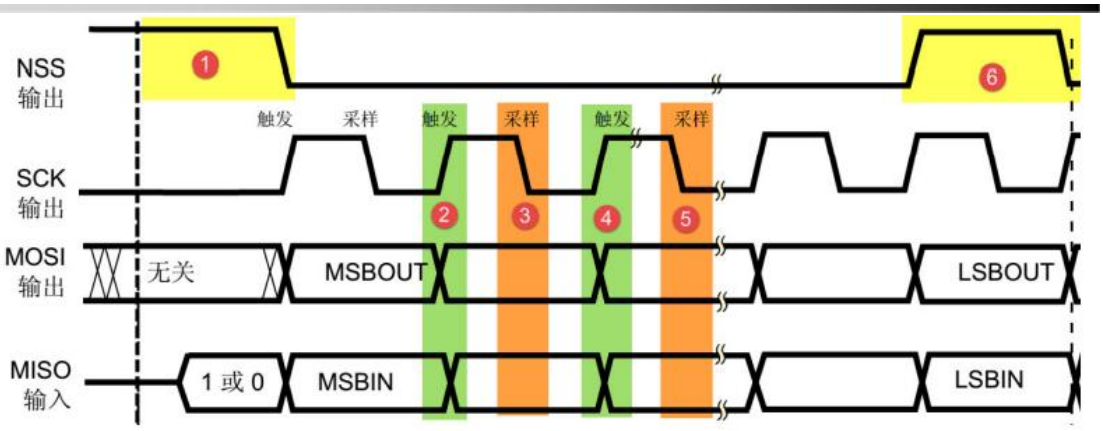
SCLK（Serial Clock）：时钟信号线，用于通讯数据同步。它由通讯主机产生，决定了通讯的速率，不同的设备支持的最高时钟频率不一样，BS32F030 从机速率只有主机的一半,两个设备之间通讯时，通讯速率受限于低速设备。

MOSI（Master Output，Slave Input）：主设备输出/从设备输入引脚。主机的数据从这条信号线输出，从机由这条信号线读入主机发送的数据，即这条线上数据的方向为主机到从机。

MISO（Master Intput，Slave Output）：主设备输入/从设备输出引脚。主机从这条信号线读入数据，从机的数据由这条信号线输出到主机，即在这条线上数据的方向为从机到主机。

协议层

与 IIC 类似，SPI 协议定义了通讯的起始和停止信号、数据有效性、时钟同步等环节。SPI 的通讯时序如下：



这是一个主机的通讯时序。NSS、SCLK、MOSI 信号都由主机控制产生，而 MISO 的信号由从机产生，主机通过该信号读取从机的数据。MOSI 与 MISO 的信号只在 NSS 为低电平的时候才有效，在 SCLK 的每个时钟周期 MOSI 与 MISO 传输一位数据。通信流程中各信号的分解如下：

- **通讯的起始和停止信号**

在上图中标号 1 位置，NSS 信号由高变低，是 SPI 通讯的起始信号。NSS 是每个从机各自独占的信号线，当从机在自己的 NSS 线检测到起始信号后，即被主机选中，准备开始通讯。在上图标号 6 的位置，NSS 信号由低变高，是 SPI 通讯的停止信号，表示本次通讯结束，从机的选中状态被取消。

- **数据有效性**

SPI 使用 MOSI 与 MISO 信号线来传输数据，使用 SCLK 信号线进行数据同步。MOSI 与 MISO 数据线在 SCLK 的每个时钟周期传输一位数据，且数据输入输出是同时进行的。数据输出时，MSB 先行或 LSB 先行并没有作硬性规定，但要保证两个 SPI 通讯设备之间使用相同的配置，一般默认采用 MSB 先行模式。

如图标号 2 处所示，MOSI 与 MISO 的数据在 SCLK 的上升沿期间变化输出，在 SCLK 的下降沿时被采样。即在 SCLK 的下降沿时刻，MOSI 与 MISO 的数据有效，高电平时表示数据“1”，低电平时表示数据“0”。在其他时刻，数据无效，MISO 与 MOSI 为下一次表示数据做准备。

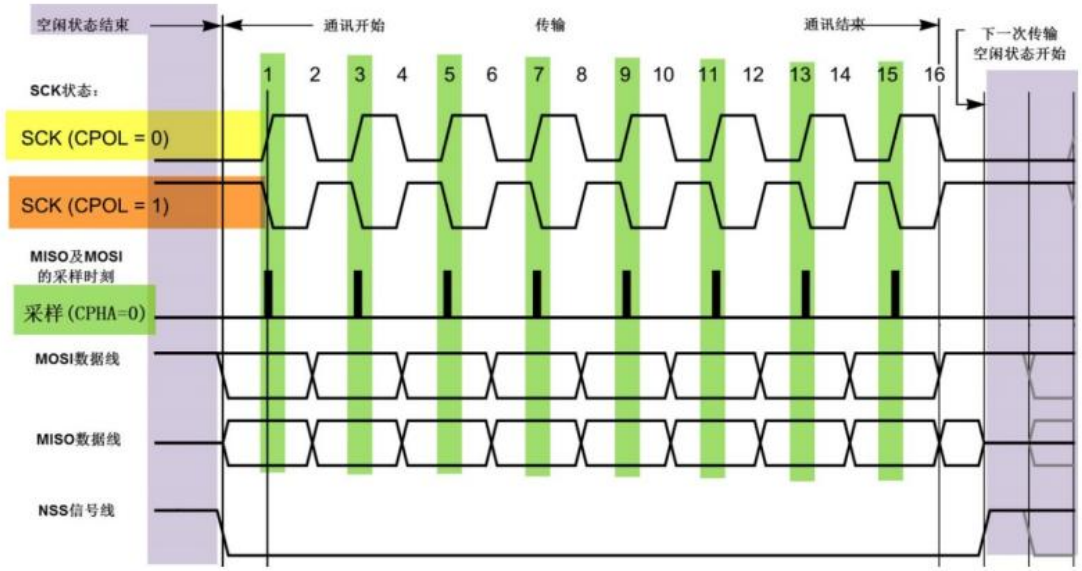
BS32F030 主控上的 SPI 外设的传输数据长度在 4-16 位与 32 位可选，参考寄存器手册配置，若写入未使用值，数据位宽将被强制设为 8 位。

- **CPOL/CPHA 及通讯模式**

上图中的时序只是 SPI 其中一种通讯模式，SPI 共有四种通讯模式，他们的主要区别是总线空闲时 SCLK 的时钟状态以及数据采样时刻。这里引入“时钟极性 CPOL”和“时钟相位 CPHA”的概念。

时钟极性 CPOL 是指 SPI 通讯设备处于空闲状态时，SCLK 信号线的电平信号（即 SPI 通讯开始前，NSS 线为高电平时 SCLK 的状态）。CPOL=0 时，SCLK 在空闲状态时为低电平，CPOL=1 时，SCLK 在空闲状态时为高电平。

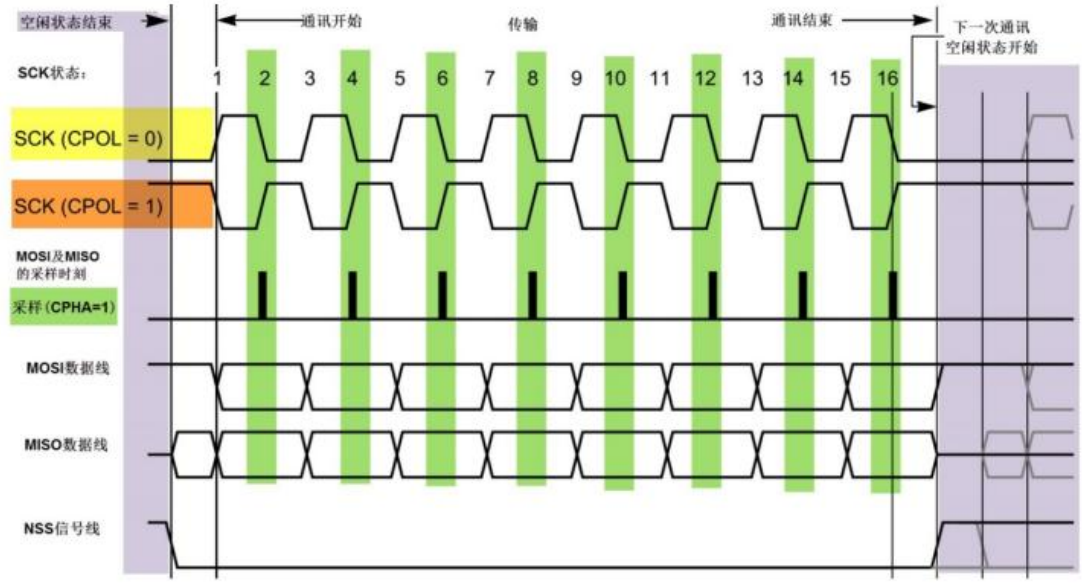
时钟相位 CPHA 是指数据的采样的时刻，当 CPHA=0 时，MOSI 与 MISO 数据线上的信号将会在 SCLK 时钟线的“奇数边沿”被采样。当 CPHA=1 时，数据线在 SCLK 的“偶数边沿”采样。如下图所示为 CPHA=0 时的 SPI 通讯时序图：



上图中 CPHA=0 的时序图，根据 SCLK 在空闲状态时的电平，分为两种情况：SCLK 信号线在空闲状态为低电平时，CPOL=0；空闲状态为高电平时，CPOL=1。

无论 CPOL=0 还是=1，因为我们配置的时钟相位 CPHA=0，如图可知，采样时刻都是在 SCLK 的奇数边沿。当 CPOL=0 的时候，时钟的奇数边沿是上升沿，而 CPOL=1 的时候，时钟的奇数边沿是下降沿。所以 SPI 的采样时刻不是由上升/下降沿决定的。MOSI 与 MISO 数据线的有效信号在 SCLK 的奇数边沿保持不变，数据信号将在 SCLK 奇数边沿时被采样，在非采样时刻，MOSI 与 MISO 的有效信号才发生切换。

与此类似，当 CPHA=1 时，不受 CPOL 的影响，数据信号在 SCLK 的偶数边沿被采样，见下图所示的 CPHA=1 时 SPI 通讯时序图：

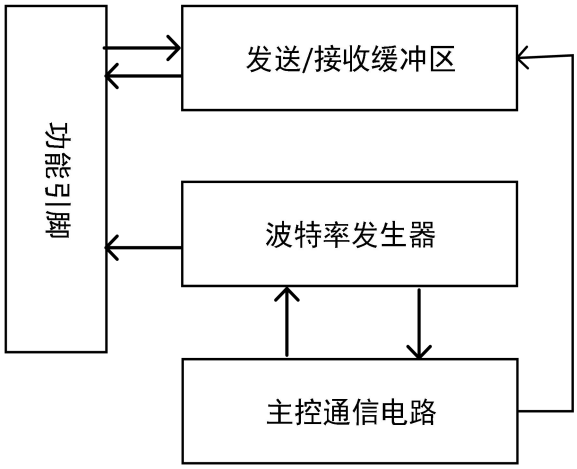


由 CPOL 与 CPHA 的不同状态，SPI 分为四种模式，主机与从机需要工作在相同的模式下才可正常通讯，实际工程中采用最多的是“模式 0”与“模式 3”，具体模式分类如下表所示：

SPI 模式	CPOL	CPHA	空闲状态 SCLK 时钟	采样时刻

0	0	0	低电平	奇数边沿
1	0	1	低电平	偶数边沿
2	1	0	高电平	奇数边沿
3	1	1	高电平	偶数边沿

4.7.2 BS32 的 SPI



BS32F030 内置 2 个 IIC 外设，具有如下特性：

1. 支持主机或从机模式，CPOL 和 CPHA 四种通信格式。
2. 支持全双工，半双工接收，半双工发送模式。
3. 4-16/32bit 数据大小选择，可配置数据 MSB/LSB 先发。
4. 支持 2/4/8/16/32/64/128/256 波特率预分频。
5. 主机片选信号由 SPI 的 CR1 寄存器 SSN 位控制，也可由硬件自动控制（NSSP 脉冲模式）。
6. 具有四个 32bit buffer(接收/发送各两个)，其数据量显示在 SR 寄存器的状态位中。
7. 可选硬件 CRC 校验，CRC 校验码长度可选 8/16 位，可编程 CRC 多项式，具有 CRC 错误标志和中断，计算得到的 CRC 校验码可读，可在所有数据发送完成后自动发送 CRC 帧。
8. CRC 和只接收模式下，需配置传输数据量 SPI_NUM，来控制时钟的停止。
9. 具有发送空和接收满状态标志，可选轮询/中断/DMA 方式处理数据。
10. 带 BUSY 标志位。
11. buffer 溢出，CRC 校验错误等错误事件有标志位和错误中断。
12. 从机最高支持 8M 通信，主机 16M。

通讯引脚

BS32F030 具有多个 SPI 外设，它们的 SPI 通讯信号引出到不同的 GPIO 引脚上，使用时须配置到这些指定的引脚，如下表所示：

引脚	SPI1	SPI2
NSS	PA4/PA15/PB0	PA8/PB9/PB12
CLK	PA1/PA5/PB3	PA0/PB8/PB10/PB13
MISO	PA6/PA11/PB4	PA3/PA9/PB2/PB6/PB14
MOSI	PA2/PA7/PA12/PB5	PA4/PA10/PB7/PB11/PB15

时钟控制逻辑

SCK 线的时钟信号，由波特率发生器根据“控制寄存器 CR2”中的 BR（Baud Rate 波特率控制）位控制，该位是对 f_{HCLK} 时钟的分频因子，对 f_{HCLK} 的分频结果就是 SCK 引脚的输出时钟频率。

注意：虽然 BS32F030 的 SPI 经由 f_{HCLK} 的理论最大分频频率是 48M/2=24M，但实际上因为 IO 翻转的速率有限，SPI 外设主机模式下 SCK 的最大值为 16M，从机模式下最大值为 8M。

通过配置“控制寄存器 CR1”的“CPOL 位”与“CPHA 位”可以把 SPI 设置为前面提到的四种 SPI 模式。

数据控制逻辑

SPI 的 MOSI 与 MISO 都连接到数据移位寄存器上，数据移位寄存器的数据流向为目标接收、发送缓冲区以及 MISO、MOSI 线。当向外发送数据的时候，数据移位寄存器以“发送缓冲区”为数据源，把数据一位一位地通过数据线发送出去；当从外部接收数据的时候，数据移位寄存器把数据线采样到的数据一位一位地存储到“接收缓冲区”中。通过写 SPI 的“数据寄存器 DR”把数据填充到发送缓冲区中，通讯读“数据寄存器 DR”，可以获取接收缓冲区中的内容。其中数据帧长度可以通过“控制寄存器 CR2”的“DS 位（DATASIZE）”配置为 4-16 位或 32 位模式；配置“控制寄存器 CR1”的“LSBFIRST 位”可选择 MSB 先行还是 LSB 先行。

整体控制逻辑

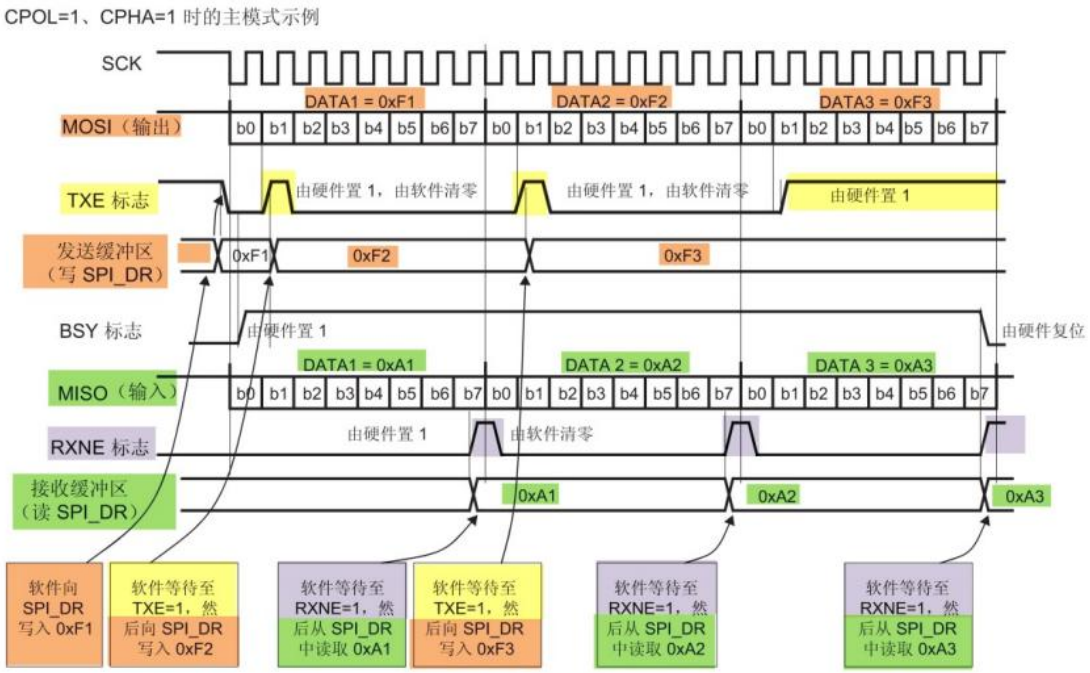
整体控制逻辑负责协调整个 SPI 外设，控制逻辑的工作模式根据我们配置的“控制寄存器（CR1/CR2）”的参数而改变，基本的控制参数包括前面提到的 SPI 模式、波特率、LSB 先行、主从模式、全双工（发送接收）/半双工（只发送/只接收）模式等等。在外设工作时，控制逻辑会根据外设的工作状态修改“状态寄存器 SR”，我们只要读取状态寄存器相关的寄存器位，就可以了解 SPI 的工作状态了。除此之外，控制逻辑还根据要求，负责控制产生 SPI 中断信号、DMA 请求及控制 NSS 信号线。

实际应用中，我们可以不使用 BS32 SPI 外设的标准 NSS 信号线，而是简单使用普通的 GPIO 口，软件控制它们的电平输出，从而产生通讯起始和停止信号。

4.7.3 BS32 SPI 通讯过程

BS32 使用 SPI 外设通讯时，在通讯的不同阶段它会对“状态寄存器 SR”的不同数据位写入参数，我们通过读取这些寄存器标志来了解通讯状态。

下图为 SPI 通讯过程中 BS32 作为主机端时的数据收发过程。



主机模式收发流程及事件说明：

1．控制 NSS 信号线，产生起始信号；

2．把要发送的数据写入到“数据寄存器 DR”中，该数据会被存储到发送缓冲区；

3．通讯开始，SCK 时钟开始运行。MOSI 把发送缓冲区中的数据一位一位地传输出去；MISO 则把数据一位一位地存储进接收缓冲区中；

4．当发送完一帧数据的时候，“状态寄存器 SR”中的“TXE 标志位”会被置 1，表示传输完一帧，发送缓冲区为空；类似的，当接收完一帧数据后，“RXNE 标志位”会被置 1，表示传输完一帧，接收缓冲区非空；

5．等待到“TXE 标志位”为 1 的时候，若还要继续发送数据，则再次往“数据寄存器 DR”写入数据即可（此时硬件会置位 TXE 为 0）；等待到“RXNE 标志位”为 1 时，通过读取“数据寄存器 DR”可以获取接收缓冲区中的内容（此时硬件会置位 RXNE 为 0）。

当我们使能了 TXE 或 RXNE 中断，TXE 或 RXNE 置 1 会产生 SPI 中断信号，进入同一个中断服务函数，到 SPI 中断服务程序后，可通过检查寄存器位来了解是哪一个事件，再分别进行处理。也可以使用 DMA 方式来收发“数据寄存器 DR”中的数据。

4.7.3.1 SPI 主机时钟和片选信号

时钟

SPI 使能打开并配置为主机时，时钟信号由内部模块产生并输出到 IO。主机发送时钟使用 SPI 内部分频后的时钟，主机接收时钟使用从 IO 口输入的时钟，从机发送接收都使用 IO 输入的时钟（一般就是从 SCK 线获取的主机时钟再经分频后的时钟）。

- 当 SPI 配置为主机只接收模式时，只要配置片选信号 SSN 拉低，时钟就会自动产生并连续

输出，直到接受完所需的数据量（通过 SPI_NUM 寄存器配置），此后 SPI 停止产生时钟，RXONLY 使能会被硬件自动拉低。只接收模式下必须配置数据量（SPI_NUM 寄存器）。

- 当 SPI 被配置为主机非只接收模式时，片选拉低后，SPI 自动检测当前是否有待发送数据。如有数据，就产生时钟并发送数据，直到一帧数据发送的末尾再次判断是否还有待发送数据。如有数据，则 SPI 连续发送数据，否则时钟停止，直到总线下一次写入数据。

满足数据发送条件时，预分频器开始工作，按照 BAUD[6:4]位配置的波特率对 hclk 进行预分频生成时钟。当使能 CRC 时，硬件会自动在最后一帧数据传输完成后传输 CRC 帧。

片选 NSS：

当 SPI 配置成主机时，NSS 信号由 SPI 内部产生并输出到 IO。有以下两种方式控制主机的 NSS 信号：

- Sfr（特殊功能寄存器）控制：通过配置 SPI_CR1 寄存器的 SSN 位控制片选信号，在此模式下必须保持 NSSP 位清零。
- 硬件控制（NSSP 模式）：通过置位 SPI_CR1 寄存器的 NSSP 位可使能 NSS 脉冲模式，NSSP 模式只有在 CPHA=0 时才能正常工作。此模式下片选信号完全由硬件自动控制，只有在数据传输期间保持低电平，其他时间保持高电平。且在数据连续传输过程中，NSS 信号会在两笔数据之间插入一个时钟周期的高电平，相当于用 NSS 脉冲将数据两两之间“断开”。注意，使用此脉冲模式时，CRC 校验和只接收模式不可用。

主机发送接收的片选都使用寄存器配置的 SSN 片选信号，但 NSSP 模式下主机接收使用从 IO 口返回的片选信号，从机发送接收都使用 IO 输入的片选信号。

4.7.3.2 SPI 发送控制

SPI 模块内含两个 32bit 的 TXBUF，用于存储总线写入的待发送数据。SPI 使能关闭时，TXBUF 将被复位。

TXBUF_NUM[5:4]用于标注当前 TXBUF 内含有多少数据，可通过状态寄存器将其读出。当 TXBUF_NUM =2 时，如果总线写入数据，则发生溢出，数据不写入 TXBUF，同时拉高 OVR 溢出标志和错误中断（错误中断使能打开时）。当 TXBUF_NUM =0 时，SPI 发送完当前数据帧后停止通信，直到总线写入下一笔数据。

TXE 位标记了 TXBUF 的数据状态，当其置 1 时，代表 TXBUF 未满，总线可写入数据。用户可选择以下三种方式处理发送数据：

- 轮询状态寄存器：读取 SPI_SR 寄存器，通过 TXE 位或 TXBUF_NUM[1:0]位来判断 TXBUF 的状态，当 TXBUF 未满时，写入数据。这种处理方式速度最快。
- TXE 中断：打开 TXEIE 中断使能，当 TXBUF 未满时，会产生 SPI 中断。在中断服务函数中读取 SPI_SR 寄存器判断 TXBUF 状态，并写入数据。如果系统处理中断的响应速度较慢，建议在 SPI 使能打开后先将 TXBUF 写满，再拉低 SS_N。

TXDMA：需要先配置好相应的 DMA 发送通道和数据，并打开 TXDMA_EN 使能。SPI 和 TXDMA_EN 使能都打开后会立即产生 TXDMA 请求，由 DMA 通过总线依次写入待发送数据。一次 TXDMA 传输完成后，必须关闭 TXDMA_EN 使能，下次使用时再打开，否则 TXDMA 请求

会一直保持为高。

在从机发送模式下，如果想要连续通信，则下一笔数据需要在当前数据帧发送完成前写入 TXBUF，否则发送会出错。

发送模式下，主机模式下使用内部生成的时钟和片选信号，从机模式下使用 IO 输入的时钟和片选信号。

如果使能了 CRC，则 SPI 将根据配置的 CRC 长度和 CRC 多项式在每个时钟采样沿对 TXD 端输出的数据进行 CRC 计算，一直到 SPI_NUM 寄存器配置的数据量传输完成，此后停止 CRC 计算，并自动将计算得到的 CRC 帧附在数据帧后面发送出去，CRC 帧与数据帧的大小端格式一致，帧长度由 CRCL 位决定。如遇到 DS（SPI 传输数据长度）与 CRC 长度不一致的情况，硬件可自动在 CRC 帧传输过程中将 DS 修正为对应值。

使能 CRC 时，NSS 在整个传输过程中必须保持为低，因此 NSSP 模式和 CRC 不兼容。在 CRC 模式下，发送完一批数据后（通过 SPI_NUM 寄存器配置),）CRC 使能被硬件拉低，下次使用时需要重新打开 CRC 使能。计算得到的 CRC 校验码将在传输完成后存储到 SPI_TXCRCR 寄存器中。

注：CRC 模式下只允许 DS 位配置为 8/16/32bit 这些 8 的倍数，CRC 长度只能配置为 8bit 或 16bit，但两者无需一致。

4.7.3.3 SPI 接收控制

SPI 在每个时钟采样沿对 RXD 端口输入的数据进行采样并保存。SPI 模块内含两个 32bit 的 RXBUF，用于存储 SPI 接收到的数据。SPI 使能关闭时，RXBUF 将被复位。

RXBUF_NUM[3:2]用于标注当前 RXBUF 内含有多少数据，可通过状态寄存器将其读出。当 RXBUF_NUM =2 时，如果接收到新的数据，则发生溢出，数据不存入 RXBUF（丢失），同时拉高 OVR 溢出标志和错误中断(错误中断使能打开时)。当 RXBUF_NUM =0 时，总线读取 SPI_DR 寄存器会返回一个无效的数据（前一次读取的数据），不会影响 RXBUF 及其状态位。因此，在读取 SPI_DR 寄存器之前必须读取 SPI_SR 寄存器，确保 RXBUF 内有数据，否则读出的数据无效。

RXNE 位标记了 RXBUF 的数据状态，当其置 1 时，代表 RXBUF 内有未读取的数据。用户可选择以下三种方式处理接收数据：

轮询状态寄存器：读取 SPI_SR 寄存器，通过 RXNE 位或 RXBUF_NUM[1:0]位来判断 RXBUF 的状态，当 RXBUF 存有数据时，总线将数据读走。这种处理方式速度最快。

RXNE 中断：打开 RXNEIE 中断使能，每次接收到一笔数据时，会产生 SPI 中断。在中断服务函数中读取 SPI_SR 寄存器判断 RXBUF 状态，并读取数据。

RXDMA：需要先配置好相应的 DMA 接收通道，并打开 RXDMA_EN 使能。每次接收到一笔数据时会产生 RXDMA 请求，由 DMA 通过总线依次读取数据。一次 RXDMA 传输完成后，必须关闭 RXDMA_EN 使能，下次使用时再打开。

如果配置为只接收模式，需要将传输数据量写入 SPI_NUM 寄存器（无需加上 CRC 帧），只接收模式下，NSS 在整个传输过程中必须保持为低，因此 NSSP 模式和只接收模式不兼容。接收

