# AN013 — 用 SMBus 实现串行通信

### 相关器件

本应用笔记适用于下列器件：

C8051F000、C8051F001、C8051F002、C8051F005、C8051F006、C8051F010、C8051F011 和 C8051F012。

## 引言

C8051F0xx系列器件有一个符合系统管理总线标准 1.1 版以及$I^2C$串行总线标准的SMBus串行I/O器件。SMBus是一个双向、二线接口，能与多个器件通信。SMBus是英特尔公司的商标；$I^2C$是菲力浦半导体公司的商标。

本应用笔记介绍 SMBus 总线的配置和操作。本文提供示例汇编代码和‘C’代码：（1）与单个具有一字节地址空间的 EEPROM 接口的汇编语言程序；（2）与多个具有二字节地址空间的 EEPROM 接口的 C 语言程序；（3）两个 C8051F0xx 点对点通信的 C 语言程序。

## SMBus 规范

本节介绍 SMBus 协议。对 SMBus 的讨论从下一节 --“使用 SMBus”开始。

### *SMBus 结构*

一个 SMBus 系统是一个二线网络，网络中的每一个器件有一个唯一的地址并可以被网络中的其它器件访问。所有的传输过程都由一个主器件启动；如果一个器件识别出自己的地址并回应，它就是那次传输的从器件。值得注意的是，没有必要指定一个主器件。对于任何一次数据传输，任何一个器件都可以作为主器件或从器件。当两个器件试图同时启动一次传输时，仲裁机制将强迫一个器件放弃总线。这种仲裁机制是非破坏性的（一个器件赢得总线，但没有信息丢失）。我们将在仲裁一节深入讨论仲裁机制。

SMBus 通信使用两根线：SDA（串行数据）和 SCL（串行时钟）。每根线都是双向的，其方向取决于器件所处的工作方式。主器件总是提供 SCL；主、从器件都可以在 SDA 上传输数据。两根线都应通过一个上拉电路接到正电源。SMBus 线上的所有器件都应有漏极开路或集电极开路输出，这样可使总线空闲时保持高电平。如果一个或多个器件输出低电平信号，总线被拉为低电平。要使总线保持在高电平，所有的器件都必须输出高电平。第二页中的图 1 给出一个典型的 SMBus 总线配置。

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Email: mcuinfo@silabs.com
Internet: www.silabs.com

新华龙电子有限公司
深圳市福田区华强北路现代之窗大厦 A 座 13F C 室(518013)
电话：0755-83645240 83645242 83645244 83645251
电邮：shenzhen@xhl.com.cn
网址：www.xhl.com.cn

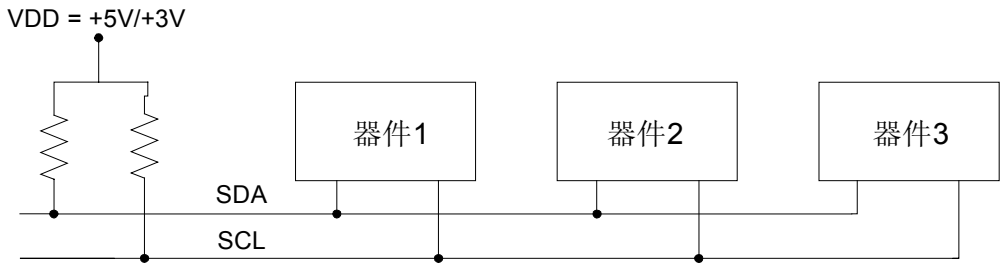



**图 1. 典型的 SMBus 总线配置**

## *握手*

SMBus 采用多种线路条件作为器件间的握手信号。注意，在一次数据传输中，SDA 只能在 SCL 为低时改变电平。在 SCL 为高电平时 SDA 发生改变则是代表如下的开始和停止信号：

