

# IIC 通讯实验

## 1 概述

本实验实验 LKS32MC081 芯片进行 IIC 主模式数据发送接收实验，本实验使用 P0.3 作为 IIC 时钟线 SCL，使用 P0.4 作为 IIC 数据线 SDA，需要注意 P0.3 和 P0.4 需要硬件外接上拉电阻。LKS32MC08x 系列 IIC 不使用 DMA 功能时需要利用 IIC 各种中断进行判断，从而实现 IIC 数据的传输。

I2C 总线接口连接微控制器和串行 I2C 总线。它提供多主机功能，控制所有 I2C 总线特定的时序、协议、仲裁和定时。支持标准和快速两种模式。

### 1.1 主要特点

多主机功能：该模块既可做主设备也可做从设备。

I2C 主设备功能：产生时钟、START 和 STOP 事件。

I2C 从设备功能：可编程的 I2C 硬件地址比较（仅支持 7 位硬件地址）、停止位检测。

根据系统分频，实现不同的通讯速度。 $I2C \text{ 工作时钟} = MCLK / (CLK\_DIV0 + 1)$ ，其中 MCLK 由 SYS\_CLK\_CFG 分频系数决定，CLK\_DIV0 通过 SYS\_CLK\_DIV0 外设时钟分频寄存器 0 配置。

