			10 :字 11:保留	
RxTrigger	[16:11]	读/写	在中断模式，接收FIFO触发电平从 6’h0～6’h40。该值是接收FIFO的字节数。	6’b0
TxTrigger	[10:5]	读/写	在中断模式，发送FIFO触发电平从 6’h0～6’h40。该值是发送FIFO的字节数。	6’b0
reserved	[4:3]	-	-	-
RxDMA On	[2]	读/写	DMA模式打开/关闭。 0 : DMA 模式关闭 1 : DMA 模式打开	1’b0
TxDMA On	[1]	读/写	DMA模式打开/关闭 0 : DMA 模式关闭 1 : DMA 模式打开	1’b0
DMA transfer	[0]	读/写	DMA 传输类型，单个或者四个脉冲。 0 : 单个 1 : 4 个脉冲 设置DMA传输大小必须和SPI中一样。	1’b0

通道传输大小应当小于或等于总线传输大小。

寄存器	地址	读/写	描述	初始值
Slave_slection_reg(Ch0)	0x7F00B00C	读/写	从属器选择寄存器。	0x1
Slave_slection_reg(Ch1)	0x7F00C00C	读/写	从属器选择寄存器。	0x1

Slave_slection_reg	位	读/写	描述	初始状态
nCS_time_count	[9:4]	读/写	nSSout 无效时间 = ((nCS时间计数 +3)/2) × SPICLKout)	6’b0
reserved	[3:2]		保留。	
Auto_n_Manual	[1]	读/写	芯片选择设置分手动或自动选择。 0: 手动 1: 自动	1’b0
nSSout	[0]	读/写	从属器选择信号（只用于手动） 0: 有效 1: 无效	1’b1

寄存器	地址	读/写	描述	初始值
SPI_INT_EN(Ch0)	0x7F00B010	读/写	SPI中断启动寄存器。	0x0
SPI_INT_EN(Ch1)	0x7F00C010	读/写	SPI中断启动寄存器。	0x0

SPI_INT_EN	位	读/写	描述	初始状态
IntEnTrailing	[6]	读/写	用于结尾计数设置为0的中断启动。 0: 无效 1:有效	1’b0
IntEnRxOverrun	[5]	读/写	用于接收超限运行的中断启动。 0: 无效 1:有效	1’b0
IntEnRxUnderrun	[4]	读/写	用于接收欠载运行的中断启动。 0: 无效 1:有效	1’b0
IntEnTxOverrun	[3]	读/写	用于发送超限运行的中断启动。 0: 无效1:有效	1’b0
IntEnTxUnderrun	[2]	读/写	用于发送欠载运行的中断启动。在从属器模式，转向从属器发送路径后，该位必须首先清除。 0:无效 1:有效	1’b0
IntEnRxFifoRdy	[1]	读/写	用于 RxFifoRdy(中断模式)的中断启动。 0: 无效 1:有效	1’b0
IntEnTxFifoRdy	[0]	读/写	用于TxFifoRdy(中断模式)的中断启动。 0: 无效 1:有效	1’b0

寄存器	地址	读/写	描述	初始值
SPI_STATUS(Ch0)	0x7F00B014	读	SPI状态寄存器。	0x0
SPI_STATUS(Ch1)	0x7F00C014	读	SPI状态寄存器。	0x0

SPI_STATUS	位	读/写	描述	初始状态
TX_done	[21]	读	表示传输完成。 0 ： 所有的情况除了熔断情况	1’b0

			1 ： 发送FIFO和移位寄存器为空时	
Trailing_byte	[20]	读	表示结尾计数是0。	1’b0
RxFifoLvl	[19:13]	读	数据水平在接收 FIFO。 0～7’h40 字节	7’b0
TxFifoLvl	[12:6]	读	数据水平在发送 FIFO。 0～7’h40字节	7’b0
RxOverrun	[5]	读	接收FIFO超限错误。 0：无错误，1：错误	1’b0
RxUnderrun	[4]	读	接收 FIFO欠载运行错误。 0：无错误，1：错误	1’b0
TxOverrun	[3]	读	发送 FIFO 超限错误。 0：无错误r，1：错误	1’b0
TxUnderrun	[2]	读	发送 FIFO欠载运行错误。	1’b0
RxFifoRdy	[1]	读	0 ： FIFO中数据少于触发器电平。 1 :FIFO中数据多于触发器电平。	1’b0
TxFifoRdy	[0]	读	0 ： FIFO中数据多于触发器电平。 1 ： FIFO中数据少于触发器电平。	1’b0