**开始：**该条件启动一次传输过程。当 SCL 为高电平时 SDA 上出现一个下降沿。

**结束：**该条件结束一次传输过程。当 SCL 为高电平时 SDA 上出现一个上升沿。

**应答：**也称为 ACK，接收器件发送该信号表示确认。例如，在器件 X 收到一个字节后，它将发送一个 ACK 确认传输成功。ACK 条件是在 SCL 为高时采样到 SDA 为低电平。

**非应答：**也称为 NACK，这是在 SCL 为高电平时采样到 SDA 为高电平。当接收器件不能产生 ACK 时，发送器件看到的是 NACK。在典型的数据传输中，收到 NACK 信号表示所寻址的从器件没有准备好或不在总线上。一个处于接收状态的主器件发送 NACK 表示这是传输的最后一个字节。在下一节中将对这两种情况进行进一步讨论。图 2 给出了握手信号时序。

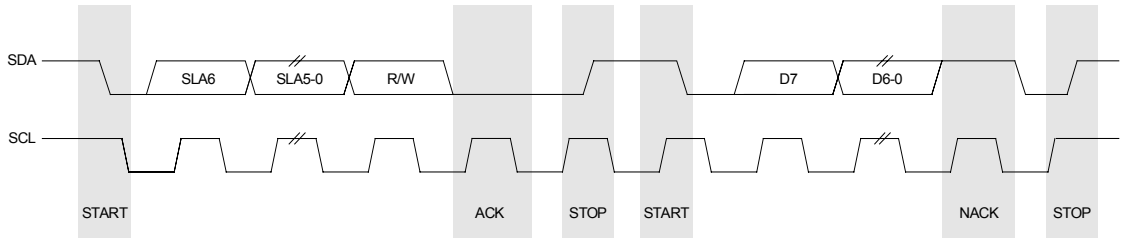



**图 2. SMBus 时序**

## *传输方式*

有两种可能的传输方式：写（从主器件到从器件）和读（从从器件到主器件）。在一次传输中，任何一个器件都可以是四种角色之一。这四种角色将在下面说明。注意，‘从地址+R/W’是指一个 8 位传输（7 位地址，1 位 R/W）。

1）主发送器：在该方式下，器件在 SDA 上发送串行数据，在 SCL 上输出时钟。器件用一个起始条件启动传输过程，发送从地址+W，然后等待从器件的 ACK。收到 ACK 后，器件发送一个或多个字节数据，每个字节都要由从器件确认。在发送完最后一个字节后，器件发送一个停止条件。

2）主接收器：在该方式下，器件在 SDA 上接收串行数据，在 SCL 上输出时钟。器件用一个起始条件启动传输过程，之后发送从地址+R。在收到从器件对地址的 ACK 后，在 SCL 上输出时钟并在 SDA 上接收数据。在接收完最后一个字节后，器件将发送一个 NACK 和一个停止条件。

3）从发送器：在该方式下，器件在 SDA 上输出串行数据，在 SCL 上接受时钟。器件接收一个起始条件和它自己的从地址+R，然后发出 ACK 并进入从发送方式。器件在 SDA 上发送数据，在发送完每个字节后都要收到一个 ACK。在传输完最后一个字节后，主器件发送一个 NACK 和一个停止条件。

4）从接收器：在该方式下，器件收到来自主器件的起始条件和和它自己的从地址+W。然后发出 ACK 并进入从接收方式。现在器件在 SDA 上接收串行数据，在 SCL 上接收时钟。在接收完每个字节后都要发送一个 ACK，在接收到主器件的停止条件后退出从接收方式。图 3 示出典型的写操作情况。（1）示出一个成功的传送过程。

在（2）中，主器件在发送完从地址+W 后收到一个 NACK。这种情况发生在从器件‘离线’时，表示它不能回应其从地址。在这种情况下主器件应发出一个停止条件或重新发出起始条件。为了重试传输过程，主器件在发出停止条件后重新发送起始条件和从地址+W。主器件将一直重复该循环过程，直到收到一个 ACK 为止。这被称为“应答查询”。