状态标志：发送器/接收器模式标志、字节发送结束标志、I2C 总线忙标志。

错误标志：主模式时的仲裁丢失、地址/数据传输后的应答(ACK)错误、检测到错位的起始或停止条件。

IIC 有一个中断向量，包含五个中断源：总线错误中断源、完成中断源、NACK 中断源、硬件地址匹配中断源和传输完成中断源。

## 1.2 IIC 接口描述

I2C 接口同外界通讯只有 SCL 和 SDA 两根信号线。

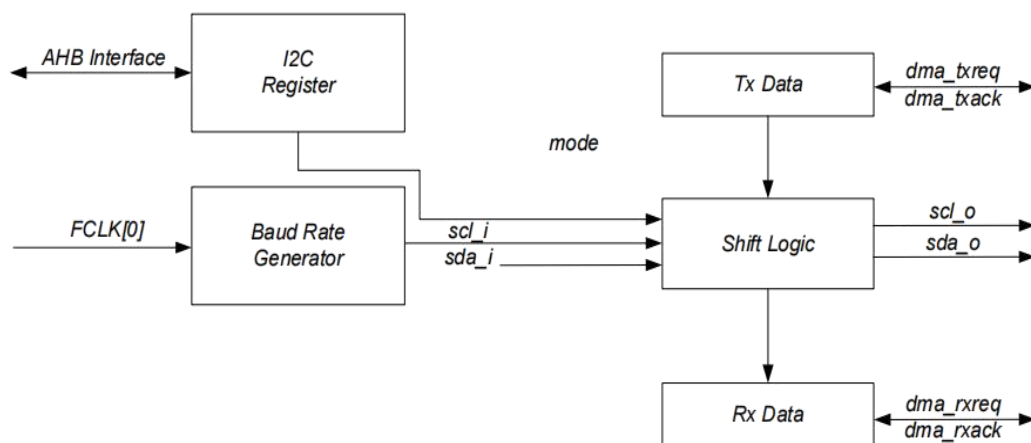


图 16.1 I2C 模块顶层功能框图

**scl\_i:** 时钟信号。当 I2C 接口配置为从模式时，此为 I2C 总线的时钟输入信号。

**sda\_i:** 数据信号。当 I2C 接口接收数据时（无论主模式还是从模式），此为 I2C 总线的数据输入信号。

**scl\_o:** 时钟信号。当 I2C 接口配置为主模式时，此为 I2C 总线的时钟输出信号。

**sda\_o:** 数据信号。当 I2C 接口发送数据时（无论主模式还是从模式），此为 I2C 总线的数据输出信号。

**sda\_oe:** 数据使能信号。当 sda\_o 输出时，sda\_oe 有效；当 sda\_i 输入时，sda\_oe 无效。

## 1.3 IIC 传输时序

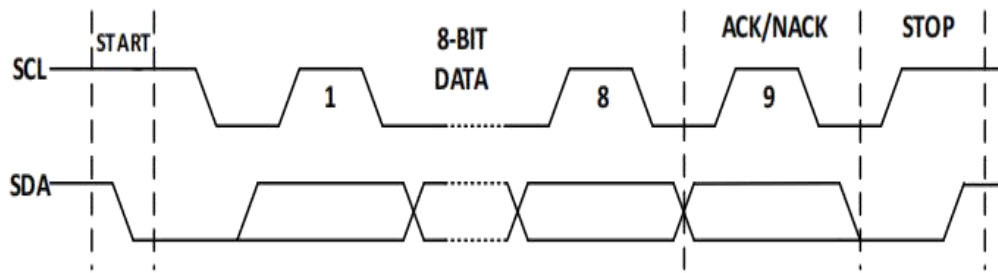


图 16.2 基本 I2C 传输时序图

数据和地址按 8 位/字节进行传输，高位在前。跟在起始条件后的 1 个字节是地址，地址数据中高 7 位为地址数据，最低位为读写控制位，最低位为 0 表示主设备发送数据，为 1 表示设备主设备接收数据。地址只在主模式发送。

在一个字节传输的 8 个时钟后的第 9 个时钟期间，接收器必须回送一个应答位(ACK)给发送器，每个字节后会跟随一个 ACK 信号。ACK bit 使得接收者通知发送者已经成功接收数据并准备接收下一个数据。所有的时钟脉冲包括 ACK 信号对应的时钟脉冲都是由 master 产生的。

**ACK 信号：**发送者在 ACK 时钟脉冲期间释放 SDA 线，接收者可以将 SDA 拉低并在时钟信号为高时保持低电平。

**NACK 信号：**当在第 9 个时钟脉冲的时候 SDA 线保持高电平，就被定义为 NACK 信号。Master 要么产生 STOP 条件来放弃这次传输，或者重复 START 条件来发起一个新的开始。

## 1.3 通讯速度设置

I2C 接口的工作时钟来自系统时钟的分频，分频寄存器为 SYS 模块的 CLK\_DIV0。

I2C 接口采用同步设计，需要对外部设备的信号进行同步采样，同步时钟为 I2C 接口工作时钟。数据和时钟信号的时钟频率为接口工作时钟/16。

$I2C \text{ 模块工作时钟频率} = \text{系统频率} / (CLK\_DIV0 + 1)。$

$I2C \text{ 波特率} = I2C \text{ 模块工作时钟频率} / 17。$

## 1.4 IIC 传输流程

### 1) 主模式传输

每次传输完一个字节的的数据后，将产生中断判断是否还要继续传输。下图为主模式传输的总线示意图。从图可知，流程如下：

- 1、判断总线是否空闲，若空闲，准备开始传输。
- 2、首先，发送从地址，若地址匹配，才继续后续传输，否则停止。
- 3、若是接收模式，一个字节接收完毕后，产生中断，软件判断是否继续接收，返回 ACK/NACK 响应。
- 4、若是发送模式，一个字节发送完毕后，等待响应（ACK/NACK），产生中断，根据响应判断后续操作。
- 5、发送总线 STOP 事件，本次传输完成。

