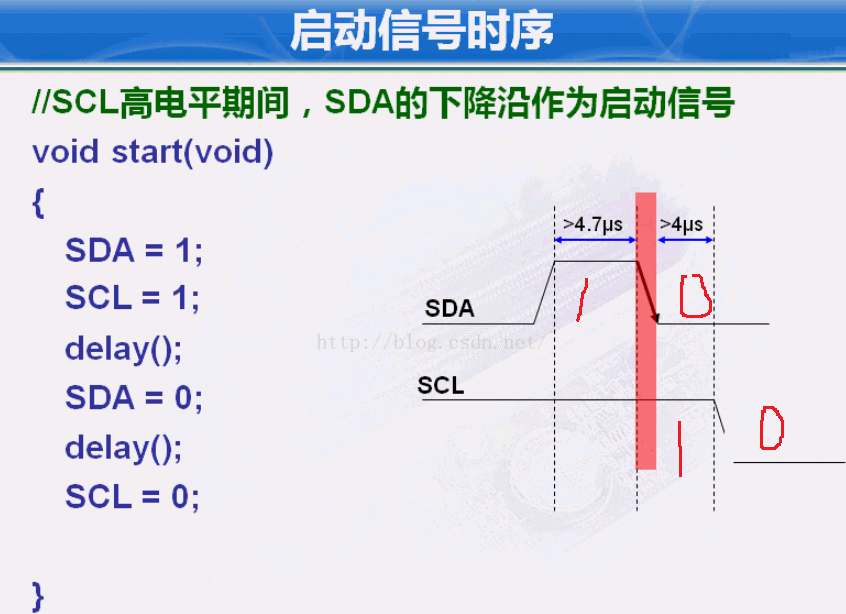
# I2c协议

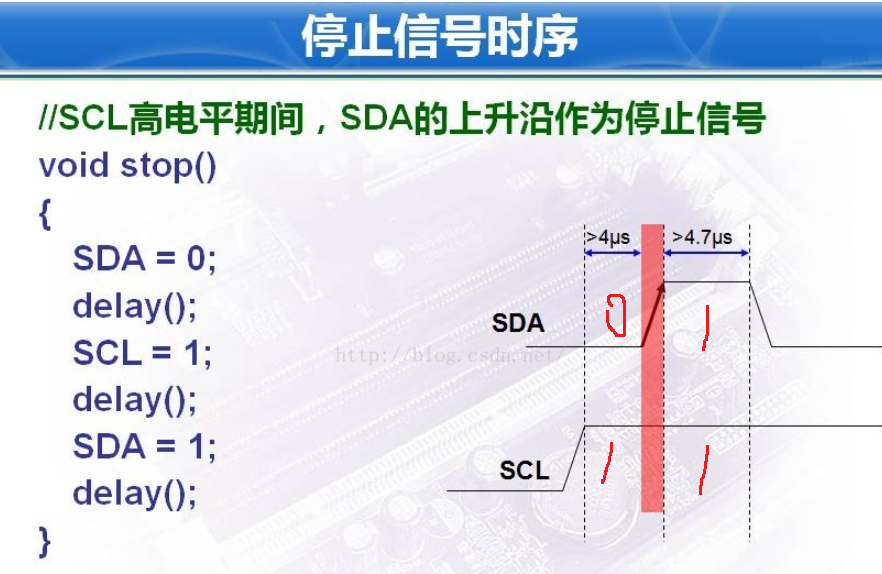
## I2c总线时序



## 启动信号



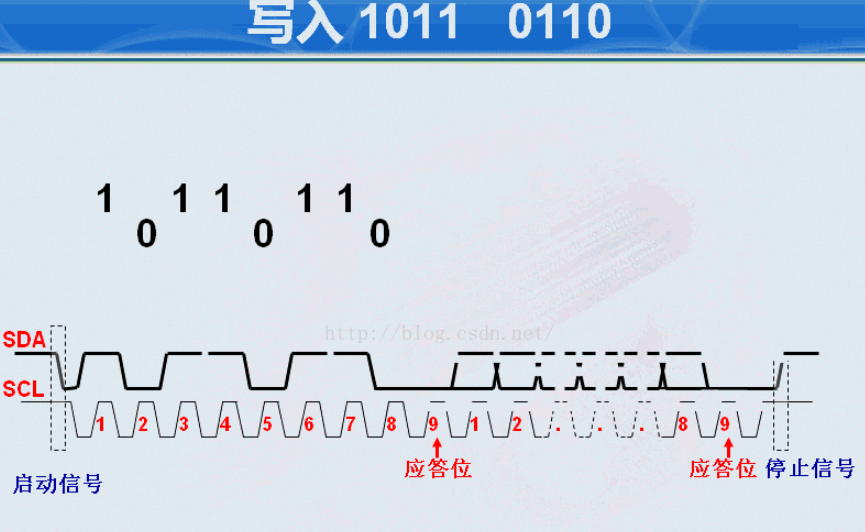
## 停止信号



## 字节传输



## 写入 1011 0110



## IIC发应答和检测应答

MASTER 指主控制端，在一般系统中就是我们常说的单片机了；SLAVE是指具备I2C协议的专用IC

1，**SCL一直由Master控制**，**SDA依照数据传送的方向，读数据时由Slave控制SDA，写数据时由Master控制SDA。当8位数据传送完毕之后，应答位或者否应答位的SDA控制权与数据位传送时相反。**