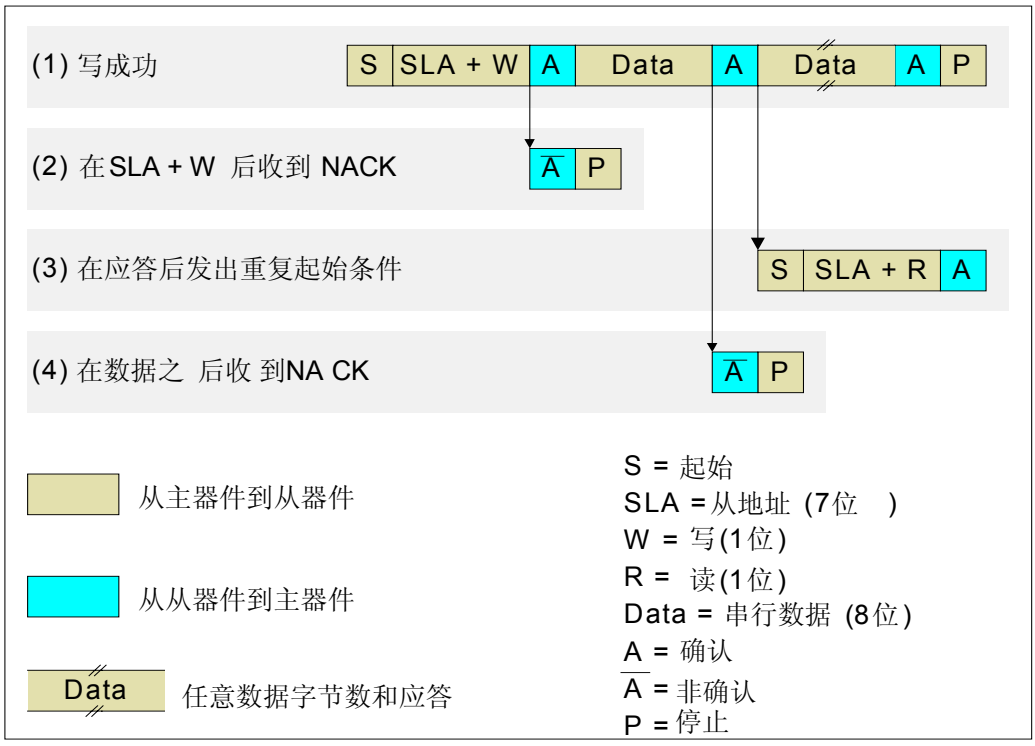



**图 3. 典型的写操作情况**

在（3）中，主器件在收到一个 ACK 后重新发出起始条件。这一过程允许主器件在不放弃总线的情况下启动一个新的传输过程（例如，从写操作切换到读操作）。重复起始条件通常在访问 EEPROM 时使用，因为一个读操作前面必须有一个写存储器地址的操作。在所提供的三个代码示

例中都有重复起始条件。

在（4）中，NACK 是在接收完一个数据字节后收到的。在典型的 SMBus 总线中，这是接收器指示错误的方式。主器件或者象在（2）中那样发送一个停止条件后重试传输过程，或者放弃这次传输。注意：NACK 的使用并不仅限于发生错误的情况，应答级别是一个用户可定义的特性，可以随着应用的不同而变化。

图 4 示出典型的读操作情况。（1）示出一个成功的读过程。

在（2）中，主器件在发送完从地址+R 后收到一个 NACK。这种情况的处理与在写操作的（2）中讨论的一样。主器件可用应答查询来重试传输过程或放弃这次传输。（3）示出主器件在发送完一个字节数据后重复发出起始条件的过程。这与写操作中讨论的重复起始条件状态是一样的。一个主器件可以在传输完任何一个字节后重新发出起始条件，可以在重复起始条件之后启动读或写操作。一般来说，重复起始条件用于改变方向（R/W）或改变地址（从器件）。

注意，在读和写的示意图中所看到的只是典型情况。总线错误、超时和总线竞争都有可能发生。超时用于检测一次传输过程是否停止或总线何时空闲。常常有这样的情况：一个器件保持 SCL 为低电平，直到它准备好继续传输过程为止。这种过程允许一个低速从器件与一个快速的主器件通信，因为停止总线操作实际上相当于降低了 SCL 频率。系统管理总线协议规定：SMBus 系统中的所有器件必须将任何大于 25ms 的 SCL 低电平视为“超时”。发生种情况时，总线上的所有器件必须进行通信复位。也有可能发生高电平 SCL 超时。如果 SDA 和 SCL 同时为高电平的时间大于 50 微秒，则可认为总线处于空闲状态。