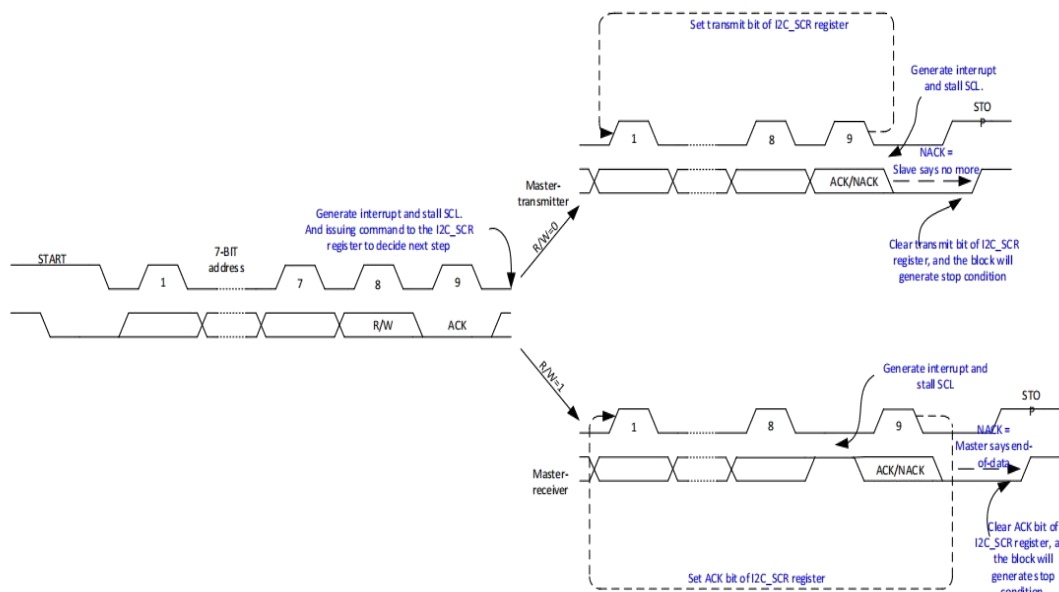


图 16.3 主模式下单字节传输示意图

## 2 寄存器

### 2.1 SYS\_CLK\_DIV0

SYS\_CLK\_DIV0 外设时钟分频寄存器 0

位置	位名称	说明
[31:16]		未使用
[15:0]	DIV0	I2C 工作时钟= $MCLK/(CLK\_DIV0+1)$ 其中 MCLK 由 SYS_CLK_CFG 分频系数决定。

### 16.2.2 SYS\_ADDR

I2C\_ADDR 地址寄存器

位置	位名称	说明
[31:8]		未使用
[7]	ADDR_CMP	I2C 硬件地址比较使能开关，默认值为 0。 0: 关闭 1: 开启 <b>从模式使用，用于自动匹配地址</b>
[6:0]	ADDR	仅用于从模式下，I2C 设备硬件地址。主模式下，从设备地址写入 I2C_DATA 寄存器。

### 2.3 I2C\_CFG

I2C\_ADDR 地址寄存器

位置	位名称	说明
[31:8]		未使用
[7]	IE	I2C 中断使能信号。默认值为 0。 1: 使能 I2C 中断 0: 关闭 I2C 中断
[6]	TC_IE	I2C 数据传输完成中断使能信号。默认值为 0。 1: 使能此中断源 0: 屏蔽此中断源
[5]	BUS_ERR_IE	I2C 总线错误事件中断使能信号。默认值为 0。 1: 使能此中断源 0: 屏蔽此中断源
[4]	STOP_IE	I2CSTOP 事件中断使能信号。默认值为 0。 1: 使能此中断源 0: 屏蔽此中断源
[3:2]		未使用
[1]	MST_MODE	I2C 主模式使能信号。默认值为 0。 1: 使能主模式 0: 关闭主模式
[0]	SLV_MODE	I2C 从模式使能信号。默认值为 0。 1: 使能从模式 0: 关闭从模式

## 2.4 I2C\_SCR

LKS32MC05x 系列芯片因为内部没有 DMA 模块，所以 IIC 的整个发送接收过程都是通过 I2C 中断内判断 I2C\_SCR 状态位进行操作，实现数据的发送和接收操作。

I2C\_SCR 状态控制寄存器

位置	位名称	说明
[31:8]		未使用
[7]	STT_ERR	总线错误状态标志位，用于主模式发送/主模式接收，写 0 清除。 0: 无 START/STOP 总线错误 1: 有 START/STOP 总线错误
[6]	LOST_ARB	总线仲裁丢失状态标志位，用于主模式发送/主模式接收，发生总线仲裁丢失事件将此位置 1，无中断事件产生，在字节完成中断中需查此位。总线上任何 START 事件将导致硬件清除此位。 0: 无总线仲裁丢失错误发生 1: 有总线仲裁丢失错误发生
[5]	STOP_EVT	STOP 事件状态标志位，用于主模式发送/从模式发送/主模式接收/从模式接收。写 0 清除。 0: 无 STOP 事件 1: 有 STOP 事件
[4]	BYTE_CMPLT	ACK 控制位，用于主模式接收/从模式接收。发送方发送完毕当前字节，接收方对此的响应。若是发送方，此位保留 0 值。接收方，根据实际情况配置。 0: 字节发送完成，返回 NACK 回应，表示接收方不能接收更多数据 1: 字节发送完成，返回 ACK 回应，表示接收方可以继续接收数据
[3]	ADDR_DATA	地址数据标志位，用于主模式发送/从模式发送/主模式接收/从模式接收。START 后，第一个字节为地址数据，此位是一个提示位。写 0 清除。 0: 当前传输的数据非地址数据。 1: 当前传输的数据是地址数据。
[2]	DATA_DIR	发送或接收控制位，主模式发送/从模式发送，此位置 1，触发发送，硬件自动清零；主模式接收/从模式接收，此位置 0，等待接收。 0: 接收 1: 触发发送
[1]	RX_ACK	接收响应标志位，用于主模式发送/从模式发送，告知发送方，接收方的反馈。发送方收到反馈后，对该位执行清零操作。 0: 本 I2C 接口发送数据，接收到 ACK 响应。 1: 本 I2C 接口发送数据，接收到 NACK 响应。
[0]	Done	传输完成状态标志位，用于主模式发送/从模式发送/主模式接收/从模式接收。写 0 清除。 0: 传输未完成 1: 传输已完成

