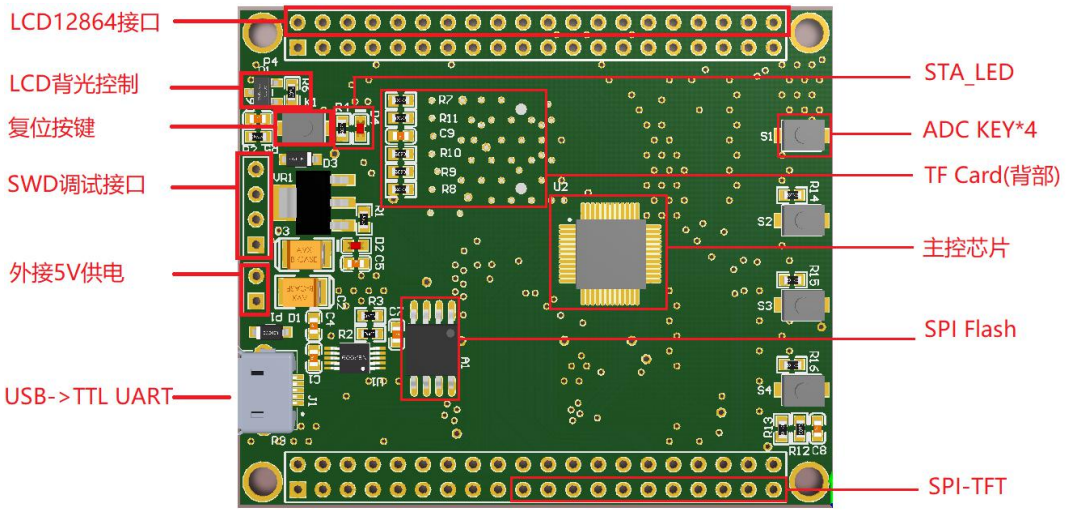
ITX-190 用户手册



日期　：2020 年 5 月 16 日

Vision　：V1.0

目录

1 开发板介绍

1.1 芯片及开发板资源介绍

ITX-190 使用的是华芯微特的 MCU，型号为 SWM190-CBT7，此芯片的主要配置如下:

- 内核
  - 32 位 ARM® Cortex™-M0 内核
  - 24 位系统定时器
  - 工作频率最高 60MHz
  - 硬件单周期乘法
  - 集成嵌套向量中断控制器（NVIC），提供最多 32 个、 4 级可配置优先级的中断
  - 通过 SWD 接口烧录
- SRAM 存储器 --> 20KB
- FLASH 存储器 --> 120KB
- 串行接口
  - UART*4，具有独立 8 字节 FIFO，最高支持主时钟 16 分频
  - SPI*2，具有 8 字节独立 FIFO，支持 SPI、 SSI 协议，支持 master/slave 模式
  - I2C*2， 支持 8 位、 10 位地址方式，支持 master/slave 模式
- PWM 控制模块
- 定时器模块
- DMA 模块
- 除法器模块
- 旋转坐标计算模块
- GPIO
- 模拟外设
  - 2 路 12 位 8 通道高精度 SAR ADC
  - 3 路模拟比较器
  - 4 路运算放大器
- 时钟源
  - 24MHz、 48MHz 精度可达 1%的片内时钟源
  - 支持片上 PLL，最高支持 60MHZ 时钟
  - 32KHZ 片内时钟源
  - 片外 2～32Mhz 片外晶振
  - 片外 32KHZ 时钟，供 RTC 使用

对于 ITX-190，并没有完全使用该芯片的所有功能，通过观察图 1，可以明显看到，其板上资源如下：

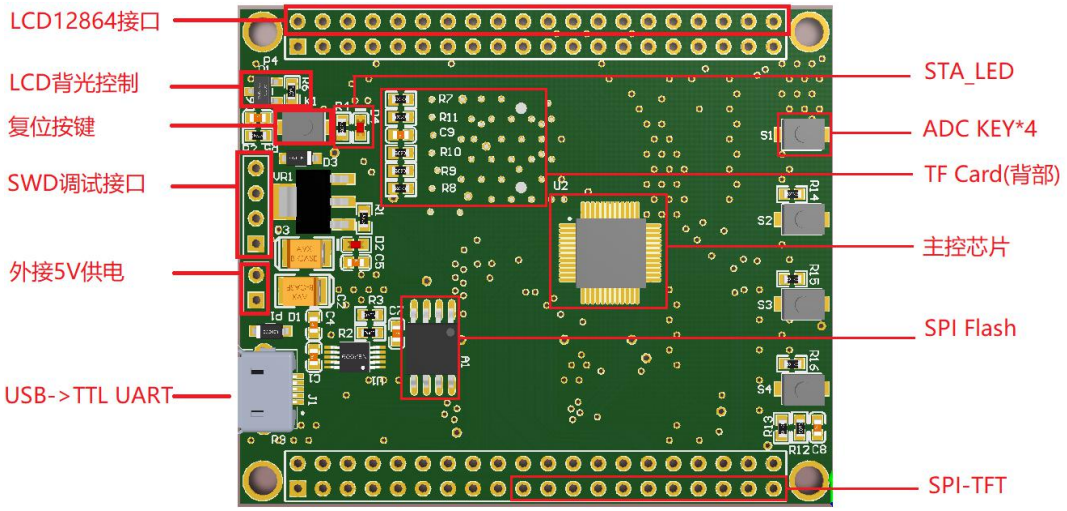


图 1-1 ITX-190 板载资源示意图

1.2 ITX-190 提供哪些资料

本开发板随板赠送资料如下:

(1)基本外设驱动——《初级用户手册》

①GPIO 驱动	:	点亮状态灯
②UART 通信	:	串口打印数据
③ADC 检测	:	ADC 实现按键检测
④SysTick Timer	:	提供 1ms 滴答
⑤Timer	:	定时器/基本定时器
⑥SPI 通信	:	读写 SPI Flash

(2)增值资料，板载资源实验——《中级用户手册》

①串口控制台	:	console
②文件下载协议	:	X-modem
③SPI Flash 存储字库	:	FontLib
④TF 卡——文件系统	:	FATFS
⑤LCD12864 显示	:	FrameBuffer
⑥迷你 UI 框架	:	ItxUI

(3)官方基本例程

1.2 软件的安装

1.2.1 软件包介绍

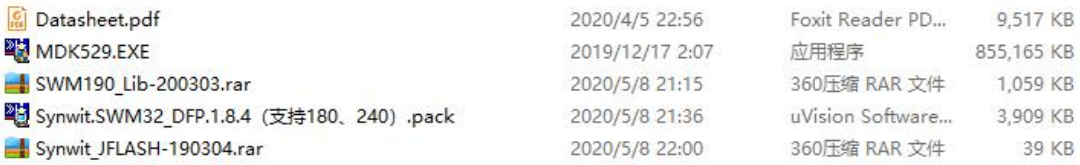


图 1-2 官方提供的文件

如果 1-2 所示，官方提供了一些开发相关的文档，分别是:

(1)Datasheet.pdf

SWM190 的芯片编程手册

(2)MDK529.EXE

编程软件 ARM-Keil-529

(3)SWM190_Lib-200303.rar

官方提供的 SWM190 例程

(4)Synwit.SWM32_DFP.1.8.4（支持 180、240）.pack

针对 SWM 芯片的 Keil 支持包

(5)Synwit_JFLASH-190304.rar

针对 SWM 芯片的 Keil 软件程序下载算法

(6)MDKCM525.EXE

Cortex-M0 架构支持包，不安装也能编译，但是可能会一直弹警告。

1.2.2 安装流程

(1)打开 MDK529.EXE，安装 Keil 的安装，并填写注册码，进行认证注册。

(2)打开 Synwit.SWM32_DFP.1.8.4，完成型号支持包的安装。

打开 MDKCM525.EXE，完成架构支持包的安装。

(3)解压 Synwit_JFLASH-190304.rar，将里面的 FLM 文件复制到 Keil 安装目录下的 /ARM/Flash 文件夹中。

(4)解压 SWM190_Lib-200303.rar，获得例程。

(5)随意使用一个例程，编译过后，选择 Debug 为相应的调试器，即可完成下载操作。

支持 ST-Link 以及 J-link 调试/下载程序。

1.2.3 官方工程介绍

打开一个工程，如果 1-3 所示。可以看到，官方的例程代码使用了一个公共文件夹，提供基础驱动库。所以在复制例程的时候，需要将库文件所在的文件夹也一起复制。

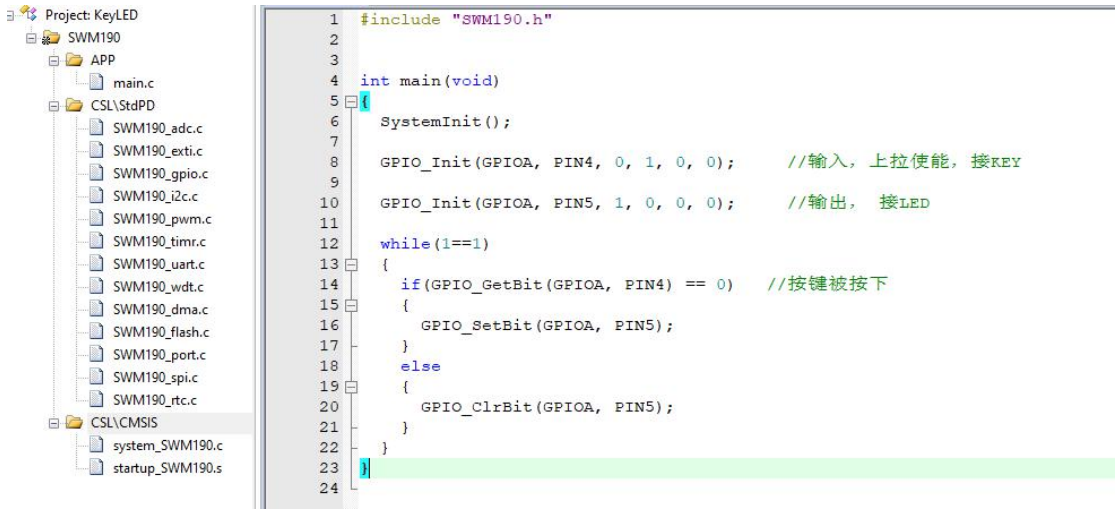


图 1-3　GPIO 例程展示

追踪函数 SystemInit() 的代码，如下方代码 Code-1-1 所示:

Code-1-1 时钟初始化函数 SystemInit()源码

```
#define SYS_CLK_24MHz                0        //0 内部高频 24MHz RC 振荡器
#define SYS_CLK_3MHz                 1        //1 内部高频 3MHz RC 振荡器
#define SYS_CLK_48MHz                2        //2 内部高频 48MHz RC 振荡器
#define SYS_CLK_6MHz                 3        //3 内部高频 6MHz RC 振荡器
#define SYS_CLK_XTAL                 4        //4 外部晶体振荡器（2-30MHz）
#define SYS_CLK_XTAL_DIV8            5        //5 外部晶体振荡器（2-30MHz） 8 分频
#define SYS_CLK_PLL                  6        //6 锁相环输出
#define SYS_CLK_PLL_DIV8             7        //7 锁相环输出 8 分频
#define SYS_CLK_32KHz                8        //8 内部低频 32KHz RC　振荡器
#define SYS_CLK_XTAL_32K             9        //9 外部低频 32KHz 晶体振荡器
#define SYS_CLK     SYS_CLK_48MHz
void SystemInit(void)
{
  SYS->CLKEN0 |= (1 << SYS_CLKEN0_ANAC_Pos);
  switch(SYS_CLK)
  {
      case SYS_CLK_24MHz:     switchTo24MHz();      break;
      case SYS_CLK_3MHz:      switchTo3MHz();       break;
      case SYS_CLK_48MHz:     switchTo48MHz();      break;
      case SYS_CLK_6MHz:      switchTo6MHz();       break;
      case SYS_CLK_XTAL:      switchToXTAL(0);      break;
      case SYS_CLK_XTAL_DIV8: switchToXTAL(1);      break;
      case SYS_CLK_PLL:           switchToPLL(0);       break;
      case SYS_CLK_PLL_DIV8:      switchToPLL(1);       break;
```

```
            case SYS_CLK_32KHz:      switchTo32KHz();        break;
            case SYS_CLK_XTAL_32K:   switchToXTAL_32K();     break;
        }
    SystemCoreClockUpdate();
    }
```

可以发现，系统时钟使用的是内部高频 48MHz RC 振荡器，速度为 48MHz。显然跟官宣的 60MHz 有差别。但是为了快速开发，可以先跳过这个环节。

我们观察手册，得知以下的时钟信息:

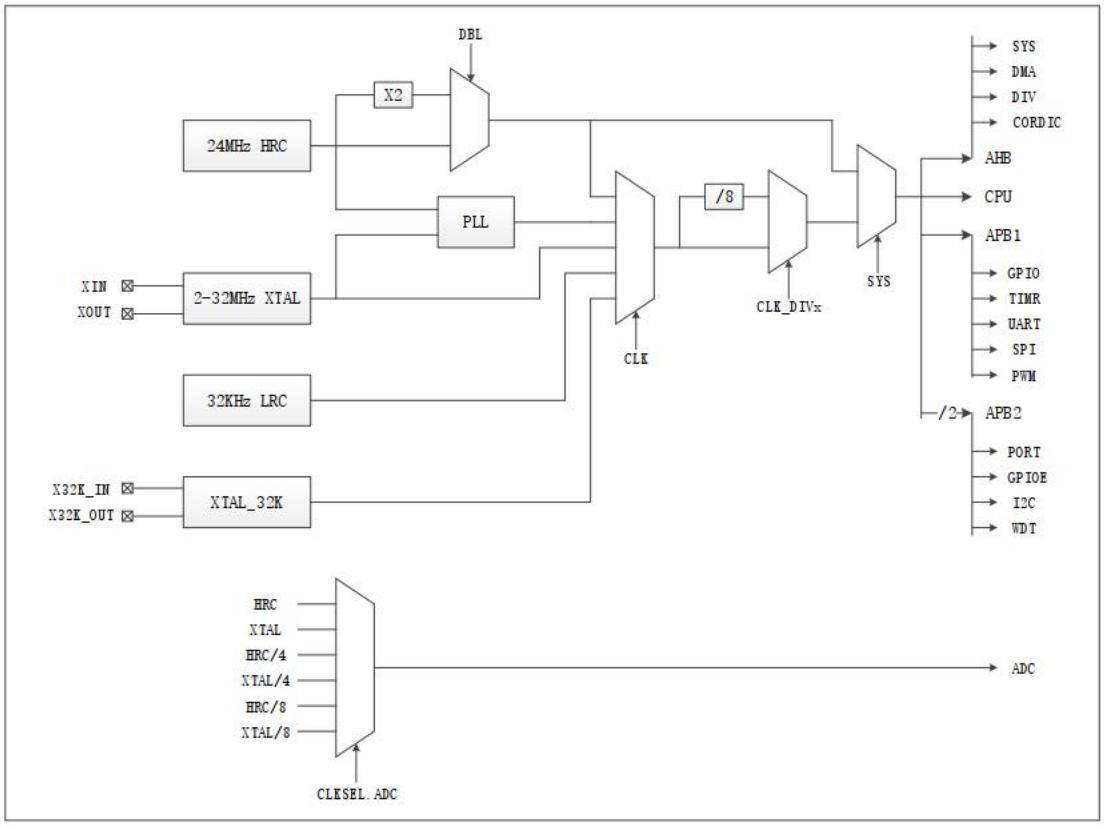


图 1-4 时钟树

注：

系统时钟 ->SYSCLK；

AHB 时钟 ->HCLK；

APB1 时钟 ->PCLK1；

APB2 时钟 ->PCLK2。

SYSCLK = HCLK = PCLK1 = 2* PCLK2。

例如：系统时钟 SYSCLK 选择的是 48MHz，那么 PCLK1=48 MHz， PCLK2=24 MHz。

对于本文档所有的例程，SYSCLK= 48MHz。

2　基本驱动

2.1　GPIO

2.1.1　使用官方驱动代码建立自己的工程

新建名为 Base 的文件夹，作为基本驱动的工作目录，然后从官方例程中复制图 2-1 所示的两个文件夹过来。

图 2-1　Base 工作文件夹示意图

其中,SWM190_StdPeriph_Driver 文件夹里存放的是官方驱动接口，GPIO 是相应的工程文件夹。

2.1.2 GPIO 操作函数

(1)初始化一个 GPIO

```
void GPIO_Init(
    GPIO_TypeDef * GPIOx,      指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n,                    指定 GPIO 引脚，PIN0~PIN15
    uint32_t dir,              引脚方向，0 输入          1 输出
    uint32_t pull_up,              上拉使能
    uint32_t pull_down,            下拉使能
    uint32_t open_drain            开漏使能
    );
```

(2)置位一个 GPIO

```
void GPIO_SetBit(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n                     指定 GPIO 引脚，PIN0~PIN15
    );
```

(3)清零一个 GPIO

```
void GPIO_ClrBit(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n                     指定 GPIO 引脚，PIN0~PIN15
    );
```

(4)翻转一个 GPIO 电平

```
    void GPIO_InvBit(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n                     指定 GPIO 引脚，PIN0~PIN15
    );
```

(5)获取一个 GPIO 的状态

```
uint32_t GPIO_GetBit(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n                     指定 GPIO 引脚，PIN0~PIN15
);
```

(6)连续置位多个 GPIO

```
void GPIO_SetBits(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n,                指定起始 GPIO 引脚
    uint32_t w                     指定结束 GPIO 引脚
);
```

(7)连续清零多个 GPIO

```
void GPIO_ClrBits(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n,                指定起始 GPIO 引脚
    uint32_t w                     指定结束 GPIO 引脚
);
```

(8)连续翻转多个 GPIO

```
void GPIO_InvBits(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n,                    指定起始 GPIO 引脚
    uint32_t w                     指定结束 GPIO 引脚
);
```

(9)获取连续多个 GPIO 状态

```
uint32_t GPIO_GetBits(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n,                    指定起始 GPIO 引脚
    uint32_t w                     指定结束 GPIO 引脚
    );
```

(10)原子操作-置位一个 GPIO

```
void GPIO_AtomicSetBit(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n                     指定 GPIO 引脚，PIN0~PIN15
    );
```

(11)原子操作-清零一个 GPIO

```
void GPIO_AtomicClrBit(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n                     指定 GPIO 引脚，PIN0~PIN15
    );
```

(12)原子操作-翻转一个 GPIO

```
void GPIO_AtomicInvBit(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n                     指定 GPIO 引脚，PIN0~PIN15
    );
```

(13)原子操作-置位多个连续的 GPIO

```
void GPIO_AtomicSetBits(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n,                    指定起始 GPIO 引脚
    uint32_t w                     指定结束 GPIO 引脚
    );
```

(14)原子操作-清零多个连续的 GPIO

```
void GPIO_AtomicClrBits(
    GPIO_TypeDef * GPIOx,          指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n,                    指定起始 GPIO 引脚
    uint32_t w                     指定结束 GPIO 引脚
    );
```

(15)原子操作-翻转多个连续的 GPIO

```
void GPIO_AtomicInvBits(
    GPIO_TypeDef * GPIOx,        指定 GPIO 端口，GPIOA/GPIOB/GPIOC/GPIOD/GPIOE
    uint32_t n,                  指定起始 GPIO 引脚
    uint32_t w                   指定结束 GPIO 引脚
);
```

实验 2-1-1:实现 GPIO 点亮 LED(GPB0),并闪烁

如何编写代码？使用官方提供的函数函数。代码如下:

```
#include "SWM190.h"
/*-------------------------------------------------------
实验现象:
位于 GPB0 上的 LED 灯闪烁
-------------------------------------------------------*/
int main(void)
{
    SystemInit();
    GPIO_Init(GPIOB, PIN0, 1, 1, 0, 0);          //输出， 接 LED
    while(1==1)
    {
        GPIO_SetBit(GPIOB, PIN0);                //输出高电平
        for(volatile unsigned int a=0;a<1000000;a++);
        GPIO_ClrBit(GPIOB, PIN0);                //输出低电平
        for(volatile unsigned int a=0;a<1000000;a++);
    }
}
```

2.2 UART

2.2.1 使用流程

查看原理图，观察串口所对应的引脚。在 ITX-190 开发板中，串口引脚的连接如下:

CH340	MCU
RX	UART0_TX(GPA1)
TX	UART0_RX(GPA0)

编程流程大致如下:

(1)在使用串口之前，初始化 IO 口的引脚功能。

(2)分配一个结构体变量——串口配置，并填充它

(3)使用配置函数，将串口配置的数据生效。

(4)使能串口。。

先从小节 2.2.2 看一看官方提供哪些函数，再进行具体编程操作。

2.2.2 UART 操作函数

(1)串口初始化函数

```
void UART_Init(
    UART_TypeDef * UARTx,                  //指定 UART，UART0/UART1/UART2/UART3
    UART_InitStructure * initStruct        //包含 UART 串口相关设定值的结构体
);
```

(2)UART 串口打开

```
void UART_Open(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(3)UART 串口关闭

```
void UART_Close(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(4)发送一个字节数据

```
void UART_WriteByte(
    UART_TypeDef * UARTx,                  //指定 UART，UART0/UART1/UART2/UART3
    uint8_t data                           //要发送的字节
);
```

(5)读取一个字节数据，并指出数据是否 Valid

```
uint32_t UART_ReadByte(
    UART_TypeDef * UARTx,                  //指定 UART，UART0/UART1/UART2/UART3
    uint32_t * data                        //接收到的数据
);
    返回: 0 无错误      UART_ERR_PARITY 奇偶校验错误
```

(6)判断发送是否正忙

```
uint32_t UART_IsTXBusy(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
```

```
);
    返回: 1 UART 正在发送数据      0 数据已发完
```

(7)判断接收 FIFO 是否为空

```
uint32_t UART_IsRXFIFOEmpty(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
    返回: 1 接收 FIFO 空      0 接收 FIFO 非空
```

(8)判断发送 FIFO 是否为满

```
uint32_t UART_IsTXFIFOFull(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
    返回: 1 发送 FIFO 满      0 发送 FIFO 不满
```

(9)设置波特率

```
void UART_SetBaudrate(
    UART_TypeDef * UARTx,               //指定 UART，UART0/UART1/UART2/UART3
    uint32_t baudrate                   //指定 UART 的波特率值
);
```

注意:不要在串口工作时更改波特率，使用此函数前请先调用 UART_Close()关闭串口

(10)查询波特率

```
uint32_t UART_GetBaudrate(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
    返回: 当前波特率值
```

(11)UART CTS 流控配置

```
void UART_CTSConfig(
    UART_TypeDef * UARTx,           //指定 UART，UART0/UART1/UART2/UART3
    uint32_t enable,                //1 使能 CTS 流控      0 禁止 CTS 流控
    uint32_t polarity               //0 CTS 输入为低表示可以发送数据
    //1 CTS 输入为高表示可以发送数据
);
```

(12)UART CTS 线当前状态

```
uint32_t UART_CTSLineState(
    UART_TypeDef * UARTx       //指定 UART，UART0/UART1/UART2/UART3
);
    返回: 0 CTS 线当前为低电平      1 CTS 线当前为高电平
```

(13)UART RTS 流控配置

```
void UART_RTSConfig(
    UART_TypeDef * UARTx,           //指定 UART，UART0/UART1/UART2/UART3
    uint32_t enable,                //1 使能 RTS 流控      0 禁止 RTS 流控
    uint32_t polarity,              //0 RTS 输出低表示可以接收数据
    //1 RTS 输出高表示可以接收数据
    uint32_t threshold              //RTS 流控的触发阈值
);
```

RTS 流控的触发阈值，可取值范围为：

UART_RTS_1BYTE、UART_RTS_2BYTE、UART_RTS_4BYTE、UART_RTS_6BYTE

(14)UART RTS 线当前状态

```
uint32_t UART_RTSLineState(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(15)UART LIN 功能配置

```
void UART_LINConfig(
    UART_TypeDef * UARTx,           //指定 UART，UART0/UART1/UART2/UART3
    uint32_t detectedIEn,           //检测到 Break 中断使能
    uint32_t generatedIEn           //Break 发送完成中断使能
);
```

(16)UART LIN 产生/发送 Break

```
void UART_LINGenerate(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(17)UART LIN 是否检测到 Break

```
uint32_t UART_LINIsDetected(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(18)UART LIN Break 是否发送完成

```
uint32_t UART_LINIsGenerated(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

返回: 1 LIN Break 发送完成 0 LIN Break 发送未完成

(19)UART 自动波特率检测开始

```
void UART_ABRStart(
    UART_TypeDef * UARTx,           //指定 UART，UART0/UART1/UART2/UART3
    uint32_t detectChar             //用于自动检测、计算波特率的检测字符
);
```

用于自动检测、计算波特率的检测字符:

①8 位数据时可取值：0xFF、0xFE、0xF8、0x80，分别表示:

发送方必须发送 0xFF、0xFE、0xF8、0x80

②9 位数据时可取值：0x1FF、0x1FE、0x1F8、0x180，分别表示:

发送方必须发送 0x1FF、0x1FE、0x1F8、0x180

注意事项: 自动波特率检测时不能开启奇偶校验

(20)判断 UART 自动波特率是否完成

```
uint32_t UART_ABRIsDone(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

返回:

```
0 未完成
UART_ABR_RES_OK 已完成，且成功
```

```
    UART_ABR_RES_ERR 已完成，但失败、出错
```

(21)使能 RX FIFO 中数据高位中断

当 RX FIFO 中数据个数 >= RXThreshold 时 触发中断

```
void UART_INTRXThresholdEn(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(22)禁用 RX FIFO 中数据高位中断

当 RX FIFO 中数据个数 >= RXThreshold 时 不触发中断

```
void UART_INTRXThresholdDis(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(23)查询 RX FIFO 中数据个数是否高位

查询 RX FIFO 中数据个数是否>RXThreshold

```
uint32_t UART_INTRXThresholdStat(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
    返回: 1 RX FIFO 中数据个数 >= RXThreshold        0 RX FIFO 中数据个数 < RXThreshold
```

(24)使能 TX FIFO 中数据低位中断

当 TX FIFO 中数据个数 <= TXThreshold 时 触发中断

```
void UART_INTTXThresholdEn(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(25)禁用 TX FIFO 中数据低位中断

当 TX FIFO 中数据个数 <= TXThreshold 时 不触发中断

```
void UART_INTTXThresholdDis(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(26)查询 TX FIFO 中数据是否低位

查询 TX FIFO 中数据个数是否 <= TXThreshold

```
uint32_t UART_INTTXThresholdStat(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(27)使能接收超时中断

串口数据接收超时的时候 触发中断

```
void UART_INTTimeoutEn(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(28)禁用接收超时中断

串口数据接收超时的时候 不触发中断

```
void UART_INTTimeoutDis(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(29)查询串口数据接收是否超时

是否发生了接收超时，即超过 TimeoutTime/(Baudrate/10) 秒没有在 RX 线上接收到数据时触发中断请求。

```
uint32_t UART_INTTimeoutStat(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(30)使能 FIFO 空/发送移位寄存器空中断

使能发送 FIFO 空且发送移位寄存器空中断

```
void UART_INTTXDoneEn(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(31)禁用 FIFO 空/发送移位寄存器空中断

禁用发送 FIFO 空且发送移位寄存器空中断

```
void UART_INTTXDoneDis(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

(32)查询 FIFO/发送移位寄存器是否为空

发送 FIFO 空且发送移位寄存器空中断状态

```
uint32_t UART_INTTXDoneStat(
    UART_TypeDef * UARTx            //指定 UART，UART0/UART1/UART2/UART3
);
```

实验 2-2-1 使用串口打印数据

编程流程下:

一、初始化串口

(1)在使用串口之前，初始化 IO 口的引脚功能。

```
PORT_Init(PORTA, PIN1, PORTA_PIN1_UART0_TX, 0);          //GPIOA.1 配置为 UART0 TXD
PORT_Init(PORTA, PIN0, PORTA_PIN0_UART0_RX, 1);          //GPIOA.0 配置为 UART0 RXD
```

(2)分配一个结构体变量——串口配置，并填充它

```
    UART_InitStructure UART_initStruct;
    /*设置波特率----------------------------------------------------*/
    UART_initStruct.Baudrate          = 115200;
    /*设置数据位数----------------------------------------------------*/
    UART_initStruct.DataBits          = UART_DATA_8BIT;
    /*设置奇偶校验----------------------------------------------------*/
    UART_initStruct.Parity            = UART_PARITY_NONE;
    /*设置停止位----------------------------------------------------*/
    UART_initStruct.StopBits          = UART_STOP_1BIT;
    /*设置接收 FIFO 高位提醒阈值---------------------------------*/
    UART_initStruct.RXThreshold       = 0;
    UART_initStruct.RXThresholdIEn    = 1;
    /*设置发送 FIFO 低位提醒阈值---------------------------------*/
```

```
UART_initStruct.TXThreshold                = 1;
UART_initStruct.TXThresholdIEn     = 0;
/*设置接收超时时间  单位:个(字符周期)----------*/
UART_initStruct.TimeoutTime        = 10;
/*是否使能字符接收超时中断----------*/
UART_initStruct.TimeoutIEn = 1;
```

(3)使用配置函数，将串口配置的数据生效。

```
UART_Init(UART0, &UART_initStruct);
```

(4)使能串口。

```
UART_Open(UART0);
```

二、编写相应的中断服务函数

串口中断里将接收到的数据，存储在缓冲区 UART_RXBuffer[]中。

```
void UART0_Handler(void)
{
    uint32_t chr;
    /*------------------------------------------------------------*/
    //接收类型中断:接收高位中断/接收超时中断
    if(UART_INTRXThresholdStat(UART0) || UART_INTTimeoutStat(UART0))
    {
        while(UART_IsRXFIFOEmpty(UART0) == 0)
        {//成立，说明 RX FIFO 非空
          if(UART_ReadByte(UART0, &chr) == 0)
          {
              if(UART_RXIndex < UART_RX_LEN)
              {
                  UART_RXBuffer[UART_RXIndex] = chr;
                  UART_RXIndex++;
              }
          }
        }
    }
}
```

三、编写程序读取缓冲区的数据

```
uint32_t UART_GetChars(char *data)
{
    uint32_t len = 0;
    if(UART_RXIndex != 0)
    {
      /*关中断--------------------------------------*/
      NVIC_DisableIRQ(UART0_IRQn);
      /*读数据--------------------------------------*/
      memcpy(data, UART_RXBuffer, UART_RX_LEN);
      len = UART_RXIndex;
      UART_RXIndex = 0;
      /*开中断--------------------------------------*/
      NVIC_EnableIRQ(UART0_IRQn);
    }
    return len;
}
```

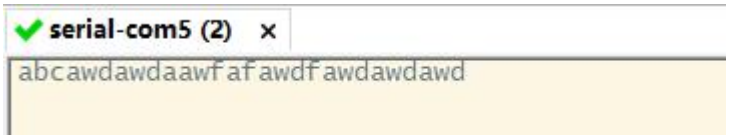
四、消耗接收到的数据

```
int main(void)
{
    uint32_t len, i;
    char buffer[UART_RX_LEN] = {0};
```

```
    SystemInit();
    SerialInit();
    while(1==1)
    {
      if((len = UART_GetChars(buffer)) != 0)
      {
          for(i = 0; i < len; i++) printf("%c", buffer[i]);
      }
    }
}
```

五、打开串口工具，观察现象。

现象:你发送什么数据，就会返回什么数据。



2.3　ADC

对于本开发板 ITX-190，将 ADC0 的 CH0(GPE4)外接了按键电路。如下方的图 2-3-1 所示

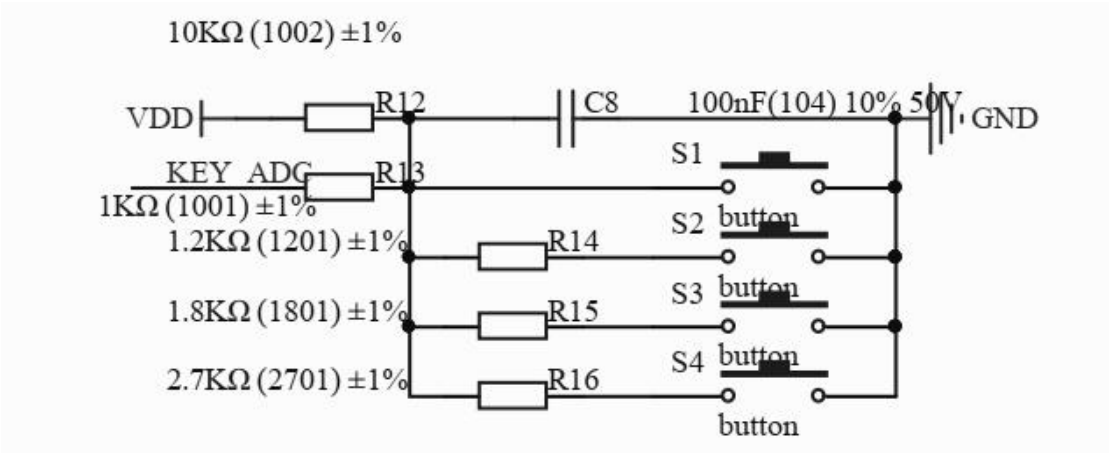


图 2-3-1　ADC 按键电路

2.3.1 ADC 按键开发流程

(1)算出各个按键按下时，ADC 应该是什么电压。

VDD = 3.3V

〇常态时,U(KEY_ADC) =　VDD　≈　3.3V

①按下 S1 时,U(KEY_ADC) = 0V

②按下 S2 时,U(KEY_ADC) = VDD/(R12+R14)*R14 = (3.3V)/(10K+1.2K)*1.2K　≈　0.353V

③按下 S3 时,U(KEY_ADC) = VDD/(R12+R15)*R15 = (3.3V)/(10K+1.8K)*1.8K　≈　0.503V

③按下 S4 时,U(KEY_ADC) = VDD/(R12+R16)*R16 = (3.3V)/(10K+2.7K)*2.7K　≈　0.702V

(2)初始化 ADC，并读取 ADC 线上的数字信号值

(3)量化数据

a, 将 ADC 读出来的 数字信号值 转化为电压值;

b, 或者将按键的不通状态的电压转化为 数字信号值 。

(4)判断当前 ADC 引脚上的按键处于什么状态

具体实现代码先不理会，先看看官方提供了什么驱动函数。

2.3.2 ADC 操作函数

(1)ADC 模数转换器初始化

```
void ADC_Init(
    ADC_TypeDef * ADCx,              //指定设置的 ADC，ADC0/ADC1
    ADC_InitStructure * initStruct   //包含 ADC 各相关定值的结构体
    );
```

(2)使能 ADC

```
void ADC_Open(
    ADC_TypeDef * ADCx,              //指定设置的 ADC，ADC0/ADC1
    );
```

(3)禁用 ADC

```
void ADC_Close(
    ADC_TypeDef * ADCx,              //指定设置的 ADC，ADC0/ADC1
```

```
    );
```

(4)启动 ADC 转换

```
void ADC_Start(
    ADC_TypeDef * ADCx,             //指定设置的 ADC，ADC0/ADC1
    );
```

(5)停止 ADC 转换

```
void ADC_Stop(
    ADC_TypeDef * ADCx,             //指定设置的 ADC，ADC0/ADC1
    );
```

(6)读取 ADC

```
uint32_t ADC_Read(
    ADC_TypeDef * ADCx,             //指定设置的 ADC，ADC0/ADC1
    uint32_t chn                    //指定通道号,ADC_CH0~ADC_CH7
    );
```

返回: 读取到的转换结果

(7)查询通道是否 End Of Conversion

```
uint32_t ADC_IsEOC(
    ADC_TypeDef * ADCx,             //指定设置的 ADC，ADC0/ADC1
    uint32_t chn                    //指定通道号,ADC_CH0~ADC_CH7
    );
```

返回: 1 该通道完成了转换 0 该通道未完成转换

(8)选通 ADC 通道

ADC 通道选通，模数转换会在选通的通道上依次采样转换

```
void ADC_ChnSelect(
    ADC_TypeDef * ADCx,             //指定设置的 ADC，ADC0/ADC1
    uint32_t chns                   //指定通道号,ADC_CH0~ADC_CH7
    );
```

(9)使能 ADC 转换完成中断请求

```
void ADC_IntEOCEn(
    ADC_TypeDef * ADCx,             //指定设置的 ADC，ADC0/ADC1
    uint32_t chns                   //指定通道号,ADC_CH0~ADC_CH7
    );
```

(10)禁用 ADC 转换完成中断请求

```
void ADC_IntEOCDis(
    ADC_TypeDef * ADCx,             //指定设置的 ADC，ADC0/ADC1
    uint32_t chns                   //指定通道号,ADC_CH0~ADC_CH7
    );
```

(11)清除 ADC 转换完成中断标志

```
void ADC_IntEOCClr(
    ADC_TypeDef * ADCx,             //指定设置的 ADC，ADC0/ADC1
    uint32_t chns                   //指定通道号,ADC_CH0~ADC_CH7
    );
```

(12)查询 ADC 是否转换完成

```
uint32_t ADC_IntEOCStat(
    ADC_TypeDef * ADCx,          //指定设置的 ADC，ADC0/ADC1
    uint32_t chns                //指定通道号,ADC_CH0~ADC_CH7
    );
```

(13)使能数据溢出中断请求

```
void ADC_IntOVFEn(
    ADC_TypeDef * ADCx,          //指定设置的 ADC，ADC0/ADC1
    uint32_t chns                //指定通道号,ADC_CH0~ADC_CH7
    );
```

(14)禁用数据溢出中断请求

```
void ADC_IntOVFDis(
    ADC_TypeDef * ADCx,          //指定设置的 ADC，ADC0/ADC1
    uint32_t chns                //指定通道号,ADC_CH0~ADC_CH7
    );
```

(15)清除数据溢出中断标志