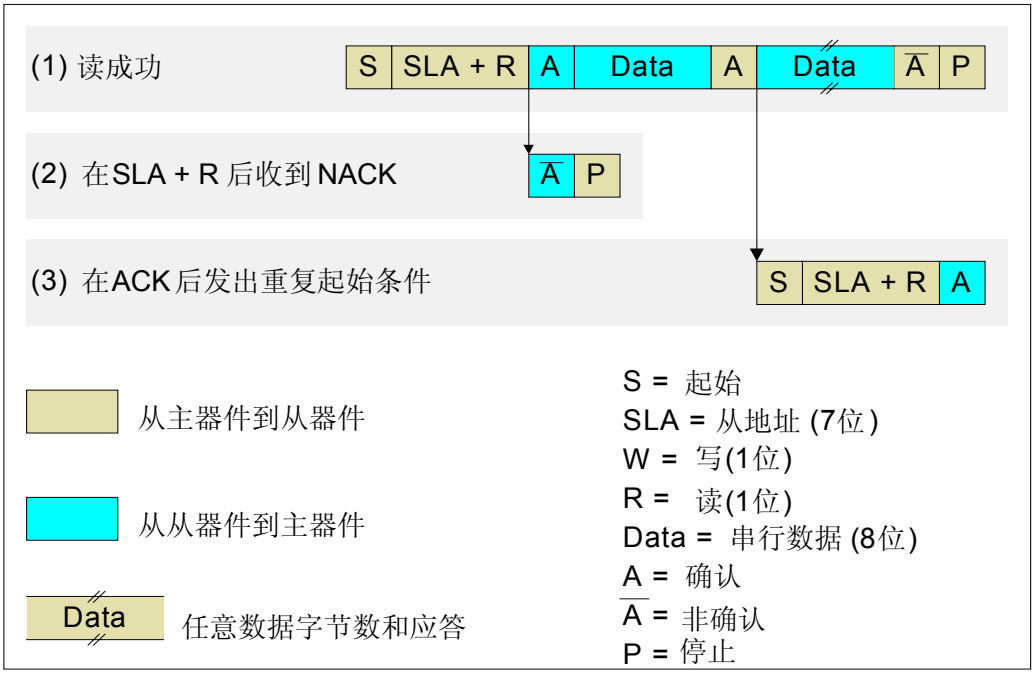



**图 4. 典型的读操作情况**

### *总线竞争*

如果在同一个 SMBus 系统中有多个主器件，有可能出现两个主器件同时启动传输过程的情况。如果发生这种情况，总线仲裁机制就会强迫一个器件放弃总线。

总线仲裁机制是什么：两个主器件继续发送过程，直到其中一个试图发送高电平而另一个试图发送低电平为止。由于总线是漏极开路的，试图发送低电平的器件将获得总线控制权。发送高电平的器件放弃总线，其它器件继续其传输过程。注意，这种总线争用是非破坏性的：总会有一个器件赢得总线。

总线仲裁机制如何工作：假设器件 X 和器件 Y 争用总线。赢得总线的器件 X 不受仲裁机制的任何影响。因为数据是在移位寄存器中移出、移入，所以器件 Y 不丢失任何数据。图 5 给出了两个器件在总线竞争期间输出时序示例。注意：器件 Y 在放弃总线后开始接收数据。