## 2.5 I2C\_DATA

IIC 发送的地址和数据都是通过对该寄存器进行写入。

I2C\_DATA 数据寄存器

位置	位名称	说明
[31:8]		未使用
[7:0]	DATA	数据寄存器,用于主模式发送/从模式发送/主模式接收/从模式接收。发送方,写入发送数据;接收方,读取接收数据。注意,地址数据也是数据,主模式只能将要发送地址数据写入此寄存器。

## 2.6 I2C\_MSCR

I2C\_MSCR 主模式寄存器

位置	位名称	说明
[31:4]		未使用
[3]	BUSY	I2C 总线, 闲忙状态。 0: 检测到 STOP 事件, 空闲。 1: 检测到 START 事件, 忙碌。
[2]	MST_CHECK	主模式争抢总线标志位。争抢到总线, 置 1; STOP 事件或者发生总线冲突本模块释放总线, 置 0。
[1]	RESTART	再次触发 START 事件, 写 1 有效。发送 START 完毕, 硬件清 0。I2C_CFG[1]置 1, 才能实现写 1 操作。
[0]	START	触发 START 事件并发送地址数据至总线, 写 1 有效。I2C_CFG[1]置 1, 才能实现写 1 操作。

## 2.7 I2C\_BCR

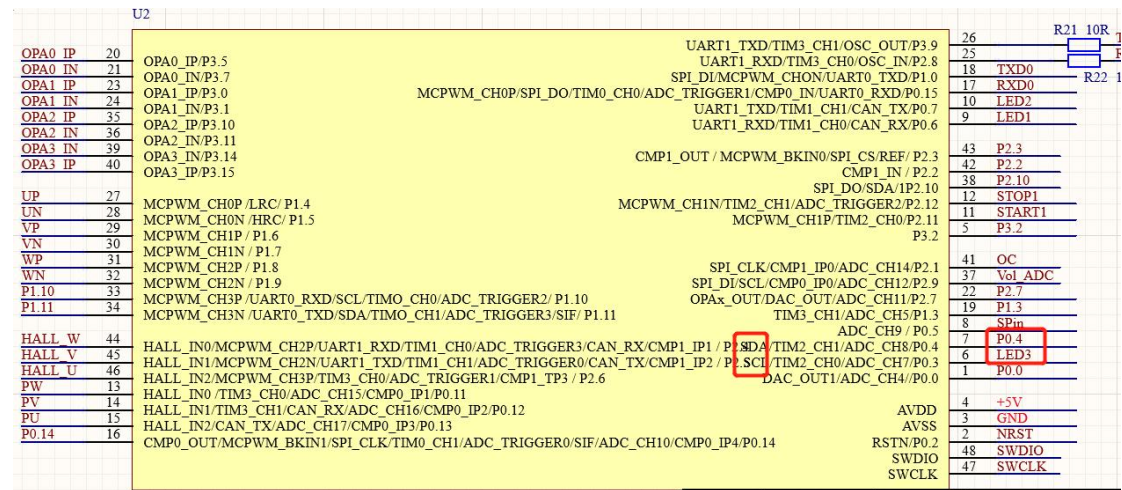
I2C\_BCR 传输控制寄存器

位置	位名称	说明
[31:8]		未使用
[7]	BURST_NACK	I2C 传输, NACK 事件中断使能信号。 1: 使能此中断源 0: 屏蔽此中断源
[6]	BURST_ADDR_CMP	I2C 传输, 硬件地址匹配中断使能信号。 1: 使能此中断源 0: 屏蔽此中断源
[5]	BUSRT_EN	I2C 多数据传输使能, 需要采用 DMA 方式。 1: 使能 0: 关闭
[4]	SLV_DMA_PREF	I2C 多数据传输。从模式执行 DMA 方式发送, 触发硬件预取第一个字节。硬件自动清零。 1: 使能 0: 关闭
[3:0]	BURST_SIZE	I2C 数据传输长度寄存器, 用于多字节传输。 实际传输字节数 = B[3:0] + 1