2，**开始位“Start”和停止位“Stop”，只能由Master来发出。**

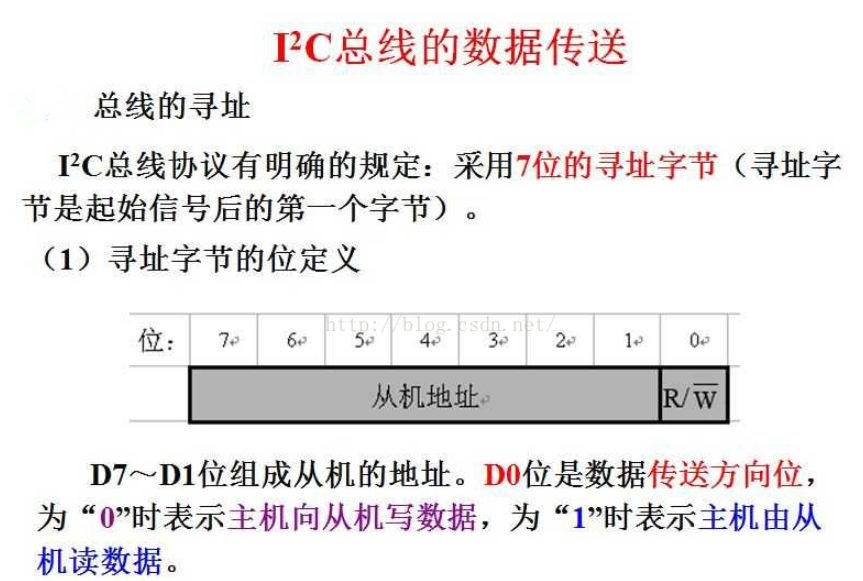
3，**地址的8位传送完毕后，成功配置地址的Slave设备必须发送“ACK”。否则否则一定时间之后Master视为超时，将放弃数据传送，发送“Stop”。**

4，**当写数据的时候，Master每发送完8个数据位，Slave设备如果还有空间接受下一个字节应该回答“ACK”，Slave设备如果没有空间接受更多的字节应该回答“NACK”，Master当收到“NACK”或者一定时间之后没收到任何数据将视为超时，此时Master放弃数据传送，发送“Stop”。**

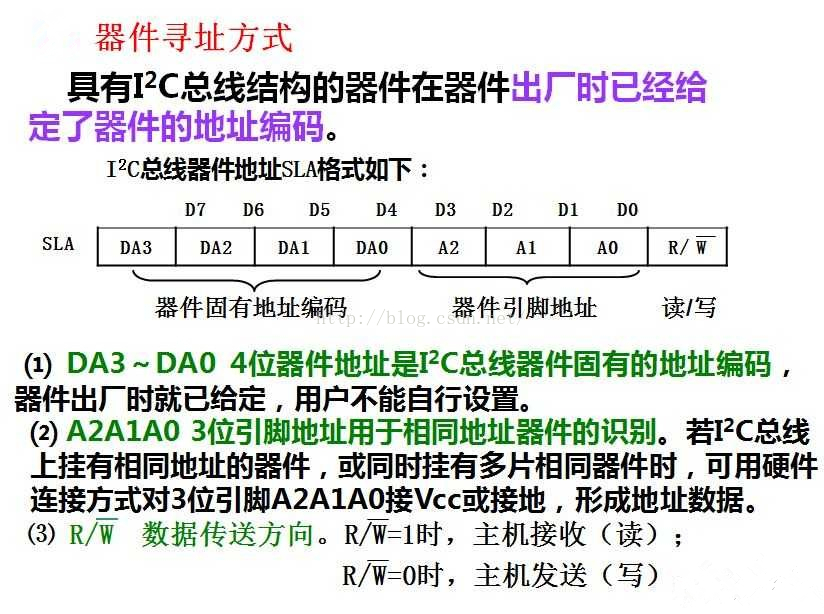
5，**当读数据的时候，Slave设备每发送完8个数据位，如果Master希望继续读下一个字节，Master应该回答“ACK”以提示Slave准备下一个数据，如果Master不希望读取更多字节，Master应该回答“NACK”以提示Slave设备准备接收Stop信号。**

6，**当Master速度过快Slave端来不及处理时，Slave设备可以拉低SCL不放（SCL=0将发生“线与”）以阻止Master发送更多的数据。此时Master将视情况减慢或结束数据传送。**

## 器件的地址



## 地址编码规则

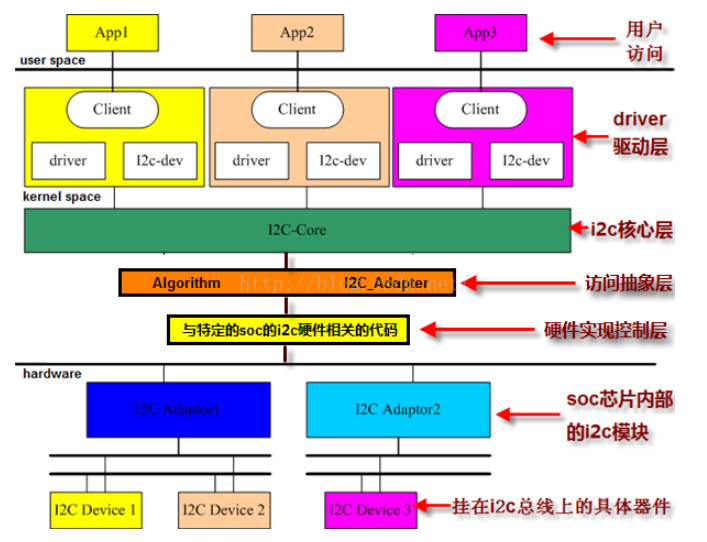


## 常用地址码



# Linux下IIC驱动架构

Linux定义了系统的IIC驱动体系结构，在Linux系统中，IIC驱动由3部分组成，即IIC核心、IIC总线驱动和IIC设备驱动。这3部分相互协作，形成了非常通用、可适应性很强的IIC框架。



## I2C驱动体系结构三部分详细分析

### a -- IIC核心

IIC 核心提供了IIC总线驱动和设备驱动的注册、注销方法，IIC通信方法（即“algorithm”，笔者认为直译为“运算方法”并不合适，为免引起误解， 下文将直接使用“algorithm”）上层的、与具体适配器无关的代码以及探测设备、检测设备地址的上层代码等。

在我们的Linux驱动的i2c文件夹下有algos，busses，chips三个文件夹，另外还有i2c-core.c和i2c-dev.c两个文件。

i2c-core.c文件实现了I2Ccore框架，是Linux内核用来维护和管理的I2C的核心部分，其中维护了两个静态的List，分别记录系统中的I2Cdriver结构和I2Cadapter结构。I2Ccore提供接口函数，允许一个I2Cadatper，I2Cdriver和I2Cclient初始化时在I2Ccore中进行注册，以及退出时进行注销。同时还提供了I2C总线读写访问的一般接口，主要应用在I2C设备驱动中。