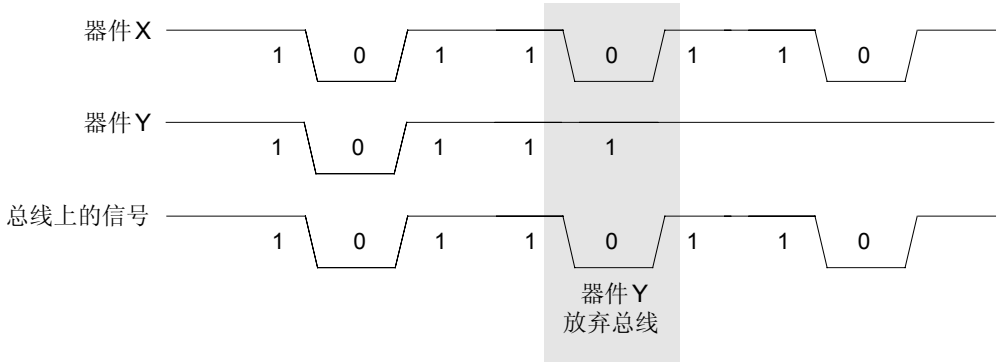



**图 5. 总线竞争过程**

## 使用 SMBus

SMBus 可以工作在主和从方式。总线硬件为串行传输提供了时序和移位控制；字节控制是用户定义的。SMBus 硬件完成下列与应用无关的任务：

时序控制：在主方式，硬件在 SCL 上产生时钟信号用来同步 SDA 上的数据。硬件还识别超时和总线错误。

串行数据传输：硬件控制所有在 SDA 上移入和移出的数据，包括应答级别。应答级别是用户定义的，细节在下面的寄存器定义中解释。

从地址识别：硬件能识别来自另一个器件的起始条件，并读取随后的从器件地址。如果从地址与 SMBus 地址寄存器（下面将定义）中的内容匹配，则硬件应答该地址。注意：只有在 AA（地址应答）被置位时该功能才被允许。

### *配置与控制*

SMBus 的工作由下列寄存器中的内容决定。

**SMB0STA**。SMBus 状态寄存器保持着代表 SMBus 当前状态的 8 位状态代码。只有在 SI 位被

置‘1’时 SMB0STA 才有意义。共有 28 种可能的状态，所有的状态都有一个唯一的代码（代码是 8 的整数倍）。永远不要对 SMB0STA 写入。第 12 页中的表 1 给出了这 28 种状态和它们的说明。

**SMB0CN。**SMBus 控制寄存器用于允许 SMBus 总线并操纵可能的 SMBus 状态。该寄存器包含起始和停止控制以及中断、应答和超时控制。

一次传输过程是通过将 STA 位置 1 来启动的。SMBus 硬件将等待到总线空闲后发送一个起始条件。**注意：STA 不是由硬件来清除的**。用户必须用软件来清除 STA，以避免产生不希望的重复起始条件。

一次传输过程是通过将 STO 位置 1 来中止的。在主方式，置位 STO 将导致产生一个停止条件。如果在 STO 置‘1’时 STA 也被置‘1’，则在起始条件后会跟随一个停止条件。在从方式，置位 STO 会导致硬件产生象接收到一个停止条件那样的动作，尽管实际上发送没有停止条件。

当进入到 28 种可能状态中的任何一个（空闲状态除外）时，SI 位被置‘1’。注意：在 SI 位被置‘1’期间，SCL 保持低电平。这意味着在 SI 被清除之前总线将停止工作，使主器件与从器件同步。

AA 位决定了在应答期间返回的应答类型。如果 AA=1，将发送一个 ACK；如果 AA=0，将发送一个 NACK。这意味着只有在 AA 为被置‘1’时器件才能应答它的从地址。