### 3 硬件电路

LKS32MC08x 的 IIC 硬件需要在 SCL 和 SDA 口接上拉电阻,才能正常通讯。





## 4 程序配置

### 4.1 GPIO 配置

实验中使用 P0.3 为 SCL 时钟信号线，P0.4 为 SDA 数据信号线。主模式下 P0.3 配置为输出模式，从模式配置为输入模式，而主模式存在发送和接收数据，所以 P0.4（SDA）输入输出模式都需要配置。

```
void GPIO_init(void)
{
    // P0.3 SCL P0.4 SDA
    GPIO1_PIE |= BIT10 | BIT11;    // P0.3 和 P0.4 输入使能
    GPIO1_POE |= BIT10 | BIT11;    // P0.3 和 P0.4 输出使能
    // P0.3 复用功能 SCL
    GPIO_PinAFConfig(GPIO1, GPIO_PinSource_10, AF6_I2C);
    // P0.4 复用功能 SDA
    GPIO_PinAFConfig(GPIO1, GPIO_PinSource_11, AF6_I2C);
}
```

### 4.2 IIC 初始化主模式配置

```
void IIC_init(u32 div0)
{
    I2C_InitTypeDef I2C_InitStruct;
    I2C_StructInit(&I2C_InitStruct);
    // I2C 硬件地址比较使能开关，只有在 DMA 模式下开启才有效。
    I2C_InitStruct.ADRCMP = DISABLE ;
    I2C_InitStruct.MST_MODE = ENABLE ; // I2C 主模式使能
    I2C_InitStruct.SLV_MODE = DISABLE ; // I2C 从模式使能
    I2C_InitStruct.DMA = DISABLE ; // I2C DMA 传输使能
    I2C_InitStruct.BaudRate = div0 ; // I2C 波特率
    I2C_InitStruct.IE = ENABLE ; // I2C 中断使能
    I2C_InitStruct.TC_IE = ENABLE ; // I2C 数据传输完成中断使能
    // I2C 总线错误事件中断使能
    I2C_InitStruct.BUS_ERR_IE = DISABLE ;
    I2C_InitStruct.STOP_IE = ENABLE ; // I2C STOP 事件中断使能
    // I2C 传输，NACK 事件中断使能
    I2C_InitStruct.BURST_NACK = ENABLE ;
    // I2C 传输，硬件地址匹配中断使能
    I2C_InitStruct.BURST_ADDR_CMP = DISABLE ;
    I2C_Init(I2C, &I2C_InitStruct);
}
```

```

/* 保存 IIC 波特率设置，在传输错误中断会重新继续硬件初始化*/
I2C_Par.IIC_div_t = div0;
I2C_Par.Idle_RX_Flag = 1; /* I2C 接收时检查空闲标志位初始化*/
I2C_Par.Idle_TX_Flag = 1; /* I2C 发送时检查空闲标志位初始化*/
}

```

IIC 主模式不需要开启硬件地址比较（从模式使用）；初始化时 IIC 发送地址数据不进行配置，后续在发送函数进行配置；初始化主要开启 IIC 主模式使能，和 IIC 的时钟分频设置；然后就是使能 IIC 的各种中断，使用主要用到 NACK，数据传输完成，停止时间（STOP）事件中断使能，总线错误中断本实验没有使用，主要因为本实验主要针对数据的发送和接收进行实验，不同的应用错误处理不一样，无法做到兼容，所以索性就没有对错误事件进行处理。

### 4.3 IIC 发送函数配置

IIC MCU 发送函数最主要就是将 7bit 地址数据移位到高 7 位，最低位写 0（写指令）操作。其它只是辅助作用，I2C\_Par.Tran\_Mode 状态变量主要是 IIC 发送需要 MCU 搬运时，所有判断处理需要软件在中断内判断 IIC 状态位完成，所以为了在一个例程中提供仅数据发送或者仅数据接收或者发送一字节数据后再转为接收一字节数据模式才定义的这个变量（不同模式处理方法不一样，具体可看中断函数）。其它对变量的写入操作只是为了在中断内发送字节提供状态使用。

addr:地址。

i2c\_data:发送数据缓冲区。

len:发送字节个数。

mode:IIC 工作模式，1:主模式只接收或者只发送，2:主模式发送数据后立刻转为接收，或者接收数据后立刻转为发送。

```

u8 I2C_TX_Function(u8 addr, u8 *i2c_data, u32 len, u8 mode)
{
    I2C_Par.Tran_Mode = mode;           //发送模式确认
    if (!i2c_delay_txok())               //等待 I2C 发送完成
    {
        I2C_Par.Data_Length_TX = len;
        I2C_Par.I2C_DATA_TX = i2c_data;
        I2C_Par.IIC_ADDR = addr << 1;
        I2C_Par.Data_Temp_Length_TX = 0;
        I2C_DATA = I2C_Par.IIC_ADDR; // 地址信号
        I2C_MSCR |= BIT0;             // 触发 I2C 发送地址
        I2C_Par.I2C_Mode = 0;         // 发送模式
        return 0x0;                   // 发送成功
    }
    else {
        return 0xff; /*发送失败*/
    }
}