### b -- IIC总线驱动

IIC总线驱动是对IIC硬件体系结构中适配器端的实现，适配器可由CPU控制，甚至直接集成在CPU内部。总线驱动的职责，是为系统中每个I2C总线增加相应的读写方法。但是总线驱动本身并不会进行任何的通讯，它只是存在那里，等待设备驱动调用其函数。

IIC总线驱动主要包含了IIC适配器数据结构i2c\_adapter、IIC适配器的algorithm数据结构i2c\_algorithm和控制IIC适配器产生通信信号的函数。经由IIC总线驱动的代码，我们可以控制IIC适配器以主控方式产生开始位、停止位、读写周期，以及以从设备方式被读写、产生ACK等。

　Busses文件夹下的i2c-mpc.c文件实现了PowerPC下I2C总线适配器驱动，定义描述了具体的I2C总线适配器的i2c\_adapter数据结构，实现比较底层的对I2C总线访问的具体方法。I2Cadapter 构造一个对I2Ccore层接口的数据结构，并通过接口函数向I2Ccore注册一个控制器。I2Cadapter主要实现对I2C总线访问的算法，iic\_xfer() 函数就是I2Cadapter底层对I2C总线读写方法的实现。同时I2Cadpter 中还实现了对I2C控制器中断的处理函数。

### c -- IIC设备驱动

IIC设备驱动是对IIC硬件体系结构中设备端的实现，设备一般挂接在受CPU控制的IIC适配器上，通过IIC适配器与CPU交换数据。设备驱动则是与挂在I2C总线上的具体的设备通讯的驱动。通过I2C总线驱动提供的函数，设备驱动可以忽略不同总线控制器的差异，不考虑其实现细节地与硬件设备通讯。

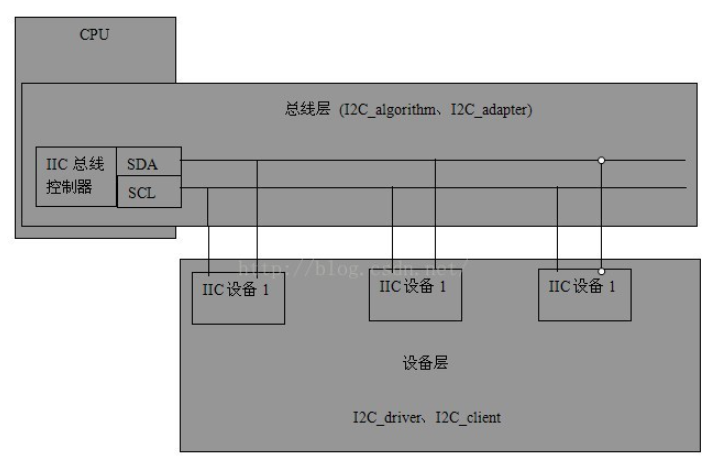
IIC设备驱动主要包含了数据结构i2c\_driver和i2c\_client，我们需要根据具体设备实现其中的成员函数。

i2c-dev.c文件中实现了I2Cdriver，提供了一个通用的I2C设备的驱动程序，实现了字符类型设备的访问接口，实现了对用户应用层的接口，提供用户程序访问I2C设备的接口，包括实现open，release，read，write以及最重要的ioctl等标准文件操作的接口函数。我们可以通过open函数打开 I2C的设备文件，通过ioctl函数设定要访问从设备的地址，然后就可以通过 read和write函数完成对I2C设备的读写操作。

通过I2Cdriver提供的通用方法可以访问任何一个I2C的设备，但是其中实现的read，write及ioctl等功能完全是基于一般设备的实现，所有的操作数据都是基于字节流，没有明确的格式和意义。为了更方便和有效地使用I2C设备，我们可以为一个具体的I2C设备开发特定的I2C设备驱动程序，在驱动中完成对特定的数据格式的解释以及实现一些专用的功能。

## 重要的结构体

这里简单的将IIC设备驱动分为设备层、总线层。理解这两个层次的重点是理解4个数据结构，这4个数据结构是i2c\_driver、i2c\_client、i2c\_algorithm、i2c\_adapter。i2c\_driver、i2c\_client属于设备层；i2c\_algorithm、i2c\_adapter属于总线型。如下图：



设备层关系到实际的IIC设备，总线层包括CPU中的IIC总线控制器和控制总线通信的方法。值得注意的是：一个系统中可能有很多个总线层，也就是包含多个总线控制器；也可能有多个设备层，包含不同的IIC设备

由IIC总线规范可知，IIC总线由两条物理线路组成，这两条物理线路是SDA和SCL。只要连接到SDA和SCL总线上的设备都可以叫做IIC设备。

### a -- i2c\_client

|  |
| --- |
| struct i2c\_client  {  unsigned short flags; 　 //标志位  unsigned short addr;　　　　　　　　　 //设备的地址，低7位为芯片地址  char name[I2C\_NAME\_SIZE];　　　　 　 //设备的名称，最大为20个字节  struct i2c\_adapter \*adapter; //依附的适配器i2c\_adapter，适配器指明所属的总线  struct i2c\_driver \*driver;　　　　　　　　　　　　　//指向设备对应的驱动程序  struct device dev;　　　　　　　　　　　　　　　　　//设备结构体  int irq;　　　　　　　　　　　　　　　　　　　　　　　//设备申请的中断号  struct list\_head list;　　　　　　　　　　　　　　　　//连接到总线上的所有设备  struct list\_head 　　detected;　　　　　　　　　　　//已经被发现的设备链表  struct completion　　released;　　　　　　　　　　　//是否已经释放的完成量  }; |

IIC设备还有一些重要的注意事项：

1、i2c\_client数据结构是描述IIC设备的“模板”，驱动程序的设备结构中应包含该结构；

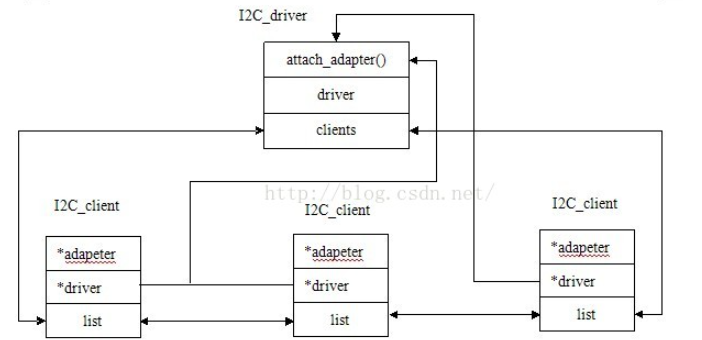
2、adapter指向设备连接的总线适配器，系统可能有多个总线适配器。内核中静态指针数组adapters记录所有已经注册的总线适配器设备；