SCL 高电平和低电平超时检测允许是通过分别将 FTE 和 TOE 位置‘1’来实现的。

通过置位 SMBus 允许位 ENSMB 来允许 SMBus。

**SMB0CR。**在器件工作于主方式时 SMBus 时钟寄存器用于控制 SCL 时钟速率。SMB0CR 中的 8 位数决定了时钟速率，公式如下：

$$SMB0CR \cong -\frac{SYSCLK}{2 \times F_{SCL}} \qquad <1>$$

其中，SMB0CR 是一个负数的补码。因此，对于 100kHz 的 SCL 频率和 16MHz 的 SYSCLK，应向 SMB0CL 装入-80，即 0xB0。

SMB0CR 还定义总线空闲时间周期的极限值（SCL 高电平超时）。总线空闲时间由下面的公式定义：

$$T_{Free} = -\frac{(10 \times SMB0CR) + 1}{SYSCLK} \qquad <2>$$

**SMB0ADR**。SMBus 地址寄存器保存器件在从方式时将要应答的从地址。位（7:1）保存从地址；位 0 是通用呼叫允许。如果位 0 被置位，器件将应答通用呼叫地址（0x00）。

**SMB0DAT**。SMBus 数据寄存器用于保存将要发送或刚刚接收的数据。只有在 SI=1 时，从该寄存器读出的数据才是有效的。当 SI 不为 1 时，SMBus 可能处在向 SMB0DAT 移入数据或从 SMB0DAT 移出数据的过程中。注意：在传输过程中，从 SMB0DAT 移出的最高位又移回到最低位，因此在一次传输完成后 SMB0DAT 中仍然保存着原始数据。

## 实现选择

用户软件基于状态变迁来控制 SMBus。每当发生状态改变时，SI 位被硬件置‘1’，并在中断被允许的情况下产生一个中断。SMBus 接着被停止，直到用户软件完成状态变化服务并清除 SI 位。SMBus 操作很容易用一个状态表定义；但是没有必要定义全部 28 个状态。例如，如果 SMBus 是系统中唯一的主器件，则可以不定义从状态和竞争状态。如果 SMBus 永远不作为主器件出现，则可以不定义主状态。如果状态没有被定义，在程序设计中应有缺省的响应来处理预想不到的或错误的情况。

SMBus 状态表在 C 程序中用开关语句处理。但是对于简单或有时间限制的系统，汇编语言状态译码可能更有效。需要注意的是，SMB0STA 中的状态代码是 8 的倍数。如果 SMBus 状态服务程序在 8 字节以内，则 SMB0STA 可以用作软件索引。在这种情况下，一个状态代码的译码在三个汇编命令以内完成。对于每个状态定义只有 8 字节的代码空间可用。对于那些需要多于 8 个字节空间的状态，程序必须从状态表跳出，以使后续状态不受干扰。

## 示例

本文提供了三个例子：与单个具有一字节地址空间的 EEPROM 接口的汇编语言程序；与多个具有二字节地址空间的 EEPROM 接口的 C 语言程序；两个器件点对点通信的 C 语言程序。每个例子都使用中断驱动操作。

### *单个 EEPROM*

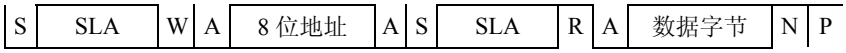
这是 SMBus 与一个 256 字节 EEPROM 接口的简单例子。SMBus 在所有时间内都作为主器件。传输过程与任何二线 EEPROM 接口类似。

发送操作是一个单字节随机写。SMBus 发出一个起始条件并接着发出三个字节：EEPROM 的器件地址+W（该地址可以在 EEPROM 数据表中查到）、待写存储器地址和数据字节。从器件应在收到每个字节后发出 ACK。如果主器件在发送完每个字节后都收到一个 ACK，它就发出停止条件结束传输过程。如果在任何时间主器件收到一个 NACK，它会用应答查询重试传输过程。如果连续对 EEPROM 进行多次要读/写操作，收到 NACK 是很正常的，这是因为大多数自定时的 EEPROM 在实际进行存储器写操作时会进入离线状态。图 6 示出单个 EEPROM 情况下 SDA 上的发送操作。

**图 6. 单个 EEPROM 发送时序**


接收操作是一个单字节随机读。与写操作一样，SMBus 首先发出一个起始条件并接着发出 EEPROM 器件地址+W（写操作用于设置 EEPROM 的当前地址）。在收到从器件的 ACK 后，主器件发出待读存储器地址。在收到一个 ACK 后，主器件发出重复起始条件和从器件地址+R。从器件在发出 ACK 后将发送由前面“放弃”的写操作所给定的地址内读出的数据字节。主器件发出一个 NACK（因为这是最后也是唯一的数据字节）并接着发出停止条件。此处用了重复起始条件，使写存储器地址和读数据字节之间不能开始其它传输过程。图 7 示出单个 EEPROM 情况下 SDA 上的读操作。