```
void ADC_IntOVFClr(
    ADC_TypeDef * ADCx,          //指定设置的 ADC，ADC0/ADC1
    uint32_t chns                //指定通道号,ADC_CH0~ADC_CH7
    );
```

(16)查询数据溢出中断标志的状态

```
uint32_t ADC_IntOVFStat(
    ADC_TypeDef * ADCx,          //指定设置的 ADC，ADC0/ADC1
    uint32_t chns                //指定通道号,ADC_CH0~ADC_CH7
    );
```

返回:1 该通道完成了转换　　0 该通道未完成转换

注意:对于 FIFO 的使用状态，存在多种中断触发方式，但是提供的函数中并没有相关函数，深度使用需要自行查阅数据手册。

实验 2-3-1 ADC-KEY 的检测

实验要求:检查 ADC-KEY 的状态，并将其状态发送到串口助手显示。

从原理图上面，可以得知，ADC-KEY 接在 GPE4 引脚上，该引脚对应 ADC0 的通道 0。具体的开发流程如下:

(1)算出各个按键按下时，ADC 应该是什么电压。

VDD = 3.3V

〇常态时,U(KEY_ADC) = VDD ≈ 3.3V

①按下 S1 时,U(KEY_ADC) = 0V

②按下 S2 时,U(KEY_ADC) = VDD/(R12+R14)*R14 = (3.3V)/(10K+1.2K)*1.2K ≈ 0.353V

③按下 S3 时,U(KEY_ADC) = VDD/(R12+R15)*R15 = (3.3V)/(10K+1.8K)*1.8K ≈ 0.503V

③按下 S4 时,U(KEY_ADC) = VDD/(R12+R16)*R16 = (3.3V)/(10K+2.7K)*2.7K ≈ 0.702V

(2)初始化 ADC，并读取 ADC 线上的数字信号值

a, 将 GPE4 设置为 ADC0-CH0

```
PORT_Init(PORTE, PIN4,  PORTE_PIN4_ADC0_IN0,  0);        //PE.4  => ADC0.CH0
```

b, 分配一个 ADC 参数结构体变量，并设置它。怎么配置呢?看着图 2-3-2，ADC 的功能框图配置。

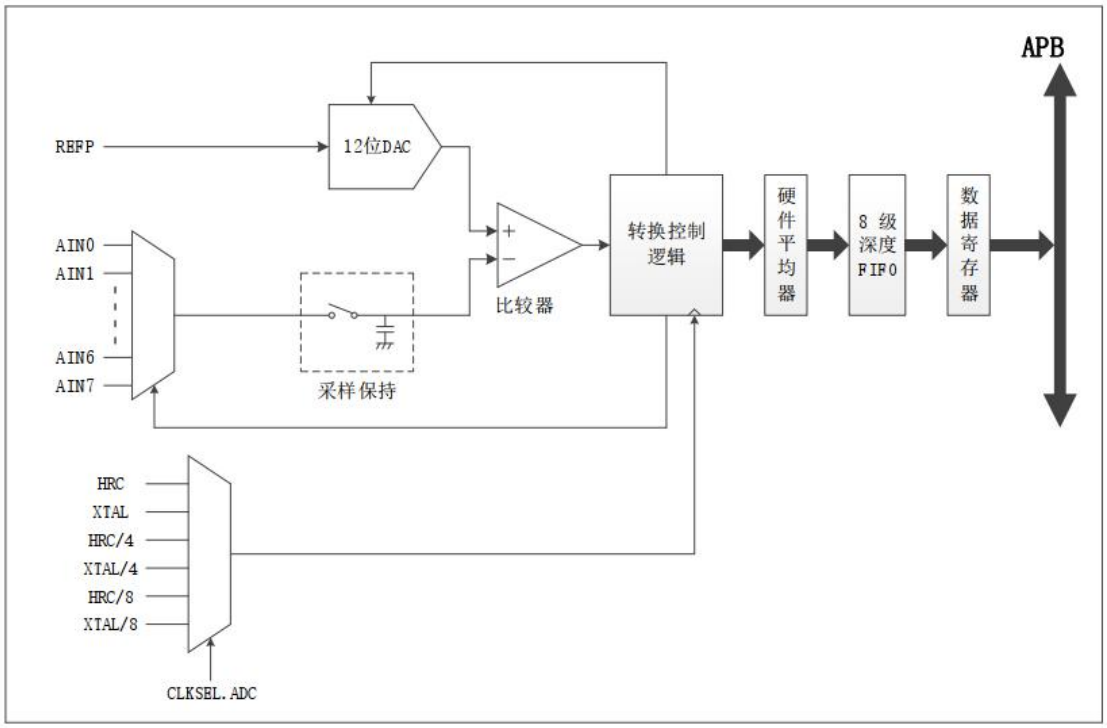


图 2-3-2 ADC 功能框图

```
/*分配一个 ADC 参数结构体变量*/
ADC_InitStructure ADC_initStruct;
/*选择 ADC 的时钟源*/
ADC_initStruct.clk_src          = ADC_CLKSRC_HRC_DIV8;
/*选择当前工作通道*/
ADC_initStruct.channels         = ADC_CH0;
/*配置硬件平均器,多少次采样输出一个平均值*/
```

```
ADC_initStruct.samplAvg       = ADC_AVG_SAMPLE2;
/*配置 ADC 触发方式,可以是软件触发/PWM 触发/定时器触发...*/
ADC_initStruct.trig_src       = ADC_TRIGGER_SW;
/*配置 ADC 触发是否为连续模式*/
ADC_initStruct.Continue       = 0;                    //非连续模式，即单次模式
/*配置 ADC 转换完成是否请求中断*/
ADC_initStruct.EOC_IEn        = 0;
/*配置 ADC FIFO 溢出是否请求中断*/
ADC_initStruct.OVF_IEn        = 0;
/*应用这些配置*/
ADC_Init(ADC0, &ADC_initStruct);                      //配置 ADC
/*使能 ADC*/
ADC_Open(ADC0);                                       //使能 ADC
```

c，在 main 函数中读取该路 ADC 的值，打印出来

```
while(1==1)
{
    /*开始转换*/
    ADC_Start(ADC0);
    /*阻塞等待转换完成*/
    while((ADC0->CH[0].STAT & 0x01) == 0);
    /*将数据贴出来*/
    printf("ADC0->CH0 %4d \r\n", ADC_Read(ADC0, ADC_CH0));
    /*延时一下,防止打印太多数据*/
    for(volatile unsigned int i=0;i<500000;i++);
}
```

(3)单独按下一个按键，观察串口数据

①按下 s1，情况如下:

```
ADC0->CH0     0
ADC0->CH0     0
```

②按下 s2，情况如下:

```
ADC0->CH0   431
ADC0->CH0   431
```

③按下 s3，情况如下:

```
ADC0->CH0   619
ADC0->CH0   619
```

④按下 s4，情况如下:

```
ADC0->CH0   864
ADC0->CH0   863
```

(4)使用按键组合按下，观察串口数据

因为 S1，没有串联按键偏置电阻。所以，S1 无论与任何按键组合，都是 0V。

使用组合键，必须给电阻串联按键偏置电阻。

①S2+S3，情况如下:

```
ADC0->CH0   269
ADC0->CH0   269
```

②S2+S4，情况如下:

```
ADC0->CH0   308
ADC0->CH0   307
```

③S3+S4，情况如下:

```
ADC0->CH0   393
ADC0->CH0   393
```

2.4 SysTick Timer

Cortex-M0 提供了一个系统滴答定时器。这个定时器是芯片架构内置，理论上，所有 Cortex-M0 系列的芯片都内置这样的一个定时器，所以，这个定时器在 Cortex-M0 系列芯片中，与芯片厂家及芯片具体型号无关。某些深入定制的厂商，会存在一定的存储器组织映射差异。

值得一提的是，这个 SysTick 定时器操作函数，并不在华芯微特提供的驱动代码库中，这是芯片架构的外设，对它的操作函数应该在架构相关的文件里。

通过遍历 Cortex-M0 的相关文档，发现对其的操作，位于 core_cm0.h 文件中，其代码如下 code-2-4-1 所示:

code-2-4-1 SysTick 配置函数

```
/** \brief  System Tick Configuration
    The function initializes the System Timer and its interrupt, and
starts the System Tick Timer.
    Counter is in free running mode to generate periodic interrupts.
    \param [in]  ticks  Number of ticks between two interrupts.
    \return          0  Function succeeded.
    \return          1  Function failed.
    \note     When the variable <b>__Vendor_SysTickConfig</b> is set to
1, then the
    function <b>SysTick_Config</b> is not included. In this case, the
file <b><i>device</i>.h</b>
    must contain a vendor-specific implementation of this function.
 */
__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)
{
   /* Reload value impossible */
   if ((ticks - 1) > SysTick_LOAD_RELOAD_Msk)  return (1);
   /* set reload register */
   SysTick->LOAD  = ticks - 1;
   /* set Priority for Systick Interrupt */
   NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
   /* Load the SysTick Counter Value */
   SysTick->VAL   = 0;
   /* Enable SysTick IRQ and SysTick Timer */
   SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                    SysTick_CTRL_TICKINT_Msk   |
                    SysTick_CTRL_ENABLE_Msk;
   /* Function successful */
   return (0);
}
```

2.4.1 SysTick 使用流程

我们大致猜测使用流程，应该就是:
①包含这个 core_cm0.h 头文件，
②调用这个__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)函数，对 SysTick 进行初始化。
③定义一个 SysTick 的中断服务函数。

这样就可以通过 SysTick 给整个项目提供一个时间基准。

既然是定时器,它肯定需要有一个时钟源,看下图图 2-4-1,可以得知 SysTick 的时钟源是 HCLK。

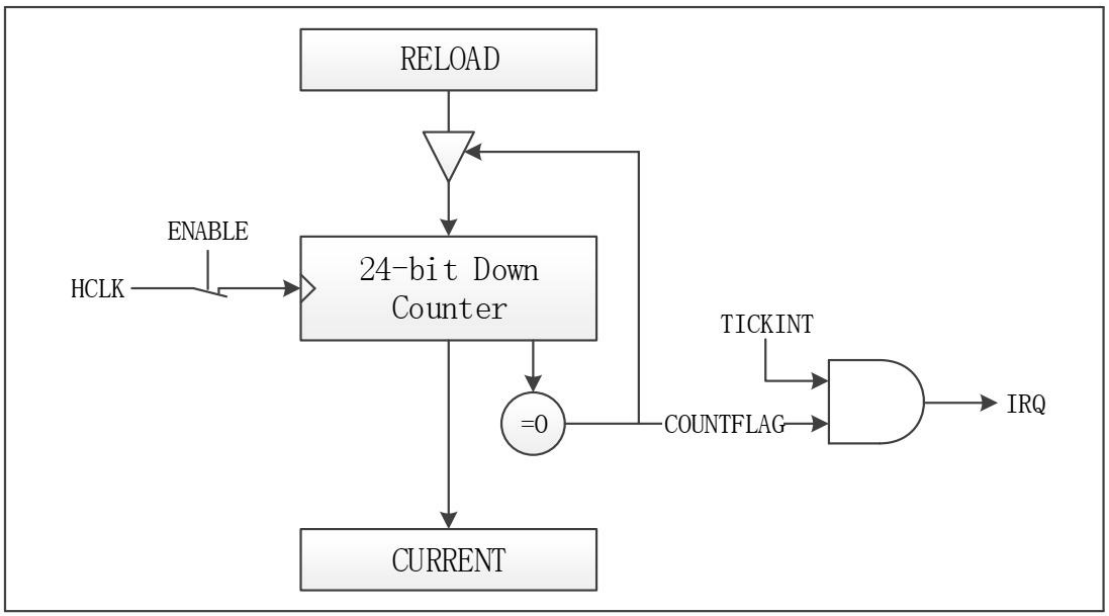


图 2-4-1 SysTick 时钟源

对于 SWM190，芯片工作于 48MHz 时，所以，HCLK=48MHz。

周期=1/频率。可以得知一个 Tick 周期是 1/48us,也就是说 48 个 Tick 是 1us。我们想得到 1ms 的定时周期，应该要设置的定时器引发中断的周期是 1000*48=48000 个 Tick 周期。

从手册上，我们可以得知，SysTick 定时器是一个自减的定时器。所以我们可以直接设置它的初值为 48000，即可获取 1ms 的时间基准。
代码如下图 2-4-2 所示:

```
unsigned int gSysTick=0;
void SerialInit(void);
int main(void)
{
  SystemInit();
  SerialInit();
  /*------------------------------------------------------------------------*/
  SysTick_Config(48000);
  while(1){
    NVIC_DisableIRQ(SysTick_IRQn);
    printf("gSysTick = %d \r\n",gSysTick);
    NVIC_EnableIRQ(SysTick_IRQn);
  }
}
/*系统滴答中断服务函数*/
void SysTick_Handler(void)
{
  gSysTick++;
}