完一批数据后(通过 SPI_NUM 寄存器配置)，RXONLY 使能被硬件拉低，下次使用时需重新打开 RXONLY 使能。

如果使能了 CRC，则 SPI 将根据配置的 CRC 长度和 CRC 多项式在每个时钟采样沿对 RXD 端输入的数据进行 CRC 计算，一直到 SPI_NUM 寄存器配置的数据量传输完成，此后停止 CRC 计算，并再接收一个 CRC 帧。如遇到 DS（8/16/32bit）与 CRC 长度（8/16bit）不一致的情况，硬件可自动在 CRC 帧传输过程中将 DS 修正为对应值。

使能 CRC 时，NSS 在整个传输过程中必须保持为低，因此 NSSP 模式和 CRC 不兼容。在 CRC 模式下，接收完一批数据后(通过 SPI_NUM 寄存器配置)，CRC 使能被硬件拉低，下次使用时需重新打开 CRC 使能。接收到的 CRC 校验码将存入 RXBUF 中，计算得到的 CRC 校验码将在传输完成后存储到 SPI_RXCRCR 寄存器中。

注：CRC 模式只允许 DATASIZE 配置为 8/16/32bit 这些 8 的倍数，CRC 长度只能配置为 8bit 或 16bit，但二者无需一致。

4.7.3.4 状态标志和中断

中断使能：

- **ERRIE**：错误中断使能。发生 buffer 溢出或 CRC 校验错误事件时，可出中断
- **TXEIE**：发送空中断。TXBUF 未满时，可出中断。
- **RXNEIE**：接收满中断。RXBUF 内有接收到的数据时，可出中断。

状态标志：

- **OVR**：buffer 溢出标志。当 TXBUF 已满时总线写入数据或者 RXBUF 已满时接收到新数据，此位置 1，软件写 0 清零。可输出错误中断。
- **CRCERR**：CRC 校验错误标志。接收到的 CRC 值与 SPI 计算值不匹配时，此位置 1，软件写 0 清零。可输出错误中断。
- **BUSY**：SPI 忙标志。SPI 当前忙于通信时，此位拉高，传输完成后硬件拉低。主机模式下，只有在输出时钟时 BUSY 才拉高，其余时间保持为低；从机模式下，只要输入的片选为低（即该从机被选中），BUSY 信号就保持为高。此位用于软件判断传输是否结束。
- **TXBUF_NUM[5:4]**：TXBUF 内的数据量。
- **RXBUF_NUM[3:2]**：RXBUF 内的数据量。
- **TXE**：发送空标志。TXBUF 未满时，此位置 1。
- **RXNE**：接收满标志。RXBUF 内有接收数据时，此位置 1。

4.7.4 SPI 初始化结构体详解

与其他外设一致，BS32 LL 库提供了 SPI 初始化结构体及初始化函数来配置 SPI 外设。初始化

结构体及函数定义在库文件“bs32f0xx_ll_spi.h”与“bs32f0xx_ll_spi.c”中，编程时我们可以结合这两个文件内的注释以及芯片数据/参考手册来掌握外设 SPI 的使用。

```
typedef struct
{
  uint32_t TransferDirection;   //传输模式：全双工、只接收、只发送
  uint32_t Mode;                //SPI 主机、从机模式
  uint32_t DataWidth;           //数据位宽
  uint32_t ClockPolarity;       //时钟极性
  uint32_t ClockPhase;          //时钟相位
  uint32_t BaudRate;            //波特率
  uint32_t BitOrder;            //帧格式，数据 MSB 在前还是 LSB 在前
  uint32_t CRCCalculation;      //CRC 计算使能
  uint32_t CRCPoly;             //CRC 多项式
  uint32_t CRCLength;           //CRC 长度
} LL_SPI_InitTypeDef;
```

这些结构体成员说明如下：

TransferDirection

该成员配置 SPI 的通讯方向，可设置全双工（LL_SPI_FULL_DUPLEX），只接收（LL_SPI_HALF_DUPLEX_RX），只发送（LL_SPI_HALF_DUPLEX_TX）。

Mode

该成员配置 SPI 在主机模式(LL_SPI_MODE_MASTER)或从机模式(LL_SPI_MODE_SLAVE)，这两个模式的最大区别为 SPI 的 SCK 信号线的时序，SCK 的时序是由通讯中的主机产生的。若被配置为从机模式，BS32 的 SPI 外设将接受外来的 SCK 信号。

DataWidth

该成员可以选择 SPI 通讯的数据帧大小（LL_SPI_DATAWIDTH_NBIT），N 取值 4-16 或 32。

ClockPolarity 与 ClockPhase

这两个成员配置 SPI 的时钟极性 CPOL 与时钟相位 CPHA,这两个配置影响到 SPI 的通讯模式，具体参考前面的讲解。

时钟极性 CPOL，可设置为高电平（LL_SPI_POLARITY_HIGH）或低电平（LL_SPI_POLARITY_LOW）。

时钟相位 CPHA 则可以设置为 LL_SPI_PHASE_1EDGE（在 SCK 的奇数边沿采集数据）或 LL_SPI_PHASE_2EDGE（在 SCK 的偶数边沿采集数据）。

BaudRate

该成员设置波特率分频因子，分频后的时钟即为 SPI 的 SCK 信号线的时钟频率。成员参数可选：LL_SPI_BAUDRATEPRESCALER_DIV（2、4、8、16、32、64、128、256）。

BitOrder

所有串行的通讯协议都会有 MSB 先行（高位数据在前）还是 LSB 先行（低位数据在前）的问题，而 BS32 的 SPI 模块可以通过这个结构体成员，对这个特性进行控制：LSB 先行（LL_SPI_LSB_FIRST）、MSB 先行（LL_SPI_MSB_FIRST）。

CRCCalculation

该成员设置 CRC 计算使能:使能 LL_SPI_CRCCALCULATION_ENABLE；关闭 LL_SPI_CRCCALCULATION_DISABLE。

CRCPoly

该成员是 SPI 的 CRC 校验中的多项式，若我们使用 CRC 校验时，就使用这个成员的参数（多项式），来计算 CRC 的值。

CRCLength

该成员设置 CRC 的长度：8 位 CRC 长度（LL_SPI_CRC_8BIT）；16 位 CRC 长度（LL_SPI_CRC_16BIT）。

配置完这些结构体成员后，我们要调用 LL_SPI_Init 函数把这些参数写入到寄存器中，实现 SPI 的初始化。

4.7.5 SPI 编程要点（读写串行 FLASH）

为使工程更有条理，用户自定义过程中可以把读写 FLASH 相关的代码独立分开存储，方便以后移植。在“工程模板”之上新建“bsp_spi_flash.c”与“bsp_spi_flash.h”文件来实现目标。

编程要点

1. 初始化 SPI 通讯使用的目标引脚及端口时钟；
2. 使能 SPI 外设的时钟；
3. 配置 SPI 外设的模式、地址、速率等参数并使能 SPI 外设；
4. 编写基本 SPI 按字节收发的函数；
5. 编写对 FLASH 擦除及读写操作的函数；
6. 编写测试程序，对读写数据进行校验。

4.7.6 SPI 应用流程举例

4.7.6.1 主机全双工通信 中断处理数据

1. 配置 SPE=1：使能 SPI
2. 配置 MSTR=1：配置为主机模式
3. 配置 CPOL、CPHA、LSBFIRST 位：配置时钟极性、时钟相位、数据大小端

4. 配置 ERRIE=1：使能错误中断
5. 配置 RXNEIE=1：使能接收非空中断
6. 配置 BAUD 和 DATASIZE：配置波特率与数据传输长度
7. 预先写入两笔发送数据到 SPI_DR：否则 SPI 检测无数据，时钟停止
8. 拉低 SSN，开始通信
9. 在中断函数中读取 SPI_SR 寄存器，如果错误标志置位，则通信出现错误， 需要重新发起通信；如果 RXNE=1 或 RXBUF_NUM[3:2]不为零，表示 RXBUF 里有数据待读取，需要读取 SPI_DR 寄存器直到 RXBUF 空；如果 TXE=1 或 TXBUF_NUM[5:4]小于 2，表示 TXBUF 未满，需要向 SPI_DR 寄存器写入发送数据直到 TXBUF 满。
10. 接收到最后一笔数据后，在中断里轮询 SPI_SR 寄存器，直到 BUSY=0，表示传输完成。
11. 再次确认 SPI_SR 寄存器的错误标志位没有拉高，如果拉高，则通信出错，需要重新发起通信。
12. 拉高 SSN
13. 关闭 SPE 使能：禁用 SPI

4.7.6.2 主机只发送模式 NSSP 脉冲模式 DMA 处理数据

1. 配置好 DMA 发送通道和数据，打开 DMA 通道使能中断
2. 配置 SPE=1：使能 SPI
3. 配置 NSSP=1：使能 NSS 脉冲模式
4. 配置 MSTR=1：配置为主机模式
5. 配置 TXONLY=1：使能只发送模式
6. 配置 CPOL、CPHA、LSBFIRST 位：配置时钟极性、时钟相位、数据大小端
7. 配置 ERRIE=1：使能错误中断
8. 配置 BAUD 和 DATASIZE：配置波特率与数据传输长度
9. 配置 TXDMA_EN=1
10. 等待 DMA 通道完成所有数据传输并出中断，在中断函数中轮询 SPI_SR 寄存器，如果错误标志置位，则通信出现错误，需要重新发起通信；如果 BUSY=0，表示数据传输完成。
11. 配置 TXDMA_EN=0
12. 再次确认 SPI_SR 寄存器的错误标志位没有拉高，如果拉高，则通信出错，需要重新发起通信。
13. 配置 NSSP=0：禁用 NSS 脉冲模式
14. 关闭 SPE 使能：禁用 SPI

4.7.6.3 主机只接收模式 CRC 校验

1. 配置 SPE=1：使能 SPI
2. 配置 RXONLY=1：使能只接收模式
3. 配置 CRCEN=1：使能 CRC 计算
4. 配置 CRCL：CRC 长度配置
5. 配置 MSTR=1：配置为主机模式
6. 配置 CPOL、CPHA、LSBFIRST 位：配置时钟极性、时钟相位、数据大小端
7. 配置 ERRIE=1：使能错误中断
8. 配置 RXNEIE=1：使能接收非空中断
9. 配置 BAUD 和 DATASIZE（只能配成 8 的倍数）
10. 配置 SPI_NUM 寄存器，写入需要传输的数据量
11. 配置 SPI_CRCPR 寄存器：配置用于 CRC 计算的多项式
12. 拉低 SSN，开始通信
13. 在中断函数中读取 SPI_SR 寄存器，如果错误标志置位，则通信出现错误，需要重新发起通信；如果 RXNE=1 或 RXBUF_NUM[3:2]不为零，表示 RXBUF 里有数据待读取，需要读取 SPI_DR 寄存器。
14. 在中断函数中读取 SPI_SR 寄存器，如果 BUSY 等于 0，表示数据传输完成，将 RXBUF 内的数据全部读出来，注意最后一笔数据为接收到的 CRC 帧；如果 CRCERR 位置 1，表示 CRC 校验错误，需要重新发起通信。读取 SPI_RXCRCR 可读出 SPI 计算得到的 CRC 校验码。
15. 如需再次发起一次只接收或 CRC 传输，需要再次开启 RXONLY 或 CRC 使能。
16. 再次确认 SPI_SR 寄存器的错误标志位没有拉高，如果拉高，则通信出错，需要重新发起通信。
17. 拉高 SSN
18. 关闭 SPE 使能：禁用 SPI

4.7.6.4 从机全双工通信 DMA 处理数据

1. 配置好 DMA 发送与接收通道，打开 DMA 通道中断使能
2. 配置 SPE=1：使能 SPI
3. 配置 CPOL、CPHA、LSBFIRST 位：配置时钟极性、时钟相位、数据大小端
4. 配置 ERRIE=1：使能错误中断

5. 配置 BAUD 和 DATASIZE：配置波特率与数据传输长度
6. 配置 CR2_TXDMA_EN=1，CR2_RXDMA_EN=1：使能 DMA 发送与接收
7. 等待主机拉低对应从机片选，开始通信
8. 在 DMA 中断函数中轮询 SPI_SR 寄存器直到 BUSY 信号拉低（从机模式下，只有片选拉高以后，BUSY 信号才会拉低），表示传输完成，如果错误标志置位，则通信出现错误，需要重新发起通信。
9. 配置 CR2_TXDMA_EN=0，CR2_RXDMA_EN=0
10. 关闭 SPE 使能：禁用 SPI

4.7.6.5 注意事项

- 主机只接收模式下时钟连续产生，必须配置传输数据量，否则无法控制时钟停止
- 使能 CRC 时，必须配置传输数据量（不包含 CRC 帧），否则 CRC 无法正常校验
- 使能 CRC 时，DATASIZE 必须配置为 8 的倍数；接收到的 CRC 帧存入 RXBUF
- NSSP 模式只有在 CPHA=0 时才能正常工作，且 NSSP 模式下无法使用 CRC 和只接收模式（（片选拉高会导致计数器被清零，无法计数））
- 建议先读取 SPI_SR 寄存器，确保 RXBUF 内有数据后再读取 SPI_DR，否则读出的数据无效
- CRC 校验模式或只接收模式下，一批数据（通过 SPI_NUM 配置）传输完成后，CRC 使能或 RXONLY 使能会被硬件拉低，下次使用时软件需重新开启使能
- 每次 DMA 通道数据处理完成后，必须关闭 SPI 内部的 DMA 使能，下次使用时再打开

4.8 ADC 模块介绍

BS32F030XX 内嵌一个 12 位与 8 位分辨率可选的线性逐次逼近 ADC，8 位分辨率的数据采用右对齐格式。该 ADC 有 41 个通道（39 个外部输入通道+1 个内部参考电压通道+1 个内部温度传感器通道）。

该 ADC 支持单通道转换模式与连续多通道 DMA 模式。采样时间和转换速度可配置，ADC 在转换完成以后产生中断，时间触发标志可通过软件触发和硬件触发。ADC 支持 Sleep Mode 模式下唤醒。

启动 ADC 转换的方式：

- 软件触发
- 内部事件（定时器 1 事件）硬件触发

4.8.1 ADC 功能解析

掌握 ADC 的基础功能，就可以对 ADC 有一个整体把握，在编程时可以更轻松。

电压输入范围

我们设计原理图时一般把 VDD 和 VREF+接到一起，ADC 的输入电压范围就取决于 VDD 的值，常用的值是 3.3V。

如果我们想让输入的电压范围变宽，去到可以测试负电压或者更高的正电压，我们可以在外部加一个电压调理电路，把需要转换的电压抬升或者降压到 VREF+，这样 ADC 就可以测量。

输入通道

我们确定好 ADC 输入电压之后，还需要明确电压怎么输入到 ADC 中。这里引入通道的概念，BS32 的 ADC 多达 41 个通道，其中 39 个外部通道对应着不同的 IO 口，具体对应关系在 SDK 的 bs32f0xx_ll_adc.h 中可以查到。剩下的两个内部通道一个是内部参考电压通道与内部温度传感器通道。

```
/** @defgroup ADC_LL_EC_CHANNEL_ADDRESS ADC Channel Address 通道地址
#define LL_ADC_CHANNEL_0                    (0x00) /*!< ADC channel 0  on PA0 */
#define LL_ADC_CHANNEL_1                    (0x01) /*!< ADC channel 1  on PA1 */
#define LL_ADC_CHANNEL_2                    (0x02) /*!< ADC channel 2  on PA2 */
#define LL_ADC_CHANNEL_3                    (0x03) /*!< ADC channel 3  on PA3 */
#define LL_ADC_CHANNEL_4                    (0x04) /*!< ADC channel 4  on PA4 */
#define LL_ADC_CHANNEL_5                    (0x05) /*!< ADC channel 5  on PA5 */
#define LL_ADC_CHANNEL_6                    (0x06) /*!< ADC channel 6  on PA6 */
#define LL_ADC_CHANNEL_7                    (0x07) /*!< ADC channel 7  on PA7 */
#define LL_ADC_CHANNEL_8                    (0x08) /*!< ADC channel 8  on PA8 */
#define LL_ADC_CHANNEL_9                    (0x09) /*!< ADC channel 9  on PA9 */
#define LL_ADC_CHANNEL_10                   (0x0a) /*!< ADC channel 10 on PA10*/
#define LL_ADC_CHANNEL_11                   (0x0b) /*!< ADC channel 11 on PA11*/
```

```
#define LL_ADC_CHANNEL_12                  (0x0c) /*!< ADC channel 12 on PA12*/
#define LL_ADC_CHANNEL_13                  (0x0d) /*!< ADC channel 13 on PA13*/
#define LL_ADC_CHANNEL_14                  (0x0e) /*!< ADC channel 14 on PA14*/
#define LL_ADC_CHANNEL_15                  (0x0f) /*!< ADC channel 15 on PA15*/
#define LL_ADC_CHANNEL_16                  (0x10) /*!< ADC channel 16 on PB0 */
#define LL_ADC_CHANNEL_17                  (0x11) /*!< ADC channel 17 on PB1 */
#define LL_ADC_CHANNEL_18                  (0x12) /*!< ADC channel 18 on PB2 */
#define LL_ADC_CHANNEL_19                  (0x13) /*!< ADC channel 19 on PB3 */
#define LL_ADC_CHANNEL_20                  (0x14) /*!< ADC channel 20 on PB4 */
#define LL_ADC_CHANNEL_21                  (0x15) /*!< ADC channel 21 on PB5 */
#define LL_ADC_CHANNEL_22                  (0x16) /*!< ADC channel 22 on PB6 */
#define LL_ADC_CHANNEL_23                  (0x17) /*!< ADC channel 23 on PB7 */
#define LL_ADC_CHANNEL_24                  (0x18) /*!< ADC channel 24 on PB8 */
#define LL_ADC_CHANNEL_25                  (0x19) /*!< ADC channel 25 on PB9 */
#define LL_ADC_CHANNEL_26                  (0x1a) /*!< ADC channel 26 on PB10*/
#define LL_ADC_CHANNEL_27                  (0x1b) /*!< ADC channel 27 on PB11*/
#define LL_ADC_CHANNEL_28                  (0x1c) /*!< ADC channel 28 on PB12*/
#define LL_ADC_CHANNEL_29                  (0x1d) /*!< ADC channel 29 on PB13*/
#define LL_ADC_CHANNEL_30                  (0x1e) /*!< ADC channel 30 on PB14*/
#define LL_ADC_CHANNEL_31                  (0x1f) /*!< ADC channel 31 on PB15*/
#define LL_ADC_CHANNEL_34                  (0x22) /*!< ADC channel 34 on PC13*/
#define LL_ADC_CHANNEL_35                  (0x23) /*!< ADC channel 35 on PC14*/
#define LL_ADC_CHANNEL_36                  (0x24) /*!< ADC channel 36 on PC15*/
#define LL_ADC_CHANNEL_41                  (0x29) /*!< ADC channel 41 on PF0 */
#define LL_ADC_CHANNEL_42                  (0x2a) /*!< ADC channel 42 on PF1 */
#define LL_ADC_CHANNEL_43                  (0x2b) /*!< ADC channel 43 on PF6 */
#define LL_ADC_CHANNEL_44                  (0x2c) /*!< ADC channel 44 on PF7 */
#define LL_ADC_CHANNEL_VREF                (0x2d) /*!< ADC channel 45 is vref*/
#define LL_ADC_CHANNEL_TEMPSENSOR          (0x2e) /*!< ADC channel 46 is tempsensor */
```

触发源

通道选好后，通过 ADC_INSELx 寄存器来选择对应通道的 ADC 功能，转换顺序按照其寄存器位由低到高依次转换。开始触发首先要通过 ADC_CFG3 寄存器写 ADCTRG 位来进行转换触发选择：软件触发与硬件触发。

当选择软件触发时，给 ADSTART[0]写 1 就能发起转换。当选择硬件触发时，TIMER1 触发后就能发起转换。

转换时间

- ADC 时钟

ADC 外设时钟 ADC_CLK 由系统时钟（BS32F030 为 48M）经过分频产生，最大是 24M，分频因子由 ADC_CFG1 寄存器的 ADCK 位来配置，可以是 2/3/4/8/16/32 分频。

- 采样时间

ADC 使用若干个 ADC_CLK 周期对输入的电压进行采样，采样的周期数可通过 ADC 采样时间配置寄存器 ADC_SPT 进行配置。

ADC 的转换时间与 ADC 的输入时钟和采样时间以及 ADC_CFG1 寄存器的 ADCWNUM 位配置有关，公式为：ADC 转换时间 T= sample_time（采样时间）+time2（采样转换间隔时间+采样延迟时间）+ time3（固定数据转换时间）。

sample_time= (ADC_SPT+1)*1*Tadc_clk(ADC_SPT 配置 0/1/2 时，sample_time=3)

time2=（ADCWNUM+4+SAMDEL）* Tadc_clk(SAMDEL=0:0；1:2；2:4；3:8；4:16；5:32)

time3=（2*1+12）*Tadc_clk（2 为固定时间，12 与采样分辨率有关）

时序要求：（ADCWNUM+4）>= 4，(ADC_SPT+1)*1>= 3。具体时序如下图所示：

如上图所示，在 12 位分辨率的前提下，ADC 采样最短时间周期由 21 个 ADC_CLK 组成。

数据寄存器

ADC 转换后的数据存放在 12 位的扫描结果寄存器 ADC_RDATA 中。通道有 41 个，但是结果寄存器只有一个，若使用多通道转换，为防止数据被覆盖，应当开启 DMA 模式或通道转换完成后就把数据取走。

中断

数据转换结束后，可产生 ADC 转换结束中断。和其他中断一致，有相应的中断标志位和中断使能位，我们可以根据中断类型写相应配套等待中断服务程序。

电压转换

模拟电压经过 ADC 转换后，是一个 12 位的数字值，如果通过串口以 16 进制打印出来，可读性较差。因此可以把数字电压转换为模拟电压，且可通过万用表与实际的模拟电压对比，观察转换是否准确。

常用的开发板会把 ADC 的输入电压范围设定在：0-3.3V，因为 ADC 是 12 位的，那么 12 位满量程对应的就是 3.3V，12 位满量程对应的数字值是：2^{12}。数值 0 对应的就是 0V。如果转换后的数值为 X，X 对应的模拟电压为 Y，那么就有等式成立：$2^{12}/3.3=X/Y$，即 Y=（3.3*X）/2^{12}。

4.8.2 ADC 初始化结构体详解

LL 库函数对每个外设都建立了一个初始化结构体 xxx_InitTypeDef(xxx 为外设名称)，结构体成员用于配置外设工作参数，并由 LL 库函数 xxx_Init()调用这些设定参数进入设置外设相应的寄存器，达到配置外设工作环境的目的。

结构体 xxx_InitTypeDef 和库函数 xxx_Init 配合使用可以高效进行外设的开发，理解了结构体 xxx_InitTypeDef 每个成员意义基本上就对外设的应用有了初步的了解。结构体 xxx_InitTypeDef

定义在 bs32f0xx_ll_xxx.h 文件中，库函数 xxx_Init 定义在 bs32f0xx_ll_xxx.c 文件中，编程时我们可以结合这两个文件内注释使用。

LL_ADC_InitTypeDef 结构体

LL_ADC_InitTypeDef 结构体定义在 bs32f0xx_ll_adc.h 文件内，具体定义如下：

```
typedef struct
{
  uint32_t ChannelAddr;        //通道地址
  uint32_t Clock;              //时钟频率
  uint32_t Resolution;         //采样分辨率
  uint32_t SampleDelayTime;    //采样延迟时间
  uint32_t SamCompInterval;    //采样时序与比较时序间隔时间
  uint32_t SamConverInterval;  //采样完到 ADC 转换间隔时间
  uint32_t Current;            //偏置电流选择
  uint32_t CompCtrlSel;        //比较器失调消除选择
  uint32_t FilterSel;          //滤波选择
  uint32_t SampleTime;         //采样时间=(SampleTime+1)*4 Tadc_clk
  uint32_t TriggerSource;      //设置 ADC 转换触发源
  uint32_t OverrunMode;        //设置 ADC 溢出管理
} LL_ADC_InitTypeDef;
```

ChannelAddr:通过 ADC_CHADDR 寄存器进行 ADC 通道地址配置。单次扫描时（非 DMA 模式）配置，配置范围：0-44（引脚输入通道）；45（参考电压 v_{ref} 输入通道）；46（内部温度 ts 输入通道）。在连续模式（DMA 模式）下，地址随扫描通道而变化，读地址寄存器时可能会因时间不够而读得不对。

Clock：通过 ADC_CFG1 寄存器的 ADCK 位进行 ADC 的时钟分频选择，可选分频 2/3/4/8/16/32。

Resolution：通过 ADC_CFG1 寄存器的 RES 位进行 ADC 的数据分辨率配置，可选 8 位和 12 位分辨率。

SampleDelayTime：通过 ADC_CFG1 寄存器的 SAMDEL 位进行 ADC 的采样延迟时间配置，可选采样延迟时间为 0/2/4/8/16/32(ADC_CLK)。

SamCompInterval:通过 ADC_CFG1 寄存器的 SAMBG 位进行 ADC 的采样时序与比较时序间隔选择，可选间隔 0/1 个(ADC_CLK)，不计入 AD 转换周期。

SamConverInterval：通过 ADC_CFG1 寄存器的 ADCWNUM 位进行 ADC 的采样完毕后到 ADC 转换的间隔时间,其值为 4+ADCWNUM(ADC_CLK)。

Current：通过 ADC_CFG2 寄存器的 ADC_I_SEL 位进行 ADC 的偏置电流大小选择，包含比较器偏置电流与运放偏置电流两部分。

CompCtrlSel：通过 ADC_CFG2 寄存器的 ADC_CTRL_SEL 位进行 ADC 的比较器失调消除选择配置，置 0 时先采样再失调消除；置 1 时所有开关一起断开，默认值为 1。

FilterSel：通过 ADC_CFG2 寄存器的 ADC_FILTER_R_SEL 位进行 ADC 的输入信号滤波选择配置，置 0 时不加 RC 滤波；置 1 时加 RC 滤波。

SampleTime：通过 ADC_SPT 寄存器进行 ADC 的采样时间配置，最小值为 3 个(ADC_CLK)。

TriggerSource：通过 ADC_CFG3 寄存器的 ADCTRG 位进行 ADC 的转换触发源选择。置 0 时为禁止硬件触发，选择软件触发，此时给 ADSTART[0]写 1 就能发起转换；置 1 时选择硬件(TIMER1 事件)触发开始转换。

OverrunMode：通过 ADC_CFG3 寄存器的 OVRMOD 位进行 ADC 的溢出管理模式。置 0 时检测到溢出，ADC_RDATA 寄存器保留原有数据；置 1 时检测到溢出，ADC_RDATA 寄存器会被上一转换数据覆盖。

4.8.3 BS32 ADC 功能概述

4.8.3.1 通道选择

单通道选择

管脚控制寄存器（ADC_INSELx）用来开启或禁止模拟输入管脚的 ADC 控制功能，每个通道都有专用的选择位。ADC 通道地址选择寄存器（ADC_CHADDR）用来控制当前扫描通道的选择。对于 39 路 GPIO 引脚的输入，在选择通道后要开启对应管脚的 ADC 功能。2 路内部模拟输入（温度传感器、内部参考电压）只需通道地址 ADC_CHADDR 选择。

多通道选择

41 路通道都通过编程（ADC_INSELx）寄存器控制，编程管脚控制寄存器来确定需要转换的通道，每个（ADC_INSELx）位均使能一个通道，编程 1 使能通道；通道地址寄存器(ADC_CHADDR)则会随扫描通道而变化，一次扫描很快完成，扫描过程中若读地址寄存器时可能会因时间不够而读得不对。通道扫描顺序是从 ADC_INSELx 寄存器的低位到高位，一次最多可扫描 41 个通道。