```

#### 4.4 IIC 接收函数配置

接收函数主要功能也是对地址的写入，先对 7bit 地址数据移位后最低为写 1（表示接收数据），其它操作与发送函数一样。

```

u8 I2C_RX_Function(u8 addr, u8 *i2c_data, u32 len, u8 mode)
{
    I2C_Par.Tran_Mode = mode;
    if (!i2c_delay_rxok())               //等待 I2C 接收完成
    {
        I2C_Par.IIC_ADDR = addr << 1;
        I2C_Par.Data_Length_RX = (len - 1);
        I2C_Par.I2C_DATA_RX = i2c_data;
        I2C_Par.Data_Temp_Length_RX = 0;
        I2C_DATA = I2C_Par.IIC_ADDR | 0x01; // 地址信号
        I2C_MSCR |= BIT0;                   // 触发 I2C 发送地址
        I2C_Par.I2C_Mode = 1;               // 接收模式
        return 0x0;                         // 发送成功
    }
    else
    {
        return 0xff; /*发送失败*/
    }
}

```

## 4.5 IIC 中断函数配置

```
void I2C0_IRQHandler(void)
{
    switch (I2C_SCR){
    case 0x01://byte complete
        //last receive data, len will stop at 8
        if (I2C_Par.Data_Temp_Length_RX >= I2C_Par.Data_Length_RX)
        {
            I2C_Par.I2C_DATA_RX[I2C_Par.Data_Temp_Length_RX] = I2C_DATA;
            I2C_SCR = 0x00;//字节接收完成, 返回 NACK 回应
            break;
        }
        I2C_Par.I2C_DATA_RX[I2C_Par.Data_Temp_Length_RX] = I2C_DATA;
        I2C_Par.Data_Temp_Length_RX += 1;
        I2C_SCR = BIT4;    //字节接收完成, 返回 ACK 回应
        break;
    case 0x05://发送模式 and I2C 数据发送完成 且接收到 ACK 信号
        //传输最后 1Byte 数据 I2C_Par.Data_Length
        if (I2C_Par.Data_Temp_Length_TX >= I2C_Par.Data_Length_TX)
        {
            I2C_SCR = 0x00; /*清零 SCR 寄存器触发 STOP 信号*/
            break;
        }else{
            //再次发送数据
            I2C_DATA = I2C_Par.I2C_DATA_TX[I2C_Par.Data_Temp_Length_TX];
            I2C_Par.Data_Temp_Length_TX += 1;
            I2C_SCR = BIT2; /*触发 I2C 发送数据*/
            break;
        }
    case 0x07: // 发送, 字节完成和接收到 NACK
        /*发送模式, 检查从机返回 NACK, 该内容根据实际情况进行处理*/
        I2C_SCR = 0x00;
        break;
    case 0x09: //当前传输为地址数据且接收到 ACK 响应
        if (!I2C_Par.I2C_Mode) //发送模式
        {
            // transmit first data
            I2C_DATA = I2C_Par.I2C_DATA_TX[I2C_Par.Data_Temp_Length_TX];
            I2C_Par.Data_Temp_Length_TX += 1;
            I2C_SCR = BIT2; /*触发 I2C 发送数据*/
        }
    }
```

```

else
{ //接收模式
    I2C_SCR = 0x00;
}
break;
case 0x0B: // slave address not match;
    /*检查从模式地址匹配错误事件，该内容根据实际情况进行处理*/
    I2C_SCR = 0x00;
break;
case 0x20: // stop;
    if (I2C_Par.Tran_Mode == I2C_TX_OR_RX)
    {
        I2C_Par.Idle_TX_Flag = 1;
        I2C_Par.Idle_RX_Flag = 1;
    }
    else if (I2C_Par.Tran_Mode == I2C_TX_AND_RX)
    {
        if (!I2C_Par.I2C_Mode) //发送模式
        {
            I2C_Par.Idle_RX_Flag = 1;
        }
        else
        {
            I2C_Par.Idle_TX_Flag = 1;
        }
    }

    I2C_SCR = 0x00;
    break;
default:
    IIC_init(I2C_Par.IIC_div_t); //传输错误，复位 IIC
    I2C_SCR = 0x00;
    break;