/*串口初始化函数*/
void SerialInit(void)
{
  UART_InitStructure UART_initStruct;
  PORT_Init(PORTA, PIN1, PORTA_PIN1_UART0_TX, 0);    //GPIOA.1配置为UART0 TXD
  PORT_Init(PORTA, PIN0, PORTA_PIN0_UART0_RX, 1);    //GPIOA.0配置为UART0 RXD
  UART_initStruct.Baudrate = 115200;
  UART_initStruct.DataBits = UART_DATA_8BIT;
  UART_initStruct.Parity = UART_PARITY_NONE;
  UART_initStruct.StopBits = UART_STOP_1BIT;
  UART_initStruct.RXThresholdIEn = 0;
  UART_initStruct.TXThresholdIEn = 0;
  UART_initStruct.TimeoutIEn = 0;
  UART_Init(UART0, &UART_initStruct);
  UART_Open(UART0);
}
```

图 2-4-2 使用 SysTick 定时器示例代码

观察串口输出，要看到上面疯狂打印信息，如图 2-4-3 所示:

```
gSysTick = 285093
gSysTick = 285095
gSysTick = 285097
gSysTick = 285099
gSysTick = 285101
gSysTick = 285103
gSysTick = 285105
gSysTick = 285107
gSysTick = 285108
gSysTick = 285110
gSysTick = 285112
gSysTick = 285114
gSysTick = 285116
gSysTick = 285118
gSysTick = 285120
gSysTick = 285122
gSysTick = 285124
gSysTick = 285126
gSysTick = 285128
gSysTick = 285130
gSysTick = 285132
gSysTick = 285134
gSysTick = 285136
```

图 2-4-3 系统滴答值打印

2.5 Timer

2.5.1 概述

通过选型指南表(图 2-5-1)可以得知:ITX-190，所使用的芯片 SWM190-CBT7，内置的通用定时器个数为 8+1 个。

Part Number	Voltage(V)	Flash(KB)	SRAM(KB)	I/O	RTC	WDT	TIME	PWM	DMA	UART	I2C	SPI	SARADC	OPA	CMP	BOD	CORDIC	DIVIDER	Package
SWM190RBT6-50	2.3~3.6	120	20	51	1	1	8+1	16	4	4	2	2	2(15)	4	3	1	1	1	LQFP64
SWM190CBT7-50	2.3~3.6	120	20	38	1	1	8+1	15	4	4	1	2	2(13)	4	3	1	1	1	LQFP48
SWM190K6T7-80	2.3~3.6	32	4	26	1	1	8+1	15	4	4	1	2	2(10)	1	3	1	1	1	LQFP32

图 2-5-1 选型指南表

具体是什么样的情况呢?观察数据手册。

通过阅读手册，可以得知，除了 SysTick 之外，SWM190-CBT7 还内置了 2 种定时器，分别是:

(1)BTIMER *4 :

- 4 路 24 位通用定时器
- 每路均具备独立 8 位预分频
- 独立中断源

(2)TIMER *4 :

- 4 路 32 位通用定时器
- 可单独配置计时触发条件为内部时钟或者外部输入
- 支持脉冲捕获及宽度测量，检测脉冲极性可配
- 支持脉冲发送功能，可作为 PWM 使用
- TIMER0 支持 HALL 功能，可采集霍尔传感器角度
- TIMER2、 TIMER3 支持 ADC 采样触发功能

2.5.2 基本定时器 BTIMER

从手册获取到功能框图:

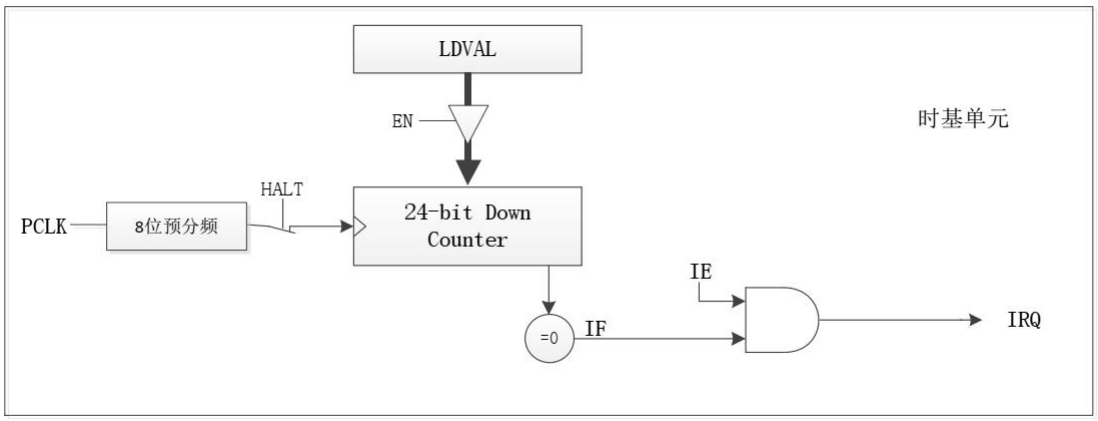


图 2-5-2-1 BTIMER 模块结构框图

从图 2-5-2-1 可以得知

(1)时钟源:PCLK

由于系统存在 2 个 PCLK，分别是 PCLK1、PCLK2，所以需要看看是具体是哪一个时钟。

回头浏览，“1.2.3 官方工程介绍”，可以发现，TIMER 使用的是 PCLK1，也就是说，TIMER 的输入时钟为 48MHz。

(2)从手册上得知，每一个 BTIMER 内部核心是一个 24 位的自减型计数器，计数器的初值来源是 LDVALx 寄存器。

计数器的时钟源是 PCLK1/分频器。

在配置这个寄存器的时候，可以通过向装载位（RELOAD）写 1，

BTIMERx 将立刻重新装载改变值并计数（RELAOD 位硬件自动清 0）。

(3)当计数器 x 自减为 0 时，会置位 IFx。可以通过配置 IEx，来配置其是否允许其产生中断请求。

(4)IFx 位清零方式为“写 1 清零”。

(5)对于高精度计算时，可在计数过程中，可通过对当前值寄存器（CVALx）进行读取，获取当前计数值。

本想在此列出官方提供的驱动函数，但是瞄了一眼，发现官方将高级定时器和普通定时器混在一起写了，因此先不列出定时器的函数。

2.5.3 增强定时器 TIMER

从数据手册上面获取增强定时器的功能框图，如图 2-5-3-1 所示。

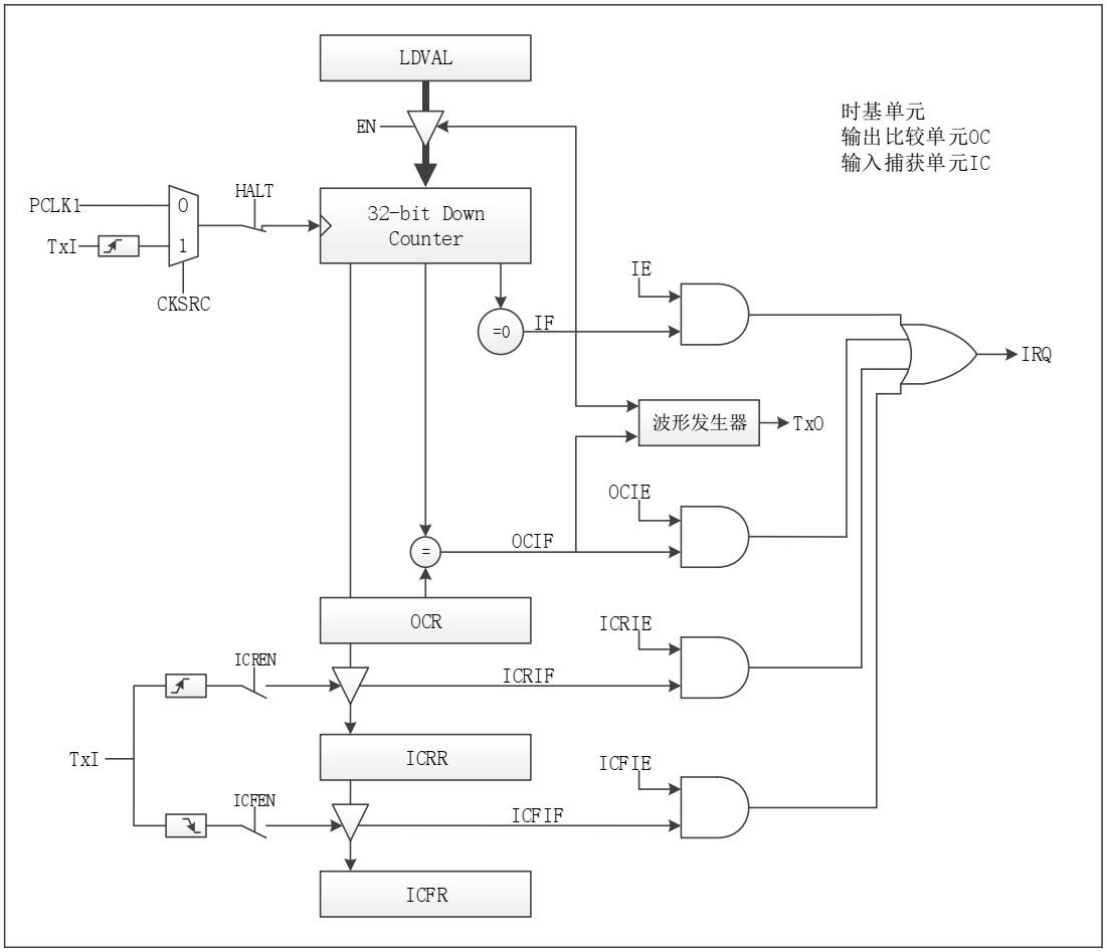


图 2-5-3-1 增强定时器功能框图

高级定时器就功能多了，看起来还有点麻烦，但是不要慌乱，一步步看，只要知道了基本用法就 OK 了。

(1)时钟源，可以是:

①PCLK1 ==>48MHz

②TxI ==>外部时钟输入

(2)从手册上可以得知，高级定时器有这么多功能:

①定时器

②计数器

③级联

④脉冲发送

⑤脉冲捕捉

⑥霍尔接口

⑦ADC 采样触发功能

看起来挺复杂的，这里只描述一下基本功能，其余请看官方手册或者代码，其实手册上面已经描述得很清楚了😄。

2.5.4 定时器操作函数

(1)某个定时器/计数器初始化

```
函数原型:void TIMR_Init(TIMR_TypeDef * TIMRx, uint32_t mode, uint32_t
period, uint32_t int_en);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
②uint32_t mode
    a,对于 TIMR0~3，它的值可以是:
      TIMR_MODE_TIMER    -->定时器模式
      TIMR_MODE_COUNTER      -->计数器模式
      TIMR_MODE_OC       -->比较模式
      TIMR_MODE_IC       -->捕获模式
    b,对于 BTIMR0~3,只能是
      TIMR_MODE_TIMER    -->定时器模式
③uint32_t period
    a,对于 TIMR0~3:  定时/计数周期，32 位
    b,对于 BTIMR0~3: period[23:0]为定时周期,period[31:24]为预分频值
④uint32_t int_en
    是否使能中断
返回值 无
```

(2)启动某个定时器

```
函数原型:void TIMR_Start(TIMR_TypeDef * TIMRx);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(3)停止某个定时器

```
函数原型:void TIMR_Stop(TIMR_TypeDef * TIMRx);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(4)暂停某个定时器

```
函数原型:void TIMR_Halt(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(5)恢复某个被暂停的定时器

```
函数原型:void TIMR_Resume(TIMR_TypeDef * TIMRx);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(6)设置定时/计数周期

```
函数原型:void TIMR_SetPeriod(TIMR_TypeDef * TIMRx, uint32_t period)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
②uint32_t period
    a,对于 TIMR0~3:  定时/计数周期，32 位
    b,对于 BTIMR0~3: period[23:0]为定时周期,period[31:24]为预分频值
返回值 无
```

(7)获取定时器/计数器周期

```
函数原型:uint32_t TIMR_GetPeriod(TIMR_TypeDef * TIMRx);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 uint32 类型
      a,对于 TIMR0~3:  定时/计数周期，32 位
      b,对于 BTIMR0~3: period[23:0]为定时周期,period[31:24]为预分频值
```

(8)获取某个定时器当前计数值

```
函数原型:uint32_t TIMR_GetCurValue(TIMR_TypeDef * TIMRx);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 uint32_t 类型 当前计数值
```

(9)使能某个定时器中断请求

```
函数原型:void TIMR_INTEn(TIMR_TypeDef * TIMRx);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(10)禁用某个定时器的中断请求

```
函数原型:void TIMR_INTDis(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(11)清除某个定时器的中断标志

