IIC （又叫 I2C），是单片机芯片和它的外围设备进行数据传送的一种方式。

I2C 串行总线一般有两根信号线，一根是双向的数据线 SDA，另一根是时钟线 SCL。所有接到 I2C 总线设备上的串行数据 SDA 都接到总线的 SDA 上，各设备的时钟线 SCL 接到总线的 SCL 上。SCL 相当于为信息的传递打拍子，拿摩斯电码举例，多长时间发送一个信号就是信息传递的节拍。SDA 为数据线，相当于摩斯电码中的长或短。

为了避免总线信号的混乱，要求各设备连接到总线的输出端时必须是漏极开路（OD）输出或集电极开路（OC）输出，这一点很重要，在 GPIO 初始化的时候一定要设置对该方式。IIC 所用的两条线路都需要设置成上拉，这样 SCL 或 SDA 电平才能被从机设备所改变。设备上的串行数据线 SDA 接口电路应该是双向的，即在数据传送过程中既要为输出模式，又需要其输入模式。输出电路用于向总线上发送数据，输入电路用于接收总线上的数据。而串行时钟线也应是双向的，作为控制总线数据传送的主机，一方面要通过 SCL 输出电路发送时钟信号，另一方面还要检测总线上的 SCL 电平，以决定什么时候发送下一个时钟脉冲电平；作为接受主机命令的从机，要按总线上的 SCL 信号发出或接收 SDA 上的信号，也可以向 SCL 线发出低电平信号以延长总线时钟信号周期。(事实上在我们的模拟 IIC 中 SCL 是单向的)。

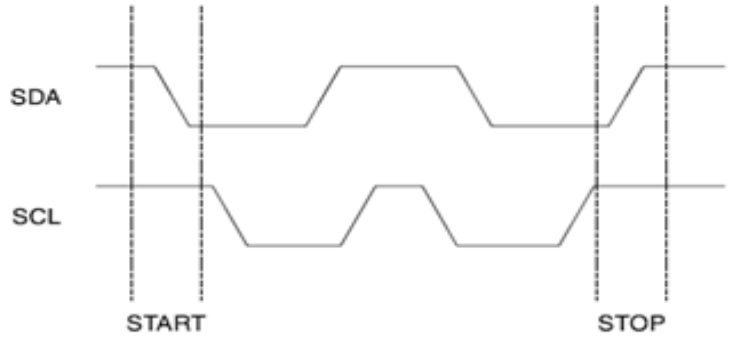
由于 STM32F103 系列的硬件 IIC 模块有一定的设计缺陷，所以大多数人都选择使用模拟 IIC。我们这里提供模拟 IIC 读取 MPU6050 的源码，一方面规避硬件 IIC 可能存在的问题，另一方面也能让大家从原理上明白这种硬件之间的通信方式。

IIC 的通信包括以下几个部分：

1）空闲状态：I2C 总线的 SDA 和 SCL 两条信号线同时处于高电平时，规定为总线的空闲状态。

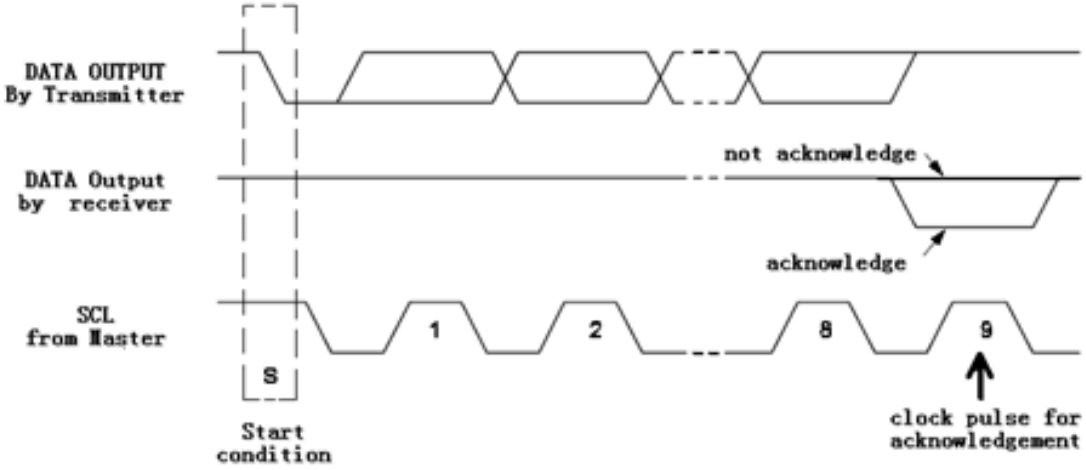
2）起始信号：当 SCL 为高期间，SDA 由高到低的跳变；启动信号是一种电平跳变时序信号（是一个变化的过程），而不是一个电平信号。