3、driver是指向设备驱动程序，这个驱动程序是在系统检测到设备存在时赋值的；

### b -- IIC设备驱动 i2c\_driver

|  |
| --- |
| struct i2c\_driver  {  int id; //驱动标识ID  unsigned int class; //驱动的类型  int (\*attach\_adapter)(struct i2c\_adapter \*); //当检测到适配器时调用的函数  int (\*detach\_adapter)(struct i2c\_adapter\*); //卸载适配器时调用的函数  int (\*detach\_client)(struct i2c\_client \*) \_\_deprecated; //卸载设备时调用的函数    //以下是一种新类型驱动需要的函数，这些函数支持IIC设备动态插入和拔出。如果不想支持只实现上面3个。要不实现上面3个。要么实现下面5个。不能同时定义  int (\*probe)(struct i2c\_client \*,const struct i2c\_device\_id \*); //新类型设备探测函数  int (\*remove)(struct i2c\_client \*); //新类型设备的移除函数  void (\*shutdown)(struct i2c\_client \*); //关闭IIC设备  int (\*suspend)(struct i2c\_client \*,pm\_messge\_t mesg); //挂起IIC设备  int (\*resume)(struct i2c\_client \*); //恢复IIC设备  int (\*command)(struct i2c\_client \*client,unsigned int cmd,void \*arg); //使用命令使设备完成特殊的功能。类似ioctl（）函数  struct devcie\_driver driver; //设备驱动结构体  const struct i2c\_device\_id \*id\_table; //设备ID表  int (\*detect)(struct i2c\_client \*,int kind,struct i2c\_board\_info \*); //自动探测设备的回调函数    const struct i2c\_client\_address\_data \*address\_data; //设备所在的地址范围  struct list\_head clients; //指向驱动支持的设备  }; |

结构体i2c\_driver和i2c\_client的关系较为简单，其中i2c\_driver表示一个IIC设备驱动，i2c\_client表示一个IIC设备。关系如下图：



### c -- i2c\_adapter

IIC总线适配器就是一个IIC总线控制器，在物理上连接若干个IIC设备。IIC总线适配器本质上是一个物理设备，其主要功能是完成IIC总线控制器相关的数据通信：

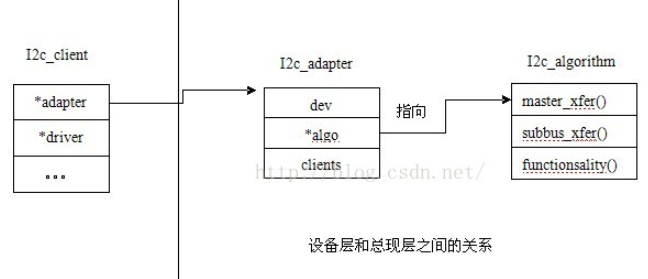
|  |
| --- |
| struct i2c\_adapter  {  struct module \*owner; //模块计数  unsigned int id; //alogorithm的类型，定义于i2c\_id.h中  unsigned int class; //允许探测的驱动类型  const struct i2c\_algorithm \*algo; //指向适配器的驱动程序  void \*algo\_data; //指向适配器的私有数据，根据不同的情况使用方法不同  int (\*client\_register)(struct i2c\_client \*); //设备client注册时调用  int (\*client\_unregister(struct i2c\_client \*); //设备client注销时调用  u8 level;  struct mutex bus\_lock; //对总线进行操作时，将获得总线锁  struct mutex clist\_lock ; //链表操作的互斥锁  int timeout; //超时  int retries; //重试次数  struct device dev; //指向 适配器的设备结构体  int nr ;  struct list\_head clients; //连接总线上的设备的链表  char name[48]; //适配器名称  struct completion dev\_released; //用于同步的完成量  }; |

### d -- i2c\_algorithm

每一个适配器对应一个驱动程序，该驱动程序描述了适配器与设备之间的通信方法:

|  |
| --- |
| struct i2c\_algorithm  {  int (\*master\_xfer)(struct i2c\_adapter \*adap, struct i2c\_msg \*msg, int num); //传输函数指针，指向实现IIC总线通信协议的函数，用来确定适配器支持那些传输类型  int (\*smbus\_xfer)(struct i2c\_adapter \*adap, u16 addr, unsigned short flags, char read\_write, u8 command, int size, union i2c\_smbus\_data \*data); //smbus方式传输函数指针，指向实现SMBus总线通信协议的函数。SMBus和IIC之间可以通过软件方式兼容，所以这里提供了一个函数，但是一般都赋值为NULL  u32 (\*functionality)(struct i2c\_adapter \*); //返回适配器支持的功能    }; |

IIC设备驱动程序大致可以分为设备层和总线层。设备层包括一个重要的数据结构，i2c\_client。总线层包括两个重要的数据结构，分别是i2c\_adapter和i2c\_algorithm。一个i2c\_algorithm结构表示适配器对应的传输数据方法。3个数据结构关系：