```
函数原型:void TIMR_INTClr(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(12)获取某个定时器中断状态

```
函数原型:uint32_t TIMR_INTStat(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(13)初始化输出比较功能

```
函数原型:void TIMR_OC_Init(TIMR_TypeDef * TIMRx,
     uint32_t match, uint32_t match_int_en, uint32_t init_lvl);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
②uint32_t match         当计数器的值递减到 match 时引脚输出电平翻转
③uint32_t match_int_en  当计数器的值递减到 match 时是否产生中断
④uint32_t init_lvl      初始输出电平
返回值 无
```

(14)使能输出比较功能的波形输出

```
函数原型:void TIMR_OC_OutputEn(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(15)禁止输出比较功能的波形输出

```
函数原型:void TIMR_OC_OutputDis(TIMR_TypeDef * TIMRx, uint32_t level);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
②uint32_t level         禁止输出波形后在引脚上保持的电平
返回值 无
```

(16)设置输出比较功能的比较值

```
函数原型:void TIMR_OC_SetMatch(TIMR_TypeDef * TIMRx, uint32_t match)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
②uint32_t match         输出比较功能的比较值
返回值 无
```

(17)获取输出比较功能的比较值

```
函数原型:uint32_t TIMR_OC_GetMatch(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(18)使能输出比较中断

```
函数原型:void TIMR_OC_INTEn(TIMR_TypeDef * TIMRx);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(19)禁能输出比较中断

```
函数原型:void TIMR_OC_INTDis(TIMR_TypeDef * TIMRx);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(20)清除输出比较中断标志

```
函数原型:void TIMR_OC_INTClr(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(21)获取输出比较中断状态

```
函数原型:uint32_t TIMR_OC_INTStat(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(22)输入捕获功能初始化

```
函数原型:void TIMR_IC_Init(TIMR_TypeDef * TIMRx,
     uint32_t captureH_int_en, uint32_t captureL_int_en)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
②uint32_t captureH_int_en   测量高电平长度完成中断使能
③uint32_t captureL_int_en   测量低电平长度完成中断使能
返回值 无
```

(23)获取高电平长度测量结果

```
函数原型:uint32_t TIMR_IC_GetCaptureH(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(24)获取低电平长度测量结果

```
函数原型:uint32_t TIMR_IC_GetCaptureL(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(25)使能输入捕获高电平长度测量完成中断

```
函数原型:void TIMR_IC_CaptureH_INTEn(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(26)禁能输入捕获高电平长度测量完成中断

```
函数原型:void TIMR_IC_CaptureH_INTDis(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(27)清除输入捕获高电平长度测量完成中断标志

```
函数原型:void TIMR_IC_CaptureH_INTClr(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(28)获取输入捕获高电平长度测量完成中断状态

```
函数原型:uint32_t TIMR_IC_CaptureH_INTStat(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(29)使能输入捕获低电平长度测量完成中断

```
函数原型:void TIMR_IC_CaptureL_INTEn(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(30)禁能输入捕获低电平长度测量完成中断

```
函数原型:void TIMR_IC_CaptureL_INTDis(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(31)清除输入捕获低电平长度测量完成中断标志

```
函数原型:void TIMR_IC_CaptureL_INTClr(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

(32)获取输入捕获低电平长度测量完成中断状态

```
函数原型:uint32_t TIMR_IC_CaptureL_INTStat(TIMR_TypeDef * TIMRx)
输入参数:
①TIMR_TypeDef * TIMRx       要设置的 TIMER,TIMER0~3/BTIMER0~3
返回值 无
```

实验 2-5-1 使用基本定时器提供 1ms 滴答

(1)初始化 BTIMER0

直接使用下方这个官方驱动代码提供的函数即可:

CODE 2-5-5-1 某个定时器/计数器初始化

```
函数原型:void TIMR_Init(TIMR_TypeDef * TIMRx, uint32_t mode, uint32_t
period, uint32_t int_en);
输入参数:
①TIMR_TypeDef * TIMRx   要设置的 TIMER,TIMER0~3/BTIMER0~3
②uint32_t mode
    a,对于 TIMR0~3，它的值可以是:
      TIMR_MODE_TIMER    -->定时器模式
      TIMR_MODE_COUNTER  -->计数器模式
      TIMR_MODE_OC       -->比较模式
      TIMR_MODE_IC       -->捕获模式
    b,对于 BTIMR0~3,只能是
      TIMR_MODE_TIMER    -->定时器模式
③uint32_t period
    a,对于 TIMR0~3:  定时/计数周期，32 位
    b,对于 BTIMR0~3: period[23:0]为定时周期,period[31:24]为预分频值
④uint32_t int_en
    是否使能中断
返回值 无
```

对于 BTIMER，应该这样使用:

```
/*设置预分频值*/
unsigned char PerDiv = 48;
/*设置定时器初值*/
unsigned int CounterVal = 1000;
/*组合参数*/
unsigned int periodVal =  PerDiv;
periodVal  = periodVal << 24;
periodVal  = periodVal  + CounterVal;
/*应用设置*/
TIMR_Init(BTIMR0, TIMR_MODE_TIMER, periodVal, 1);
/*启动定时器*/
TIMR_Start(BTIMR0);
```

(2)编写中断服务函数

对于 BTIMR0，对应的中断服务函数应该如此编写:

```
unsigned int gTimer0Tick = 0;
void BTIMR0_Handler(void)
{
    TIMR_INTClr(BTIMR0);
    gTimer0Tick ++;
}
```

(3)使用这个时间基准变量

```
NVIC_DisableIRQ(BTIMR0_IRQn);
printf("gTimer0Tick = %d \r\n",gTimer0Tick);
NVIC_EnableIRQ(BTIMR0_IRQn);
```

(4)使用基本定时器 BTIMER0 的总体代码如下:

```c
unsigned int gTimer0Tick = 0;
int main(void)
{
  SystemInit();
  SerialInit();
  /*设置预分频值*/
  unsigned char PerDiv = 48;
  /*设置定时器初值*/
  unsigned int CounterVal = 1000;
  /*组合参数*/
  unsigned int periodVal =  PerDiv;
  periodVal  = periodVal << 24;
  periodVal  = periodVal  + CounterVal;
  /*应用设置*/
  TIMR_Init(BTIMR0, TIMR_MODE_TIMER, periodVal, 1);
  /*启动定时器*/
  TIMR_Start(BTIMR0);
  while(1==1)
  {
    NVIC_DisableIRQ(BTIMR0_IRQn);
    printf("gTimer0Tick = %d \r\n",gTimer0Tick);
    NVIC_EnableIRQ(BTIMR0_IRQn);
  }
}
void BTIMR0_Handler(void)
{
  TIMR_INTClr(BTIMR0);
  gTimer0Tick ++;
}
```

实验 2-5-2 使用高级定时器提供 1ms 滴答

(1)初始化 TIMER0

高级定时器 TIMER0 的输入时钟是 PCLK1，所以一个周期是 1/48MHz(s)，想要的到 1ms 的中断周期，需要将其初值设置为 48*1000 = 48000。

```
    /*没有组合参数，直接就是定时器初值*/
    unsigned int periodVal =  48000;
    /*应用设置*/
    TIMR_Init(TIMR0, TIMR_MODE_TIMER, periodVal, 1);
    /*启动定时器*/
    TIMR_Start(TIMR0);
```

(2)编写中断服务函数

对于 TIMR0，对应的中断服务函数应该如此编写:

```
unsigned int gTim0Tick = 0;
void TIMR0_Handler(void)
{
    TIMR_INTClr(TIMR0);
    gTim0Tick ++;
}
```

(3)使用这个时间基准变量

```
NVIC_DisableIRQ(TIMR0_IRQn);
printf("gTim0Tick = %d \r\n",gTim0Tick );
NVIC_EnableIRQ(TIMR0_IRQn);
```

(4)总体代码如下:

```
unsigned int gTim0Tick = 0;
void SerialInit(void);

int main(void)
{
  SystemInit();
  SerialInit();
  /*-------------------------------------------------------*/
  /*没有组合参数，直接就是定时器初值*/
  unsigned int periodVal =  48000;
  /*应用设置*/
  TIMR_Init(TIMR0, TIMR_MODE_TIMER, periodVal, 1);
  /*启动定时器*/
  TIMR_Start(TIMR0);
  while(1==1)
  {
    NVIC_DisableIRQ(TIMR0_IRQn);
    printf("gTim0Tick = %d \r\n",gTim0Tick );
    NVIC_EnableIRQ(TIMR0_IRQn);
  }
}
void TIMR0_Handler(void)
{
  TIMR_INTClr(TIMR0);
  gTim0Tick ++;
}
```

2.6 SPI 通信

2.6.1 SPI 通信概述

SWM190 的 SPI 模块有以下特性:

SPI 模式

- 支持主机模式和从机模式
- 支持 SPI 和 SSI 两种帧结构
- 内置深度为 8 的 FIFO，作为接收和发送数据的缓存
- 支持 DMA
- 数据位数 4~16bit 可配置
- 可编程时钟极性和相位
- 支持 LSB 和 MSB 可配置

I2S 模式

- 支持全双工或半双工通讯
- 支持主模式或从模式
- 8 位可编程线性预分频器,可实现精确的音频采样频率(8kHz 到 192kHz)
- 数据格式 8 位、 16 位、 24 位或 32 位可配置
- 支持 I2S Philips 标准、
  支持 MSB justified 标准、
  支持 PCM 标准（长帧和短帧同步）
- 支持 DMA
- 支持 LSB 和 MLB 可配置

SPI FLASH 模式

- 仅支持 4 线快速读操作 SWM190 系列
- Dummy clock 个数可配置
- 读命令可配置

其模块框图如图 2-6-1 所示:

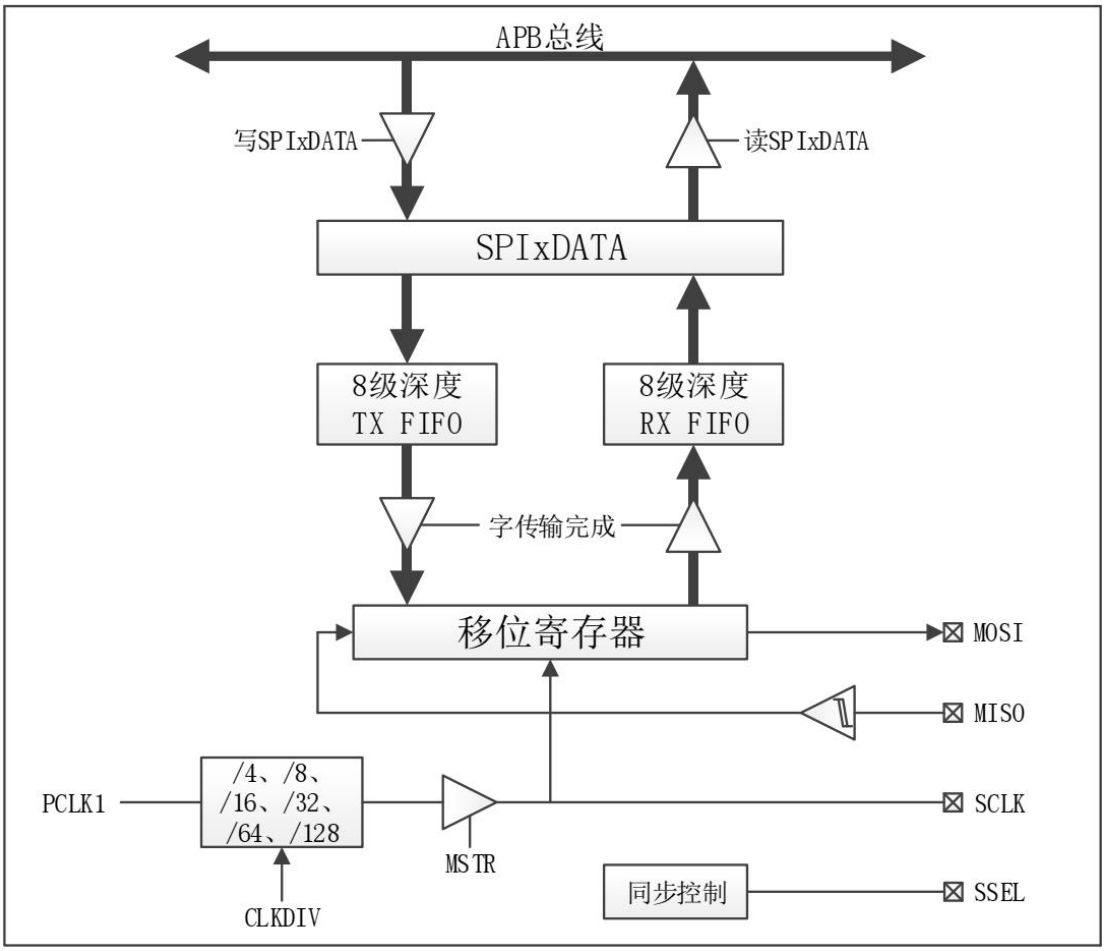


图 2-6-1　SPI 模块功能框图

从这个框图中，可以明显的得到以下信息:

①时钟

SPI 模块的时钟源为 PCLK1。

可以通过分频器 CLKDIV，降低时钟速度。

②SPI 的核心是一个移位寄存器，上面的图 2-6-1 并不贴切，抛开芯片相关的内容，我们根据图 2-6-2 单独剖析 SPI 的数据收发流程:

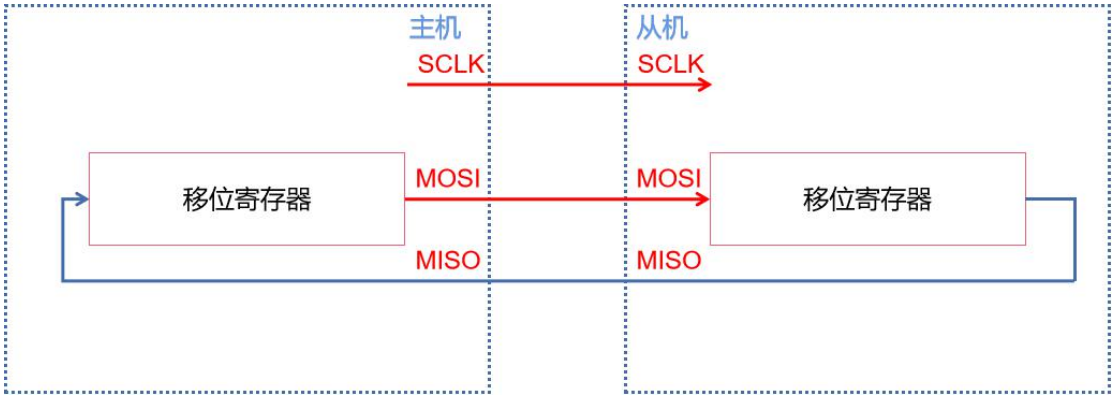


图 2-6-2 SPI 工作原理框图

如图 2-6-2 所示，主机的移位寄存器与从机的移位寄存器构成了一个环形结构。可以这样说，移位寄存器相当于一个管道。

对于主机:

a,主机的时钟来源于芯片本身，对于本芯片，来源于 PCLK1 分频后的低速时钟。
b,主机发送数据时，每来一个边沿信号，管道就从左侧引脚(MISO)进来一位数据
c,管道空间有限,原本位于管道右侧的数据被挤出,呈现在右侧引脚(MOSI)上。

对于从机:(若是已经被使能，或者说被片选)

a,从机接收数据时被动的，它的边沿信号由主机的 SCLK 引脚提供。
b,主机在发送数据的时候，给从机提供了边沿信号，每来一个对应的边沿信号，从机的管道左侧(MOSI)就会进入一位数据。
c,管道空间有限，原本位于管道右侧的数据被挤出，呈现在引脚(MOSI)上。

如果一片单片机作为主机，想要接多个从设备，应该怎么接?看下面的图 2-6-3，一目了然。

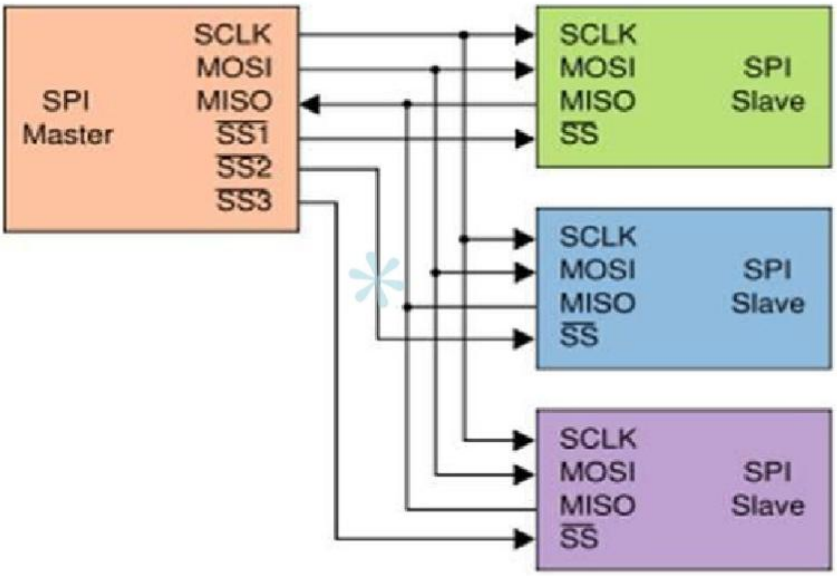


图 2-6-3 SPI 主机接多个从设备

从上面的描述可以得知:**MCU 的 SPI 模块作为主机发送数据的时候,需要收取一个数据，即使收到的这个数据时没用的。**

如果是科普过或者学习过 SPI 的同学，应该知道:

a,在传输数据的时候，SPI 模块对引脚的数据采样是一个瞬间，这个时间点，可以是上升沿，也可以是下降沿。
b,SPI 主机提供 SCLK，送到从机设备，在 SPI 模块空闲时，它应该是暂停对外提供 SLCK 时钟信号的，此时就对应一个空闲时 SCLK 引脚时什么状态，可以设置为低电平，也可以设置为高电平。

话不多说，对于 SPI 的工作控制，看看官方提供什么接口?

2.6.2 SPI 驱动接口

(1)SPI 初始化函数

函数原型:

```
void SPI_Init(SPI_TypeDef * SPIx, SPI_InitStructure * initStruct);
```

输入参数:

```
①SPI_TypeDef * SPIx                 要设置的 SPI，SPI0/SPI1
②SPI_InitStructure * initStruct     包含 SPI 相关设定值的结构体
返回值: 无
   SPI_InitStructure * initStruct 的具体格式，请阅读代码。
```

(2)打开某个 SPI 模块

```
函数原型:void SPI_Open(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                 要设置的 SPI，SPI0/SPI1
返回值: 无
```

(3)关闭某个 SPI 模块

```
函数原型:void SPI_Close(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                 要设置的 SPI，SPI0/SPI1
返回值: 无
```

(4)读取一个数据

```
函数原型:uint32_t SPI_Read(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                 要设置的 SPI，SPI0/SPI1
返回值: uint32_t 类型     读取到的数据
```

(5)写入一个数据

```
函数原型:void SPI_Write(SPI_TypeDef * SPIx, uint32_t data)
输入参数:
①SPI_TypeDef * SPIx                 要设置的 SPI，SPI0/SPI1
②uint32_t data                      要写入的数据
返回值: 无
```

(6) 写入一个数据并等待数据完全发送出去

```
函数原型:void SPI_WriteWithWait(SPI_TypeDef * SPIx, uint32_t data);
①SPI_TypeDef * SPIx                 要设置的 SPI，SPI0/SPI1
②uint32_t data                      要写入的数据
返回值: 无
```

(7)发送一个数据，并返回发送过程中接收到的数据

```
函数原型:uint32_t SPI_ReadWrite(SPI_TypeDef * SPIx, uint32_t data);
输入参数:
①SPI_TypeDef * SPIx                 要设置的 SPI，SPI0/SPI1
②uint32_t data                      要写入的数据
返回值: uint32_t 类型 接收到的数据
```

(8) 判断接收 FIFO 是否空

```
   判断接收 FIFO 是否空，如果不空则可以继续 SPI_Read()。
函数原型:uint32_t SPI_IsRXEmpty(SPI_TypeDef * SPIx);
输入参数:
①SPI_TypeDef * SPIx                 要设置的 SPI，SPI0/SPI1
返回值: 1 接收 FIFO 空     0 接收 FIFO 非空
```

(9)判断发送 FIFO 是否满

```
    发送 FIFO 是否满，如果不满则可以继续 SPI_Write()。
函数原型:uint32_t SPI_IsTXFull(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                    要设置的 SPI，SPI0/SPI1
返回值: 1 发送 FIFO 满     0 发送 FIFO 不满
```

(10)判断发送 FIFO 是否空

```
函数原型:uint32_t SPI_IsTXEmpty(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                    要设置的 SPI，SPI0/SPI1
返回值:1 发送 FIFO 空     0 发送 FIFO 非空
```

(11)使能接收 FIFO 半满中断

```
函数原型:void SPI_INTRXHalfFullEn(SPI_TypeDef * SPIx);
输入参数:
①SPI_TypeDef * SPIx                    要设置的 SPI，SPI0/SPI1
返回值:无
```

(12)禁用接收 FIFO 半满中断

```
函数原型:void SPI_INTRXHalfFullDis(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                    要设置的 SPI，SPI0/SPI1
返回值:无
```

(13)清除接收 FIFO 半满中断标志

```
函数原型:void SPI_INTRXHalfFullClr(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                    要设置的 SPI，SPI0/SPI1
返回值:无
```

(14)查询接收 FIFO 半满中断状态

```
函数原型:uint32_t SPI_INTRXHalfFullStat(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                    要设置的 SPI，SPI0/SPI1
返回值:    1 接收 FIFO 达到半满     0 接收 FIFO 未达到半满
```

(15)使能接收 FIFO 满中断

```
函数原型:void SPI_INTRXFullEn(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                    要设置的 SPI，SPI0/SPI1
返回值:无
```

(16)禁用接收 FIFO 满中断

```
函数原型:void SPI_INTRXFullDis(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(17)清除接收 FIFO 满中断标志

```
函数原型:void SPI_INTRXFullClr(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(18)查询接收 FIFO 满中断状态

```
函数原型:uint32_t SPI_INTRXFullStat(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(19)使能接收 FIFO 溢出中断

```
函数原型:void SPI_INTRXOverflowEn(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(20)禁用接收 FIFO 溢出中断

```
函数原型:void SPI_INTRXOverflowDis(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(21)清除接收 FIFO 溢出中断标志

```
函数原型:void SPI_INTRXOverflowClr(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(22)查询接收 FIFO 溢出中断状态

```
函数原型:uint32_t SPI_INTRXOverflowStat(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:1 接收 FIFO 溢出     0 接收 FIFO 未溢出
```

(23)使能发送 FIFO 空中断

```
函数原型:void SPI_INTTXEmptyEn(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(24)禁止发送 FIFO 空中断

```
函数原型:void SPI_INTTXEmptyDis(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(25)清除发送 FIFO 空中断标志

```
函数原型:void SPI_INTTXEmptyClr(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(26)查询发送 FIFO 空中断状态

```
函数原型:uint32_t SPI_INTTXEmptyStat(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:1 发送 FIFO 空     0 发送 FIFO 非空
```

(27)使能发送 FIFO 空且发送移位寄存器空中断

```
函数原型:void SPI_INTTXCompleteEn(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(28)禁用发送 FIFO 空且发送移位寄存器空中断

```
函数原型:void SPI_INTTXCompleteDis(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(29)清除发送 FIFO 空且发送移位寄存器空中断状态

```
函数原型:void SPI_INTTXCompleteClr(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(30)查询发送 FIFO 空且发送移位寄存器空中断状态

```
函数原型:uint32_t SPI_INTTXCompleteStat(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(31)使能接收 FIFO 中数据个数≥CTRL.RXTHR 中断

```
函数原型:void SPI_INTRXThresholdEn(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(32)禁用接收 FIFO 中数据个数≥CTRL.RXTHR 中断

```
函数原型:void SPI_INTRXThresholdDis(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(33)清除接收 FIFO 中数据个数≥CTRL.RXTHR 中断标志

```
函数原型:void SPI_INTRXThresholdClr(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(34)查询接收 FIFO 中数据个数≥CTRL.RXTHR 中断状态

```
函数原型:uint32_t SPI_INTRXThresholdStat(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:
    1 接收 FIFO 中数据个数≥CTRL.RXTHR
    0 接收 FIFO 中数据个数不≥CTRL.RXTHR
```

(35)使能发送 FIFO 中数据个数≤CTRL.TXTHR 中断

```
函数原型:void SPI_INTTXThresholdEn(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(36)禁止发送 FIFO 中数据个数≤CTRL.TXTHR 中断

```
函数原型:void SPI_INTTXThresholdDis(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(37)清除发送 FIFO 中数据个数≤CTRL.TXTHR 中断标志

```
函数原型:void SPI_INTTXThresholdClr(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:无
```

(38)查询发送 FIFO 中数据个数≤CTRL.TXTHR 中断状态

```
函数原型:uint32_t SPI_INTTXThresholdStat(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:
    1 发送 FIFO 中数据个数≤CTRL.TXTHR
    0 发送 FIFO 中数据个数不≤CTRL.TXTHR
```