4.8.3.2 触发选择

ADC 控制器有两种触发方式可选软件触发和硬件触发。通过 ADCTRG 来选择转换的触发类型。当选择软件触发时，写入 ADSTART 就能发起转换。当选择硬件触发时，TIMER1 事件到来触发后就能发起转换。

- 选择软件触发：ADSTART=0 时，ADC 模块禁止。配置一次 adc_ start =1，进行一次 ADC 扫描，转换完成后，结果保存在数据寄存器（ADC_RDATA）中，然后触发中断。
- 选择硬件触发：无 TIMER1 事件时，ADC 模块禁止。TIMER1 事件到来时，进行一次 ADC 扫描，转换完成后，结果保存在数据寄存器（ADC_RDATA）中，然后触发中断。

4.8.3.3 转换模式

ADC 有两种转换模式可选，通过配置 ADC_CFG3 寄存器的 DMAEN 位：

- 非 DMA 模式单次转换（DMAEN=0）：

ADC 触发后开始扫描，在扫描一次结束后，硬件自动关闭 ADC，若要开启下一次扫描，则需软

件/硬件再次触发；扫描过程中若置 ADSTART =0，则此次扫描立即停止，模块内部相关信号复位。

注意：ADSTART =1，检测到中断时，硬件让 ADSTART =0。扫描过程中不允许配置 ADSTART。同时如果通道地址对应的 IO 口没有使能为 ADC 口，扫描也无法启动。

➢ DMA 模式连续转换（DMAEN=1）：

选择连续转换时要开启 DMAEN，配置 ADC_INSELx 来确定需要转换的通道。ADC 触发后开始扫描，在每一个通道扫描结束后，开启下一个通道扫描，直至配置的通道扫描完成；

每个通道扫描结束后都产生一个 DMA 请求，该请求信号直到 DMA 读取数据后拉低。如果数据未由 DMA 及时读取，则会产生溢出，ADC 可以继续转换。达到 DMA 设置的接收数据长度后，DMA 传输结束，产生 DMA 中断。

DMA 中断在 DMA 模块中产生，ADC 的转换中断使能关闭，溢出中断使能打开。

如果转换中关闭 DMAEN 使能，则此时为单次转换模式（DMAEN=0），若此时通道地址 ADC_CHADDR 对应的 IO 口(ADC_INSELx)使能为 ADC 口，则此通道扫描完后停止，内部相关信号复位；若此时通道地址对应的 IO 口没有使能为 ADC 口，则扫描立即停止；若要重新扫描所配置的通道，则需再配置打开 DMAEN 使能和 ADSTART。

使用 DMA 时需要配置的寄存器：

- 配置 ADC_INSELx 来确定需要转换的通道；
- 配置 DMA_CCDBPR 来确定主数据地址(DMA_SRAM_BASE)；
- 配置 DMA_CFGR 的 EN 来打开 DMA 控制器使能；
- 配置 DMA_CENSR 和 DMA_CIER 打开 DMA 中 ADC 的传输通道使能和通道中断；
- 配置 DMA 通道的 DMA_CSEAR 指向 ADC_RDATA 地址 (源地址)；
- 配置 DMA 通道的 DMA_CDEAR 指向 SRAM 地址(目的地址)；
- 配置 DMA_CCFGRx 来确定传输数据大小、传输数据量以及传输模式；
- 配置 ADC_CFG3 的 DMAEN 打开 ADC 的 DMA 使能；
- 配置 ADC_ICSR 的 OVREN 来打开 ADC 溢出使能；
- 配置触发 ADC。

注意：DMAEN 位完全由软件编程，若要从连续模式切换为单次模式，则要先软件编程 DMAEN 位为 0，否则再次触发启动还是连续模式。

4.8.3.4 ADC 启动转换

➢ 通过软件将 ADSTART 置 1 的方式启动 ADC 转换。

在 ADCTRG= 0 时，ADSTART 置 1 后会立即开始转换（软件触发）。

ADSTART 位由软件清零：在软件触发模式（ADCTRG= 0）下，软件给 ADSTART 位写 0

清零；

➢ 通过硬件将 ADSTART 置 1 的方式启动 ADC 转换。

在 ADCTRG≠0 时，会在硬件触发事件到来后将 ADSTART 置 1 开始转换，软件写 ADSTART 无效。

ADSTART 位由硬件清零：单次模式下，只要转换结束就清零；连续模式下，所有配置通道转换结束时就清零。

注意：所有情况下，软件给 ADSTOP 位写 1 后清零。

4.8.3.5 ADC 转换停止

可通过软件将 ADC_CFG3 寄存器中的 ADSTOP 置 1，停止任何正在进行的转换。此操作会复位 ADC，ADC 将处空闲状态，准备好进行新操作。

如果 ADSTOP 位由软件置 1，则会中断任何正在进行的转换，并会丢弃转换结果（ADC_RDATA 寄存器不会更新为当前转换结果）。扫描序列也会中止并会复位（这意味着重启 ADC 将重新开始新的序列）。此过程完成后，ADSTOP 位和 ADSTART 位均会被硬件清零，软件必须等待 ADSTART=1，然后才能开始新的转换。

- ADTRG=0

  选择软件触发启动 ADC，可配置 ADSTART 和 ADSTOP 位来停止。

- ADTRG=1

  选择硬件触发启动 ADC，配置 ADSTOP 位来停止，配置 ADSTART 位无效。

4.8.3.6 DMA 溢出

在连续模式下，如果转换后的数据未由 DMA 及时提取，在新转换生成数据之前，会由溢出标志指示数据溢出事件。即如果 ADC_INT_CFG 寄存器中的 ADC_OVR_EN 位置 1，可产生溢出中断。

如果发生溢出情况，ADC 会保持工作状态并可继续进行转换，除非通过软件将 ADSTOP 位寄存器置 1，从而停止并复位序列。溢出标志可通过由软件向 ADC_OVR_CLR 写入 1 的方式来清零。

可对 ADC_CFG3 寄存器中的 OVRMOD 位进行编程，从而配置发生溢出事件时是要保留数据还是要覆盖数据：

- OVRMOD=0

  溢出事件会保留 ADC_RDATA 数据寄存器的数据，防止其被覆盖：保留原数据，并丢弃新的转换结果。如果溢出标志保持为 1，可继续进行转换，但会丢弃所得的数据。

- OVRMOD=1

  数据寄存器会由上一次转换结果覆盖，之前未读出的数据会丢失。如果溢出标志保持为 1，

可继续进行转换，ADC_RDATA 寄存器始终为最新转换得出的数据。

4.8.3.7 注意事项

- 触发启动

  无论是非 DMA 模式还是 DMA 模式，先把相关寄存器配置好：

  - 若选择的触发方式是软件触发，则最后配置 ADSTART =1 启动；扫描过程中不允许配置 ADSTART。
  - 若选择硬件触发，则最后配置 ADCTRG =1 等待启动；扫描过程中不允许配置 ADCTRG。

- 模式切换

  DMAEN 位完全由软件编程，若要从连续模式切换为单次模式，则要先软件编程 DMAEN 位为 0，否则再次触发启动还是连续模式。

- 时钟分频

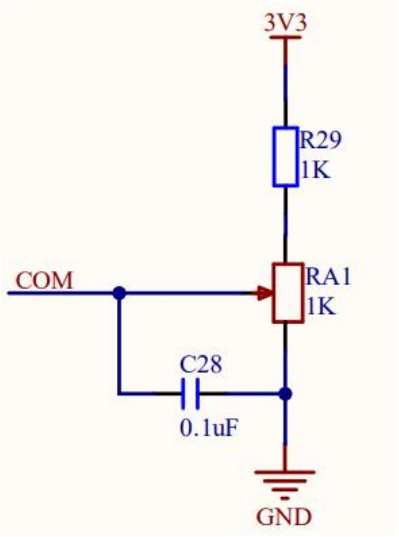
  pclk 时钟频率很慢时，数据不会更新，也不会产生溢出信号，不建议配置 pclk 时钟慢而 adc_clk 时钟快。若要 ADC 正确更新数据，配置时钟分频时应当注意这点。

4.8.4 ADC 采集实验（单通道）

单通道采集是 BS32 的 ADC 基础功能，我们可以通过改变开发板上可调电阻的触点，来实现引脚电压的采集变化并可打印至 PC 端串口调试助手。单通道采集适用 AD 转换完成中断，在中断服务函数中读取数据，不使用 DMA 传输，一般在多通道采集时才使用 DMA 传输。

硬件设计

开发板板载一个可调电阻，电路设计如下：



可调电阻的动触点通过连接 BS32 芯片的 ADC 通道引脚。当我们调节可调电阻阻值时，其动触

点电压也随之改变，电压变化范围为 0-3.3V，或是开发板默认的 ADC 电压采集范围。

软件设计

我们可以编写两个 ADC 驱动文件，bsp_adc.h 和 bsp_adc.c,用来存放 ADC 所用 IO 引脚的初始化函数以及 ADC 配置相关函数。

编程要点

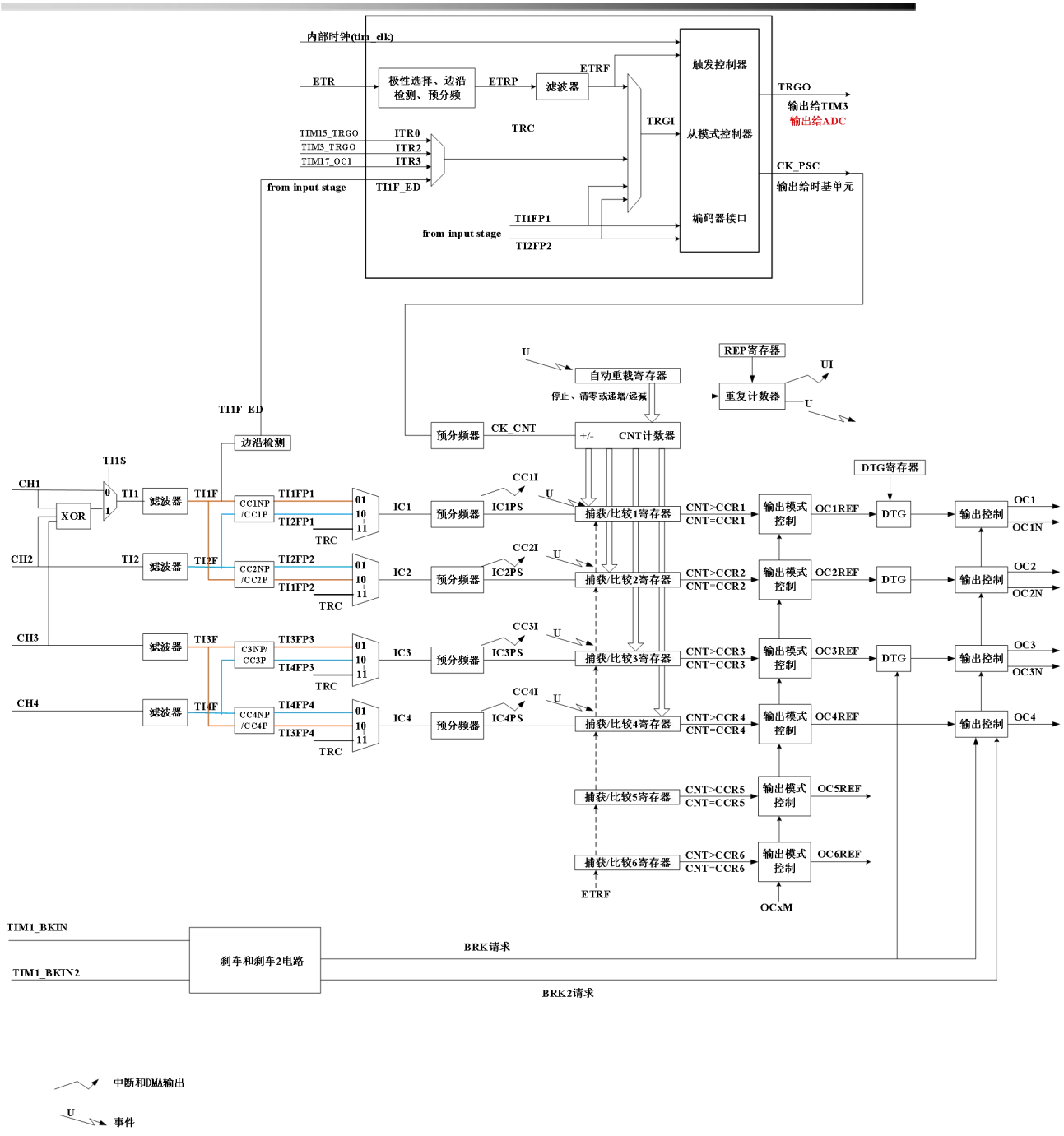
1. 初始化 ADC 用到的 GPIO；
2. 设置 ADC 的工作参数并初始化；
3. 设置 ADC 工作时钟；
4. 设置 ADC 采样转换相关时间参数；
5. 配置使能 ADC 转换完成中断，在中断内读取转换完数据；
6. 使能 ADC；
7. 使能软件触发 ADC 转换。

ADC 转换结果数据使用中断方式读取，此处没有使用 DMA 进行数据传输，连续多通道采样时使用 DMA 传输。

4.9 TIMER 模块介绍

BS32F030XX 芯片的高级控制定时器（TIM1）包含一个 16 位自动重载计数器，该计数器由可编程预分频器驱动。TIM1 定时器的主要特性如下：

- 16 位递增、递减、递增/递减自动重载计数器
- 16 位可编程预分频器，计数器时钟频率的分频系数为 1 ~ 65536 之间的任意整数值
- 多达 6 个独立通道，可用于：
  - 输入捕获（通道 5 和通道 6 除外）
  - 输出比较
  - PWM 生成（边沿和中心对齐模式）
  - 单脉冲模式输出
- 带可编程死区时间的互补输出
- 使用外部信号控制定时器且实现多个定时器互连的同步电路
- 重复计数器，用于在给定数目的计数器周期后更新定时器寄存器
- 2 个刹车输入，用于将定时器的输出信号置于用户可选的安全配置中
- 发生如下事件时生成中断/DMA 请求：
  - 更新：计数器上溢/下溢、计数器初始化（通过软件或内部/外部触发）
  - 触发事件（计数器启动、停止、初始化或通过内部/外部触发计数）
  - 输入捕获
  - 输出比较
  - 刹车输入（中断请求）
- 支持用于定位的增量（正交）编码器和霍尔传感器电路
- 触发输入 ETR 用作外部时钟或逐周期电流管理



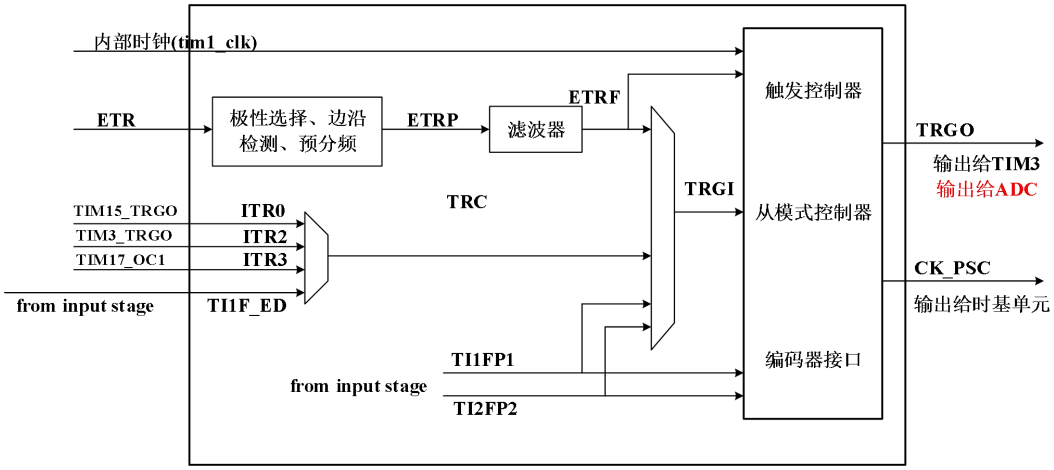
高级定时器控制框图

4.9.1 时钟/触发控制器

时钟/触发控制器允许用户选择计数器的时钟源，输入触发信号和输出信号。

控制器部分包括触发控制器、从模式控制器以及编码器接口。触发控制器用来针对片内外设输出触发信号，比如为其他定时器提供时钟和触发 ADC 转换。

编码器接口专门针对编码器计数而设计。从模式控制器可以控制计数器进行复位、启动、递增/递减、计数操作。



4.9.1.1 预分频时钟（CK_PSC）

高级控制定时器的计数器时钟有四个时钟源可选：

- 内部时钟源 tim_clk
- 外部时钟模式 1：外部输入引脚或内部触发输入（ITRx）
- 外部时钟模式 2：外部触发输入 ETR
- 编码器模式

内部时钟源 tim_clk

内部时钟 tim_clk 来自于芯片内部，其时钟源可选 HSE（4-16M）或 PLL_CLK（48M）经过 CLK_PLL_DIV1 或 CLK_PLL_DIV2 分频后。如果通过 TIM1_SMCR 寄存器禁止从模式控制器（SMS=000），则 TIM1_CR1 寄存器的 CEN 位、DIR 位和 TIM1_EGR 寄存器的 UG 位为实际控制位，并且只能通过软件进行更改（UG 除外，仍保持自动清零）。当对 CEN 位写入 1 使能计数器时，预分频器的时钟直接由内部时钟 tim_clk 提供。

外部时钟源模式 1

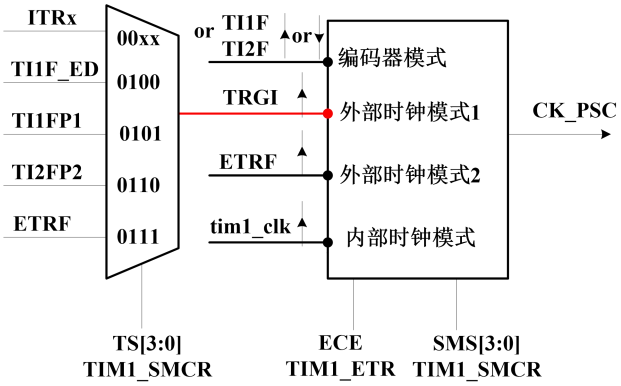
当使用外部时钟模式 1 的时候，时钟信号来自于定时器的输入通道或内部定时器触发输入。

时钟信号来自于定时器输入通道时，考虑到外部时钟信号的频率过高或者混杂有高频信号的话，我们需要使用滤波器对信号重新采样，来达到降频或者去除高频干扰的目的。之后需要对滤波器的输出进行边沿检测，决定是上升沿有效还是下降沿有效。选取触发源后，最后需要把信号连接到 TRGI 引脚，让触发信号成为外部时钟模式 1 的输入，此时使能计数器开始计数，外部时钟 1 的配置完成。

其中内部触发输入是使用一个定时器作为另一个定时器的预分频器。硬件上高级控制定时器和通用定时器在内部连接在一起，可以实现定时器同步或级联。主模式的定时器可以对从模式定时器执行复位、启动、停止或提供时钟。

当 TIM1_SMCR 寄存器中的 SMS=111 时，选择外部时钟源模式 1。计数器在所选触发信号（TRGI）

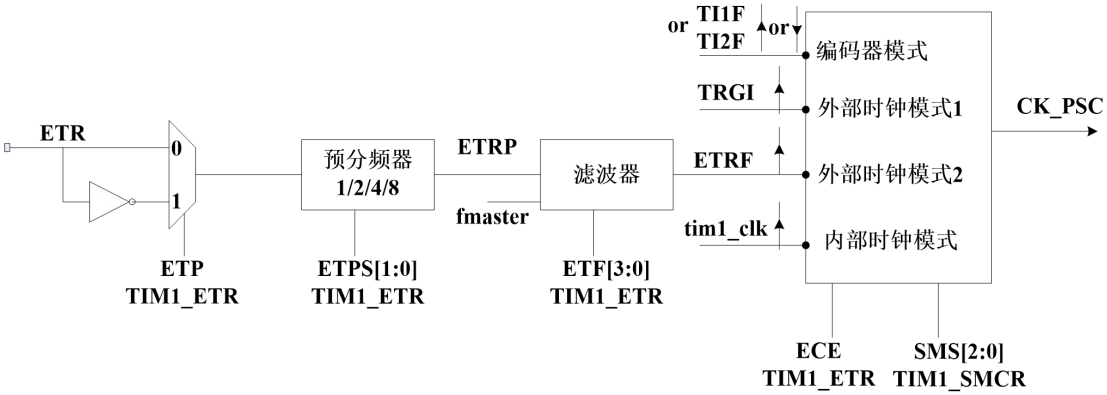
上出现上升沿时计数。



外部时钟模式 2

当使用外部时钟模式 2 的时候，时钟信号来自于定时器的特定输入通道 ETR，来自 ETR 引脚输入的信号可以选择为上升沿或者下降沿。当 ETRP 的信号频率过高的时候，需使用分频器来降频。若 ETRP 的信号频率过高或者混杂有高频干扰信号的话，就需要使用滤波器对 ETRP 信号重新采样，来实现降频或去除高频干扰的目的。经过滤波器滤波的信号连接到 ETRF 引脚后，触发信号成为外部时钟模式 2 的输入，此时使能计数器开始计数，外部时钟模式 2 的配置完成。

当 TIM1_SMCR 寄存器中的 ECE=1 时，选择外部时钟源模式 2。计数器可在外部触发输入 ETR 出现上升沿或下降沿时计数。



编码器模式

根据 TIM1_SMCR 寄存器中的 SMS 位段取值来决定不同的编码器模式。

编码器模式 1（0001）：计数器根据 TI2FP2 电平在 TI1FP1 边沿递增/ 递减计数。

编码器模式 2（0010）：计数器根据 TI1FP1 电平在 TI2FP2 边沿递增/ 递减计数。

编码器模式 3（0011）：计数器在 TI1FP1 和 TI2FP2 的边沿计数，计数的方向取决于另外一个输入的电平。

4.9.1.2 定时器同步

TIM1 定时器从内部与别的定时器连接在一起，以实现定时器同步或级联。它们可以在以下几种

模式下实现同步：复位模式、门控模式和触发模式。

从TIM	ITR0	ITR1	ITR2	ITR3
TIM1	TIM15	—	TIM3	TIM17_OC1

- **从模式：复位模式**

当触发输入信号发生变化时，计数器及其预分频器可重新初始化。此外，如果 TIM1_CR1 寄存器中的 URS 位处于低电平，则会生成更新事件 UEV。然后，所有预装载寄存器（TIM1_ARR 和 TIM1_CCRx）都将更新。

在 TIM1_CR1 寄存器中写入 CEN=1，启动计数器。计数器开始根据内部时钟计数，直到出现触发输入（TRGI）上升沿时，计数器清零，重新从 0 开始计数。同时，触发标志（TIM1_SR 寄存器中的 TIF 位）置 1，使能中断或 DMA 后，还可发送中断或 DMA 请求（取决于 TIM1_DIER 寄存器中的 TIE 和 TDE 位）。

- **从模式：门控模式**

输入信号的电平可用于使能/禁止计数器。

在TIM1_CR1寄存器中写入CEN=1，启动计数器。只要TRGI为高电平，计数器根据内部时钟计数，直到TRGI变为低电平时停止计数。计数器启动或停止时，TIM1_SR寄存器中的TIF标志都会置1。

在门控模式下，如果CEN=0，则无论触发输入电平如何，计数器都不启动。在门控模式下，当TRGI为高电平时，如果CEN=0，则在6个TIM_CLK后停止计数。再次CEN=1，则在3个TIM_CLK后开启计数。

- **从模式：触发模式**

当出现所选触发输入信号时，可以启动计数器。当 TRGI 出现上升沿时，计数器开始根据内部时钟计数，并且 TIF 标志置 1。

- **从模式：组合复位+触发模式**

在出现所选触发输入 (TRGI) 上升沿时，重新初始化计数器，生成一个寄存器更新事件，并启动计数器。该模式用于单脉冲模式。

- **从模式：外部时钟模式 2+触发模式**

外部时钟模式 2 可与另一种从模式（外部时钟模式 1 和编码器模式除外）结合使用。这种情况下，ETR 信号用作外部时钟输入，选择另一个输入作为触发输入 TRGI(在复位模式、门控模式或触发模式下)。禁止通过 TIM1_SMCR 寄存器中的 TS 位来选择 ETR 作为 TRGI。

TRGI 出现上升沿时将使能计数器并且 TIF 标志置 1。然后计数器在 ETR 信号的每个上升沿处计数。

4.9.1.3 ADC 同步

定时器可通过多种内部信号产生 ADC 触发事件，例如复位、使能或比较事件。也可是由内部边

沿检测器生成发出的脉冲，例如：

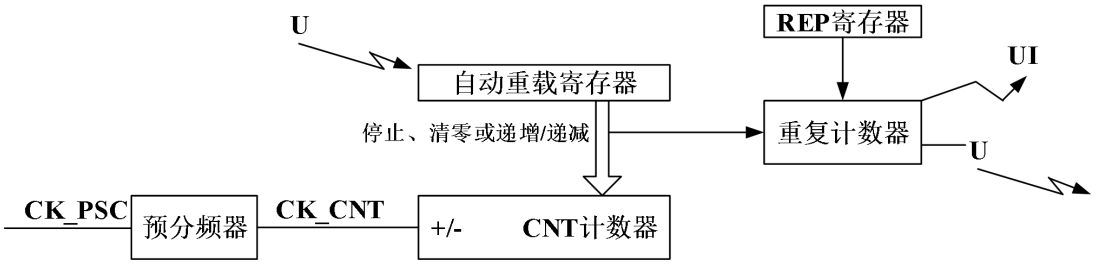
- OC4ref 的上升沿和下降沿
- OC5ref 的上升沿或 OC6ref 的下降沿

在重定向到 ADC 的 TRGO2 内部线路上发出触发信号。共有 16 个可能的事件，它们可通过 TIM1_CR2 寄存器中的 MMS2[3:0]位选择。

注意：

- 必须先使能接收 TRGO 或 TRGO2 信号的从外设（定时器、ADC 等）的时钟，才能从主定时器接收事件；并且从主定时器接收触发信号时，不得实时更改时钟频率（预分频器）。
- 必须先使能 ADC 时钟，才能从主定时器接收事件；从定时器接收触发信号时，不得实时更改 ADC 时钟频率（预分频器）。

4.9.2 时基单元



可编程高级控制定时器的主要模块是一个 16 位计数器及其相关的自动重载寄存器。计数器可递增计数、递减计数或交替进行递增和递减计数。计数器的时钟可通过预分频器进行分频。

计数器、自动重载寄存器和预分频器可通过软件进行读写。在计数器计数时，自动重载寄存器和预分频器寄存器可执行读写，计数器可执行读操作禁止写操作。

时基单元包含四个寄存器：计数器寄存器（TIM1_CNT）、预分频器寄存器(TIM1_PSC)、自动重载寄存器(TIM1_ARR)、重复计数器寄存器(TIM1_RCR)。

TIM1_PSC、TIM1_ARR、TIM1_RCR 寄存器是预装载的，对其执行写入或读取操作时会访问预装载寄存器。TIM1_ARR 寄存器的内容既可以直接传送到影子寄存器，也可以在每次发生更新事件时传送到影子寄存器，取决于 TIM1_CR1 寄存器中的自动重载预装载使能位(ARPE)，TIM1_PSC、TIM1_RCR 在发生更新事件时才送到影子寄存器。更新事件也可由软件产生。