```

中断函数处理看起来很长，其实我们把它拆分看就很容易了，主要可拆分为发送数据和接收数据处理两部分。

首先在 I2C\_TX\_Function（）函数内触发 IIC 发送一字节地址数据，此时如果 IIC 总线产生 START 后，第一个字节为地址数据，此时 I2C\_SCR BIT3 会发生置位，我们通过发送完成进入中断，如果第一个地址数据发送返回 NACK，则 I2C\_SCR=0x07;本实验直接 I2C\_SCR 清零后，硬件会自动产生 STOP 信号，然

后产生 STOP 中断,在 STOP 中断内根据发送模式对相应的状态变量置位, 表明 I2C 数据发送完成, 总线处于空闲状态(用户可以根据自己的应用在接口内添加自己的操作)。如果第一个地址数据发送返回 ACK,则中断内判断 I2C\_SCR=0x09;即数据发送完成且第一个发送字节为地址且接收到 ACK 应答, 然后继续要发送数据第一个字节的发送; 如果第一个数据发送成功从设备返回 ACK 应答信号, 则产生发送完成中断, 在中断内判断 I2C\_SCR=0x05;数据触发发送, 发送完成且接收到 ACK 信号, 则对 I2C\_DATA 写入下一字节发送数据, 操作 I2C\_SCR 的 BIT2 位触发发送数据, 直到发送最后一个直接后清零 I2C\_SCR, 硬件自动产生 STOP 信号, 在中断判断 I2C\_SCR BIT5 有 STOP 事件结束本次发送, 如果任意字节发送接收到 NACK 信号, 就会进入 I2C\_SCR=0x07; 软件对 I2C\_SCR 清零然后产生 STOP 信号结束发送。

主模式接收时, 在 I2C\_RX\_Function () 函数内触发 IIC 发送一字节地址数据, 此时如果 IIC 总线产生 START 后, 第一个字节为地址数据, 此时 I2C\_SCR BIT3 会发生置位, 我们通过发送完成进入中断, 如果第一个地址数据发送返回 NACK, 则操作与发送数据一样; 如果第一个数据发送成功从设备返回 ACK 应答信号, 将 I2C\_SCR 的 BIT2 清零等待数据接收。当主模式接收一字节数据后, 参数接收完成中断, 在中断内判断 I2C\_SCR=0x01;然后读取 I2C\_DATA 数据存储到接收数组内, 然后将 I2C\_SCR 的 BIT4 置 1 硬件产生 ACK 应答信号, 继续接收从机数据, 直到主模式接收字节设定的最后一个字节后将 I2C\_SCR 的 BIT4 清零硬件产生 NACK 信号, 告诉不在需要数据, 最后主设备产生 STOP 事件和 STOP 中断, 在中断将对应接收状态变量置 1, 表示数据接收完成, 总线处于空闲状态。

STOP 状态中根据不同模式对发送状态变量和接收状态变量置 1, 该变量主要用于在执行 I2C\_TX\_Function () 和 I2C\_RX\_Function () 函数时判断 IIC 总线是否空闲, 如果置 1 表示总线空闲, 可以进行地址数据的发送, 而 I2C\_TX\_OR\_RX 是支持发送和接收数据操作间隔时使用, 即发生一字节后紧跟着接收从机一字节或者接收从机一字节后紧跟着发送一字节时使用。三种模式(仅发送数据, 仅接收数据, 发送接收交替进行模式), 对应状态变量的不同处理, 便于在程序中来回使用三种模式, 使能不会对总线空闲产生误判。



#### 4.6 等待 IIC 发送空闲函数配置

等待 IIC 发送空闲函数的核心思想就是判断对应完成变量是否置 1，如果置 1，将发送和接收状态变量初始化清零，返回 0，表示总线空闲，如果长时间对应状态标志位还未置 1，为了防止程序卡死，t 临时变量计数到 0xffff 反馈 0xff，表示此时总线忙碌。

```
u8 i2c_delay_txok(void)
{
    u16 t = 0;
    if (I2C_Par.Tran_Mode == I2C_TX_OR_RX)
    {
        // 等待 I2C 空闲;判断 I2C_Par.FF 置 1
        while((!I2C_Par.Idle_RX_Flag)||(!I2C_Par.Idle_TX_Flag))
        {
            t++;
            if (t == 0xffff)
            {
                //避免程序卡死
                return 0xff; /*总线忙*/
            }
        }
    }
    else
    {
        while (!I2C_Par.Idle_TX_Flag) // 等待 I2C 空闲;判断 I2C_Par.FF 置 1
        {
            t++;
            if (t == 0xffff)
            {
                //避免程序卡死
                return 0xff; /*发送失败*/
            }
        }
        I2C_Par.Idle_TX_Flag = 0;
        I2C_Par.Idle_RX_Flag = 0;
        return 0x0; /*总线空闲/
    }
}
```