寄存器	地址	读/写	描述	初始值
SPI_TX_DATA(Ch0)	0x7F00B018	写	SPI发送数据寄存器。	0x0
SPI_TX_DATA(Ch1)	0x7F00C018	写	SPI发送数据寄存器。	0x0

SPI_TX_DATA	位	读/写	描述	初始状态
TX_DATA	[31:0]	写	该区域包含要通过SPI通道发送的数据。	32’b0

寄存器	地址	读/写	描述	初始值
SPI_RX_DATA(Ch0)	0x7F00B01C	读	SPI 接收数据寄存器。	0x0
SPI_RX_DATA(Ch1)	0x7F00C01C	读	SPI 接收数据寄存器。	0x0

SPI_RX_DATA	位	读/写	描述	初始状态
RX_DATA	[31:0]	读	该区域包含通过SPI通道被接收到的数据。	32’b0

寄存器	地址	读/写	描述	初始值
Packet_Count_reg(Ch0)	0x7F00B020	读/写	计数，主控器收到多少数据。	0x0
Packet_Count_reg(Ch1)	0x7F00C020	读/写	计数，主控器收到多少数据。	0x0

Packet_Count_reg	位	读/写	描述	初始状态
Packet_Count_En	[16]	读/写	启动位，用于信息包计数。 0：无效 1：有效	1’b0
Count Value	[15:0]	读/写	包计数值。	16’b0

寄存器	地址	读/写	描述	初始值
SWAP_CFG(Ch0)	0x7F00B028	读/写	交换配置寄存器。	0x0
SWAP_CFG(Ch1)	0x7F00C028	读/写	交换配置寄存器。	0x0

SWAP_CFG	位	读/写	描述	初始状态
RX_Half-word swap	[7]	读/写	0: 关闭 1: 交换	1’b0
RX_Byte swap	[6]	读/写	0: 关闭 1: 交换	1’b0
RX_Bit swap	[5]	读/写	0: 关闭 1: 交换	1’b0
RX_SWAP_en	[4]	读/写	交换启动。 0 : 正常 1 : 交换	1’b0
TX_Half-word swap	[3]	读/写	0: 关闭 1: 交换	1’b0
TX_Byte swap	[2]	读/写	0: 关闭 1: 交换	1’b0
TX_Bit swap	[1]	读/写	0: 关闭 1: 交换	1’b0

TX_SWAP_en	[0]	读/写	交换启动。 0 ： 正常 1 ： 交换	1’b0

寄存器	地址	读/写	描述	初始值
FB_Clk_sel (Ch0)	0x7F00B02C	读/写	反馈时钟选择寄存器。	0x3
FB_Clk_sel (Ch1)	0x7F00C02C	读/写	反馈时钟选择寄存器。	0x3

SWAP_CFG	位	读/写	描述	初始状态
SPICLKout delay	[2]	读/写	0 ： 没有额外延迟 1 ： 2.7ns 延迟（基于典型）	1’b0
FB_Clk_sel	[1:0]	读/写	00 ： 0ns额外延迟 01 ： 3ns额外延迟 10 ： 6ns额外延迟 11 ： 9ns额外延迟 *延迟基于典型情况。	2’b3

29.5 SPI 接口应用举例

本小节主要介绍 SPI 在 ARM11 处理器中的编程实现，结合以上对 SPI 接口的理解，相信读者很容易地掌握 SPI 接口的功能及特性。

以下是 SPI 部分代码的具体实现：

1. SPI 复位：功能是复位某一个 SPI 通道。

输入：SPI_channel

输出：NONE.

```
void SPI_reset( SPI_channel * ch ) {
// 带有时钟延迟的复位
    Outp32( &ch->m_cBase->ch_cfg, Inp32(&ch->m_cBase->ch_cfg) & ~(0x3F) ); // 清除寄存器
    Outp32( &ch->m_cBase->ch_cfg, Inp32(&ch->m_cBase->ch_cfg) | (1<<5) );
    Delay(10);
```

```
    // 释放复位信号
    Outp32( &ch->m_cBase->ch_cfg, Inp32(&ch->m_cBase->ch_cfg) & ~(1<<5) );
```