计数器由预分频器输出 CK_CNT 提供时钟，仅当 TIM1_CR1 寄存器中的计数器启动位 (CEN) 置 1 时，才会启动计数器（有关计数器使能的更多详细信息，另请参见从模式控制器的相关说明）。

注意：计数器将在 TIM1_CR1 寄存器的 CEN 位置 1 时刻的一个时钟周期后开始计数。

预分频器（TIM1_PSC）

预分频器可对计数器时钟频率进行分频，分频系数介于 1-65536 之间。预分频器 TIM1_PSC 属于 16 位计数寄存器，该控制寄存器具有缓冲功能，能够在运行时被更改，新的预分频比将在下

一更新事件发生时被采用。（在后面会提到，预分频器的分频比与计数器值两者相对独立）

计数器（TIM1_CNT）

高级控制定时器的计数器有三种计数模式：递增计数模式、递减计数模式和递增/递减（中心对齐）计数模式。

➢ 递增计数模式

在递增计数模式下，计数器从 0 计数到自动重载值（TIM1_ARR 寄存器的内容），然后重新从 0 开始计数并生成计数器上溢事件。

如果使用重复计数器，则当递增计数的重复次数达到重复计数器寄存器中设定的次数加一次（TIM1_RCR + 1）后，生成更新事件(UEV)。否则，将在每次计数器上溢时产生更新事件。

将 TIM1_EGR 寄存器的 UG 位置 1（通过软件或使用从模式控制器）时，也将产生更新事件。

通过软件将 TIM1_CR1 寄存器中的 UDIS 位置 1 可禁止更新事件。各影子寄存器(ARR、PSC、RCR 和 CCRx)的值保持不变。计数器和预分频器计数器都会重新开始计数（而预分频比保持不变）。

此外，如果 TIM1_CR1 寄存器中的 URS 位（更新请求选择）已置 1，则将 UG 位置 1 会生成更新事件 UEV，但不会将 UIF 标志置 1（因此，不会发送任何中断或 DMA 请求）。这是为了避免在捕获模式下清除计数器时，同时产生更新和捕获中断。

发生更新事件时，将更新以下寄存器且将更新标志（TIM1_SR 寄存器中的 UIF 位）置 1（取决于 URS 位）：

- 将 TIM1_RCR 寄存器的内容重新装载到重复计数器
- 将预装载值 (TIM1_ARR)装载到自动重载影子寄存器
- 将 TIM1_PSC 寄存器的内容重新装载到预分频器的影子寄存器

➢ 递减计数模式

在递减计数模式下，计数器从自动重载值（TIM1_ARR 寄存器的内容）开始递减计数到 0， 然后重新从自动重载值开始计数并生成计数器下溢事件。

如果使用重复计数器，则当递减计数的重复次数达到重复计数器寄存器中编程的次数加一次（TIM1_RCR + 1）后，将产生更新事件(UEV)。否则，将在每次计数器下溢时产生更新事件。

将 TIM1_EGR 寄存器的 UG 位置 1（通过软件或使用从模式控制器）时，也将产生更新事件。

通过软件将 TIM1_CR1 寄存器中的 UDIS 位置 1 可禁止更新事件。这可避免向预装载寄存器写入新值时更新影子寄存器。此时各影子寄存器(ARR、PSC、RCR 和 CCRx)的值保持不变。计数器和预分频器计数器都会重新开始计数（而预分频比保持不变）。

此外，如果 TIM1_CR1 寄存器中的 URS 位（更新请求选择）已置 1，则将 UG 位置 1 会生成更新事件 UEV，但不会将 UIF 标志置 1（因此，不会发送任何中断或 DMA 请求）。这是为了避免在发生捕获事件并清除计数器时，同时产生更新和捕获中断。

发生更新事件时，将更新所有寄存器且将更新标志（TIM1_SR 寄存器中的 UIF 位）置 1（取决于 URS 位）：

- 将 TIM1_RCR 寄存器的内容重新装载到重复计数器
- 将预装载值 (TIM1_ARR)装载到自动重载影子寄存器
- 将 TIM1_PSC 寄存器的内容重新装载到预分频器的影子寄存器

➢ 中心对齐计数（递增/递减计数）

在中心对齐模式下，计数器从 0 开始计数到自动重载值（TIM1_ARR 寄存器的内容）– 1，生成计数器上溢事件；然后从自动重载值开始向下计数到 1 并生成计数器下溢事件。之后从 0 开始重新计数。

中心对齐模式取决于 TIM1_CR1 寄存器中的 CMS 位。将通道配置为输出模式时，其输出比较中断标志将在以下模式下置 1，即：计数器递减计数（中心对齐模式 1，CMS =“01”）、计数器递增计数（中心对齐模式 2，CMS =“10”）以及计数器递增/递减计数（中心对齐模式 3，CMS =“11”）。

在此模式下，TIM1_CR1 寄存器的 DIR 方向位不可写入值，而是由硬件更新并指示当前计数器方向。

每次发生计数器上溢和下溢时都会生成更新事件，或将 TIM1_EGR 寄存器中的 UG 位置 1 （通过软件或使用从模式控制器）也可以生成更新事件。这种情况下，计数器以及预分频器计数器将重新从 0 开始计数。

通过软件将 TIM1_CR1 寄存器中的 UDIS 位置 1 可禁止更新事件。各影子寄存器(ARR、PSC、RCR 和 CCRx)的值保持不变。不过，计数器仍会根据当前自动重载值进行递增和递减计数，计数器和预分频器计数器会重新开始计数（而预分频比保持不变）。

此外，如果 TIM1_CR1 寄存器中的 URS 位（更新请求选择）已置 1，则将 UG 位置 1 会产生 UEV 更新事件，但不会将 UIF 标志置 1（因此，不会发送任何中断或 DMA 请求）。这是为了避免在发生捕获事件并清除计数器时，同时产生更新和捕获中断。

发生更新事件时，将更新所有寄存器且将更新标志（TIM1_SR 寄存器中的 UIF 位）置 1（取决于 URS 位）：

- 将 TIM1_RCR 寄存器的内容重新装载到重复计数器
- 将预装载值 (TIM1_ARR)装载到自动重载影子寄存器
- 将 TIM1_PSC 寄存器的内容重新装载到预分频器的影子寄存器

重复计数器(TIM1_RCR)

当重复计数器达到零时，即每当发生 N+1 个计数器上溢或下溢（其中，N 是 TIM1_RCR 重复计数器寄存器中的值），才会生成更新事件。

这意味着，每当发生 N+1 个计数器上溢或下溢（其中，N 是 TIM1_RCR 重复计数器寄存器中的值），数据就将从预装载寄存器转移到影子寄存器（TIM1_ARR 自动重载寄存器、TIM1_PSC 预分频器寄存器以及比较模式下的 TIM1_CCRx 捕获/比较寄存器）。

重复计数器在下列情况下递减：

- 递增计数模式下的每个计数器上溢

- 递减计数模式下的每个计数器下溢
- 中心对齐模式下每个计数器上溢和计数器下溢

重复计数器是自动重载类型，当更新事件由软件（通过将 TIM1_EGR 寄存器的 UG 位置 1）或硬件（通过从模式控制器）生成时，无论重复计数器的值为多少，更新事件都将立即发生，并且将 TIM1_RCR 寄存器的内容重新装载到重复计数器中。

在中心对齐模式下，如果 RCR 值为奇数，更新事件将在上溢或下溢时发生，这取决于何时写入 RCR 寄存器以及何时启动计数器：如果在启动计数器前写入 RCR，则 UEV 在下溢时发生。如果在启动计数器后写入 RCR，则 UEV 在上溢时发生。

4.9.3 输入捕获

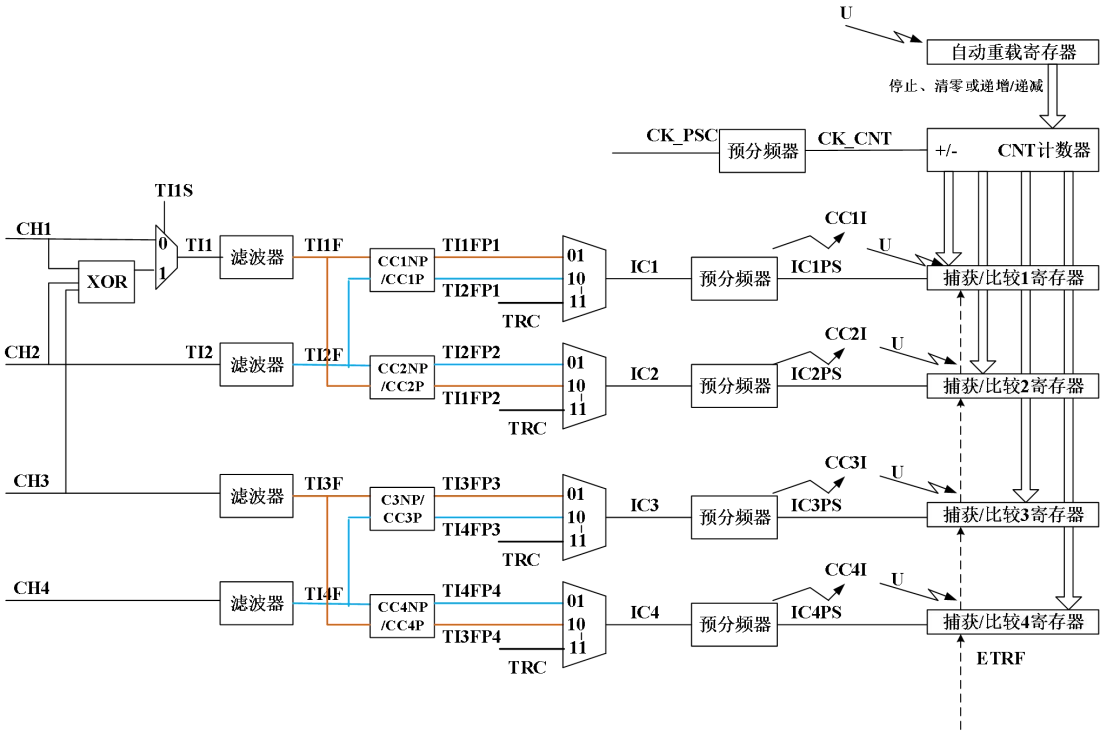
捕获/比较通道说明

每个捕获/比较通道均围绕一个捕获/比较寄存器（包括一个影子寄存器）、一个捕获输入部分（数字滤波、多路复用和预分频器，通道 5 和通道 6 除外）和一个输出部分（比较器和输出控制）构建而成。

输入阶段对相应的 TIx 输入进行采样，生成一个滤波后的信号 TIxF。然后，带有极性选择功能的边沿检测器生成一个信号(TIxFPx)，该信号可用作从模式控制器的触发输入，也可用作捕获命令。该信号先进行预分频(ICxPS)，而后再进入捕获寄存器。

注意事项：

- 捕获/比较模块由一个预装载寄存器和一个影子寄存器组成。始终可通过读写操作访问预装载寄存器。
- 在捕获模式下，捕获实际发生在影子寄存器中，然后将影子寄存器的内容复制到预装载寄存器中。
- 在比较模式下，预装载寄存器的内容将复制到影子寄存器中，然后将影子寄存器的内容与计数器进行比较。



输入捕获可以对输入的信号的上升沿、下降沿或者双边沿进行捕获，常用的有测量输入信号的脉宽，和测量 PWM 输入信号的频率和占空比这两种。

输入捕获的大概原理就是，当捕获到信号的跳变沿的时候，把计数器 CNT 的值锁存到捕获寄存器 CCR 中，把前后两次捕获到的 CCR 寄存器中的值相减，就可以算出脉宽或者频率。如果捕获的脉宽的时间长度超过你的捕获定时器的周期，就会发生溢出，此时需要额外处理。

输入通道

需要被测量的信号从定时器的外部引脚 CH1/2/3/4 进入，通常叫 TI1/2/3/4，在后面的捕获流程中对于被测量的信号都以 TIx 为标准叫法。

输入滤波器和边沿检测器

当输入的信号存在高频干扰的时候，我们需要对输入信号进行滤波，即进行重新采样，根据采样定律，采样的频率必须大于等于两倍的输入信号。比如输入信号为 1M，又存在高频信号干扰，那么此时就需要进行滤波，我们可以设置采样频率为 2M，这样可以保证采样到有效信号的基础上把高于 2M 的高频干扰信号过滤掉。

滤波器的配置由 TIM1_CCMR1(输入捕获)寄存器的 IC1F[3:0]位与 TIM1_CR1 寄存器的 CKD[1:0]位控制。从 IC1F[3:0]位的描述可知，采样频率 f_{SAMPLE} 可以由 f_{CK_INT} 和 f_{DTS} 分频后的时钟提供，其中 f_{CK_INT} 是定时器内部时钟，f_{DTS} 是 f_{CK_INT} 经过分频后得到的频率，分频因子由 CKD[1:0]决定，可以为不分频，2 分频或者是 4 分频。

边沿检测器是用来设置信号在捕获的时候是什么边沿有效，可以是上升沿、下降沿或者是双边沿，具体的由 TIM1_CCER 寄存器的位 CCxP 和 CCxNP 决定。

捕获通道

捕获通道就是图中 IC1/2/3/4，每个捕获通道都有相对应的捕获寄存器 CCR1/2/3/4，当发生捕获

的时候，计数器 CNT 的值就会被锁存到捕获寄存器中。

这里我们要搞清楚输入通道和捕获通道的区别，输入通道是用来输入信号的，捕获通道是用来捕获输入信号的通道，一个输入通道的信号可以同时输入给两个捕获通道。比如输入通道 TI1 的信号经过滤波边沿检测器之后的 TI1FP1 和 TI1FP2 可以进入到捕获通道 IC1 和 IC2，其实这就是后面会提到的 PWM 输入捕获，只有一路输入信号（TI1）却占用了两个捕获通道（IC1 和 IC2）。当只需要测量输入信号的脉宽时候，用一个捕获通道即可。输入通道和捕获通道的映射关系具体由 TIM1_CCMR1(输入捕获)寄存器的 CCxS[1:0]位配置。

预分频器

ICx 的输出信号会经过一个预分频器，用于决定发生所少个事件时进行一次捕获。具体的是由 TIM1_CCMR1(输入捕获)寄存器的 ICxPSC[1:0]位配置，如果希望捕获信号的每一个边沿，则无需分频。

捕获寄存器

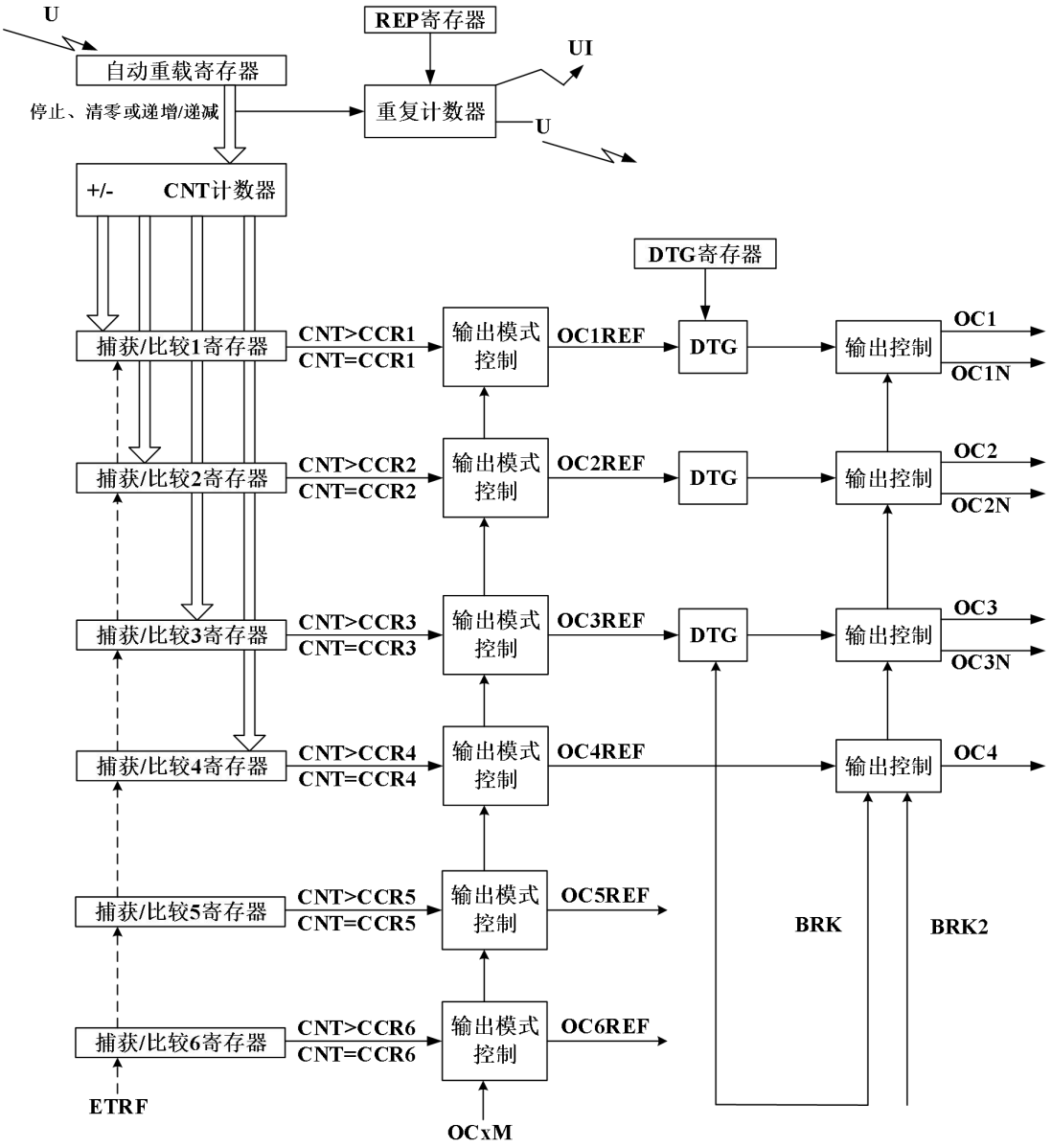
经过预分频的信号 ICxPS 是最终被捕获的信号，当发生捕获时（第一次），计数器 CNT 的值会被锁存到捕获寄存器 CCR 中，还会产生 CCxI 中断，相应的中断位 CCxIF（TIM1_SR 寄存器中）会被置位，通过软件或者读取 CCR 中的值可以将 CCxIF 清零。如果发生第二次捕获（即重复捕获：CCR 寄存器中已捕获到计数器值且 CCxIF 标志已置 1），则捕获溢出标志位 CCxOF 会被置位，CCxOF 只能通过软件清零。

注意

1. 发生输入捕获时：
   - 发生有效跳变沿时，TIM1_CCRx 寄存器会获取计数器的值
   - 将 CCxIF 标志置 1（中断标志）。如果至少发生了两次连续捕获，但 CCxIF 标志未被清零，这样 CCxOF 捕获溢出标志会被置 1
   - 根据 CCxIE 位生成中断
   - 根据 CCxDE 位生成 DMA 请求
2. 要处理重复捕获，建议在读出捕获溢出标志之前读取数据。这样可避免丢失在读取捕获溢出标志之后与读取数据之前可能出现的重复捕获信息。
3. 通过软件将 TIM1_EGR 寄存器中的相应 CCxG 位置 1 可生成 IC 中断或 DMA 请求。
4. 配置 TIM1_CCMRx 寄存器时，分两次配置：第一次配置 TIM1_CCMRx 寄存器中的 CCxS 位，选择输入模式的映射；第二次配置 TIM1_CCMRx 寄存器的 ICxF 和 ICxPSC 选择滤波和预分频系数。

4.9.4 输出比较

输出比较就是通过定时器的外部引脚对外输出控制信号，有冻结、将通道 X（x=1,2,3,4）设置为匹配时输出有效电平、将通道 X 设置为匹配时输出无效电平、翻转、强制变为无效电平、强制变为有效电平、PWM1 和 PWM2 这八种模式，具体使用哪种模式由 TIM1_CCMR1(输出比较)寄存器的 OC1M[3:0]位配置。其中 PWM 是输出比较中使用频率最多的功能。



输出比较用于输出波形。通道 1 到通道 4 可用作输出，而通道 5 和通道 6 只能在器件内部使用（用于产生混合波形或触发 ADC）。当捕获/比较寄存器与计数器之间相匹配时，输出比较功能如下：

- 产生极性、持续时间和频率都可编程的脉冲输出到相应引脚上。当计数器值与 CCR 值匹配时，输出引脚既可保持其电平 (OCxM=0000)，也可设置为有效电平(OCxM=0001)、无效电平(OCxM=0010) 或进行翻转(OCxM=0011)。
- 将捕获/比较中断标志置 1（CCxIF 位）。
- 如果相应中断使能位（CCxIE 位）置 1，将产生中断。
- 如果相应 DMA 请求使能位（CCxDE 位）置 1，将产生 DMA 请求。

使用 TIM1_CCMRx 寄存器中的 OCxPE 位，可将 TIM1_CCRx 寄存器配置为带或不带预装载寄存器。

在输出比较模式下，更新事件 UEV 对 OCxREF 和 OCx 输出毫无影响。同步的精度可以达到计

数器的一个计数周期。输出比较模式也可用于输出单脉冲（在单脉冲模式下，OPM=1）。

应用步骤

1. 选择计数器时钟（内部、外部、预分频器）
2. 在 TIM1_ARR 和 TIM1_CCRx 寄存器中写入所需数据
3. 如果要生成中断请求，则需将 CCxIE 位置 1
4. 选择输出模式，例如：
   - 写入 OCxM = 0011，当 CNT 与 CCRx 匹配时，翻转 OCx 输出引脚
   - 写入 OCxPE = 0 以禁止预装载寄存器
   - 写入 CCxP = 0 以选择高电平有效极性
   - 写入 CCxE = 1 以使能输出
5. 通过将 TIM1_CR1 寄存器中的 CEN 位置 1 来使能计数器

可通过软件随时更新 TIM1_CCRx 寄存器以控制输出波形，前提是未使能预装载寄存器（OCxPE="0"，否则 TIM1_CCRx 影子寄存器仅在下一更新事件 UEV 发生时进行更新）。

比较寄存器

当计数器 CNT 的值跟比较寄存器 CCR 的值相等的时候，输出参考信号 OCxREF 的信号的极性就会改变，其中 OCxREF=1（高电平）称之为有效电平，OCxREF=0（低电平）称之为无效电平，并且会产生比较中断 CCxI,相应的标志位 CCxIF（TIM1_SR 寄存器中）会置位。然后 OCxREF 再经过一系列控制之后就成为真正的输出信号 OCx/OCxN。

死区发生器

在生成的参考波形 OCxREF 基础上，可以插入死区时间，用于生成两路互补的输出信号 OCx 与 OCxN，死区时间的大小具体由 TIM1_BDTR 寄存器的 DTG 位配置。死区时间的大小必须根据与输出信号相连接的器件及其特性来调整。

以半桥驱动电路为例，Q1 导通、Q2 截止时，想要 Q1 截止、Q2 导通，先要让 Q1 截止一段时间之后才让 Q2 导通，这段等待时间就是死区时间，因为 Q1 关闭需要时间（由 MOS 管工艺决定）。若 Q1 关闭后，马上打开 Q2，此时一段时间内相当于 Q1 和 Q2 都导通，会导致电路短路。

输出控制

在输出比较的输出控制中，参考信号 OCxREF 在经过死区发生器之后会产生两路带死区的互补信号 OCx_DT 和 OCxN_DT（通道 1-3 才有互补信号，通道 4 没有，其余跟通道 1-3 一样），这两路带死区的互补信号然后就进入输出控制电路，如果没有加入死区控制，那么进入输出控制电路的信号就直接是 OCxREF。

进入输出控制电路的信号会被分成两路，一路是原始信号，一路是被反向的信号，具体的由 TIM1_CCER 寄存器的 CCxP 和 CCxNP 控制。经过极性选择的信号是否由 OCx 引脚输出到外部引脚 CHx/CHxN 则由 TIM1_CCER 寄存器的 CxE/CxNE 配置。

如果加入了刹车功能，则断路和死区寄存器 BDTR 的 MOE、OSSI 和 OSSR 这三个位会共同影

响输出的信号。

输出引脚

输出比较的输出信号最终是通过定时器的外部 IO 来输出的，分别为 CH1/2/3/4，其中前面三个通道还有互补的输出通道 CH1/2/3N。

4.9.5 刹车功能

刹车功能的目的是保护由 TIM1 定时器生成的 PWM 信号所驱动的功率开关。两个刹车输入通常连接到功率级和三相逆变器的故障输出。激活时，刹车电路会关闭 PWM 输出，并将其强制为预定义的安全状态。

4.9.5.1 刹车基础内容

有两个刹车通道。刹车通道源只包括应用故障（来自输入引脚），可以在死区持续时间后将输出强制为预定义的电平（有效或无效）。刹车 2 通道只包括应用故障，能够将输出强制为无效状态。

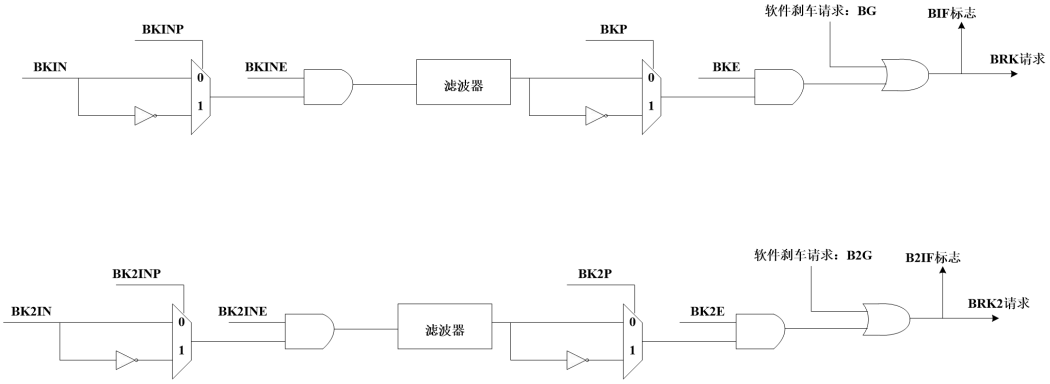
刹车期间的输出使能信号和输出电平取决于多个控制位：

- TIM1_BDTR 寄存器中的 MOE 位，允许通过允许通过软件使能/禁止输出，在发生刹车和刹车 2 事件时复位。
- TIM1_BDTR 寄存器中的 OSSI 位，定义定时器将输出控制在无效状态下，还是释放对 GPIO 控制器的控制（通常使其处于高阻态模式）。
- TIM1_CR2 寄存器中的 OISx 和 OISxN 位，将输出设置为空闲电平（有效或无效）。无论 OISx 和 OISxN 的值为何，不能将 OCx 和 OCxN 输出同时设置为有效电平。

刹车 (BRK) 通道的源为连接到 BKIN 引脚的外部源（由 TIM1_AF1 寄存器设置），具有极性选择和可选的数字滤波。

刹车 2 (BRK2)的源为连接到 BK2IN 引脚的外部源（由 TIM1_AF2 寄存器设置），具有极性选择和可选的数字滤波。

也可由软件通过 TIM1_EGR 寄存器中的 BG 和 B2G 位产生刹车事件。无论 BKE 和 BK2E 使能位的值如何，都可以使用 BG 和 B2G 通过软件产生刹车。