### e -- i2c\_msg

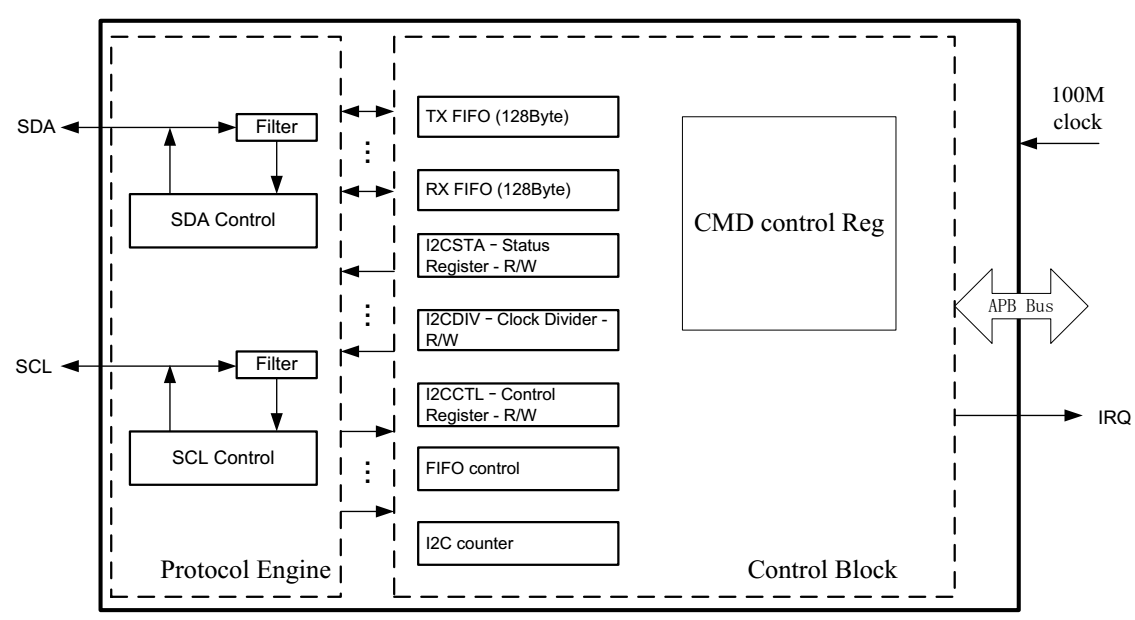
i2c\_algorithm中的关键函数master\_xfer()用于产生i2c访问周期需要的start stop ack信号，以i2c\_msg（即i2c消息）为单位发送和接收通信数据。

　　i2c\_msg也非常关键，调用驱动中的发送接收函数需要填充该结构体

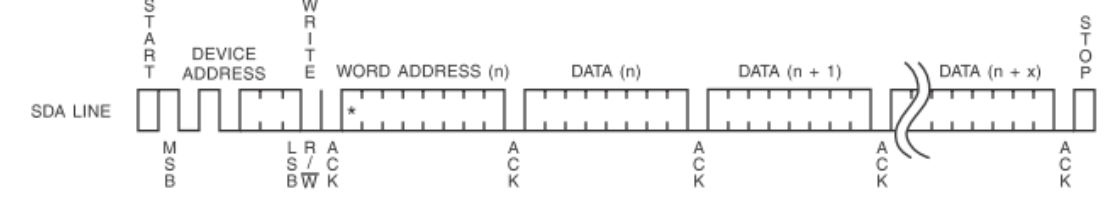
|  |
| --- |
| struct i2c\_msg {  \_\_u16 addr; /\* slave address \*/  \_\_u16 flags;  \_\_u16 len; /\* msg length \*/  \_\_u8 \*buf; /\* pointer to msg data \*/  }; |