2. SPI 通道初始化：功能是初始化某一个 SPI 通道。

输入：SPI_channel

输出：NONE.

```
SPI_channel* SPI_channel_Init( int channel ) {
    SPI_channel* ch = &SPI_current_channel[channel];
    memset ( (void*)ch, 0, sizeof(SPI_channel) );
    ch->m_ucChannelNum = channel;
    if ( channel == 0 ) {
        ch->m_cBase = (SPI_SFR*)SPI0_BASE;
        ch->m_ucIntNum = NUM_SPI0;
        ch->m_fDMA= SPI_DMADoneChannel0;
        ch->m_fISR = SPI_interruptChannel0;
#ifdef SPI_NORMAL_DMA
        ch->m_ucDMACon = DMA0;
        SYSC_SelectDMA( eSEL_SPI0_TX, 1 );    // 标准的 DMA 设置
        SYSC_SelectDMA( eSEL_SPI0_RX, 1 );    // 标准的 DMA 设置
#else
        ch->m_ucDMACon = SDMA0;
        SYSC_SelectDMA( eSEL_SPI0_TX, 0 );    // 安全的 DMA 设置
        SYSC_SelectDMA( eSEL_SPI0_RX, 0 );    // 安全的 DMA 设置
#endif
    }
    else if ( channel == 1 ) {
        ch->m_cBase = (SPI_SFR*)SPI1_BASE;
        ch->m_ucIntNum = NUM_SPI1;
        ch->m_fDMA = SPI_DMADoneChannel1;
```

```
            ch->m_fISR = SPI_interruptChannel1;
#ifdef SPI_NORMAL_DMA
            ch->m_ucDMACon = DMA1;
            SYSC_SelectDMA( eSEL_SPI1_TX, 1 );    // 标准的 DMA 设置
            SYSC_SelectDMA( eSEL_SPI1_RX, 1 );    // 标准的 DMA 设置
#else
            ch->m_ucDMACon = SDMA1;
            SYSC_SelectDMA( eSEL_SPI1_TX, 0 );    // 安全的 DMA 设置.
            SYSC_SelectDMA( eSEL_SPI1_RX, 0 );    // 安全的 DMA 设置.
#endif
        }
        else {
            Assert(0);
        }
        Outp32(&ch->m_cBase->slave_sel, Inp32(&ch->m_cBase->slave_sel) | (1<<0) );          // 片选 OFF – 激
活 LOW.
        SPI_GPIOPortSet(channel);  // 设置通道 GPIO.
        return ch;
}
```

3. SPI 基本寄存器的设置

输入：SPI_channel

输出：NONE

```
void SPI_setBasicRegister( SPI_channel* ch ) {
Outp32( &ch->m_cBase->ch_cfg,       //清除寄存器
(ch->m_eClockMode<<4)|              // 主/从模式
        (ch->m_eCPOL<<3)|                   // CPOL 高态有效/行
        (ch->m_eCPHA<<2) );                 // CPHA 传输格式

        Outp32( &ch->m_cBase->clk_cfg, (Inp32(&ch->m_cBase->clk_cfg) & ~(0x7ff))|
```

```
                                                                    //清除寄存器
      (ch->m_eClockSource<<9) |                                     //时钟设置
( ( (ch->m_eClockMode==SPI_MASTER)?(1):(0) )<<8) |        //时钟使能
ch->m_cPrescaler);                                                  //  预定标器设置

    Outp32( &ch->m_cBase->mode_cfg, (Inp32(&ch->m_cBase->mode_cfg)&(u32)(1<<31)) |
                                        //清除寄存器
(ch->m_eChSize<<29)|                //通道传输大小
(ch->m_uTraillingCnt<<19)|          // trailling 计数
(ch->m_eBusSize<<17)|               //  总线传输大小
      (ch->m_ucRxLevel<<11)|              // Rx  触发级
(ch->m_ucTxLevel<<5)|           // Tx  触发级
(ch->m_eDMAType<<0) );              // DMA  类型
}
```