发生刹车之一（其中一个刹车输入上出现所选电平）时，会进行以下操作：

- MOE 位异步清零，使输出 OCx/OCxN 处于无效状态、空闲状态甚至释放控制权给 GPIO 控制器（通过 OSSI 位进行选择）。
- 使用互补输出时：
  - 输出首先置于无效状态（取决于极性 OCxP/OCxNP）。
  - 死区时间后以 OISx 和 OISxN 位中编程的电平输出。禁止同时将 OISx 和 OISxN 位编程为有效电平。
  - 如果 OSSI=0，定时器释放输出控制（由 GPIO 控制器接管）。
- 将刹车状态标志（TIM1_SR 寄存器中的 BIF 和 B2IF 位）置 1。如果 TIM1_DIER 寄存器中的 BIE 位置 1，则会产生中断。
- 如果 TIM1_BDTR 寄存器中的 AOE 位置 1，则 MOE 位会在发生下一更新事件(UEV)时自动再次置 1。

注意：

1. 刹车输入为电平有效。因此，当刹车输入有效电平时，不能将 MOE 位置 1（自动或通过软件）。同时，不能将状态标志 BIF 和 B2IF 清零。
2. 当 CCxE 和 CCxNE 都为 0 时，不管刹车无效还是有效，OCx 和 OCxN 都禁止输出，IO 口释放给 GPIO 控制器。

除刹车输入和输出管理外，刹车电路内部还实施了写保护，用以保护应用的安全。通过该功能，用户可冻结多个参数配置（死区持续时间、OCx/OCxN 极性和禁止时的状态、OCxM 配置、刹车使能和极性）。应用可以通过 TIM1_BDTR 寄存器中的 LOCK 位，从 3 种保护级别中进行选择。MCU 复位后只能对 LOCK 位执行一次写操作。

两个刹车输入针对定时器输出具有不同的行为：

- BRK 输入可禁止（无效状态）PWM 输出，也可将 PWM 输出强制为预定义的安全状态。
- BRK2 只能禁止（无效状态）PWM 输出。

4.9.5.2 双向刹车输入

TIM1 具有双向刹车 I/O，与输入模式不同的是，双向刹车可以输入或输出到刹车引脚，可以用作故障检测信号。使用 TIM1_BDTR 寄存器的 BKBID 和 BK2BID 位，将刹车和刹车 2 输入配置为双向模式。可以使用 TIM1_BDTR 寄存器中的 LOCK 位，将 BKBID 编程位锁定在只读模式（锁定级别 1 或更高）。

双向模式可以用于刹车和刹车 2 输入，需要将 I/O 配置为低电平有效极性的开漏模式（使用 BKINP/BKP、BK2INP/BK2P 位配置有效极性，通过 GPIO 输出类型寄存器 GPIOx_OTYPER 设置为开漏模式）。刹车请求会强制将刹车输入置为低电平。

注：配置寄存器时，应先配置低电平有效极性，即先配置 BKINP、BKP、BK2INP 和 BK2P 位；再配置 TIM1_BDTR 寄存器的 BKBID 和 BK2BID 位，将刹车和刹车 2 输入配置为双向模式。如果 I/O 不是低电平有效极性，则 TIM1_BDTR 寄存器的 BKBID 和 BK2BID 位配置写不进去，无法配置为双向模式。

软件刹车事件（BG 和 B2G）也会导致刹车输入 I/O 被强制置为低电平，从而向外部组件指示定时器已进入刹车状态。当 BKDSRM(BK2DSRM)位置 1 时，会释放刹车输出以清除故障信号，从而使系统能够重新获得保护。

在任何情况下都无法禁止刹车保护电路：

刹车输入路径始终有效：即使 BKDSRM (BK2DSRM)位置 1 且释放开漏控制，刹车事件也仍然有效。这样可以在发生刹车期间防止 PWM 输出重新启动。

使能输出（MOE 位置 1）后，BK(2)DSRM 位不能解除刹车保护。

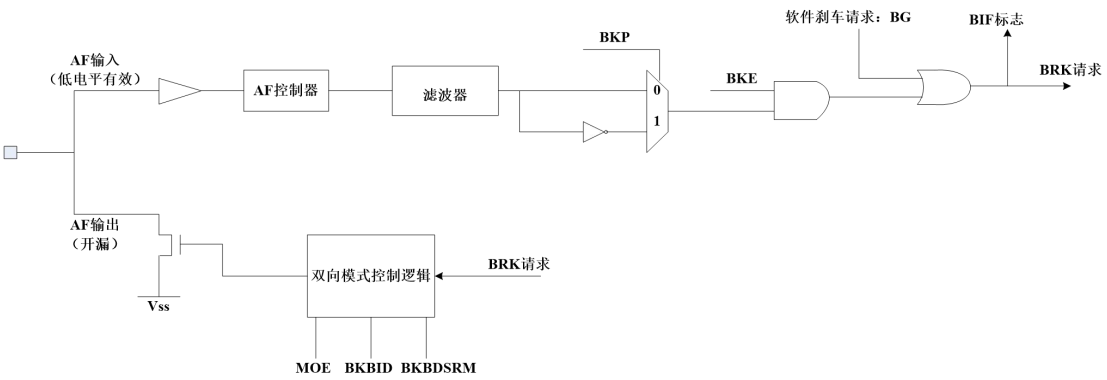
4.9.5.3 启动和重新启动刹车电路

默认情况下（外设复位配置）会启动刹车电路（在输入或双向模式下）。

发生刹车（刹车 2）事件后，必须按照以下步骤重新启动保护：

- 必须将 BKDSRM(BK2DSRM) 位置 1，以释放输出控制
- 软件必须轮询 BKDSRM (BK2DSRM)位，直到该位由硬件清零（当应用刹车条件消失时）

此后，刹车电路即启动并激活，可以通过将 MOE 位置 1 来重新使能 PWM 输出。



4.9.6 输入捕获应用

输入捕获一般应用在两个方面，一个方面是脉冲跳变沿时间测量，另一方面是 PWM 输入测量。

4.9.6.1 测量脉宽或频率

测量频率

当捕获通道 TIx 上出现上升沿时，发生第一次捕获，计数器 CNT 的值会被锁存到捕获寄存器 CCR 中，而且还会进入捕获中断，在中断服务程序中记录一次捕获（可以用一个标志变量来记录），并把捕获寄存器中的值读取到 value1 中。当出现第二次上升沿时，发生第二次捕获，计数器 CNT 的值会再次被锁存到捕获寄存器 CCR 中，并再次进入捕获中断，在捕获中断中，把捕获寄存器的值读取到 value3 中，并清除捕获记录标志。利用 value3 与 value1 的差值我们可以算出信号的周期（频率）。

测量脉宽

当捕获通道 TIx 上出现上升沿时，发生第一次捕获，计数器 CNT 的值会被锁存到捕获寄存器 CCR 中，而且还会进入捕获中断，在中断服务程序中记录一次捕获（可以用一个标志变量来记录），并把捕获寄存器中的值读取到 value1 中。然后把捕获边沿改为下降沿捕获，目的是捕获后面的下降沿。当下降沿到来的时候，发生第二次捕获，计数器 CNT 的值会再次被锁存到捕获寄存器 CCR 中，并再次进入捕获中断，在捕获中断中，把捕获寄存器的值读取到 value2 中，并清除捕获记录标志。然后把捕获边沿设置为上升沿捕获。利用 value2 与 value1 的差值我们可以算出信号的高电平脉宽。

在测量脉宽过程中需要来回的切换捕获边沿的极性，如果测量的脉宽时间比较长，定时器就会发生溢出，溢出的时候会产生更新中断，我们可以在中断里对溢出进行记录处理。

4.9.6.2 PWM 输入模式

测量脉宽和频率还有一个简单的方法就是使用 PWM 输入模式，该模式是输入捕获的特例。与上面只使用一个捕获寄存器测量脉宽和频率的方法相比，PWM 输入模式需要占用两个捕获寄存器。其特性如下：

- 两个 ICx 信号被映射至同一个 TIx 输入
- 这两个 ICx 信号在边沿处有效，但极性相反
- 选择两个 TIxFP 信号之一作为触发输入，并将从模式控制器配置为复位模式

我们以输入通道 TI1 工作在 PWM 输入模式为例来说明具体的工作原理。PWM 信号由输入通道 TI1 进入，因为是 PWM 输入模式，信号被分为两路，一路是 TI1FP1，另一路是 TI1FP2。其中一路是周期，另一路是占空比，具体哪一路信号对应周期还是占空比，得从程序上设置哪一路信号作为触发输入，作为触发输入的那一路信号对应的就是周期，另一路就是占空比。作为触发输入的那一路信号还需要设置极性，是上升沿还是下降沿捕获，一旦设置好触发输入的极性，另外

一路硬件就会自动配置为相反的极性捕获，无需软件配置。简而言之：选定输入通道，确定触发信号，然后设置触发信号的极性即可，因为是 PWM 输入，另一路由硬件配置，无需软件配置。

当使用 PWM 输入模式的时候必须将从模式控制器配置为复位模式（配置 TIM1_SMCR 寄存器的 SMS[3:0]位实现），即当我们启动触发信号开始进行捕获的时候，同时把计数器 CNT 复位清零。

PWM 信号由输入通道 TI1 进入，配置 TI1FP1 为触发信号，上升沿捕获。当上升沿的时候 IC1 和 IC2 同时捕获，计数器 CNT 清零，到了下降沿的时候，IC2 捕获，此时计数器 CNT 的值被锁存到捕获寄存器 CCR2 中，到了下一个上升沿的时候，IC1 捕获，计数器 CNT 的值被锁存到捕获寄存器 CCR1 中。其中 CCR2+1 测量的是脉宽，CCR1+1 测量的是周期。

从软件上来说，用 PWM 输入模式测量脉宽和周期更容易，付出的代价是需要占用两个捕获寄存器。

4.6.7 输出比较应用

输出比较模式共有八种，具体由 TIM1_CCMRx(输出比较)寄存器的 OCxM[3:0]位配置。这里对常用输出模式进行说明，其他模式参考数据手册即可。

4.6.7.1 强制输出模式

在强制输出模式下，通过向 TIM1_CCMRx 寄存器中的 OCxM 位写入 0101 或 0100，将输出比较参考信号（OCxREF）强制设置为有效电平或无效电平（OCxREF 始终为高电平有效），而无需考虑输出比较寄存器和计数器之间的任何比较结果。OCx/OCxN 再根据 CCxP/CCxNP 极性位配置输出。

要将输出比较信号(OCxREF/OCx) 强制设置为有效电平，用户只需向相应 TIM1_CCMRx 寄存器中的 OCxM 位写入 0101。OCxREF 进而强制设置为高电平（OCxREF 始终为高电平有效），同时 OCx 获取 CCxP 极性位的相反值。

例如：OCxM=0101，CCxP=0（OCx 高电平有效）=> 将 OCx 强制设置为高电平。

通过向 TIM1_CCMRx 寄存器中的 OCxM 位写入 0100，可将 OCxREF 信号强制设置为低电平。

无论如何，TIM1_CCRx 影子寄存器与计数器之间的比较仍会执行，而且允许将标志置 1。因此可发送相应的中断和 DMA 请求。

4.6.7.2 PWM 输出模式

PWM 模式可以生成一个频率和占空比可编程的信号，频率由 TIM1_ARR 寄存器值决定，占空比由 TIM1_CCRx 寄存器值决定。

各通道可以独立选择 PWM 模式（每个 OCx 输出对应一个 PWM），只需向 TIM1_CCMRx 寄存器的 OCxM 位写入“0110”（PWM 模式 1）或“0111”（PWM 模式 2）。必须通过将 TIM1_CCMRx 寄存器中的 OCxPE 位置 1 使能相应预装载寄存器，最后通过将 TIM1_CR1 寄存器中的 ARPE 位置 1 使能自动重载预装载寄存器（在向上计数或中心对齐模式下）。

如果 OCxPE 位或 ARPE 位置 1，使能相应的预装载寄存器。由于只有发生更新事件时预装载寄存器才会传送到影子寄存器，因此启动计数器之前，必须通过将 TIM1_EGR 寄存器中的 UG 位置 1 产生更新事件来初始化所有寄存器。

OCx 的极性可以通过软件在 TIM1_CCER 寄存器中的 CCxP 位设置。可将其设置为高电平有效或低电平有效。通过 CCxE、CCxNE、MOE、OSSI 和 OSSR 位（TIM1_CCER 和 TIM1_BDTR 寄存器）的组合使能 OCx 输出。

PWM 根据计数模式，可以分为边沿对齐模式和中央对齐模式。

PWM 边沿对齐模式

- 递增计数配置

当 TIM1_CR1 寄存器中的 DIR 位为 0 时，计数器递增计数。

PWM 模式 1 下，只要 TIM1_CNT < TIM1_CCRx，参考信号 OCxREF 为高电平，否则为低电平。如果 TIM1_CCRx 中的比较值大于自动重载值（TIM1_ARR 中），则 OCxREF 保持为“1”。如果比较值为 0，则 OCxREF 保持为“0”。

PWM 模式 2 下，只要 TIM1_CNT < TIM1_CCRx，参考信号 OCxREF 为低电平，否则为高电平。如果 TIM1_CCRx 中的比较值大于自动重载值（TIM1_ARR 中），则 OCxREF 保持为“0”。如果比较值为 0，则 OCxREF 保持为“1”。

- 递减计数配置

当 TIM1_CR1 寄存器中的 DIR 位为高时执行递减计数。PWM 模式 1 下，只要 TIM1_CNT > TIM1_CCRx，参考信号 OCxREF 为低电平，否则为高电平。如果 TIM1_CCRx 中的比较值大于自动重载值（TIM1_ARR 中），则 OCxREF 保持为“1”。如果比较值为 0，则 OCxREF 保持为“0”。PWM 模式 2 下，只要 TIM1_CNT > TIM1_CCRx，参考信号 OCxREF 为高电平，否则为低电平。如果 TIM1_CCRx 中的比较值大于自动重载值（TIM1_ARR 中），则 OCxREF 保持为“0”。如果比较值为 0，则 OCxREF 保持为“1”。

PWM 中心对齐模式

当 TIM1_CR1 寄存器中的 CMS 位不为“00”，中心对齐模式生效。CMS 位的其余配置，区别在于在计数器递增计数、递减计数或同时递增和递减计数时将比较标志置 1。TIM1_CR1 寄存器中的方向位(DIR)由硬件更新，不得通过软件更改。

注意事项：

- 启动中心对齐模式时将使用当前的递增/递减计数配置。这意味着计数器将根据写入 TIM1_CR1 寄存器中 DIR 位的值进行递增或递减计数。但注意，在启动计数器前或者计数过程中产生一个软件更新（将 TIM1_EGR 寄存器中的 UG 位置 1），将从 0 开始计数(即先递增计数)。
- 禁止在运行中心对齐模式时对计数器执行写操作。

不对称 PWM 模式

在不对称模式下，生成的两个中心对齐 PWM 信号间允许存在可编程相移。频率由 TIM1_ARR 寄存器的值确定，而占空比和相移则由一对 TIM1_CCRx 寄存器确定。两个寄存器分别控制递增

计数和递减计数期间的 PWM，这样每半个 PWM 周期便会调节一次 PWM：

- OC1REFC（或 OC2REFC）由 TIM1_CCR1 和 TIM1_CCR2 控制
- OC3REFC（或 OC4REFC）由 TIM1_CCR3 和 TIM1_CCR4 控制

两个通道可以独立选择不对称 PWM 模式（每对 CCR 对应一个 OCx 输出），只需向 TIM1_CCMRx 寄存器的 OCxM 位写入“1110”（不对称 PWM 模式 1）或“1111”（不对称 PWM 模式 2）。

注意：给定通道用作不对称 PWM 通道时，其辅助通道也可使用。例如，如果通道 1 上产生 OC1REFC 信号（不对称 PWM 模式），通道 2 上可输出 OC2REF 信号（PWM 模式）或 OC2REFC 信号（不对称 PWM 模式），取决于通道 2 选择的模式。

组合 PWM 模式

在组合 PWM 模式下，生成的两个边沿或中心对齐 PWM 信号的各个脉冲间允许存在可编程延时和相移。频率由 TIM1_ARR 寄存器的值确定，而占空比和延时则由两个 TIM1_CCRx 寄存器确定。产生的信号 OCxREFC 由两个参考 PWM 的逻辑或运算或者逻辑与运算组合组成。

- OC1REFC（或 OC2REFC）由 TIM1_CCR1 和 TIM1_CCR2 控制
- OC3REFC（或 OC4REFC）由 TIM1_CCR3 和 TIM1_CCR4 控制

两个通道可以独立选择组合 PWM 模式（每对 CCR 寄存器对应一个 OCx 输出），只需向 TIM1_CCMRx 寄存器的 OCxM 位写入“1100”（组合 PWM 模式 1）或“1101”（组合 PWM 模式 2）。

当给定通道用作组合 PWM 通道时，其互补通道必须在相反的 PWM 模式下配置（例如，一个通道在组合 PWM 模式 1 下配置，另一个通道在可在组合 PWM 模式 2 下配置）。

组合三相 PWM 模式

在组合三相 PWM 模式下，产生的一至三个中心对齐 PWM 信号与一个可编程信号间允许在脉冲中间进行逻辑与运算。OC5REF 信号用于定义产生的组合信号。凭借 TIM1_CCR5 中的 3 位 GC5C[3:1]，可以选择 OC5REF 与哪个参考信号组合。产生的信号 OCxREFC 由两个参考 PWM 的逻辑与运算组合组成。

- 如果 GC5C1 置 1，则 OC1REFC 由 TIM1_CCR1 和 TIM1_CCR5 控制
- 如果 GC5C2 置 1，则 OC2REFC 由 TIM1_CCR2 和 TIM1_CCR5 控制
- 如果 GC5C3 置 1，则 OC3REFC 由 TIM1_CCR3 和 TIM1_CCR5 控制

通道 1 到通道 3 可独立选择组合三相 PWM 模式，只需将 3 位 GC5C[3:1]中的至少一位置 1。

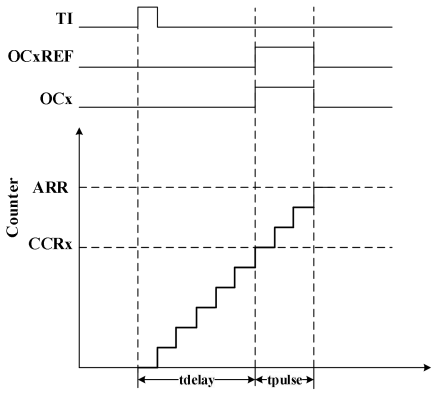
单脉冲模式

单脉冲模式(OPM)是上述模式的一个特例。在这种模式下，计数器可以在一个激励信号的触发下启动，并可在一段可编程的延时后产生一个脉宽可编程的脉冲。

可以通过从模式控制器启动计数器。可以在输出比较模式或 PWM 模式下生成波形。通过将 TIM1_CR1 寄存器中的 OPM 位置 1 选择单脉冲模式。这样，发生下一更新事件 UEV 时，计数器将自动停止。

只有当比较值与计数器初始值不同时，才能正确产生一个脉冲。启动前（定时器等待触发时），必须进行如下配置：

- 递增计数时：CNT<CCRx<=ARR(特别注意，0<CCRx)
- 递减计数时：CNT>CCRx



特殊情况（OCx 快速使能）：

在单脉冲模式下，TIx 输入的边沿检测会将 CEN 位置 1，表示使能计数器。然后，在计数器值与比较值之间发生比较时使输出翻转。但是，完成这些操作需要多个时钟周期，这会限制可能的最小延迟（tdelay 最小值）。

如果需要以最小延时输出波形，可以将 TIM1_CCMRx 寄存器中的 OCxFE 位置 1。这样会强制 OCxRef（和 OCx）对激励信号做出响应，而不再考虑比较的结果。其新电平与发生比较匹配时相同。仅当通道配置为 PWM1 或 PWM2 模式时，OCxFE 才会起作用。

可再触发单脉冲模式

该模式允许计数器可以在一个激励信号的触发下启动，并且能产生长度可编程的脉冲，但与不可再触发单脉冲模式间存在以下差别：

- 发生触发时，脉冲立即产生（无可编程延时）
- 如果在上一个触发完成前发生新的触发，脉冲将延长

定时器必须处于从模式，TIM1_SMCR 寄存器中的位 SMS[3:0] =“1000”（组合复位 + 触发模式），针对可再触发 OPM 模式 1 或模式 2 将 OCxM[3:0]位设置为“1000”或“1001”。

定时器配置为递增计数模式时，相应的 CCRx 必须置 0（ARR 寄存器设置脉冲长度）。如果定时器配置为递减计数模式，CCRx 必须高于或等于 ARR。

注意：此模式不能与中心对齐 PWM 模式一起使用。在 TIM1_CR1 中必须设置 CMS[1:0]=00。

4.6.8 电机功能应用

4.6.8.1 互补输出和死区插入

高级控制定时器（TIM1）可以输出两路互补信号，并管理输出的关断与接通瞬间。这段时间通常称为死区，用户必须根据与输出相连接的器件及其特性（电平转换器的固有延迟、开关器件产生的延迟...）来调整死区时间。死区延迟对于所有通道均相同，可通过 TIM1_BDTR 寄存器中的 DTG 位进行编程。

互补信号 OCx 和 OCxN 通过以下多个控制位的组合进行激活：TIM1_CCER 寄存器中的 CCxE 和 CCxNE 位以及 TIM1_BDTR 和 TIM1_CR2 寄存器中的 MOE、OISx、OISxN、OSSI 和 OSSR 位。

CCxE 和 CCxNE 位同时置 1 并且 MOE 位置 1（如果存在刹车）时，将使能死区插入。每个通道有一个 10 位死区发生器。将基于参考波形 OCxREF 生成 2 个输出 OCx 和 OCxN。

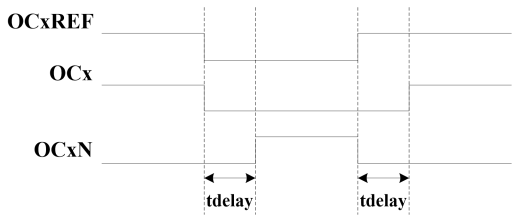
如果 OCx 和 OCxN 为高电平有效：

- 输出信号 OCx 与参考信号相同，其上升沿相对参考上升沿存在延迟。
- 输出信号 OCxN 与参考信号相反，其上升沿相对参考下降沿存在延迟。

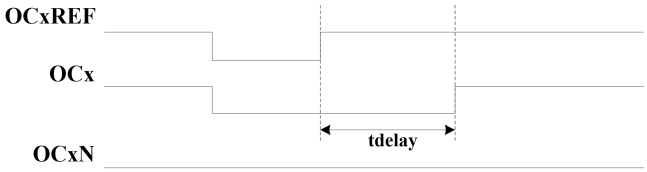
如果 OCx 和 OCxN 为低电平有效：

- 输出信号 OCx 与参考信号相反，其下降沿相对参考下降沿存在延迟。
- 输出信号 OCxN 与参考信号相同，其下降沿相对参考下降沿存在延迟。

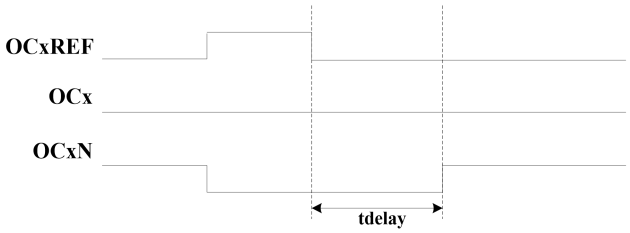
如果延迟时间大于有效输出（OCx 或 OCxN）的宽度，则不会产生相应的脉冲。



带死区插入的互补输出



延迟时间大于负脉冲宽度的死区波形



延迟时间大于正脉冲宽度的死区波形

将 OCxREF 重定向到 OCx 或 OCxN

在输出模式（强制输出模式、输出比较模式或 PWM 模式）下，通过配置 TIM1_CCER 寄存器中的 CCxE 和 CCxNE 位，可将 OCxREF 重定向到 OCx 输出或 OCxN 输出。

通过此功能，可以在一个输出上发送特定波形（如 PWM 或静态有效电平），而同时使互补输出保持其无效电平。或者，使两个输出同时保持无效电平，或者两个输出同时处于有效电平，两者互补并且带死区。

注意：如果仅使能 OCxN (CCxE=0, CCxNE=1)，两者不互补，一旦 OCxREF 为高电平，OCxN 即变为有效。例如，如果 CCxNP=0，则 OCxN=OCxREF。另一方面，如果同时使能 OCx 和 OCxN (CCxE=CCxNE=1)，OCx 在 OCxREF 为高电平时变为有效，而 OCxN 则与之互补，在 OCxREF 为低电平时变为有效。

4.6.8.2 生成六步 PWM

当通道使用互补输出时，OCxM、CCxE 和 CCxNE 位上提供预装载位。发生 COM 换向事件时，这些预装载位将传输到影子位。因此，用户可以预先编程下一步骤的配置，并同时更改所有通道的配置。COM 可由软件通过将 TIM1_EGR 寄存器中的 COM 位置 1 而生成，也可以由硬件在 TRGI 上升沿生成。
发生 COM 事件时，某个标志位 COMIF（位于 TIM1_SR 寄存器）将会置 1。这时，如果 TIM1_DIER 寄存器中的 COMIE 位置 1，将产生中断；如果 TIM1_DIER 寄存器中的 COMDE 位置 1，则将产生 DMA 请求。

4.6.8.3 编码器接口模式

选择编码器接口模式时，如果计数器仅在 TI1 边沿处计数，在 TIM1_SMCR 寄存器中写入 SMS="001"（编码器模式 1）；如果计数器仅在 TI2 边沿处计数，写入 SMS="010"（编码器模式 2）；如果计数器在 TI1 和 TI2 边沿处均计数，则写入 SMS="011"（编码器模式 3）。

通过编程 TIM1_CCER 寄存器的 CC1P 和 CC2P 位，选择 TI1 和 TI2 极性。必要时，也可以对输入滤波器进行编程。CC1NP 和 CC2NP 必须保持低电平。

TI1 和 TI2 两个输入用于连接正交编码器。如果使能计数器（在 TIM1_CR1 寄存器的 CEN 位中写入"1"），则计数器的时钟由 TI1FP1 或 TI2FP2 上的每次有效信号转换提供。TI1FP1 和 TI2FP2 是进行输入滤波器和极性选择后 TI1 和 TI2 的信号，如果不进行滤波和反相，则 TI1FP1=TI1，TI2FP2=TI2。将根据两个输入的信号转换序列，产生计数脉冲和方向信号。根据该信号转换序

列，计数器相应递增或递减计数，同时硬件对 TIM1_CR1 寄存器的 DIR 位进行相应修改。任何输入（TI1 或 TI2）发生信号转换时，都会计算 DIR 位，无论计数器是仅在 TI1 或 TI2 边沿处计数，还是同时在 TI1 和 TI2 处计数。

编码器接口模式就相当于带有方向选择的外部时钟。这意味着，计数器仅在 0 到 TIM1_ARR 寄存器中的自动重载值之间进行连续计数（根据具体方向，从 0 递增计数到 ARR，或从 ARR 递减计数到 0）。因此必须在启动之前配置 TIM1_ARR。同样，捕获、比较、重复计数器和触发输出功能继续正常工作。编码器模式和外部时钟模式 2 不兼容，因此不能同时选择。

注意：使能编码器模式时，预分频器必须设置为零。

在此模式下，计数器会根据正交编码器的速度和方向自动进行修改，因此，其内容始终表示编码器的位置。计数方向对应于所连传感器的旋转方向。下表汇总了可能的组合（假设 TI1 和 TI2 不同时切换）。