# I2c芯片设计

## I2c框架

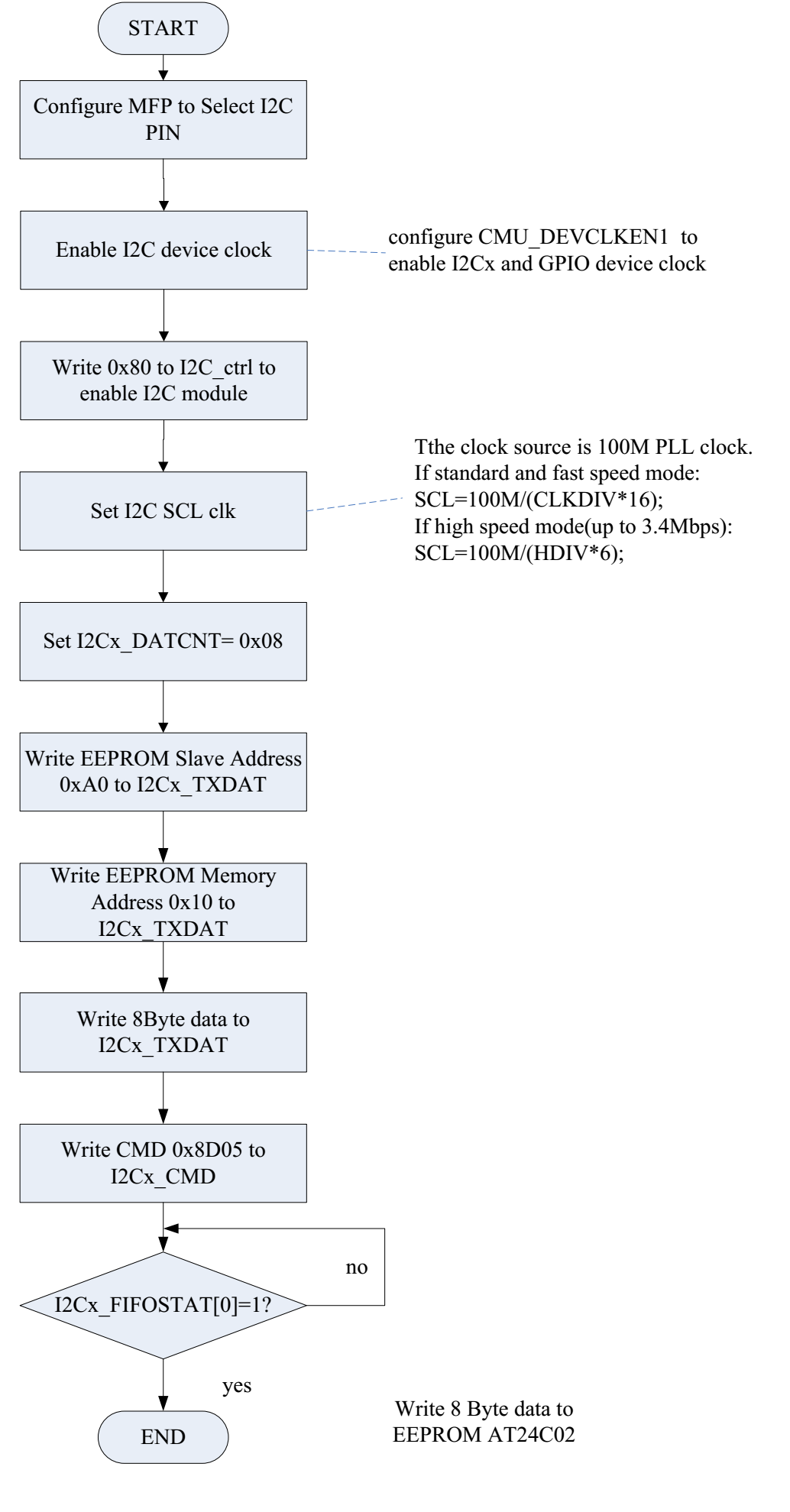


## I2c寄存器配置 : 写数据

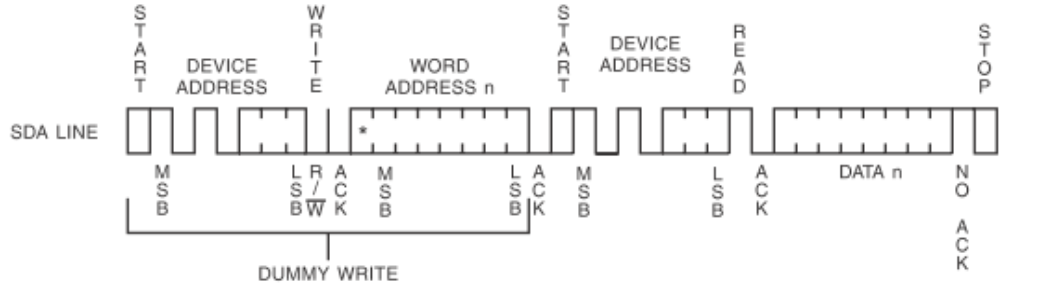


**往 AT24C02 EEPROM 的 0x10 地址写 8Byte 数据，其数据流如下：**

其命令流程为 S+A+Reg1+D+P 访问时序写 8Byte 数据

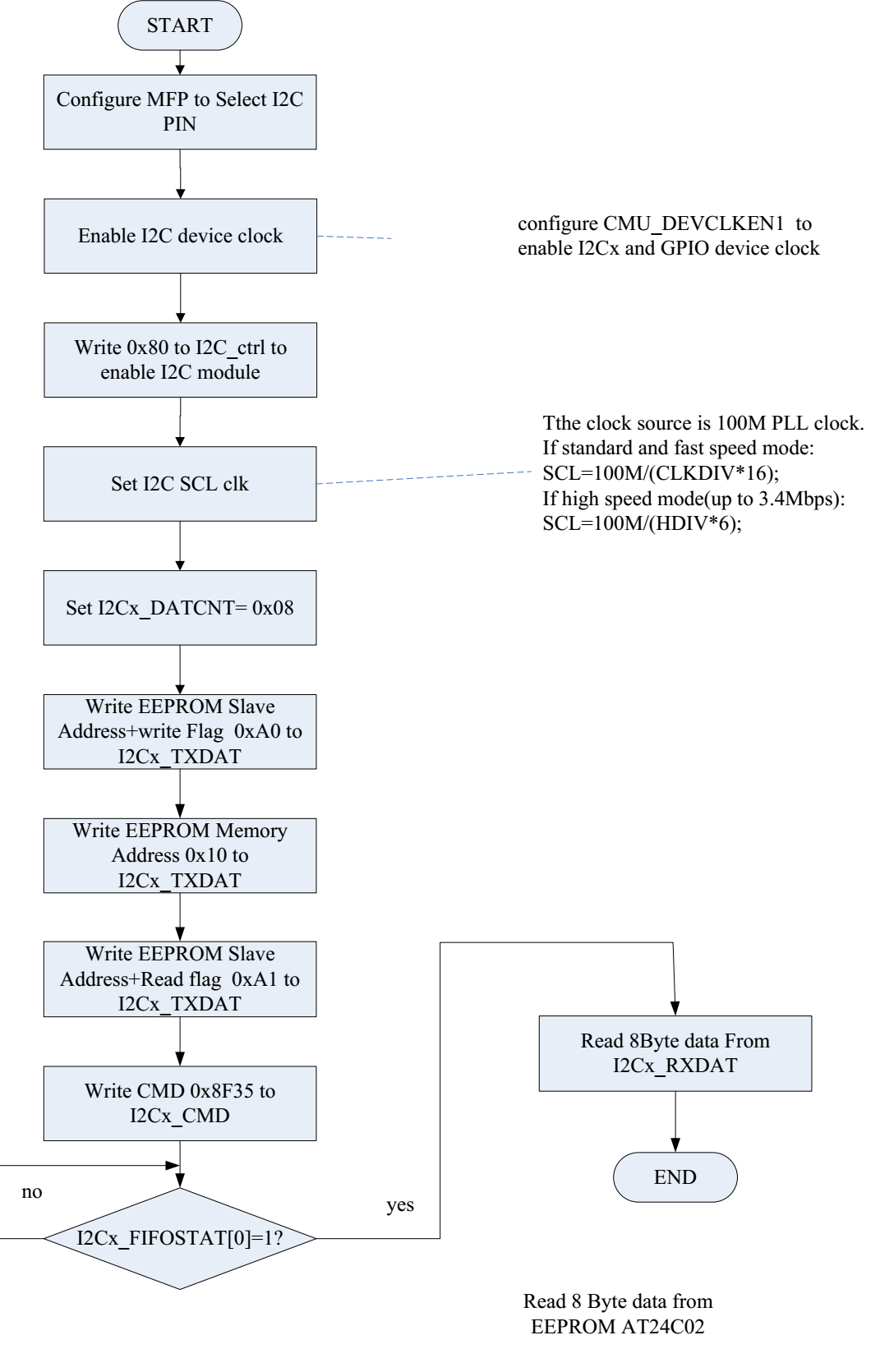


## I2c寄存器配置 :读数据



从 AT24C02 EEPROM 的 0x10 地址读 10Byte 数据，其数据流如下：

其命令流程为 S+A+ Reg1 +RS+Ar+D+P 访问时序读 10Byte 数据，



## I2c寄存器配置模型

|  |
| --- |
| 注： S=Start; RS=ReStart; P=Stop; A=Slave address; Ar=Slave address+Read Flag; SAR= Reg; D=data (nByte); RL=Release Bus  作主设备写数据：  1、 S+A+Reg1+D+P  2、 S+A+ Reg1+ Reg2+D+P  3、 S+A+ Reg1+ D+P  作主设备读数据：  4、 S+A+ Reg1+RS+Ar+D+P  5、 S+A+ Reg1+ Reg2+RS+Ar+D+P  6、 S+A+ Reg1+P+S+Ar+ D+P （中间是 Stop 后再重新发 Start 来进行读）  作从设备写数据：  7、 D+RL  作从设备读数据：  8、 RL+D |
| **寄存器配置 : Sample**  如果是 S+A+ Reg1+ Reg2+RS+Ar+D+P 访问时序读 10Byte 数据，  5、 I2Cx\_CNT 要配置为 10.  6、 往 I2Cx\_TxDAT 写进 1Byte 的从设备地址， 1Byte 的从设备内部 Memory 地址，  7、 再写 1Byte 的从设备地址，  8、 配置 CMD 寄存器的数值为： 0x8F37，  9、 然后才是从 I2Cx\_RxDAT 读回 10Byte 的数据。  如果是 S+A+Reg1+D+P 访问时序写 10Byte 数据  5、 I2Cx\_CNT 要配置为 10.  6、 往 I2Cx\_TxDAT 写进 1Byte 的从设备地址， 1Byte 的从设备内部 Memory 地址  7、 继续往 I2Cx\_TxDAT 写进 10Byte 要发送的数据  8、 配置 CMD 寄存器的数据值为： 0x8D05,  如果是特殊组合： S+A+ Reg1+P+S+Ar+ D+P（中间是 Stop 后再重新发 Start 来进行读），读 10Byte 数据，那么就要拆分成两次配置， S+A+Reg1+P 和 S+Ar+ D+P  1、 I2Cx\_CNT 要配置为 10.  2、 往 I2Cx\_TxDAT 写进 1Byte 的从设备地址， 1Byte 的从设备内部 Memory 地址  3、 先配置 CMD 寄存器的数据值为： 0x8C05  4、 等该命令执行完后，再配置 CMD 寄存器的数据值为： 0x8F03，  5、 然后才是从 I2Cx\_RxDAT 读回 10Byte 的数据。  如果是作从设备写 10Byte 数据访问时序： D+RL  1、 I2Cx\_CNT 要配置为 10.  2、 往 I2Cx\_TxDAT 写进 10Byte 要发送的数据  3、 配置 CMD 寄存器的数据值为： 0x8100，  作从设备读数据并在最后一 Byte 读完后产生 NACK： RL+D；  1、 I2Cx\_CNT 要配置为 10.  2、 配置 CMD 寄存器的数据值为： 0x9300，  3、从 I2Cx\_RxDAT 读进 10Byte 数据  如果是更特殊的，可以在不发 Stop 的前提下，多次配置 CMD 寄存器，最后一次才发 Stop，凑成特殊的序列，或多主、和作从设备 |

## I2c寄存器

### I2Cx\_CTL ： I2C Control Register, Offset=0x0000

|  |
| --- |