(39)I2S 音频串行接口初始化

```
    包括帧格式、数据长度、时钟频率、中断设定、FIFO 触发设定
函数原型:
void I2S_Init(SPI_TypeDef * SPIx, I2S_InitStructure * initStruct);
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
②I2S_InitStructure * initStruct    包含 I2S 相关设定值的结构体
返回值:    无
    I2S_InitStructure 类型，具体成员，请阅读代码。
```

(40)I2S 打开，允许收发

```
函数原型:void I2S_Open(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:    无
```

(41)I2S 关闭，禁止收发

```
函数原型:void I2S_Close(SPI_TypeDef * SPIx)
输入参数:
①SPI_TypeDef * SPIx                要设置的 SPI，SPI0/SPI1
返回值:    无
```

(42)I2S MCLK 时钟输出配置

函数原型:void I2S_MCLKConfig(SPI_TypeDef * SPIx, uint32_t output_enable, uint32_t mclk_freq)
输入参数:

①SPI_TypeDef * SPIx	要设置的 SPI，SPI0/SPI1
②uint32_t output_enable	是否输出 MCLK 时钟
③uint32_t mclk_freq	MCLK 时钟频率

返回值: 无

2.6.3 SPI-Flash 读写

(1)认识一类 SPI Flash

W25X16/W25X32/W25X64 等芯片是 SPI NorFlash 芯片，分别有 16Mbit/32Mbit/64Mbit 的存储空间，也就是常说的，2MB/4MB/8MB 空间。

芯片的最大编程单位为页，每一页的大小为 256 字节，所以这三个型号分别有 8192/16384/32768 个可编程页，。

使用“页编程指令”每次可以编程 256 个字节；

使用“扇区擦除指令”每次可以擦除 16 页，也就是 4KB。

使用“使用块擦除指令”每次可以擦除 256 页，也就是 64KB。

使用“整片擦除命令”可以擦除整个芯片的数据；

对于 Flash，它的设定，是这样的:

①编程原理:位翻转，并且只能将 1 为 0，不能将 0 翻转为 1。

②想要得到数据 1，需要将对应的位所在的扇区擦除，擦除后，默认数据为 1。

所以，操作流程是这样:

①开辟一块内存 Buf[4096]

②先读出一个扇区的数据倒 Buf

③修改 Buf 里对应的数据

④擦掉这一个扇区

⑤将数回写到 Flash 中。

所以，对于写入操作，编程流程应该是“读==>改==>擦==>写”。

从手册上面得到操作命令，如图 2-6-3-1 所示:

指令	说明	操作码周期[1]	地址周期[2]	空周期	数据周期	最大频率
读	以 25 MHz 的频率读存储器	0000 0011b （03H）	3	0	1 至 ∞	25 MHz
高速读	以 80 MHz 的频率读存储器	0000 1011b （0BH）	3	1	1 至 ∞	80 MHz
4 KB 扇区擦除[3]	擦除 4 KB 的存储器阵列	0010 0000b （20H）	3	0	0	80 MHz
32 KB 块擦除[4]	擦除 32 KB 块的存储器阵列	0101 0010b （52H）	3	0	0	80 MHz
64 KB 块擦除[5]	擦除 64 KB 块的存储器阵列	1101 1000b （D8H）	3	0	0	80 MHz
全片擦除	擦除全部存储器阵列	0110 0000b （60H） 或 1100 0111b （C7H）	0	0	0	80 MHz
字节编程	编程一个数据字节	0000 0010b （02H）	3	0	1	80 MHz
AAI 字编程[6]	自动地址递增编程	1010 1101b （ADH）	3	0	2 至 ∞	80 MHz
RDSR[7]	读取状态寄存器	0000 0101b （05H）	0	0	1 至 ∞	80 MHz
EWSR	使能写状态寄存器	0101b 0000b （50H）	0	0	0	80 MHz
WRSR	写状态寄存器	0000 0001b （01H）	0	0	1	80 MHz
WREN	写使能	0000 0110b （06H）	0	0	0	80 MHz
WRDI	写禁止	0000 0100b （04H）	0	0	0	80 MHz
RDID[8]	读 ID	1001 0000b （90H） 或 1010 1011b （ABH）	3	0	1 至 ∞	80 MHz
JEDEC-ID	JEDEC ID 读	1001 1111b （9FH）	0	0	3 至 ∞	80 MHz
EBSY	在 AAI 编程期间使能 SO 以输出 RY/BY# 状态	0111 0000b （70H）	0	0	0	80 MHz
DBSY	在 AAI 编程期间禁止 SO 为 RY/BY# 状态	1000 0000b （80H）	0	0	0	80 MHz

图 2-6-3-1 SPI Flash 操作命令

(2)了解 SPI Flash 的操作流程

①读取数据流程

```
a,判断 SPI Flash 是否处于忙状态，如果忙，则等待 Flash 响应
b,发送命令让 SPI Flash 进入读状态
c,发送想要读取的数据的起始地址
d,将数据读取出来，可以连续读。
e,发送结束命令，结束本次读操作
```

②擦扇区流程

```
a,判断 SPI Flash 是否处于忙状态，如果忙，则等待 Flash 响应
b,发送命令使能 SPI Flash 的写操作
c,发送命令让 SPI Flash 进入擦除扇区状态
d,发送地址给 SPI Flash，告诉要从什么地方开始擦除
e,发送结束命令，结束本次读操作
```

③写数据流程

对于 SPI Flash 而言，写数据操作，实际上就是编程操作。所谓编程，就是位翻转操作，可以选择性将对应的数据位翻转为 0。SPI Flash 支持“字节编程”和“页编程”，我们统一使用“页编程”操作,编程流程如下:

```
a,判断 SPI Flash 是否处于忙状态，如果忙，则等待 Flash 响应
b,发送命令使能 SPI Flash 的写操作
c,发送命令让 SPI Flash 进入 页编程状态
d,发送地址给 SPI Flash，告诉要从什么地方开始编程
e,将数据写进去，可以连续写
f,发送结束命令，结束本次读操作
```

(3)SPI Flash 常用操作函数

对于上面的操作流程，其实已经做好相应的封装，在开发过程中，常用的就那么几个函数。

①SPI Flash 读函数

```
函数原型:
    void SPI_Flash_Read(void *vbuf,uint16_t len,uint32_t eep_addr);
输入参数:
    void *vbuf          读到的数据存储在什么地方
    uint16_t len        读多长的数据? 单位:字节
    uint32_t eep_addr   从什么地方开始读?
返回值:无
```

②SPI Flash 擦除扇区函数

```
函数原型: void SectorErase(uint32_t addr)
输入参数:
    uint32_t addr       需要擦除的扇区的地址
返回值:无
```

③SPI Flash 写函数

函数原型:

```
void SPI_Flash_Write(void *vbuf, uint16_t len, uint32_t eep_addr);
```

输入参数:

void *vbuf	需要写的数据
uint16_t len	需要写多少数据? 单位:字节
uint32_t eep_addr	从什么地方开始写?

返回值:无

实验 2-6-1 读写 SPI Flash 实验

上面的原理，学不会，别着急，实际上，懂得使用就可以了。

实验目的:读写 SPI Flash

实验原理:

①准备一堆数据，这堆数据写满了 0xCC

②对 SPI Flash 某个扇区执行擦操作，

③从 SPI Flash 读出这个扇区的数据，并打印出来。(此时打印出来的数据应该全是 0xFF)

④将准备好的数据，写入该扇区。

⑤再次从 SPI Flash 读出这个扇区的数据，并打印出来(此时打印出来的数据应该全是 0xCC)

==>1、初始化 SPI 模块

ITX-190 上面使用 SPI0 挂接 SPI Flash，初始化代码如下图 2-6-3-2 所示:

```
void Init_SPI0(void)
{
  /*初始化SPI功能引脚*/
  //PORT_Init(PORTA, PIN8,  PORTA_PIN8_SPI0_SSEL,  0);
  PORT_Init(PORTA, PIN11, PORTA_PIN11_SPI0_SCLK, 0);
  PORT_Init(PORTA, PIN10, PORTA_PIN10_SPI0_MOSI, 0);
  PORT_Init(PORTA, PIN9,  PORTA_PIN9_SPI0_MISO,  1);
  GPIO_Init(GPIOC, PIN1, 1, 1, 0, 0); //CS

  /*分配一个SPI配置结构体，并设置它*/
  SPI_InitStructure SPI_initStruct;
  /*分频，降低SPI模块工作速度*/
  SPI_initStruct.clkDiv          = SPI_CLKDIV_32;    //1.5MHz
  /*配置SPI模块工作模式:SPI模式还是TI_SSI模式*/
  SPI_initStruct.FrameFormat     = SPI_FORMAT_SPI;   //选择SPI模式
  /*配置在SPI帧格式下的采样时刻:上升沿采样还是下降沿采样
    第一个边沿是上升沿，第二个边沿是下降沿*/
  SPI_initStruct.SampleEdge      = SPI_FIRST_EDGE;   //选择第一个边沿
  /*配置在SPI帧格式下，空闲时的电平:*/
  SPI_initStruct.IdleLevel       = SPI_LOW_LEVEL;    //空闲时为低电平
  /*接收字长度*/
  SPI_initStruct.WordSize        = 8;
  /*配置是否使用SPI主机模式*/
  SPI_initStruct.Master          = 1;
  /*接收中断是否使能*/
  SPI_initStruct.RXThresholdIEn = 0;
  /*发送中断是否使能*/
  SPI_initStruct.TXThresholdIEn = 0;
  /*是否允许TX FIFO或者移位寄存器空的中断请求*/
  SPI_initStruct.TXCompleteIEn  = 0;
  /*应用这些配置*/
  SPI_Init(SPI0, &SPI_initStruct);
  /*使能SPI模块*/
  SPI_Open(SPI0);
}
```

图 2-6-3-2 SPI 模块初始化流程

==>2、实验代码

代码如图 2-6-3-3 所示

```
#define BufSize   128
#define TestAddr  0
unsigned char Data[BufSize];
int main(void)
{
  SystemInit();
  SerialInit();
  Init_SPI0();
  /*----------------------------------------------------*/
  printf("Running......\r\n");
  /*----------------------------------------------------*/
  //①对SPI Flash某个扇区执行擦操作
  printf("Begin Erase SPI Flash One Sector.TestAddr = 0x%0X\r\n",TestAddr);
  SectorErase(TestAddr);
  printf("Erase SPI Flash One Sector Success!\r\n");
  /*----------------------------------------------------*/
  //②从SPI Flash读出这个扇区的数据，并打印出来。(此时打印出来的数据应该全是0xFF)
  printf("Start SPI_Flash_Read for TestAddr = 0x%0X,Size = %d\r\n",TestAddr,BufSize);
  for(unsigned int i=0;i<BufSize;i++)   {Data[i]=0;}        //清空缓冲区
  SPI_Flash_Read(Data,BufSize,TestAddr);
  for(unsigned int i=0;i<BufSize;i++)   {printf("0x%0X ",Data[i]);}
  printf("\r\nSPI_Flash_Read in the End!\r\n");
  /*----------------------------------------------------*/
  //③准备一堆数据，这堆数据写满了0xCC
  for(unsigned int i=0;i<BufSize;i++)   {Data[i]=0xCC;}
  /*----------------------------------------------------*/
  //④将准备好的数据，写入该扇区。
  printf("Start SPI_Flash_Write for TestAddr = 0x%0X,Size = %d\r\n",TestAddr,BufSize);
  SPI_Flash_Write(Data,BufSize,TestAddr);
  printf("\r\nSPI_Flash_Write in the End!\r\n");
  /*----------------------------------------------------*/
  //⑤再次从SPI Flash读出这个扇区的数据，并打印出来(此时打印出来的数据应该全是0xCC)
  printf("Start SPI_Flash_Read for TestAddr = 0x%0X,Size = %d\r\n",TestAddr,BufSize);
  for(unsigned int i=0;i<BufSize;i++)   {Data[i]=0;}
  SPI_Flash_Read(Data,BufSize,TestAddr);
  for(unsigned int i=0;i<BufSize;i++)   {printf("0x%0X ",Data[i]);}
  printf("\r\nSPI_Flash_Read in the End!\r\n");
  /*----------------------------------------------------*/
  while(1==1){
  }
}
```

图 2-6-3-3 SPI Flash 示例代码

==>3、实验结果

将程序编译烧写进单板，通过串口工具，可以看到以下图 2-6-3-4 的现象，

```
Running......
Begin Erase SPI Flash One Sector.TestAddr = 0x0
Erase SPI Flash One Sector Success!
Start SPI_Flash_Read for TestAddr = 0x0,Size = 128
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
SPI_Flash_Read in the End!
Start SPI_Flash_Write for TestAddr = 0x0,Size = 128

SPI_Flash_Write in the End!
Start SPI_Flash_Read for TestAddr = 0x0,Size = 128
0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC
0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC
0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC
0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC
0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC
0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC
0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC 0xCC
SPI_Flash_Read in the End!
```

图 2-6-3-4　SPI Flash 示例实验现象