计数方向与编码器信号关系

有效边沿	相对信号的电平（TI1FP1 对应 TI2，TI2FP2 对应 TI1）	TI1FP1 信号		TI2FP2 信号	
		上升	下降	上升	下降
仅在 TI1 计数	高	向下计数	向上计数	不计数	不计数
	低	向上计数	向下计数	不计数	不计数
仅在 TI2 计数	高	不计数	不计数	向上计数	向下计数
	低	不计数	不计数	向下计数	向上计数
在 TI1 和 TI2 上计数	高	向下计数	向上计数	向上计数	向下计数
	低	向上计数	向下计数	向下计数	向上计数

4.6.8.4 UIF 位重映射

TIM1_CR1 寄存器中的 UIFREMAP 位强制将更新中断标志 UIF 连续复制到定时器计数器寄存器的位 31(TIM1_CNT[31])中。这样便可自动读取计数器值以及由 UIFCPY 标志发出的电位翻转条件。在特定情况下，这可避免在后台任务（计数器读）和中断（更新中断）之间共享处理时产生竞争条件，从而简化计算。

UIF 和 UIFCPY 标志使能之间没有延迟。

4.6.8.5 连接霍尔传感器

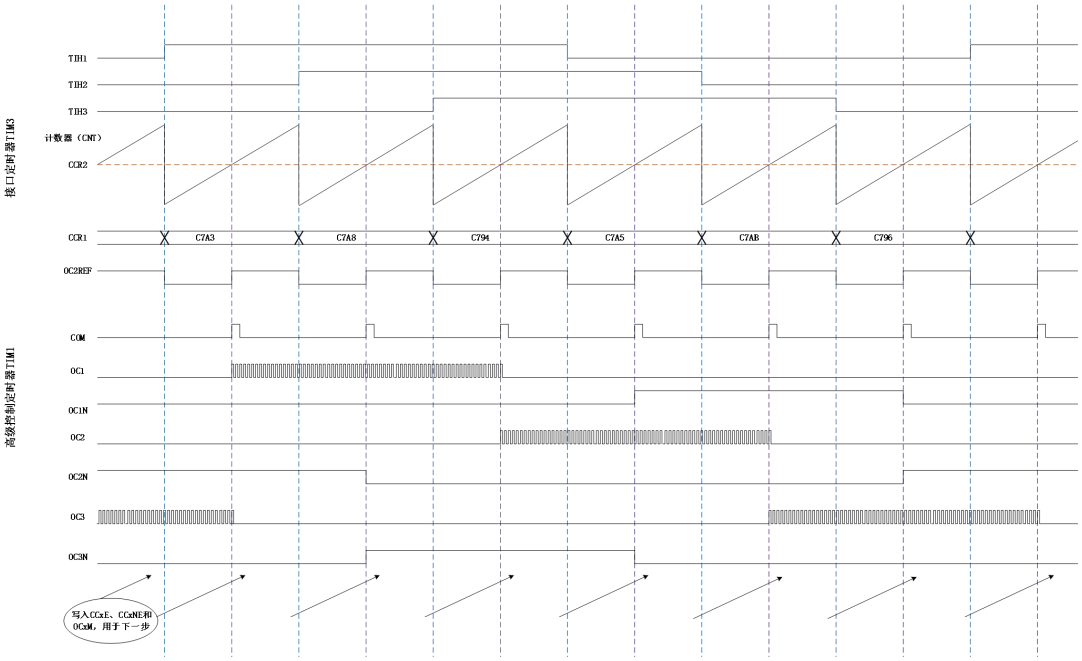
可通过用于产生电机驱动 PWM 信号的高级控制定时器(TIM1) 以及称为“接口定时器”的另一个定时器 TIM3，实现与霍尔传感器的连接。定时器的 3 个输入引脚（CH1、CH2 和 CH3）通过异或门连接到 TI1 输入通道（通过将 TIM3_CR2 寄存器中的 TI1S 位置 1 来选择），并由“接口定时器”进行捕获。

“接口定时器”的从模式控制器配置为复位模式(TIM1_SMCR 寄存器的位 SMS=”0100”)；触发选择 TI1F_ED(TIM1_SMCR 寄存器的位 TS=”00100”)。这样，每当 3 个输入中有一个输入发生切换时，计数器会从 0 开始重新计数。这样将产生由霍尔输入的任何变化而触发的时基。

在“接口定时器”上，捕获/比较通道 1 配置为捕获模式，捕获信号为 TRC(TIM1_CCMR1 寄存器的位 CC1S=”11”)。捕获值对应于输入上两次变化的间隔时间，可提供与电机转速相关的信息。

“接口定时器”可用于在输出模式下产生脉冲，以通过触发 COM 事件更改高级控制定时器(TIM1)各个通道的配置。TIM1 定时器用于生成电机驱动 PWM 信号。为此，必须对接口定时器通道进行编程，以便在编程的延迟过后产生正脉冲（在输出比较或 PWM 模式中）。该脉冲通过 TRGO 输出发送到高级控制定时器(TIM1)的 TRGI。

在高级控制定时器 TIM1 中，必须选择正确的 ITR 输入作为触发输入，定时器编程为可产生 PWM 信号，捕获/比较控制信号进行预装载（TIM1_CR2 寄存器的 CCPC=1），并且 COM 事件由触发输入控制（TIM1_CR2 寄存器的 CCUS=1）。发生 COM 事件后，在 PWM 控制位（CCxE、OCxM）中写入下一步的配置，此操作可在由“接口定时器”的 OCxREF 上升沿产生的中断子程序中完成。



4.6.9 发生外部事件时清除 OCxREF 信号

对于给定通道，在 ocref_clr_int 输入上施加高电平（相应 TIM1_CCMRx 寄存器中的 OCxCE 使能位置 1），可将 OCxREF 信号清零。OCxREF 信号将保持低电平，直到发生下一更新事件(UEV)。该功能只能在输出比较模式和 PWM 模式下使用。在强制模式下不起作用。

选择 ETRF 时，ETR 必须配置如下：

1. 必须关闭外部触发预分频器：TIM1_SMCR 寄存器中的 ETPS[1:0]位置“00”。
2. 必须禁止外部时钟模式 2：TIM1_SMCR 寄存器中的 ECE 位置“0”。
3. 外部触发极性(ETP)和外部触发滤波器(ETF)可根据用户需要进行配置。

4.6.10 定时器输入异或功能

通过 TIM1_CR2 寄存器中的 TI1S 位，可将通道 1 的输入滤波器连接到异或门的输出，从而将 TIM1_CH1 到 TIM1_CH3 这三个输入引脚组合在一起。

异或输出可与触发或输入捕获等所有定时器输入功能配合使用。

4.6.11 DMA 连续传送模式

TIM1 定时器能够根据一个事件生成多个 DMA 请求。主要目的是能够对定时器的一部分寄存器多次重新编程而无需软件开销，但也可用于定期读取一行中的多个连续的寄存器。

DMA 控制器目标唯一，必须指向虚拟寄存器 TIM1_DMAR。发生给定的定时器事件时，定时器会启动 DMA 请求序列（突发）。每次写入 TIM1_DMAR 寄存器都会重定向到其中一个定时器寄存器。

TIM1_DCR 寄存器中的 DBL[4:0]位设置 DMA 连续传送长度。当对 TIM1_DMAR 地址进行读或写访问时，定时器进行一次连续传送，即传送次数（按半字或字节）。

TIM1_DCR 寄存器中的 DBA[4:0]位定义 DMA 传送的 DMA 基址（通过 TIM1_DMAR 地址执行读/写访问时）。DBA 定义为从 TIM1_CR1 寄存器地址开始计算的偏移量：

示例：

00000：TIM1_CR1

00001：TIM1_CR2

00010：TIM1_SMCR

例如，定时器 DMA 连续传送功能用于在发生更新事件后将 CCRx 寄存器（x =1、2、3、4）的内容更新为通过 DMA 传输到 CCRx 寄存器中的多个半字。具体操作步骤如下：

1. 将相应的 DMA 通道配置如下：
   - DMA 通道外设地址为 DMAR 寄存器地址；
   - DMA 通道存储器地址为包含要通过 DMA 传输到 CCRx 寄存器的数据的 RAM 缓冲区地址；
   - 要传输的数据量= 4；
   - 禁止循环模式。
2. 通过将 DBA 和 DBL 位域配置如右来配置 DCR 寄存器：DBL=3，DBA= 0xE
3. 使能 TIM1 更新 DMA 请求（DIER 寄存器中的 UDE 位置 1）
4. 使能 TIM1
5. 使能 DMA 通道

本例适用于每个 CCRx 寄存器只更新一次的情况。如果每个 CCRx 寄存器要更新两次，则要传

输的数据量应为 8。下面以包含 data1、data2、data3、data4、data5、data6、data7 和 data8 的 RAM 缓冲区为例。数据将按照如下方式传输到 CCRx 寄存器：在第一个更新 DMA 请求期间，data1 传输到 CCR1，data2 传输到 CCR2，data3 传输到 CCR3，data4 传输到 CCR4；在第二个更新 DMA 请求期间，data5 传输到 CCR1，data6 传输到 CCR2，data7 传输到 CCR3，data8 传输到 CCR4。

4.6.12 定时器初始化结构体详解

在 LL 库函数头文件 bs32f0xx_ll_tim.h 中对定时器外设建立了六个初始化结构体，分别为时基初始化结构体 LL_TIM_InitTypeDef、输出比较初始化结构体 LL_TIM_OC_InitTypeDef、输入捕获初始化结构体 LL_TIM_IC_InitTypeDef、编码器接口初始化结构体 LL_TIM_ENCODER_InitTypeDef、霍尔传感器接口初始化结构体 LL_TIM_HALLSENSOR_InitTypeDef、刹车和死区初始化结构体 LL_TIM_BDTR_InitTypeDef，高级控制定时器 TIM1 可以用到所有初始化结构体，通用定时器除了 TIM15 外不能使用 LL_TIM_BDTR_InitTypeDef 结构体。接下来我们具体讲解这六个结构体。

LL_TIM_InitTypeDef

时基初始化结构体 LL_TIM_InitTypeDef 用于定时器基础参数设置，与 LL_TIM_Init 函数配合使用完成配置。

```
typedef struct
{
  uint16_t Prescaler;          //预分频器
  uint32_t CounterMode;        //计数模式
  uint32_t Autoreload;         //定时器周期
  uint32_t ClockDivision;      //时钟分频
  uint8_t RepetitionCounter;   //重复计数器
} LL_TIM_InitTypeDef;
```

Prescaler：定时器预分频设置，时钟源经该预分频器才是定时器计数时钟 CK_CNT,它设定 PSC 寄存器的值。计算公式为：计数器时钟频率（f_{CK_CNT}）=f_{CK_PSC}/(PSC[15:0] + 1)，可实现 1 至 65536 分频。

CounterMode：定时器计数模式，可设置为向上计数、向下计数以及中心对齐。高级控制定时器允许选择任意一种。

Autoreload：定时器周期，实际就是设定自动重载寄存器 ARR 的值，ARR 为要装载到实际自动重载寄存器（即影子寄存器）的值，可设置范围为 0 至 65535。

ClockDivision：时钟分频，设置定时器时钟 CK_CNT 频率与死区发生器以及数字滤波器采样时钟频率分频比。可以选择 1、2、4 分频。

RepetitionCounter：重复计数器，只有 8 位，只存在于高级定时器。

LL_TIM_OC_InitTypeDef

输出比较结构体 LL_TIM_OC_InitTypeDef 用于输出比较模式，与 LL_TIM_OC_Init 函数配合使用完成指定定时器输出通道初始化配置。高级控制定时器有四个定时器通道，使用时都必须单独

设置。

```
typedef struct
{
  uint32_t OCMode;         //比较输出模式
  uint32_t OCState;        //比较输出使能
  uint32_t OCNState;       //比较互补输出使能
  uint32_t CompareValue;   //脉冲宽度
  uint32_t OCPolarity;     //输出极性
  uint32_t OCNPolarity;    //互补输出极性
  uint32_t OCIdleState;    //空闲状态下比较输出状态
  uint32_t OCNIdleState;   //空闲状态下比较互补输出状态
} LL_TIM_OC_InitTypeDef;
```

OCMode：比较输出模式选择，常用的为 PWM1/PWM2。它设定 CCMRx 寄存器 OCxM[3:0]位的值。

OCState：比较输出使能，决定最终的输出比较信号 OCx 是否通过外部引脚输出。它设定 TIMx_CCER 寄存器 CCxE 位的值。

OCNState：比较互补输出使能，决定 OCx 的互补信号 OCxN 是否通过外部引脚输出。它设定 TIMx_CCER 寄存器 CCxNE 位的值。

CompareValue：比较输出脉冲宽度，实际设定比较寄存器 CCR 的值，决定脉冲宽度。可设置范围为 0 至 65535。

OCPolarity：比较输出极性，可选 OCx 为高电平有效或低电平有效。它决定着定时器通道有效电平。它设定 TIMx_CCER 寄存器 CCxP 位的值。

OCNPolarity：比较互补输出极性，可选 OCxN 为高电平有效或低电平有效。它设定 TIMx_CCER 寄存器 CCxNP 位的值。

OCIdleState：空闲状态时通道输出电平设置，可选输出 1 或输出 0，即在空闲状态（BDTR_MOE 位为 0）时，经过死区时间后定时器通道输出高电平或低电平。它设定 CR2 寄存器的 OISx 位的值。

OCNIdleState：空闲状态时互补通道输出电平设置，可选输出 1 或输出 0，即在空闲状态（BDTR_MOE 位为 0）时，经过死区时间后定时器通道输出高电平或低电平，设定值必须与 OCIdleState 相反。它设定 CR2 寄存器的 OISxN 位的值。

LL_TIM_IC_InitTypeDef

输入捕获结构体 LL_TIM_IC_InitTypeDef 用于输入捕获模式，与 LL_TIM_IC_Init 函数配合使用完成定时器输入通道初始化配置。

```
typedef struct
{
  uint32_t ICPolarity;      //输入捕获触发选择
  uint32_t ICActiveInput;   //输入通道选择
  uint32_t ICPrescaler;     //输入捕获预分频器
  uint32_t ICFilter;        //输入捕获滤波器
```

```
} LL_TIM_IC_InitTypeDef;
```

ICPolarity：输入捕获边沿触发选择，可选上升沿触发、下降沿触发或边沿跳变触发。它设定 TIMx_CCER 寄存器 CCxP 位和 CCxNP 位的值。

ICActiveInput：输入通道选择，捕获通道 ICx 的信号可来自三个输入通道，分别为 LL_TIM_ACTIVEINPUT_DIRECTTI、LL_TIM_ACTIVEINPUT_INDIRECTTI、LL_TIM_ACTIVEINPUT_TRC。它设定 CCMRx 寄存器的 CCxS[1:0]位的值。

ICPrescaler：输入捕获通道预分频器，可设置 1/2/4/8 分频，它设定 CCMRx 寄存器的 ICxPSC[1:0]位的值。如果需要捕获输入信号的每个有效边沿，则设置 1 分频即可。

ICFilter：输入捕获滤波器设置，可选设置 0x0 至 0x0F。它设定 CCMRx 寄存器 ICxF[3:0]位的值。一般我们不使用滤波器，即设置为 0。

LL_TIM_ENCODER_InitTypeDef

编码器接口结构体 LL_TIM_ENCODER_InitTypeDef 用于配置编码器工作，与 LL_TIM_ENCODER_Init 函数配合使用完成定时器编码器接口初始化配置。

```
typedef struct
{
  uint32_t EncoderMode;      //编码器模式选择
  uint32_t IC1Polarity;      //输入通道 1 触发选择
  uint32_t IC1ActiveInput;   //输入通道 1 选择
  uint32_t IC1Prescaler;     //输入捕获预分频器 1
  uint32_t IC1Filter;        //输入捕获滤波器 1
  uint32_t IC2Polarity;      //输入通道 2 触发选择
  uint32_t IC2ActiveInput;   //输入通道 2 选择
  uint32_t IC2Prescaler;     //输入捕获预分频器 2
  uint32_t IC2Filter;        //输入捕获滤波器 2
} LL_TIM_ENCODER_InitTypeDef;
```

EncoderMode：编码器模式选择提供三个模式选取：编码器模式 1（计数器根据 TI2FP2 电平在 TI1FP1 边沿递增/递减计数），编码器模式 2（计数器根据 TI1FP1 电平在 TI2FP2 边沿递增/递减计数），编码器模式 3（计数器在 TI1FP1 和 TI2FP2 的边沿计数，计数的方向取决于另外一个输入的电平）。它设定 TIM1_SMCR 寄存器的 SMS[3:0]位的值。

其他部分成员内容见输入捕获结构体成员描述。

LL_TIM_HALLSENSOR_InitTypeDef

霍尔传感器结构体 LL_TIM_HALLSENSOR_InitTypeDef 用于配置霍尔元件相关内容，与 LL_TIM_HALLSENSOR_Init 函数配合使用完成定时器与霍尔元件通讯初始化配置。

```
typedef struct
{
  uint32_t IC1Polarity;        //输入捕获触发选择
  uint32_t IC1Prescaler;       //输入捕获预分频器
  uint32_t IC1Filter;          //输入捕获滤波器
  uint32_t CommutationDelay;   //换向延迟（脉冲宽度）
```

```
} LL_TIM_HALLSENSOR_InitTypeDef;
```

CommutationDelay：该成员换向延迟在 SDK 中实际为比较输出脉冲宽度，实际设定比较寄存器 CCR2 的值，决定脉冲宽度。可设置范围为 0 至 65535。

其他部分成员内容见输入捕获结构体成员描述。

LL_TIM_BDTR_InitTypeDef

刹车和死区初始化结构体 LL_TIM_BDTR_InitTypeDef 用于刹车和死区参数的设置，属于高级定时器专用，用于配置刹车时通道输出状态，以及死区时间。它与 LL_TIM_BDTR_Init 函数配置使用完成参数配置。

```
typedef struct
{
  uint32_t OSSRState;       //运行模式下的关闭状态选择
  uint32_t OSSIState;       //空闲状态下的关闭状态选择
  uint32_t LockLevel;       //锁定配置
  uint8_t DeadTime;         //死区时间
  uint16_t BreakState;      //刹车输入使能控制
  uint32_t BreakPolarity;   //刹车输入极性
  uint32_t BreakFilter;     //刹车滤波器
  uint32_t BreakAFMode;     //刹车输入复用模式
  uint32_t Break2State;     //刹车 2 输入使能控制
  uint32_t Break2Polarity;  //刹车 2 输入极性
  uint32_t Break2Filter;    //刹车 2 滤波器
  uint32_t Break2AFMode;    //刹车 2 输入复用模式
  uint32_t AutomaticOutput; //自动输出使能
} LL_TIM_BDTR_InitTypeDef;
```

OSSRState：运行模式下的关闭状态选择，它设定 BDTR 寄存器 OSSR 位的值。

OSSIState：空闲模式下的关闭状态选择，它设定 BDTR 寄存器 OSSI 位的值。

LockLevel：锁定级别配置，BDTR 寄存器 LOCK[1:0]位的值。

DeadTime：配置死区发生器，定义死区持续时间，可选设置范围为 0x0 至 0xFF。它设定 BDTR 寄存器 DTG[7:0]位的值。

BreakState：刹车输入功能选择，可选使能或禁止。它设定 BDTR 寄存器 BKE 位的值。

BreakPolarity:刹车输入通道 BRK 极性选择，可选高电平有效或低电平有效。它设定 BDTR 寄存器 BKP 位的值。

BreakFilter：刹车滤波器的配置，可选设置 0x0 至 0xF。它设定 BDTR 寄存器 BKF[3:0]位的值。

BreakAFMode：刹车输入复用模式的配置，可选 BRK 为输入模式或双向模式。它设定 BDTR 寄存器 BKBID 位的值。

AutomaticOutput：自动输出使能，可选使能或禁止，它设定 BDTR 寄存器 AOE 位的值。

4.10 IWDG 模块介绍

BS32F030 有两个看门狗，一个是独立看门狗（IWDG）一个是窗口看门狗（WWDG）。独立看门狗（IWDG）可以理解为 12 位递增计数器，当计数器的值从某个值一直加到配置值的时候，系统就会产生一个复位信号。如果计数没有到配置值之前刷新了计数器的值，那么就不会产生复位信号，这个动作就是我们常说的喂狗。看门狗功能由 VDD 电压域供电，在停止模式和待机模式下仍能工作。

其预分频系数可配置：4/8/16/32/64/128/256 分频。最大可配置定时时间约为 32s，时钟源为 LSI。具有寄存器写保护功能，通过配置字可配置为硬件看门狗或软件看门狗，可选窗口模式。

复位条件：

- 当计数器值大于等于配置的最大值时，产生复位
- 当计数值小于窗口值时喂狗，产生复位

4.10.1 IWDG 功能详解

IWDG 主要包括以下几部分功能：

独立看门狗时钟

独立看门狗的时钟由独立的 RC 振荡器 LSI 提供，即使主时钟发生故障它依然有效，非常独立。BS32F030 的 LSI 频率为 32.768Khz，根据温度和工作场合会有一定的漂移，适用于对时间精度要求比较低的场合。

计数器时钟

递增计数器的时钟由 LSI 经过一个 3 位的预分频器得到，我们可以操作预分频器寄存器 IWDG_PR 来设置分频因子，可选 4/8/16/32/64/128/256 分频，分频得到计数器时钟，一个计数器时钟计数器就加 1。

计数器

独立看门狗的计数器是一个 12 位的递增计数器，最大值为 0XFFF，当计数器从 0 加到最大值或窗口值时，会输出一个看门狗复位信号，让程序重新启动运行，如果在计数器从 0 加到最大值之前刷新了计数器的值的话，就不会产生复位信号，重新刷新计数器值的这个动作我们称为喂狗。

计数的最大值通过 IWDG_MAXR 寄存器配置，每次更改此寄存器值，均会导致看门狗计数器清零重新计数。

键值寄存器

键值寄存器 IWDG_KR 可以说是独立看门狗的一个控制寄存器，主要有三种控制方式，往这个寄存器写入下面三个不同的值有不同的效果。

键值	键值作用
0XDDDD	关闭看门狗（看门狗使能默认开启，写入其他任意值也会开启使能）

0X5555	解锁对其余寄存器的写访问（默认开启写保护，写入其他任意值会重新锁定）
0XAAAA	喂狗

状态寄存器

状态寄存器 SR 只有位 0：window_updating、位 1：cntmax_updating、位 2：prediv_updating 有效，这三位只能由硬件置位。必须在标志位拉低以后才能再次配置对应寄存器。

window_updating：计数器窗口值更新标志，软件写 IWDG_WINR 寄存器时，硬件将该位置 1 表示窗口值正在更新。完成更新操作后（最长需要 3 个 32k 周期），会通过硬件将此位清零。

cntmax_updating：计数器最大值更新标志，软件写 IWDG_MAXR 寄存器时，硬件将该位置 1 表示计数最大值正在更新。完成更新操作后（最长需要 3 个 32k 周期），会通过硬件将此位清零。

prediv_updating：预分频器值更新标志，软件写 IWDG_PR 寄存器时，硬件将该位置 1 表示预分频器值正在更新。完成更新操作后（最长需要 3 个 32k 周期），会通过硬件将此位清零。

4.10.2 IWDG 的使用

独立看门狗一般用来检测和解决由程序引起的故障，比如一个程序正常运行的时间是 50ms，在运行完这段程序之后紧接着进行喂狗，我们设置独立看门狗的定时溢出时间为 60ms，比我们需要监控的程序 50ms 多一点，如果超过 60ms 还未喂狗，那就说明我们监控的程序出现故障跑飞，此时产生看门狗复位，程序重新运行。

窗口模式

窗口模式无使能，默认情况下，窗口值为 0，相当于窗口模式无效。当窗口值配置为其他值时，窗口模式生效，如果在计数值小于窗口值时喂狗，则会产生看门狗复位。

看门狗复位

必须在看门狗计数值大于窗口值且小于最大值时喂狗，否则将产生看门狗复位。这表示软件必须在特定的时间范围内进行喂狗操作，过早喂狗或者不喂狗都被认为是异常，并导致系统复位。

应用流程

1. 开启配置字 IWDG 使能（出厂前已完成）。
2. 将 0x5555 写入 IWDG_KR 寄存器，解锁写保护。
3. 写 IWDG_PR 寄存器，配置预分频系数。
4. 写 IWDG_MAXR 寄存器，配置计数最大值。
5. 如需使用窗口功能，写 IWDG_WINR 寄存器配置窗口值。
6. 轮询 IWDG_SR 寄存器，直到所有位清零，代表上述写入的配置值已生效。
7. 可写入一个错误的键值到 IWDG_KR 寄存器，使写保护重新激活。
8. 上述操作会使看门狗重新开始计数，软件需要在限定的时间内喂狗（将 0xAAAA 写入 IWDG_KR 寄存器）。

9. 如需关闭看门狗，将 0xDDDD 写入 IWDG_KR 寄存器。

4.11 WWDG 模块介绍

BS32F030 有两个看门狗，一个是独立看门狗（IWDG）一个是窗口看门狗（WWDG）。我们知道独立看门狗的工作原理是一个递增计数器不断地向上递增计数，当加到最大值之前如果没有喂狗的话，产生复位。窗口看门狗配有 6 位递增计数器，预分频系数可配：1/2/4/8/16/32/64/128 分频。计数时钟源为 PCLK，经过 4096 分频后再分频。

- 喂狗操作：与 IWDG 不同的是，WWDG 通过再写 WWDG_CNTR 与 WWDG_WINR 寄存器完成喂狗，喂狗后看门狗计数器回到 0 重新开始计数。
- 窗口模式：可选窗口模式（无使能控制，默认情况下，窗口值为 0，相当于窗口功能无效）。
- 复位条件：（看门狗使能有效时）
  - 当递增计数器大于配置的最大值时，产生复位
  - 在计数值小于窗口值时喂狗，产生复位
- 中断：当递增计数器等于最大值减 1 时触发中断（需要先配置中断使能），用于提醒系统即将复位。

4.11.1 WWDG 功能详解

WWDG 主要包含以下几部分功能：

窗口看门狗时钟

窗口看门狗时钟来自 PCLK，BS32F030 的 PCLK 最大值为 24M。

计数器时钟

计数器时钟由 PCLK 经过 4096 分频后再经过 WWDG_CR 寄存器的 prediv 分频获得，可选分频系数为 1/2/4/8/16/32/64/128 分频（2^{prediv}）。

计数器

窗口看门狗的计数器是一个递增计数器，共有 6 位，其值存在 WWDG_CNTR 寄存器的 count 位中，当 6 个位全部为 1 时是 0x3f，这个是最大值。当前看门狗计数值可通过读取 WWDG_CNTR 寄存器得到。

窗口值

我们知道窗口看门狗必须在计数器的值在一个范围内才可以喂狗，其中上下边界分别为 WWDG_CNTR 寄存器设定的最大值与 WWDG_WINR 寄存器设定的窗口值。

4.11.2 WWDG 的使用

WWDG 一般被用来监测，由外部干扰或不可预见的逻辑条件造成的应用程序背离正常的运行序

列而产生的软件故障。比如一个程序段正常运行时间是 50ms，在运行完这段程序之后紧接着进行喂狗，如果在规定的时间窗口内还没有喂狗，就说明我们监控的程序出故障跑飞了，那么系统就会产生看门狗复位，让程序重新运行。