30 IIC 总线接口

这一节主要讲述S3C6410 RISC中IIC总线接口的功能和使用方法。

30.1 IIC 总线接口概述

S3C6410 RISC处理器能支持一个多主控器IIC串行接口。一个专用的串行数据线（SDA)和一个串行时钟线(SCL)在总线主控器和连接到IIC总线的外部设备之间传输数据。SDA和SCL线是双向的。

在多主控制IIC总线模式下，多个S3C6410 RISC处理器能发送（或接收）串行数据到从属设备。主控器S3C6410能开始和结束IIC总线上的数据传输。在S3C6410中IIC总线使用标准的总线仲裁程序。

为了控制多个IIC总线操作，必须将值写入下面的寄存器：

- 多主控器IIC总线控制寄存器，IICCON；
- 多主控器IIC总线控制/状态寄存器，IICSTAT；
- 多主控器IIC总线发送/接收数据移位寄存器，IICDS；
- 读主控器IIC总线地址寄存器，IICADD。

当IIC总线空闲，SDA和SCL线必须是高电平。SDA从高到低转换能启动一个开始条件。当SCL处于高电平，保持稳定时，SDA从低位到高位传输能启动一个停止条件。

主设备能一直产生开始和停止条件。开始条件产生后，主控器通过在第一次输出的数据字节中写入7位的地址来选择从属器设备。第8位用于确定传输方向（读或写）。

在到SDA总线上的每一个数据字节总数上必须是8位。在总线传输操作期间，发送或接收字节没有限制。数据一直是先从最高有效位（MSB）发送，并且每个字节后面必须立即跟随确认（ACK）位。

IIC总线模块图，如图30-1所示。

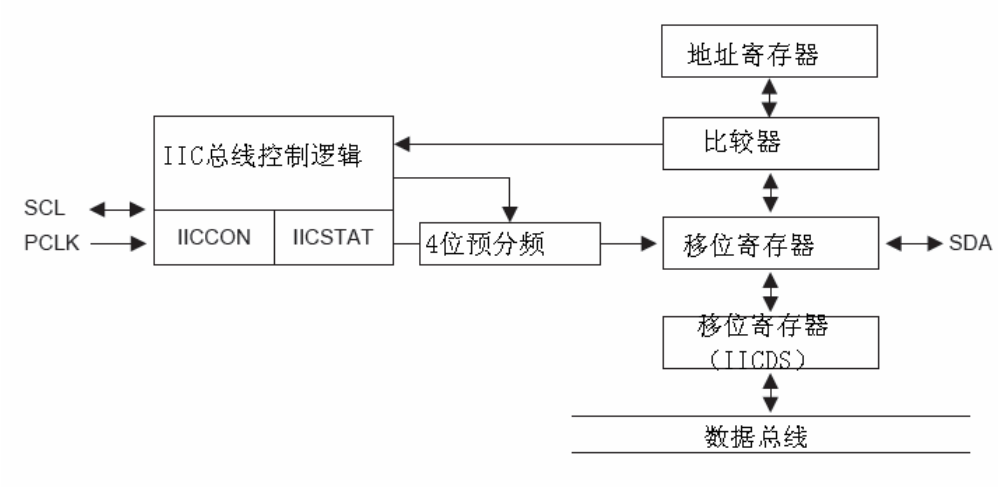

图 30-1　IIC 总线模块图

30.2 IIC 总线接口操作模式

S3C6410 IIC总线接口有四种操作模式：

- 主控器发送模式；
- 主控器接收模式；
- 从属器发送模式；
- 从属器接收模式。

这些操作模式之间的功能关系描述如下：

1. 开始和停止条件

IIC总线接口无效时，它通常是在从属器模式下。换句话说就是，在SDA线检测一个开始条件前（当时钟信号SCL在高位时，SDA线发生高位到低位的跃变，开始条件启动），接口必须在从属器模式下。当接口状态变为主控器模式时，在SDA线上的数据传输开始，并且产生SCL信号。

开始条件能通过SDA线传输一个字节的串行数据。一个停止条件能结束该数据传输。由主控器能一直产生开始和停止条件。当一个开始条件产生后，IIC总线获得繁忙信号。停止条件将使IIC总线空闲。

当主控器发起一个开始条件，它将发送一个从属地址来通知从属器设备。一个字节的地址域包含7位地址和1位传输方向指示器（表示写或读）。如果位8是0，表示写操作（发送操作）；如果位8是1，表示请求读取数据（接收操作）。