3）停止信号：当 SCL 为高期间，SDA 由低到高的跳变；停止信号也是一种电平跳变时序信号（是一个变化的过程），而不是一个电平信号。



可以清楚地看到两个跳变的过程。

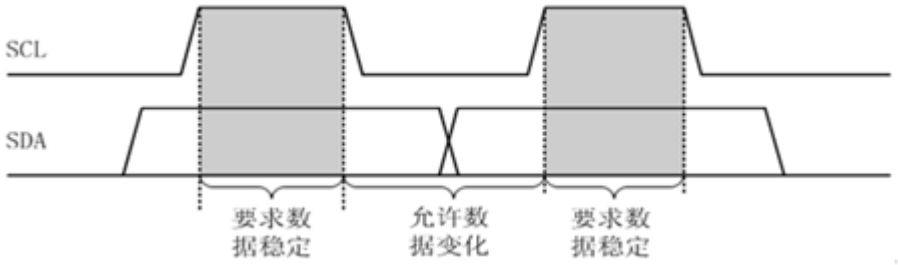
4）应答信号（ACK & NACK）：发送器每发送一个字节，就在时钟脉冲 9 期间释放数据线，由接收器反馈一个应答信号。 应答信号为低电平时，规定为有效应答位（ACK 简称应答位），表示接收器已经成功地接收了该字节；应答信号为高电平时，规定为非应答位（NACK），一般表示接收器接收该字节没有成功



I^2C 总线的响应

因为 SDA 是上拉的，所以从机可以把线路信号拉低来回应一个低电平。

4） 数据传送：I2C 总线进行数据传送时，时钟信号为高电平期间，数据线上的数据必须保持稳定，只有在时钟线上的信号为低电平期间，数据线上的高电平或低电平状态才允许变化。在我们的程序中有大量的 IIC_Delay 来使某个状态保持足够长的时间以便于正确通信。当 SCL 为低电平的时候，SDA 改变状态，输出想要输出的数据，SCL 为高电平时，SDA 数据保持稳定，从机开始读取 SDA 上的电平信号。然后 SCL 继续变为低电平，SDA 改变想要输出的信号……这样一直变换下去，大量的数据得以传送。因此可以把 IIC 协议简单的理解为两个人说话用的规则（语法）。



。

接下来我们看一下我们的实现。

在 IIC 的底层代码中，为了大家更换引脚的时候方便一些，我们将引脚号用宏定义替换来简化替换过程。

```
/*******************************************************************************
#define IIC_GPIO (GPIOB)
#define IIC_GOIO_SDA (GPIOB)
#define IIC_GPIO_SCL (GPIOB)
#define IIC_SDA (GPIO_Pin_7)
#define IIC_SCL (GPIO_Pin_6)
/*******************************************************************************
```

当你想要用其他引脚作为 IIC 的输出的时候更改该处就可以,其他的不用修改。
从图上可以看出我们的 SDA 使用的是 GPIOB 的 7 引脚，SCL 使用的是 GPIOB 的 6 引脚，这点在原理图上可以看到。
宏定义还可以用来替换代码块

```
#define SET_SDA    { GPIO_SetBits( IIC_GPIO , IIC_SDA ); }
#define RESET_SDA  { GPIO_ResetBits( IIC_GPIO , IIC_SDA );}
#define SET_SCL    { GPIO_SetBits( IIC_GPIO , IIC_SCL ); }
#define RESET_SCL   { GPIO_ResetBits( IIC_GPIO , IIC_SCL); }
#define IIC_SDA_STATE (IIC_GPIO->IDR&IIC_SDA)
```