与 IWDG 相比，WWDG 适合那些要求看门狗在精确计时窗口起作用的应用程序。

看门狗使能：WWDG 需要通过 WWDG_CR 寄存器的 wwdg_en 位配置看门狗使能，默认关闭。

窗口模式：通过 WWDG_WINR 寄存器配置窗口值，可配置为 0x00~0x3F 范围内的值。窗口值默认为 0，即默认条件下窗口功能无效。必须在计数值大于等于窗口值且小于最大值时喂狗，否则看门狗将输出复位信号。

中断：通过 WWDG_CR 寄存器的 wwdg_ie 位配置中断使能。当看门狗计数值到达最大值-1 时，会产生看门狗中断，用于提醒系统即将复位。

软件有一个计数时钟周期的时间（最短为 T_{pclk}*4096）进入中断函数中执行复位前的特定操作（如保存数据等），或通过喂狗避免产生复位。如没有在中断函数中喂狗，则看门狗计数值到达最大值时输出复位信号。

即将复位标志：当计数值等于最大值-1 时，WWDG_SR 寄存器中的 wwdg_flag 位置 1，代表即将发生复位。此位由硬件置 1，软件写 0 清零。中断不使能时此位也会拉高。

应用流程：

1. 通过 WWDG_CR 寄存器配置预分频系数，可选择打开中断使能。
2. 通过 WWDG_CNTR 配置看门狗计数最大值。
3. 如需使用窗口模式，通过 WWDG_WINR 配置窗口值。
4. 通过 WWDG_CR 寄存器打开看门狗使能，看门狗计数器开始工作。
5. 在限定时间内，通过写 WWDG_CNTR 或 WWDG_WINR 寄存器进行喂狗（相当于重新配置最大值或窗口值），可通过读 WWDG_CNTR 寄存器获得当前计数值。
6. 如果使能了中断，且中断前一直没有喂狗操作，则需要在中断函数中进行系统复位前的数据备份，或者在中断函数中喂狗以避免发生系统复位。

4.12 PMU 模块介绍

电源管理单元（Power Management Unit,PMU）用以实现对 BS32F030 的电源管控，电源是保证系统稳定的基础，且有低功耗的要求。在很多场合都有对电子设备功耗的要求，同时考虑智慧穿戴设备的小型化要求，电池体积不能太大，所以很有必要从控制功耗入手，提高设备的续航。因此，BS32 有专门的电源管理单元 PMU 来控制设备的运行模式，保证系统正常运行的前提下，尽量降低器件的功耗。

4.12.1 PMU 模块功能

为了降低功耗，系统可以进入不同的低功耗模式。SLEEP_CTRL 模块用于实现系统进入和退出 SHUTDOWN 模式时的逻辑控制。

当 SLEEP_CTRL 电路采样到系统发出的 SHUTDOWN 请求时， 所有来自数字的 1.5V 信号将被 level_shift 锁存，随后生成数字复位信号，使数字系统全部复位；之后关闭 LDO1P5 电源，数字掉电，进入 SHUTDOWN 模式。

在 SHUTDOWN 模式下，POR/BG/HSI1M/PLL/ LDO1P5 模块停止工作；其余模拟模块根据进入 SHUTDOWN 模式之前的配置决定是否工作；BGLP 模块由 BOR 模块的 PD 信号决定是否工作，BOR 不工作时，BGLP 也关闭。

退出 SHUTDOWN 模式的条件：

1. 外部中断使能和 RTC 中断使能均为开启时，立即唤醒
2. 外部中断使能开启且发生发生外部中断
3. 发生 RTC 中断
4. 发生外部复位
5. 发生 BOR 复位

发生以上任一事件时，将退出 SHUTDOWN 模式，LDO1P5 重新工作。LDO 稳定后读取配置字，系统开始正常工作。退出 SHUTDOWN 模式后，软件需要读取和清除唤醒标志位。

4.12.2 电源监控器

BS32 芯片主要通过引脚 VDD 从外部获取电源，在它的内部具有电源监控器用于检测 VDD 的电压，以实现复位功能及掉电紧急处理功能，保证系统可靠运行。

上电复位和掉电复位（POR:Power On Reset 与 PDR：Power Down Reset）

当检测到 VDD 的电压低于阈值 VPOR 及 VBOR 时，无需外部电路辅助，BS32 芯片会自动保持在复位状态，防止因电压不足强行工作带来的不良后果。

即在上电刚开始电压低于 VPOR 时，BS32 保持在上电复位状态，当 VDD 电压持续上升至大于 VPOR 后也有短时间的持续保持复位时间，然后开始正常运行。而在芯片正常运行的时候，当

检测到 VDD 电压下降至低于 VBOR 阈值，会进入掉电复位状态。

可编程欠压复位 BOR

上述 POR、PDR 功能是使用其电压阈值与外部供电电压 VDD 比较，当低于工作阈值时，会直接进入复位状态，这可防止电压不足导致的误操作。除此之外，BS32 还提供了可编程欠压复位 BOR，它也是实时检测 VDD 的电压，当检测到电压低于编程的 VBOR 阈值时，会产生欠压复位。该电压阈值可通过 BOR 掉电阈值配置寄存器设置，具体参数如下。

BOR 延时选择	BOR 阈值选择	掉电阈值	恢复阈值	迟滞 (mV)	延时档位 (us)
0	0	1.902	2.015	113	59.96
	1	1.997	2.103	106	63.83
	2	2.205	2.311	106	72.26
	3	2.496	2.612	116	83.82
	4/5/6/7	2.796	2.907	111	95.3
1	0	1.899	2.015	116	120.1
	1	1.994	2.103	109	128.4
	2	2.201	2.311	110	146.4
	3	2.491	2.612	121	171
	4/5/6/7	2.791	2.907	116	195.3

4.12.3 BS32 的电源系统

为了方便进行电源管理，BS32F030 把它的外设、内核等模块根据功能划分了供电区域。

BS32F030 的电源系统主要分为内核电路以及 ADC 电路两部分如下：

➢ ADC 电源及参考电压（V_{DDA} 供电区域）

为了提高转换精度，BS32 的 ADC 配有独立的电源接口，方便进行单独的滤波。ADC 的工作电源使用 V_{DDA} 引脚输入，使用 V_{SSA} 作为独立的地连接，V_{REF} 引脚则为 ADC 提供测量使用的参考电压。

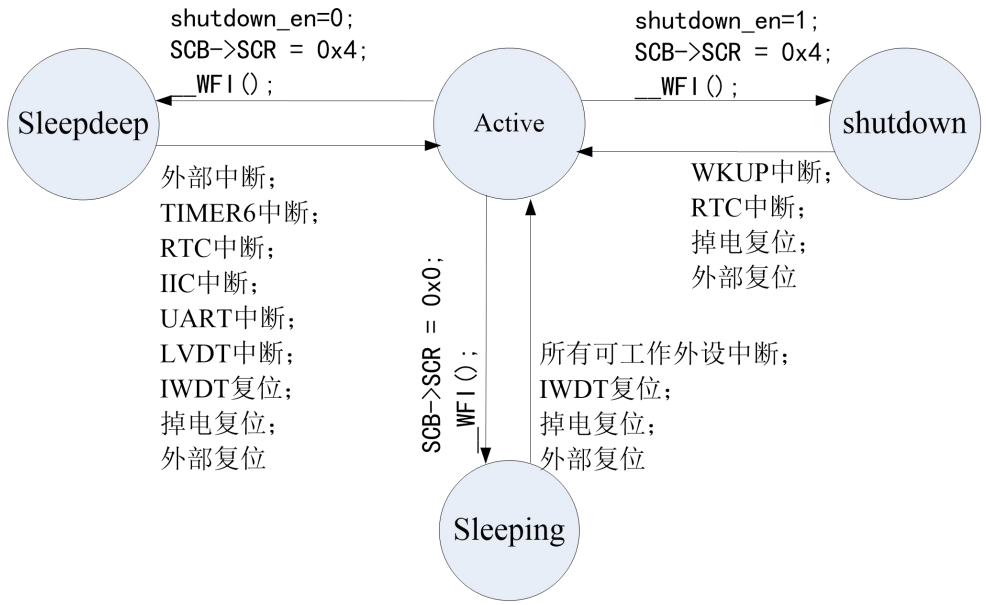
➢ 调压器供电电路（V_{DD}/1.8V 供电区域）

在 BS32 的电源系统中调压器供电的电路是最主要的部分，调压器为备份域及待机电路以外的所有数字电路供电，其中包括内核、数字外设以及 RAM，调压器的输出电压约为 1.8V，因而使用调压器供电的这些电路区域被称为 1.8V 域。

BS32 可以运行在“正常模式”、“休眠模式”、“深度休眠模式”与“关断模式”。

- 运行模式（active）：即正常工作模式，模块保持正常工作，各模块功能由软件配置控制。
- 休眠模式（sleeping）：只关闭 hclk、pclk 时钟，其余不需要 hclk、pclk 做工作时钟的模块均可工作并通过中断来退出此模式。
- 深度休眠模式（deepsleep）：此时模拟时钟源头 HSI1M/PLL 都关闭，LSI32K 工作，IWDG/TIMER6/IIC/UART/LVDT 可配置工作，CPU 和其余数字模块不工作。

- 关断模式（shutdown）：此时模拟时钟源头 HSI1M/PLL 都关闭，LSI32K 工作，LDO 关闭，数字模块掉电不工作。



配置 sleeping 与 sleepdeep 模式的寄存器在 cm0+内核里。此外，所有模块均可单独配置关闭门控，以此降低功耗。

- 退出 sleeping 模式的方式：使能 IWDG、RTC、External Interrupt、IIC、UART1/2、TIMER6、TIM1/3/14/15/16/17、ADC、LVDT，其中任意一种中断产生都可唤醒芯片，退出 SLEEPING 模式，CPU 执行中断服务程序。
- 退出 sleepdeep 模式的方式：使能 IWDG、RTC、External Interrupt、IIC、UART、TIMER6、LVDT，其中任意一种中断产生都可唤醒芯片，退出 SLEEPDEEP 模式，中断响应产生后，CPU 执行中断向量相关的中断服务程序。
- 退出 shutdown 模式的方式：使能 WKUP，RTC，其中任意一种中断产生都可唤醒芯片，退出 shutdown 模式，全局复位。

NO	分区	Module Name	时钟源	工作状态			
				Active	Sleeping	Sleepdeep	Shutdown
1	DIG	M0+	PLL48/64M	√	×	×	×
2		DMA	PLL48/64M	◎	×	×	×
3		FLASH	PLL48/64M	√	×	×	×
4		SRAM	PLL48/64M	√	×	×	×
5		SYS_CTRL	PLL48/64M	√	×	×	×
6		FLASH_CTRL	PLL48/64M	√	×	×	×
7		CRC	PLL48/64M	√	×	×	×
8		EXTI	PLL48/64M,LSI32K	◎	◎	◎	×
9		SPI1/2	PLL48/64M	◎	×	×	×
10		UART1/2	PLL48/64M,HSE4/8/16M	◎	◎	◎	×
11		IIC1/2	PLL48/64M,SCL	◎	◎	◎	×

12		TIM1/3/15	PLL48/64M,HSE4/8/16M	◎	◎	×	×
13		TIM14/16/17	PLL48/64M,HSE4/8/16M	◎	◎	×	×
14		TIMER6	LSI32K,LSE32K,HSE4/8/16M	◎	◎	◎	×
15		IWDG	LSI32K	◎	◎	◎	×
16		WWDG	PLL48/64M	◎	×	×	×
17		ADC_CTRL	PLL48/64M	◎	◎	×	×
18		GPIO	PLL48/64M,LSI32K	◎	◎	◎	×
19	ANA	RTC	PLL48/64M,LSI32K,LSE32K	◎	◎	◎	◎
20		IO	--	√	√	√	√
21		LSI32KLP	--	√	√	√	√
21		LSI32KHA	--	◎	◎	◎	◎
22		LSE32K	--	◎	◎	◎	◎
23		HSE4/8/16M	--	◎	◎	◎	×
24		HSI1M	--	√	√	×	×
25	ANA	PLL48/64M	--	√	√	×	×
26		POR	--	√	√	◎	◎
27		BOR	--	◎	◎	◎	◎
28		LDO1P5	--	√	√	√	×
29		BG	--	√	√	×	×
30		BG_LP	--	√	√	√	◎
31		ADC	--	◎	◎	◎	×
32	ANA	RTC	PLL48/64M,LSI32K,LSE32K	◎	◎	◎	◎

注：√表示工作，◎表示可选，×表示不工作

➢ 备份域电路（后备供电区域）

BS32F030XX 不支持后备电源供电。

4.12.4 不同电源模式中断唤醒

唤醒流程：在中断到来时先打开时钟 HSI1M,并记录中断状态，等待 1ms（可配置）让时钟稳定下来，然后再将相应的中断送出到内核以唤醒芯片使其正常工作。同时模块内部对时钟进行门控，在退出 sleepdeep 模式时先打开模拟时钟源 HSI1M，32us 后打开 pll48，1ms（可配置）延时后，打开数字门控信号，将时钟送给核。

模式	时钟状态	唤醒源	唤醒步骤
active	HSI1M=打开,PLL=打开		
sleeping	HSI1M=打开,PLL=打开	1. 外部中断； 2. IIC(S)中断； 3. WWDG 中断 4. LVDT 中断 5. TIMER6 中断 6. TIM1/3/15 中断	即时唤醒系统，等待 72us 后马上响应中断

		7. TIM41/16/17 中断 8. UART1/2/中断 9. ADC 中断 10. RTC 中断	
sleepdeep	LSI32K=打开， LSE32K,HSE4/8/16M 软件可配； HSI1M=关闭,PLL=关闭	1. 外部中断； 2. IIC(S)中断； 3. RTC 中断 4. LVDT 中断 5. TIMER6 中断	1.HSI1M/PLL 退出 PD 状态，LSI32K、LSE32K 状态不变； 2. HSI1M 根据配置计时 0.4/0.5/0.6/0.7/0.8/0.9/1ms 等待系统稳定，然后开启系统时钟输出到 CPU；
Shutdown	LSI32K=打开， LSE32K,HSE4/8/16M 软件可配； HSI1M=关闭,PLL=关闭	1. WKUP 中断； 2. RTC 中断	1.HSI1M/PLL 退出 PD 状态，LSI32K、LSE32K 状态不变；退出 SHUTDOWN 模式时计数 1ms，之后系统复位 1.5ms 后开始正常工作

4.12.5 PMU 应用流程

1. 配置好外部唤醒中断使能以及对应的 IO 配置，保证四个外部唤醒中断使能至少有一个开启；
2. 将需要保存的重要信息写入 RTC 的备份寄存器，以免 VDD 掉电丢失；
3. 软件配置进入 SHUTDOWN 模式；
4. 如果要唤醒系统，需要发生以下事件之一：
   - 在外部唤醒中断使能的 IO 口产生外部中断
   - 打开了 RTC 中断使能（如秒中断，天中断等）且发生了 RTC 中断
   - 使用外部复位
   - 意外情况：BOR 复位
5. 系统唤醒后，读取唤醒标志寄存器，以确定唤醒事件，之后清除所有唤醒标志；
6. 重新配置好 IO 和 BOR 相关配置后，写寄存器清除 IO 锁存信号（SHUTDOWM_IO_CLR 寄存器）；
7. 写寄存器清除 RTC 锁存信号，之后才能正常读写 RTC 寄存器。

4.13 FMC 模块介绍

FMC 是 MCU 上的 Flash 管理控制单元，在 BS32 芯片内部有一个 FLASH 存储器，它主要用于存储代码，我们在电脑上编写好应用程序后，使用下载器把编译后的代码文件烧录到该内部 FLASH 中，由于 FLASH 存储器的内容在掉电后不会丢失，芯片重新上电复位后，内核可从内部 FLASH 中加载代码并运行。

除了使用外部的工具（如下载器）读写内部 FLASH 外，BS32 芯片在运行的时候，也能对自身的内部 FLASH 进行读写，因此，若内部 FLASH 存储了应用程序后还有剩余的空间，我们可以把它像外部 SPI-FLASH 那样利用起来，存储一些程序运行时产生的需要掉电保存的数据。

由于访问内部 FLASH 的速度要比外部 SPI-FLASH 快得多，所以在紧急状态下常常会使用内部 FLASH 存储关键记录。只要 CPU 不访问闪存，闪存操作不会延缓 CPU 的执行；反之，若正在进行闪存编程或擦除时，读闪存的操作将使 CPU 被暂停，直到编程或擦除操作结束。

FMC 特性：

1. 存储空间 128Kbytes；
2. 支持 32 位整字编程，页擦除和整片擦除操作；
3. 具有支持安全保护状态，可阻止对代码或数据的非法读访问；
4. 具有擦除和编程保护状态，可阻止意外写操作。

4.13.1 内部 FLASH 的构成

BS32 的内部 FLASH 包含主存储器、系统存储器以及选项字节区域，它们的地址分布及大小见下表：

区域	名称	块地址	大小	主要功能
主存储器	页 1	0x0 0000-0x0 01FF	512bytes	存储芯片运行程序
	页 2	0x0 0200-0x0 03FF	512bytes	
	页 3	0x0 0400-0x0 05FF	512bytes	
	页 4	0x0 0600-0x0 07FF	512bytes	
	……	……	……	
	页 256	0x1 FE00-0x1 FFFF	512bytes	
系统存储区（NVR 首页）		0x2 0000-0x2 01FF	512bytes	存储用户自定义信息
选项字节（NVR 第二页）		0x2 0200-0x2 03FF	512bytes	存储配置字

各个存储区域的说明如下：

主存储器

一般我们说 BS32 内部 FLASH 的时候，都是指这个主存储器区域，它是存储用户应用程序的空间，芯片型号说明中的 128K FLASH 就是指这个区域的大小。

主存储器分为 256 页，每页大小为 512bytes，共 128KB。此处分页的概念，实质就是 FLASH 存储器的扇区，与其他 FLASH 一样，在写入数据前，要先按页（扇区）擦除。

注意上表中的主存储器是 BS32F030XX 型号芯片的参数，若使用其他大容量的产品，它们的主存储器页数量、页大小均有不同，使用的时候应注意区分。

型号范例	BS32F030XX
BS32	“BS32”表示 32bit 的 MCU
F	“F”说明产品类型
030	“030”说明具体特性
C	“C”表示 48 个引脚 其他常用：R 表示 64 引脚；V 表示 100 引脚；Z 表示 144 引脚；B 表示 208 引脚...
B	“B”表示 128KB 其他常用：6 表示 32KB；8 表示 64KB；C 表示 256KB；E 表示 512KB...
T	“T”表示 QFP 封装
6	“6”表示温度等级为 A：-40~85℃

系统存储区

系统存储区是用户不能访问的区域，它在出厂时就已经固化了启动代码，它负责实现串口、USB 以及 CAN 等 IAP 烧录功能。

选项字节

选项字节用于配置 FLASH 的读写保护、待机/停机复位、软件/硬件看门狗等功能，这部分共 16 字节。一部分可通过修改 FLASH 的选项控制寄存器修改。

4.13.2 对内部 FLASH 的功能操作

解除闪存锁

复位后，FMC 模块是被保护的，不能写入 FMC_CTRL 寄存器；通过写入特定的序列到 FMC_KEY 寄存器可以打开 FMC 模块，这个特定的序列是在 FMC_KEY 写入两个键值(KEY1=0x45670123 和 KEY2=0xCDEF89AB)；错误的操作序列都会在下次复位前锁死 FMC 模块和 FMC_CTRL 寄存器。

写入错误的键序列还会产生总线错误；总线错误发生在第一次写入的不是 KEY1，或第一次写入的是 KEY1 但第二次写入的不是 KEY2 时；FMC 模块和 FMC_CTRL 寄存器可以由程序设置 FMC_CTRL 寄存器中的 LOCK 位锁住，这时可以通过在 FMC_KEY 中写入正确的键值对 FMC 解锁。

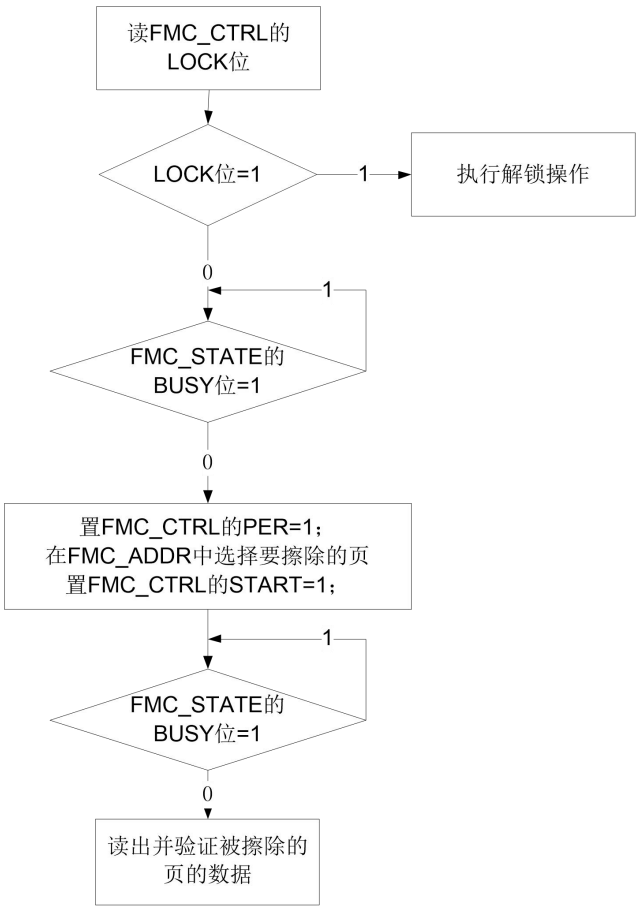
页擦除

FMC 的页擦除功能使得主存储闪存的页内容初始化为高电平。每一页都可以被独立擦除，而不影响其他页内容。FMC 擦除页步骤如下：

1. 确保 FMC_CTRL 寄存器不处于锁定状态；

2. 检查 FMC_STATE 寄存器的 BUSY 位来判定闪存是否正处于擦写访问状态，若 BUSY 位为 1， 则需等待该操作结束，BUSY 位变为 0；
3. 置位 FMC_CTRL 寄存器的 PER 位；
4. 将待擦除页的绝对地址写到 FMC_ADDR 寄存器；
5. 通过将 FMC_CTRL 寄存器的 START 位置 1 来发送页擦除命令到 FMC；
6. 等待擦除指令执行完毕，FMC_STATE 寄存器的 BUSY 位清 0；
7. 如果需要，可通过 CPU 读验证该页是否擦除成功。

当页擦除成功执行，FMC_STATE 寄存器的 END 位将置位。若 FMC_CTRL 寄存器的 ENDIE 位被置 1，则 FMC 将触发一个中断。需要注意的是，用户需确保写入的是正确的擦除地址。否则当待擦除页的地址被用来取指令或访问数据时，软件将会跑飞。该情况下，FMC 不会提供任何出错通知。另一方面，对擦写保护的页进行擦除操作将无效。如果 FMC_CTRL 寄存器的 ERIE 位被置位，该操作将触发操作出错中断。中断服务程序可通过检测 FMC_STATE 寄存器的 WPERR 位来判断该中断是否发生。下图显示了页擦除操作流程：

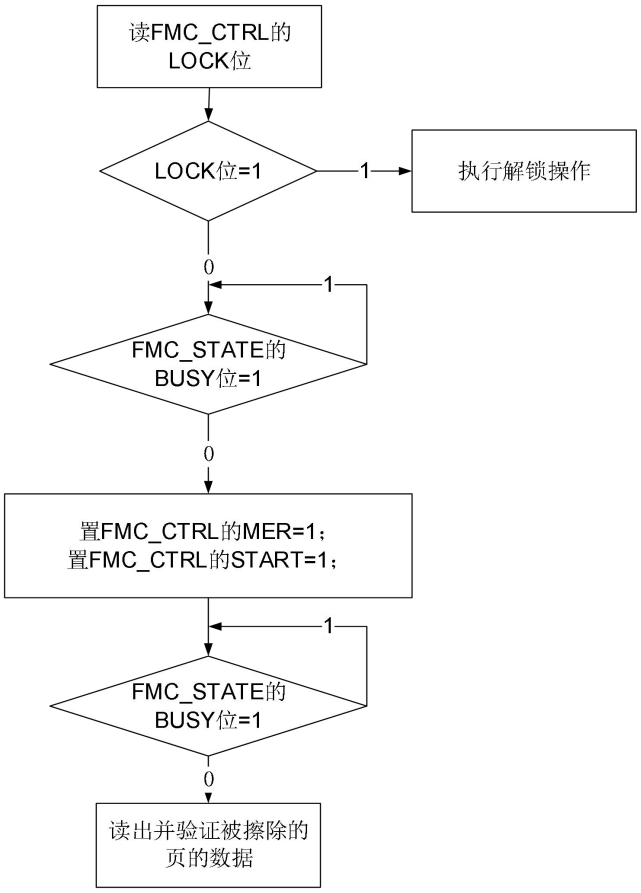


整片擦除

1. 提供了整片擦除功能可以初始化主存储闪存块的内容。整片擦除操作，寄存器设置具体步骤如下：
2. 确保 FMC_CTRL 寄存器不处于锁定状态；

3. 等待 FMC_STATE 寄存器的 BUSY 位变为 0；
4. 置位 FMC_CTRL 寄存器的 MER 位；
5. 通过将 FMC_CTRL 寄存器的 START 位置 1 来发送整片擦除命令到 FMC；
6. 等待擦除指令执行完毕，FMC_STATE 寄存器的 BUSY 位清 0；
7. 如果需要，可通过 CPU 读验证是否擦除成功。

当整片擦除成功执行，FMC_STATE 寄存器的 END 位置位。若 FMC_CTRL 寄存器的 ENDIE 位被置 1， FMC 将触发一个中断。由于所有的闪存数据都将被复位为 0xFFFF_FFFF，可以通过运行在 SRAM 中的程序或使用调试工具直接访问 FMC 寄存器来实现整片擦除操作。下图显示了整片擦除操作流程：



主存储体编程

FMC 提供了一个 32 位整字编程功能，用来修改主存储闪存块内容。编程操作使用各寄存器流程如下：

1. 确保 FMC_CTRL 寄存器不处于锁定状态；
2. 等待 FMC_STATE 寄存器的 BUSY 位变为 0；
3. 置位 FMC_CTRL 寄存器的 PG 位；
4. CPU 写一个 32 位整字到目的绝对地址（又称实际地址或物理地址）（0x000X XXXX）；