**图 7. 单个 EEPROM 接收时序**




本例中的软件是用汇编语言写的，目的是为了说明用 SMB0STA 作为软件索引的优点。SMBus 状态表写在 8 字节的存储段内（每个状态 8 字节）。这是通过对每个状态使用‘ORG’语句来实现的，从表的起始地址按对应的状态码向下偏移。例如，如果状态表的标识符为 STATE_TABLE，状态 1 是 0x08，则状态 1 的代码段应从下面的地址开始：

```
;状态 1
org         STATE_TABLE + 08h
状态 1 程序代码
```

现在，当 SMB0STA 中的状态码为 0x80 时，可按如下过程访问状态 1：

```
;装入当前状态
    mov     a, SMB0STA  ;
    将 DPTR 指向表的起始地址
    mov     DPTR, #STATE_TABLE ;
    跳转到被索引的状态
    jmp     @A+DPTR
```

这一过程使得状态译码非常有效。但是需要注意每个状态的代码空间只有 8 字节。如果一个状态所需代码空间大于 8 字节，则程序必须跳转到状态表以外的一个代码段内，以保证下一个状态定义不受干扰。

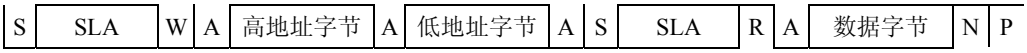
为使状态表简单且易于理解，本例的 SMBus 系统中假设只有一个主器件。从状态没有定义，竞争状态忽略任何接收到的数据。还假设重复起始状态对应的是一个读操作。第 14 页给出了源码清单。

## *多个 EEPROM*

例 2 中使用多个具有二字节地址空间的 EEPROM。软件是用 C 语言编写的。所用的三个 EEPROM 的存储空间都为 8k 字节。注意，这三个 EEPROM 是完全一样的。这些 EEPROM 有三个地址选择引脚 A0 – A2，用于设置器件的从地址。器件地址的高 4 位已在 EEPROM 内固定为“0101”；从地址的低 3 位通过设置地址引脚确定（接 VDD 为 1，接 GND 为 0）。图 9 给出了器件配置图。

本例与上例的区别在于 EEPROM 有二字节地址空间。这意味着对于每次传输过程，读和写操作必须多发送一个地址字节（见图 8）。当中断服务程序进入到“数据已发出，ACK 收到”状态时，它必须知道发出的是哪一个字节，是高地址字节、低地址字节还是数据字节。该信息保存在状态变量 BYTE_NUMBER 中。