IDR 寄存器的数据时实时反映 IO 口的状态，甚至不需要切换为输入模式。
所以最后一条语句是获取 IIC_SDA 引脚的电平状态。

1） 在使用 GPIO 之前一定要初始化 IO 口，并开启它们所对应的时钟。

```
void IIC_GPIO_Configuration( GPIO_TypeDef * GPIOx_SDA , uint16_t SDA_Pin , GPIO_TypeDef * GPIOx_SCL , uint16_t SCL_Pin )
{
  GPIO_InitTypeDef GPIO_InitStructure;                          //声明GPIO配置结构体
  uint32_t RCC_GPIOx_SDA = 0;
  uint32_t RCC_GPIOx_SCL = 0;

  //得到滤波后的引脚端口
  RCC_GPIOx_SDA = GPIO_Filter( GPIOx_SDA );                     //根据宏定义选定初始化的时钟
  RCC_GPIOx_SCL = GPIO_Filter( GPIOx_SCL );

  //使能时钟
  RCC_APB2PeriphClockCmd(RCC_GPIOx_SDA,ENABLE);
  RCC_APB2PeriphClockCmd(RCC_GPIOx_SCL,ENABLE);

  //配置引脚
  GPIO_InitStructure.GPIO_Pin = SDA_Pin;
  GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode=GPIO_Mode_Out_OD;                //设置为开漏输出模式！！！
  GPIO_Init(GPIOx_SDA, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Pin = SCL_Pin;
  GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode=GPIO_Mode_Out_OD;                //设置为开漏输出模式！！！
  GPIO_Init(GPIOx_SCL, &GPIO_InitStructure);

  //初始化ICC的模式
  SET_SDA;
  SET_SCL;
}
```

（代码在例程中 iic.c 和 iic.h 中）

2） 因为各个模式的实现函数均在例程中给出，这里拿一个函数进行分析举例。

```
void IIC_Stop(void)
{
  RESET_SDA;
  IIC_DELAY;

  SET_SCL;
  IIC_DELAY;

  SET_SDA;
  IIC_DELAY;
}
```

这是发送停止信号的函数。
过程如下：
1） SDA 输出 0
2） 延时保持
3） SCL 输出 1
4） 延时保持
5） SDA 输出 1
6） 延时保持。
SDA 出现了由低电平到高电平的变化，停止信号发出。

有了这些基础原件以后，怎么和设备通信？只用我们上面提到的五点可不可以？复杂的东西都是由简单的部分组成，不仅体现在硬件上，体现在所有方面。

```
/**************************************************
*函数名称:Single_Write_IIC
*参数:SlaveAddress从机地址，REG_Address寄存器地址，REG_data数据
*功能:单字节写入
**************************************************/

void Single_Write_IIC(u8 SlaveAddress,u8 REG_Address,u8 REG_data)
{
    IIC_Start();                    //起始信号
    IIC_SendByte(SlaveAddress);     //发送设备地址+写信号
    IIC_SendByte(REG_Address);      //内部寄存器地址
    IIC_SendByte(REG_data);         //内部寄存器数据
    IIC_Stop();                     //发送停止信号
}
```

这就是一次简单的写入通信
1） 发送开始信号
2） 发送数据，该数据为从机地址。从机地址在从机设备的数据手册上有说明。6050 的从机地址在 mpu6050.h 头文件里有定义。
3） 发送数据，该数据为想要写入的从机寄存器的地址
4） 发送数据，该数据为想要写到寄存器的地址。
5） 停止信号
那么不妨思考一下为什么是这样？IIC 是一种可以同时和多个设备通信的协议，也就是多个从机设备同时并联在同一条线路上，你发送给 A 的数据事实上 B 也能收到，但是当 B 检查你发送的从机地址发现并不是给自己就不会接受。这是从机地址的意义。一个从机设备中有很多的寄存器，建立起通信以后你发送寄存器的地址（从机设备数据手册中提供），告诉从机你要往这个寄存器写数据。最后一步把你要写的数据传过去，这就是一个完整的数据传送的过程。同理怎么读出数据？
当你将从机地址发过去，想读取的从机地址发过去，等待设备把你的数据返回。如果前两步完全一样，从机怎样判断你是想写入还是想读出呢？

```
u8 Single_Read_IIC(u8 SlaveAddress, u8 REG_Address)
{
  u8 REG_data;
    IIC_Start();                          //起始信号
    IIC_SendByte(SlaveAddress);           //发送设备地址+读信号
    IIC_SendByte(REG_Address);            //发送存储单元地址，//从0开始
    IIC_Start();                          //起始信号
    IIC_SendByte(SlaveAddress+1);         //发送设备地址+读信号
    REG_data = IIC_RecvByte();              //读出寄存器数据
  IIC_SendACK();
  IIC_Stop();                           //停止信号
    return REG_data;
}
```

可以看到在读取的时候，我们在设备地址上加 1 代表读信号。不加的时候其实是+0，代表写信号。

另对于不同的从机设备，IIC 使用的方法有细微的区别，但是最底层的步骤是绝对不会变的。关于 IIC 和 MPU6050 通信的部分请参见 MPU6050。