#### 4.7 等待 IIC 接收空闲函数配置

等待 IIC 接收空闲函数的核心思想与等待发送空闲函数一样。

```

u8 i2c_delay_txok(void)
{
    u16 t = 0;
    if (I2C_Par.Tran_Mode == I2C_TX_OR_RX)
    {
        // 等待 I2C 空闲;判断 I2C_Par.FF 置 1
        while((!I2C_Par.Idle_RX_Flag)||(!I2C_Par.Idle_TX_Flag))
        {
            t++;
            if (t == 0xffff)
            {
                //避免程序卡死
                return 0xff; /*总线忙*/
            }
        }
    }
    else
    {
        while (!I2C_Par.Idle_RX_Flag) // 等待 I2C 空闲;判断 I2C_Par.FF 置 1
        {
            t++;
            if (t == 0xffff)
            {
                //避免程序卡死
                return 0xff; /*发送失败*/
            }
        }
        I2C_Par.Idle_TX_Flag = 0;
        I2C_Par.Idle_RX_Flag = 0;
        return 0x0; /*总线空闲/
    }
}

```

#### 4.8 主函数配置

主模式主要通过全局变量 Test\_Pattern 赋值不同执行 IIC 不同模式，可通过 debug 进行操作 Test\_Pattern 变量。

```

u8 I2C_TX_BUFF[8] = {0x75,0x11,0x22,0x88,0x44,0x55,0x66,0x77};
u8 I2C_RX_BUFF[8] = {0};
u8 Test_Pattern = 0; /*三种模式测试*/
int main(void)
{
    Hardware_init(); /* 硬件初始化 */
    for (;;)
    {
        if(Test_Pattern == 0)/*发送接收：轮流执行*/
        {
            I2C_TX_Function(0x3C,I2C_TX_BUFF,8,I2C_TX_AND_RX);
            I2C_RX_Function(0x3C,I2C_RX_BUFF,8,I2C_TX_AND_RX);
        }
        if(Test_Pattern == 1)/*只发送*/
        {
            I2C_TX_Function(0x3C,I2C_TX_BUFF,8,I2C_TX_OR_RX);
        }
        if(Test_Pattern == 2)/*只接收*/
        {
            I2C_RX_Function(0x3C,I2C_RX_BUFF,8,I2C_TX_OR_RX);
        }
        SoftDelay(100);
    }
}

```

## 5 下载验证

将例程编译下载后，051 芯片接口与从设备进行硬件连接，通过 Debug 模式操作 `Test_Pattern` 进行数据接收与发送验证。

主设备发送与接收数据：

Test_Pattern	0
I2C_TX_BUFF	0x20000000 I2C_TX_BU...
[0]	0x75 'u'
[1]	0x11
[2]	0x22 ''''
[3]	0x88 '?'
[4]	0x44 'D'
[5]	0x55 'U'
[6]	0x66 'f'
[7]	0x77 'w'
I2C_RX_BUFF	0x20000008 I2C_RX_B...
[0]	0x11
[1]	0x22 ''''
[2]	0x33 '3'
[3]	0x44 'D'
[4]	0x55 'U'
[5]	0x66 'f'
[6]	0x77 'w'
[7]	0x88 '?'

从设备发送与接收数据：

I2C_TX_BUFF	0x20000018 I2C_TX_BU...
[0]	0x11
[1]	0x22 ''''
[2]	0x33 '3'
[3]	0x44 'D'
[4]	0x55 'U'
[5]	0x66 'f'
[6]	0x77 'w'
[7]	0x88 '?'
I2C_RX_BUFF	0x20000008 I2C_RX_B...
[0]	0x75 'u'
[1]	0x11
[2]	0x22 ''''
[3]	0x88 '?'
[4]	0x44 'D'
[5]	0x55 'U'
[6]	0x66 'f'
[7]	0x77 'w'