| **b[11] : CSE current-source enable bit**  0: normal operation  1: test mode, when set, enable the current-source  **b[10] : SHSM Standard high speed mode**  0: disable standard high speed mode  1: enable standard high speed mode  **b[9] : FHSM Force in High speed mode**  0: not in High speed mode  1: Force to High speed mode (3.4M)  **b[8] : AE Arbitrate enable**  0: disable  1: enable  **b[7] : EN Enable. When enable, reset the status machine to IDLE**  0: Disable  1: Enable  **b[5] : IRQE IRQ Enable.**  0: Disable  1: Enable  新加 FIFO 模式  当满足以下条件时产生 IRQ：  1、 写时 TX FIFO 空，计数器没有计数到零，则 IRQ 起  来；如果计数器计数到零，则要等 Stop 产生后才产  生  2、 读时 RX FIFO 满了，  3、 产生或收到 Stop 后。  4、 作从设备时收到符合本地从地址时  5、 仲裁失败  6、 收到 NACK（不忽略情况下）  **b[3:2] : GBCC Generating Bus Control Condition (only for master mode).**  00: No effect  01: Generating START condition  10: Generating STOP condition  11: Generating Repeated START condition  Write the slave address to the I2C\_DAT register, select start or restart, and then the start or restart command follow by the slave address will occur on the bus.  **b[1] : RB Release Bus. Write 1 to this bit will release the bus.**  **b[0] : GRAS Generate ACK or NACK Signal. When receive data :**  0: generate the ACK signal at 9th clock of SCL  1: generate the NACK signal at 9th clock of SCL |

### I2Cx\_CLKDIV : I2C Clock Divide Control Register, Offset=0x0004

|  |
| --- |
| **b[17:16] CLKCOMP Clk counter compensation**  00: no compensation  01: 10ns  10: 20ns  11: 30ns  注：如果因 CLK 的上升时间太慢导致 pin 的速度比配置速度慢的话，可以设置该寄存器来进行速度补偿。  **b[15:8] HDIV High speed mode Clock Divider Factor (only for the master mode).**  Calculating SCL is as following:  **SCL=100M/(CLKDIV\*6)**  **b[7:0] CLKDIV Clock Divider Factor (only for master mode).**  I2C clock (SCL) can select standard (100kbps) mode and fast (400kbps) mode. Calculating SCL is as following:  **SCL=100M/(CLKDIV\*16)** |