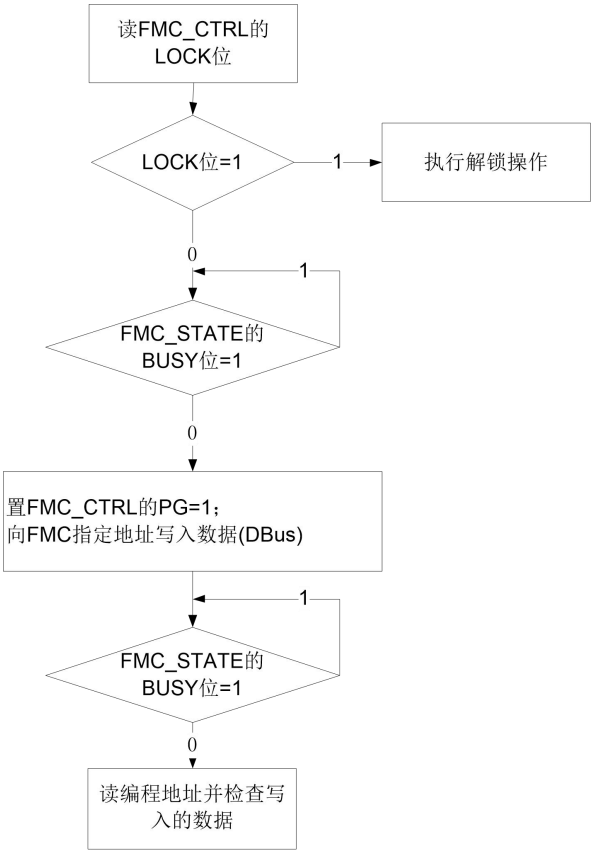
5. 等待编程指令执行完毕，FMC_STATE 寄存器的 BUSY 位清 0；

6. 如果需要，可通过 CPU 读验证是否编程成功。

Flash 物理地址表：

页码	1	2	3	…	255	256
物理地址十六进制	0	80	100	…	7F00	7F80

当主存储块编程成功执行，FMC_STATE 寄存器的 END 位置位。若 FMC_CTRL 寄存器的 ENDIE 位被置 1， FMC 将触发一个中断。需要注意的是，执行整字编程操作时需要检查目的地址是否已经被擦除。如果该地址没有被擦除，对该地址写一个非 0x0 值，FMC_STATE 寄存器的 PGERR 位将被置 1，对该地址的编程操作无效（当写内容为 0x0 时，即使目的地址没有被正常擦除，也可以正确编程）。另一方面，如果目的地址在一个处于擦除和编程保护的页中，编程不会成功且 FMC_STATE 寄存器的 WPERR 位将会置位。在这两种情形下，如果 FMC_CTRL 寄存器的 ERIE 位被置 1，FMC 将触发一次闪存操作错误中断。在中断服务程序中，可以检查 FMC_STATE 寄存器的 PGERR 位和 WPERR 位来判断哪一种错误发生了。下图显示了主存储块编程操作流程：



选项字节块的擦除（NVR）

FMC 提供了一个擦除功能用来初始化 FLASH 中 NVR（system block）存储空间，不包含(information block)，可选字节块擦除过程如下所示：

1. 确保 FMC_CTRL 寄存器不处于锁定状态；

2. 等待 FMC_STATE 寄存器的 BUSY 位变为 0；

3. 解锁 FMC_CTRL 寄存器的可选字节操作位（分别操作 FMC_KEY 与 FMC_OTPKEY 解锁）；
4. 等待 FMC_CTRL 寄存器的 OTPWRE 位置 1；
5. 置位 FMC_CTRL 寄存器的 OTPER 位；
6. 将待擦除 NVR 的绝对地址写到 FMC_ADDR 寄存器；
7. 通过将 FMC_CTRL 寄存器的 START 位置 1 来发送可选字节块擦除命令到 FMC；
8. 等待擦除指令执行完毕，FMC_STATE 寄存器的 BUSY 位清 0；
9. 如果需要，可通过 CPU 读验证是否擦除成功。

当可选字节块擦除成功执行，FMC_STATE 寄存器的 END 位置位。若 FMC_CTRL 寄存器的 ENDIE 位被置 1， FMC 将触发一个中断。

选项字节块的编程（NVR）

FMC 提供了一个 32 位整字编程功能，可用来修改可选字节块内容。可选字节块共有 8Bytes 可选字节。字节块编程操作过程如下：

1. 确保 FMC_CTRL 寄存器不处于锁定状态；
2. 等待 FMC_STATE 寄存器的 BUSY 位变为 0；
3. 解锁 FMC_CTRL 寄存器的可选字节操作位；
4. 等待 FMC_CTRL 寄存器的 OTPWRE 位置 1；
5. 置位 FMC_CTRL 寄存器的 OTPPG 位；
6. CPU 写一个 32 位整字到目的地址；
7. 等待编程指令执行完毕，FMC_STATE 寄存器的 BUSY 位清 0；
8. 如果需要，通过 CPU 读验证是否编程成功。

当可选字节块编程成功执行，FMC_STATE 寄存器的 END 位置位。若 FMC_CTRL 寄存器的 ENDIE 位被置 1， FMC 将触发一个中断。需要注意的是，执行整字/半字编程操作需要检查目的地址是否已经被擦除。如果该地址没有被擦除，对该地址写一个非 0x0 值，FMC_STATE 寄存器的 PGERR 位将被置 1，对该地址的编程操作无效（当写内容为 0x0 时，即使目的地址没有被正常擦除，也可以正确编程）。

保护

闪存中的用户代码区可以防止非法的读出；同样可以对闪存区的页加以保护，防止在程序跑飞的情况下被意外地改变，写保护的基本单位是：4 页。

1. 主存储体读保护

这项保护是通过设置 RDP 选项字节，然后在系统重新复位加载了新的 RDP 选项字节后启动的。

- 读保护开启：
  a. 只允许从用户代码中对主闪存存储器的读操作

b. 所有通过 SWD 向内置 SRAM 装载代码并执行代码的功能依然有效，亦可以通过 SWD 从内置 SRAM 启动，这个功能可以用来解除读保护

c. 当读保护的选项字节转变为存储器未保护的数值时，将会执行整片擦除过程

➢ 解除读保护的步骤：

a. 擦除整个选项字节区域，读保护码(RDP)将变为 0xFF，此时读保护仍然有效；

b. 写入正确的 nRDP(0x5A) 和 RDP(0xA5)，以解除存储器的保护，该操作将首先导致对所有用户闪存的整片擦除操作

c. 进行复位，以重新加载选项字节(和新的 RDP 代码)，此时读保护被解除

2. 主存储体写保护

写保护是以每 4 页为单位实现的。如果试图在一个受保护的页面进行编程或擦除操作，在闪存状态寄存器(FMC_STATE)中会返回一个保护错误标志。

配置选项字节 WRP[7:0]将设置写保护，随后的系统复位将加载新的 WRPx 选项字节。

所有页默认值都为写保护状态，同时第 0~3 页有独立的解除写保护操作，其它部分的存储器可以通过在主闪存存储器中执行的代码进行编程(实现 IAP 或数据存储等功能)，但不允许在调试模式下或在从内部 SRAM 启动后执行写或擦除操作(当前写保护有效时整片擦除也不能操作)。

解除写保护：

a. 擦除整个选项字节区域；

b. 重装载选项字节(包含新的 WRP[7:0]字节)；

c. 进行系统复位，复位读取新的保护状态，对应写保护被解除。

3. 选择字块写保护

默认状态下，选项字节块始终是可以读且被写保护。要想对选项字节块进行写操作(编程/擦除)首先要在 OTPKEY 中写入正确的键序列(与上锁时一样)，随后允许对选项字节块的写操作，FMC_CTRL 寄存器的 OTPWRE 位标示允许写，清除这位将禁止写操作。

4.13.3 System block（NVR 第一页）说明

选项字节共有 512 个字节，由用户根据应用的需要配置；例如：可以选择使用硬件模式的看门狗或软件的看门狗。

在选项字节中每个 32 位的字被划分为下述格式：

Bit[31:24]	Bit[23:16]	Bit[15:8]	Bit[7:0]
选择字节 1 的取反	选择字节0的取反	选择字节1	选择字节0

选择字节块必须 32 位字编程。Bit[31:16]是 Bit[15:0]取反，否则恢复默认值 0xFFFF_FFFF。

地址	Bit[31:24]	Bit[23:16]	Bit[15:8]	Bit[7:0]
0x201fc	nData1	nData0	Data1	Data0
0x201f8	nUSER	nRDP	USER	RDP

0x201f4	nWRP1	nWRP0	WRP1	WRP0
0x201f0	nWRP3	nWRP2	WRP3	WRP2
0x201ec	nWRP5	nWRP4	WRP5	WRP4
0x201e8	nWRP7	nWRP6	WRP7	WRP6

修调地址	BITS	信号	功能说明
18'h201fc	[7:0]	INFO_WR_PORTECT	information block 擦写保护使能。 0x88 ：正常擦写； others：无法擦写
18'h201f8	[7:0]	MAIN_READ_PORTECT	代码读出保护选项字节。 未保护状态（能够读取）：RDP=0xA5 已保护状态（不能读取）：RDP!=0xA5
18'h201f4	[7:0]	WDP1	WRP1：第 32~63 页的写保护
	[7:0]	WDP0	WRP0：第 0~31 页的写保护
18'h201f0	[7:0]	WDP3	WRP3：第 96~127 页的写保护
	[7:0]	WDP2	WRP2：第 64~95 页的写保护
18'h201ec	[7:0]	WDP5	WRP5：第 160~191 页的写保护
	[7:0]	WDP4	WRP4：第 128~159 页的写保护
18'h201e8	[7:0]	WDP7	WRP7：第 224~255 页的写保护
	[7:0]	WDP6	WRP6：第 192~223 页的写保护

每次系统复位后，选项字节装载器(OTP)读出信息块的数据，并保存在选项字节寄存器(FMC_OTB)中；每个选择位都在信息块中有它的反码位，在装载选择位时反码位用于验证选择位是否正确，如果有任何的差别，将产生一个选项字节错误标志(OPTERR)。当发生选项字节错误时，对应的选项字节被强置为 0xFF。当选项字节和它的反码均为 0xFF 时(擦除后的状态)，则关闭上述验证功能。

所有的选择位(不包括它们的反码位)用于配置该微控制器，CPU 可以读选项字节寄存器，详见寄存器说明。

注意：Flash 存储空间为 128K+1024byte，当 cpu 的地址大于 0x203ff 小于 0x1fff_ffff 时地址溢出。

4.14 RTC 模块介绍

BS32F030 的 RTC 外设（Real Time Clock），实质是一个掉电后还继续运行的定时器。从定时器的角度来说，相对于通用定时器 TIM 外设，它十分简单，只有很纯粹的计时和触发中断的功能；但若是芯片支持后备电源则具有掉电还能继续运行的特性。

上述的掉电，是指主电源 V_{DD} 断开的情况，为了 RTC 外设掉电继续运行，一般的开发板上可以接上后备电池给 BS32 的 RTC 通过后备电源供电。无论由什么电源供电，RTC 中的数据都保存在属于 RTC 的备份域中。备份域除了 RTC 模块的寄存器，还有 5 个 32 位的寄存器保存用户程序的数据，系统复位或电源复位时，这些数据可以提前存入到 Flash 中可以掉电保存，否则掉电将导致备份域中保存的所有数据丢失。

从 RTC 的定时器特性来说，它是一个 32 位的计数器，只能向上计数。它使用的时钟源有三种: 32 分频高速外部 HSE 时钟（HSE/4-16Mhz）、低速内部 LSI 时钟（LSI/32.768Khz）、低速外部 LSE 时钟（LSE/32.768Khz）；使用 HSE 或 LSI 的话，在主电源 V_{DD} 掉电的情况下，这两个时钟来源都会受到影响，因此无法保证 RTC 正常工作，所以 RTC 一般使用低速外部时钟 LSE，在设计中，频率通常为实时时钟模块中常用的 32.768Khz，这是因为 $32768=2^{15}$，分频容易实现，所以他被广泛应用到 RTC 模块。在主电源 V_{DD} 有效的情况下（待机），RTC 还可以配置闹钟事件使 BS32 退出待机模式。

主要特性：

1. RTC 寄存器具有写保护功能。
2. 日历具有亚秒、秒、分、小时（24 小时格式）。
3. 日历可初始化，可通过写 RTC 寄存器使 RTC 域复位。
4. RTC 计数时钟可选 LSI32KHA，LSE32K，32 分频的 HSE16M。
5. 可配置测试模式，使用 HSI1M 作为测试时钟。
6. 可配置 LSE32K 时钟错误检测，检测到错误后可出中断。
7. 一个可编程闹钟（计数范围内的任意时间）。
8. 可将闹钟匹配信号和 TAMP1/2 入侵事件输出到芯片管脚（三选一）。
9. 可将秒时钟、LSI32KHA、LSE32K 时钟输出到芯片管脚（三选一）。
10. 可检测 PC13/PA4/PA0 管脚上的外部入侵。
11. 5 个 32bit 备份寄存器，可用于保存数据，检测到入侵时可自动擦除（擦除使能可配）。
12. 闹钟，LSE32K 时钟错误，秒中断，天中断，TAMP1/2 入侵事件均可产生 RTC 中断。所有 RTC 中断都可以将器件从低功耗模式唤醒。

4.14.1 RTC 功能说明

BS32F030 的 RTC 外设主要有三大功能：时钟日历、闹钟配置、侵入检测。

RTC 在 VDD 电源有效时，RTC 可以触发秒中断与闹钟中断。若 BS32 原本处于待机状态，可由闹钟事件或 WKUP 事件（外部唤醒事件，属于 EXTI 模块，不属于 RTC）使它退出待机模式。

软件编程实现的时间戳，可通过程序转换输出实时时钟和日历的功能，修改 RTC_TR 寄存器的值可以重新设置系统当前的时间。其时钟配置系统在备份域，在系统复位或从待机模式唤醒后 RTC 的设置维持不变，且使用备份域电源可以保证 RTC 在主电源关闭的情况下仍然运行，保证时间的正确。

RTC 寄存器写保护

系统复位后，可通过电源控制外设 PMU 中的 RTC_DBP 位来保护 RTC 寄存器以防止非正常的写访问。必须将 DBP 位置 1 才能使能 RTC 寄存器的写访问。

在 DBP 位置 1 的前提下，通过向写保护寄存器 RTC_WPR 写入密钥来使能对受保护 RTC 寄存器的写操作。要解锁 RTC 寄存器的写保护，需要执行以下步骤：

1. 将 0xCA 写入 RTC_WPR 寄存器。
2. 将 0x53 写入 RTC_WPR 寄存器。

写入一个错误的关键字会再次激活写保护。

RTC 初始化配置

通过将 RTC_INITR 寄存器的 RTC_RST 位置位可将 RTC 域复位。

通过 RTC_INITR 寄存器的 RTCCLK_SEL[1:0]位可选择 RTC 的计数时钟：

- LSI32.768KHA（内部低速高精度时钟）
- LSE32.768K
- HSE16M 的 32 分频

通过 RTC_INITR 寄存器的 RTCCLK_OFF 位可关闭 RTC 时钟。

测试模式下，可通过 RTC_INITR 寄存器的 TEST_EN 位进入测试模式，固定使用 HSI1M 时钟作为测试时钟，测试模式下，日历的每一级计数器都将使用测试时钟进行计数，以加快测试速度。

RTC 日历

每一个 RTCCLK 周期，当前日历值将被同步到 RTC_TR 寄存器，同时 RTC_INITR 寄存器的 RSF 位被置 1，RSF 位可由软件写 0 清零。

➢ **日历初始化配置**

要配置初始日历值，需按照以下顺序操作：

1. 轮询 RTC_INITR 寄存器中的 INIT 位，确保 INIT 位为 0，避免在上一次初始化还未完成的情况下再次初始化。
2. 在 RTC_TR 寄存器中写入初始时间。
3. 向 INIT 位写 1，开始初始化。
4. 轮询 RTC_INITR 寄存器中的 INIT 位，当 INIT 清零时，表示初始化完成。

当初始化序列完成之后，日历从配置值开始计数。

➢ **读取日历**

硬件每次将日历值同步到 RTC_TR 寄存器时，RTC_INITR 寄存器中的 RSF 位都会被置 1。每一个 RTCCLK 周期执行一次复制。

从低功耗模式唤醒、系统复位或初始化之后，必须通过软件将 RSF 清零。之后，软件必须等待至 RSF 再次置 1 之后才可以读取 RTC_TR 寄存器。

RTC 可编程闹钟

可通过 RTC_CR 寄存器中的 ALARMA_EN 位来使能闹钟 A。注意，RTC_CR 寄存器的[4:0]位每个 RTCCLK 周期更新一次，并非实时更新，更改配置后，可通过回读寄存器值来判断是否成功更改配置。

如果日历值与闹钟寄存器 RTC_ALARMR 中编程的值相匹配，且闹钟使能打开，则 RTC_SR 寄存器中的 ALARMAF 标志会被置为 1。

可通过 RTC_CR 寄存器中的 ALARMA_IE 位来使能闹钟 A 中断。

要配置闹钟，必须执行以下步骤：

1. 将 RTC_CR 中的 ALARMA_EN 位清零以禁止闹钟 A(初次配置时不需要此步骤)；
2. 轮询 RTC_CR 中的 ALARMA_EN 位，直到其值为 0（最长需要一个 RTC_CLK_IN 周期）；
3. 编程闹钟寄存器 RTC_ALARMR，配置闹钟时间；
4. 将 RTC_CR 寄存器中的 ALARMA_EN 位置 1 以使能闹钟 A；
5. 轮询 RTC_CR 中的 ALARMA_EN 位，直到其值为 1（最长需要一个 RTC_CLK_IN 周期）。

TAMP 备份寄存器和入侵检测

TAMP 模块内含 5 个 32bit 的备份寄存器，用于用户存储关键信息。

TAMP 模块可检测 PC13/PA4/PA0 管脚的入侵（PC13/PA4 为入侵 1，PA0 为入侵 2），可通过 TAMP_CR 寄存器分别配置两个入侵的检测使能和中断使能。当检测到任意入侵事件时，RTC_SR 寄存器的对应标志位会被硬件拉高，软件可通过 RTC_SCR 寄存器写 1 清除对应标志位。可通过 TAMP_CR 寄存器分别配置检测到入侵时是否擦除备份寄存器。

RTC 输出选择

通过 RTC_CR 寄存器的 CLKOUT_SEL[1:0]位可选择将以下时钟输出到 IO：

1. 秒时钟
2. LSI32.768KHA
3. LSE32.768Khz

通过 RTC_CR 寄存器的 TAMPALARM_SEL[1:0]位可选择将以下事件输出到 IO：

1. 闹钟 A 匹配事件

2. TAMP1 入侵事件

3. TAMP2 入侵事件

RTC 中断和状态标志

➢ **中断**

1. **ALARMA_IE**：闹钟 A 中断使能，闹钟 A 使能打开且当前日历值与闹钟值匹配时，输出 RTC 中断。

2. **SECOND_IE**：秒中断使能，每次日历记到 1 秒时，输出 RTC 中断。

3. **DAY_IE**：天中断使能，每次日历记到 1 天时，输出 RTC 中断。

4. **LSE_ERR_IE**：LSE32.768K 错误中断使能，检测到错误时输出 RTC 中断。

5. **TAMP1_IE**：入侵 1 中断使能，每次检测到入侵事件 1 时，输出 RTC 中断。

6. **TAMP2_IE**：入侵 2 中断使能，每次检测到入侵事件 2 时，输出 RTC 中断。

➢ **状态标志**

ALARMAF：当闹钟 A 使能打开且日历值与闹钟值匹配时，此位置 1，软件通过 RTC_SCR 寄存器对应位写 1 清零。

SECONDF：使能秒中断且日历记满一秒时，此位置 1，软件通过 RTC_SCR 寄存器对应位写 1 清零。

DAYF：使能天中断且日历记满一天时，此位置 1，软件通过 RTC_SCR 寄存器对应位写 1 清零。

LSE32KERR：使能 LSE32K 振荡器错误检测且检测到错误时，此位置 1，软件通过 RTC_SCR 寄存器对应位写 1 清零。

TAMP1F：TAMP1 检测使能打开且在 TAMP1 输入上检测到入侵时，此位置 1，软件通过 RTC_SCR 寄存器对应位写 1 清零。

TAMP2F：TAMP2 检测使能打开且在 TAMP2 输入上检测到入侵时，此位置 1，软件通过 RTC_SCR 寄存器对应位写 1 清零。

RTC 低功耗唤醒

模式	说明
SLEEPING	不影响 RTC 工作，RTC 中断可使器件退出 SLEEPING 模式
SLEEPDEEP	不影响 RTC 工作，RTC 中断会使器件退出 SLEEPDEEP 模式
SHUTDOWN	不影响 RTC 工作，RTC 中断会使器件退出 SHUTDOWN 模式

4.14.2 RTC 应用流程

RTC 日历功能

1. 将电源模块 PMU 中的 RTC_DBP 位置 1 使能对 RTC 和备份寄存器的写访问；

2. 往 RTC_WPR 寄存器依次写入 0XCA，0X53，解锁对 RTC 寄存器的写保护；

3. 将初始化日历值写入 RTC_TR 寄存器；
4. 将 RTC_INITR 寄存器的 INIT 位置 1，开始初始化；
5. 轮询 INIT 位直至其清零，表示初始化完成，日历会从初始化值开始计数；
6. 可通过读取 RTC_TR 寄存器获取日历值。

闹钟功能

1. 按照上述日历初始化流程将日历初始化；
2. 如需输出中断，将 RTC_CR 寄存器的 ALARM_IE 位置 1，然后清零 RTC_INITR 寄存器的 RSF 位待其置 1 后，再回读 RTC_CR 寄存器，判断 ALARM_IE=1，表示闹钟中断使能成功打开；
3. 将 RTC_CR 寄存器的 ALARM_EN 位清零，然后清零 RSF 位待其置 1 后，再回读 RTC_CR 寄存器，判断 ALARM_EN=0，表示闹钟使能成功关闭（闹钟使能默认关闭，初次配置闹钟时可跳过此步骤）。每次更改闹钟配置时都需要先关闭使能；
4. 通过 RTC_ALARMR 寄存器配置闹钟值；
5. 将 RTC_CR 寄存器的 ALARM_EN 位置 1，然后清零 RSF 位待其置 1 后，再回读 RTC_CR 寄存器，判断 ALARM_EN=1，表示闹钟使能成功打开；
6. 当日历值与配置的闹钟值匹配时，将输出 RTC 中断，同时 RTC_SR 寄存器的 ALARMAF 位将拉高，需要软件将 RTC_SCR 对应位写 1 清零。

秒中断/天中断功能

1. 按照上述日历初始化流程将日历初始化；
2. 将 RTC_CR 寄存器的 SECOND_IE/DAY_IE 位置 1，然后清零 RSF 位待其置 1 后，再回读 RTC_CR 寄存器，判断 SECOND_IE/DAY_IE =1，表示秒中断/天中断使能成功打开；
3. 每次日历记满一秒/记满一天时，将输出 RTC 中断，同时 SECONDF/DAYF 标志位将拉高。

备份寄存器和入侵检测

1. 将需要备份的重要数据写入 TAMP_BKPR 寄存器；
2. 配置 TAMP_CR 寄存器，配置入侵检测使能，中断使能和擦除使能；
3. 清零 RSF 位待其置 1 后，再回读 TAMP_CR 寄存器，判断对应使能=1，表示使能成功打开；
4. 当检测到入侵事件时，将输出 TAMP 中断和拉高 RTC_SR 寄存器中对应的标志位，如果打开了擦除使能，五个备份寄存器的数据将被擦除。

4.14.3 RTC 使用注意事项

1. RTC_TR 寄存器写入的值是希望被初始化的值，读出的值为当前日历值。
2. 从低功耗模式退出，系统复位或初始化之后，必须通过软件将 RSF 清零，待其置 1 后再读取日历值，否则读取的日历值可能不正确。

3. 写 RTC_CR[4:0]/TAMP_CR 寄存器，以及配置 TEST_EN，都需要一个 RTCCLK 时钟周期才能生效，需要将 RSF 清零，待其置 1 后再回读寄存器判断是否操作成功。

4. 闹钟功能必须在闹钟使能关闭的情况下才能更改配置。

4.15 CRC 模块介绍

CRC（循环冗余校验）计算单元使用多项式发生器从一个 8 位/16 位/32 位的数据字中产生 CRC 码。它是利用除法及余数的原理来作错误侦测的。

在众多的应用中，基于 CRC 的技术还常用来验证数据传输或存储的完整性。根据功能安全标准的规定，这些技术提供了验证 Flash 完整性的方法。CRC 计算单元有助于在运行期间计算软件的签名，并将该签名与链接时生成并存储在指定存储单元的参考签名加以比较。

BS32F030 CRC 主要特性

- 3 种 CRC 多项式可选：
  - CRC-32 多项式：32’h04C11DB7（默认选择该多项式）（$X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^{8}+X^{7}+X^{5}+X^{4}+X^{2}+X+1$）
  - CRC-16 多项式：16’h1021
  - CRC-8 多项式：8’h07
- 可处理 8 位、16 位、32 位数据
- 输入数据可按字节、半字和字执行位反转
- 计算结果可按位反转
- 初始值、异或值可配置

4.15.1 CRC 功能说明

CRC 操作

CRC 计算单元具有单个 32 位读/写寄存器（CRC_DR）。它用于输入新数据（写访问）和保存之前 CRC 计算的结果（读访问）。

对数据寄存器的每个写操作都会对之前的 CRC 值（存储在 CRC_DR 中）和新值再做一次 CRC 计算。CRC 计算针对整个 32 位数据字或逐个字节完成，具体取决于数据的写入格式。

CRC_DR 寄存器可按字、半字和字节进行访问。对于其他寄存器，只允许进行 32 位访问。计算时间取决于数据宽度：

- 32 位数据需要 4 个 HCLK 时钟周期
- 16 位数据需要 2 个 HCLK 时钟周期
- 8 位数据需要 1 个 HCLK 时钟周期

可动态调整数据大小，从而能最大程度地减少给定字节数的写访问次数。例如：5 个字节进行 CRC 计算时，可先写入字，然后写入字节。

输入数据的顺序可反转，以管理各种数据存放方式（双字/单字/字节，大端/小端等等）。可对 8 位、16 位和 32 位数据执行反转操作，具体取决于 CRC_CR 寄存器的 REV_IN 位。

例如，输入数据 0x1A2B3C4D 在 CRC 计算中用作：

- 按字节执行位反转的 0x58D43CB2
- 按半字执行位反转的 0xD458B23C
- 按全字执行位反转的 0xB23CD458

通过将 CRC_CR 寄存器的 REV_OUT 位置 1 也可将输出数据反转。

该操作按位进行：例如，输出数据 0x11223344 将转换为 22CC4488。

使用 CRC_CR 寄存器的 RESET 控制位可用于复位 CRC 计算模块，装载 CRC 初始值。

可使用 CRC_INIT 寄存器对 CRC 初始值进行编程，对 CRC_INIT 寄存器进行写访问时会自动初始化 CRC_DR 寄存器。

4.15.2 CRC 应用建议

1. 配置 CRC_CR 寄存器的 RESET 位为 1，复位 CRC 计算模块，RESET 硬件清零，脉宽为一个系统时钟周期；
2. CRC 计算期间不能实时配置多项式选择寄存器，异或值寄存器，及位反转控制寄存器；
3. 配置流程：
   - 配置开启 CRC 模块时钟；
   - 配置相关寄存器（多项式选择/初始值/异或值/位反转）；
   - 配置 RESET，复位计算模块（第一次计算也必须配置该寄存器）；
   - 读标志位 CRC_CF 为 0 后，写数据寄存器 CRC_DR 开始计算 crc 校验结果；读标志位 CRC_CF 为 0，延迟一个系统时钟周期后可读出 crc 校验结果。
4. 注意事项：

   当字节或半字拼接成字进行 CRC 计算时，需按以下格式拼接：
   - {Byte1，Byte2，Byte3，Byte4}
   - {Halfword1，Halfword2}

附件

帮助文件目录

Keil 环境：
Keil5 安装包
BYD.BS32F0xx_DFP.1.0.0.pack
IAR 环境：
IAR 安装包
BS32F0xx_v1.0.0.exe
通用烧录环境：
ST-LINK 固件升级包：en.stsw-link007-v3.9.3
开发模板：
SDK：BS32F030XXx_LL_SDK_V1.0.0

版本历史

版次	日 期	修 改 内 容
V1.0.0	20221011	初版
V1.0.1	20221220	更新部分寄存器命名与新建工程内容