### I2Cx\_STAT : I2C Status Register, Offset=0x0008

|  |
| --- |
| **b[10] SRGC Slave receive general call**  0: not receive a general call  1: receive a general call  **b[9] SAMB Slave address match bit**  0: slave address not match  1: slave address match  **b[8] LBST Last Byte Status Bit.**  0: Indicate the last byte received or transmitted is address  1: Indicate the last byte received or transmitted is data  **b[7] TCB Transfer complete bit**  0: not finish transfer  1: In normal mode:  A byte transfer finish, include transfer the ACK or NACK bit  **b[6] BBB Bus busy bit**  0: Not busy  1: Busy  This bit will set to 1 while the start command detected, and set to 0 after the stop command  **b[5] STAD Start detect bit, include restart.**  The bit is clear when the I2C module is disable or when the STOP condition is detected. Writing 1 to the bit will clear it.  0: Start bit is not detected  1: Start bit is detected  **b[4] STPD Stop detect bit**  The bit is clear when the I2C module is disable or when the START condition is detected. Writing 1 to the bit will clear it.  0: Stop bit is not detected  1: Stop bit is detected  **b[3] LAB Lose arbitration bit**  0: not lose  1: lose arbitration  **b[2] IRQP IRQ Pending Bit.**  1: IRQ  0: No IRQ  Set condition:  1. transfer complete  2. detect normal stop bit ( no bus error )  3. arbit fail  **b[1] BEB Bus error bit**  0: No error occur  1: Bus error occur  The below conditions occur generate error bit:  Detect stop bit right after detect start/restart bit.  Detect stop, start bit when sending or receiving data.  **b[0] RACK Receive ACK or NACK when transmit data or address**  0: NACK  1: ACK  The bit will be updated when the 9th of next byte clock arrived |

### I2Cx\_ADDR : I2C Address Register, Offset=0x000C

|  |
| --- |
| **b[7:1] SDAD Own Slave Device Address.**  Only use in slave mode. I2C\_Addr contains the own address of the module when the device is use in slave mode.  Content of the register is irrelevant when the I2C module is functioning as a master.  **b[0] - Reserved.** |

### I2Cx\_TXDAT && I2Cx\_RXDAT

|  |
| --- |
| I2Cx\_TXDAT  **b[7:0] DA The register of Data or address to be transfer, or received to.**  I2CDAT contains the byte to be transmitted on the I2C-bus or a byte that has been received from the I2C-bus.  In master mode, along with the data byte to be transmitted, it also includes the slave address. The seven MSB’s are the slave I2C device address while the LSB is the Read/Write bit.  128 层 FIFO， 8x128 |
| I2Cx\_RXDAT  **7:0 DA The Receive data Register**  128 层 FIFO， 8x128 |

### I2Cx\_CMD : I2C Data Register, Offset=0x0018

|  |
| --- |
| **b[15] SECL Start to execute the command list**  0: not execute  1: execute command  如果没有使能该位，则 FIFO 不可用，但 I2C 模块的原来非 FIFO 的那一套可以使用。  **b[12] WRS Write or Read select**  0: write  1: read  This bit only used in Slave mode.  作主设备时的读或写标志由填进去的跟在 Start Bit 后的从地址的 Bit0 来判别。  **b[11] MSS Master or slave mode select**  0: slave mode  1: Master mode  **b[10] SE Stop enable**  0: disable  1: enable  **b[9] NS NACK select**  0: not select  1: select  generate the NACK signal at 9th clock of SCL of the last byte when read data  **b[8] DE Data enable**  0: disable  1: enable  The counts of data transmitted depend on the I2Cx\_CNT  **b[7:5] SAS Second address select**  000: no address  001: 1 byte address  010: 2 byte address  011: 3 byte address  100: 4 byte address  101: 5 byte address  110: 6 byte address  111: 7 byte address  Restart 命令后面跟着的地址域。  **b[4] RBE Restart bit enable**  0: not send restart bit  1: send restart bit  **b[3:1] AS Address select**  000: no address  001: 1 byte address  010: 2 byte address  011: 3 byte address  100: 4 byte address  101: 5 byte address  110: 6 byte address  111: 7 byte address  The address include slave address and slave internal memory address.  Start 命令后面跟着的地址域。  **b[0] SBE Start bit enable**  0: not send start bit  1: send start bit |

### I2Cx\_FIFOCTL : I2C Counter Register, Offset=0x001C

|  |
| --- |
| **b[2] TFR TX FIFO reset bit**  Write 1 to reset TX FIFO, auto clear to 0 when Tx FIFO reset complete.  **b[1] RFR RX FIFO reset bit**  Write 1 to reset RX FIFO, auto clear to 0 when Rx FIFO reset complete.  **b[0] NIB NACK Ignore Bit**  0: not ignore, when receive NACK when write, generate Error, do not continue the command list execute， generate IRQ  1: ignore NACK, when receive NACK, don’t generate error, and will continue the command list execute  但如果是收到计数器最后一 Byte 的 NACK 时，就算是使能后，也不应产生  注： Tx FIFO 和 Rx FIFO 产生 IRQ 的门槛规定为全满或全空。 |

### I2Cx\_FIFOSTAT : I2C Counter Register, Offset=0x0020

|  |
| --- |
| **b[23:16] TFD Tx FIFO level display**  This field indicate the current Tx FIFO level  **b[15:8] RFD Rx FIFO level display**  This field indicate the current Rx FIFO level  **b[6] WRS Write or read status bit when acts as slave， used only in FIFO mode**  0: master write to slave  1: master read from slave  **b[5] TFF TX FIFO full bit**  0: not full  1: full  **b[4] TFE TX FIFO empty bit**  0: empty  1: not empty  **b[3] RFF RX FIFO full bit**  0: not full  1: full  **b[2] RFE RX FIFO empty bit**  0: empty  1: not empty  **b[1] RNB Receive NACK Error bit**  0: not receive NACK  1: receive NACK when write data  当往外写数据时，如果在除最后一 Byte 外，其它 Byte的传输中收到 NACK 时,如果 FIFOCTL 寄存器 Bit0 为 0，则该位置 1，不再执行 CMD，并产生中断；如果 FIFOCTL寄存器 Bit0 为 1，则该位不置位，继续执行 CMD。  **b[0] CECB Command Execute Complete bit**  0: not complete  1: complete  该位置 1 则表示所有的命令、数据已经被读或被写。 |

### I2Cx\_DATCNT : I2C Counter Register, Offset=0x0024

|  |
| --- |
| **b[9:0] TC Data Transmit counter**  数据传输计数器，跟 CMD 寄存器的 Bit8 相对应，如果I2C\_CMD[8]不使能，则该计数器无效。  Data 域每 Shift Out 一个字节才减 1，每 Shift In 一个字节到 FIFO 才减 1。计数器寄存器的值不变，但内部的计数器作减运算。 |

### I2Cx\_RCNT ： I2C remain Counter Register， Offset=0x0028

|  |
| --- |
| **b[9:0] TC Remain counter**  显示当前还没有传输的数据个数 |

# End