**图 8. 多个 EEPROM 接收时序**



SMBus 中断服务程序是用开关语句实现的，用 SMBus 状态代码（SMB0STA）作为开关变量。本例的程序清单从第 23 页开始。

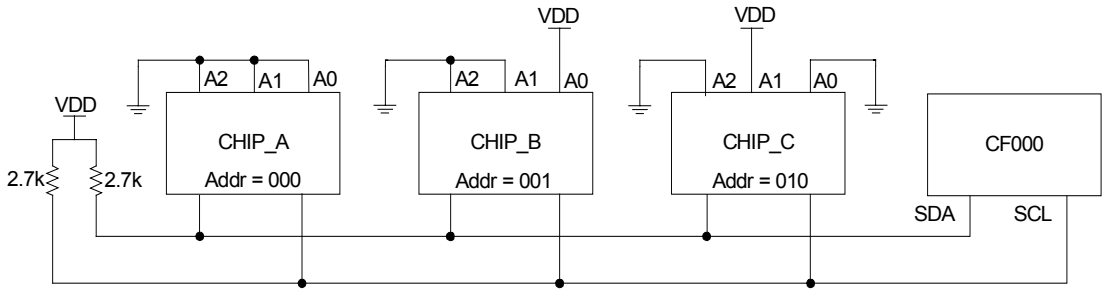



**图 9. 多个 EEPROM 配置**

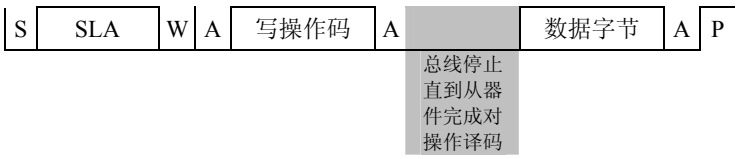
## *点对点通信接口*

最后一个例子是将两个 C8051F0xx 配置为点对点通信。点对点接口使用一组操作码执行下面的一组任务。每个器件都可以启动传输过程。

**写入到从 DAC：**主器件发出一个 WRITE_DAC 操作码并接着发出一字节数据。从器件在完成接收过程后将数据写到其 DAC0 端口。

**写入到缓冲区：**主器件发出一个 WRITE_BUF 操作码并接着发出让从器件存储在其缓冲区中的一个字节数据。WRITE_BUF 操作码的高 4 位含有缓冲区索引地址。图 10 给出点对点的写时序（写 DAC 与写缓冲区是相同的）图。

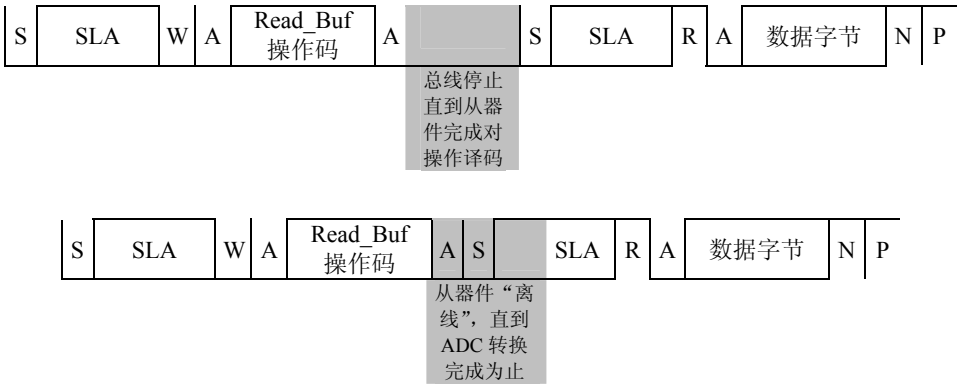
**图 10. 点对点的写时序**




读 ADC：主器件发出一个 READ_ADC 操作码并接着发出一个重复起始条件。从器件读它的 ADC 输入并将数据写入到 SMB0DAT 寄存器。在 ADC 转换期间，从器件清除 AA 位并进入‘离线’状态。在从器件离线期间，主器件在发出重复起始条件和从地址后将收到 NACK。主器件继续应答查询，直到从器件回应为止。这一技术是很有用的，因为如果该从器件的操作时间长，其它器件可以在其离线期间使用总线。从器件在准备好后将 AA 位置 1，继续传输过程。在从器件应答之后，主器件请求一次读操作。传输时序见图 11。

读缓冲区：主器件发出一个 READ_BUF 操作码并接着发出一个重复起始条件。操作码中的高 4 位是缓冲区的索引号。在对操作码译码期间从器件保持 SCL 线为低电平。在 SCL 为低电平期间，主器件不能继续传输过程。另外，其它主器件也不能试图进行传输。在从器件的延迟时间较短的情况下，总线冻结技术是很有用的。从器件在结束对操作码的译码后释放 SCL 并准备好传输数据。主器件此时发出一个重复起始条件和从地址+R。见图 11。

**图 11. 点对点的读时序**




在这个例子的中断服务程序中，用开关语句实现 SMBus 操作。所有可能的状态都被定义，包括竞争状态。如果出现竞争，未获得总线的器件保存其当前传输数据（目标从地址、操作码、相关数据）并应答接收到的操作码。在这次传输过程结束后，刚才未获得总线的器件将用保存的数据重试传输过程。

OP_CODE_HANDLER 函数运行在查询方式，处理接收到的数据。当器件收到一个有效的操作码时，OP_CODE_HANDLER 对其译码并进行相应的操作。

为了测试总线，将对 CHIP_A 的 OP_CODE_HANDLER 调用的注释符去掉。这就允许 CHIP_A 运行所提供的测试代码。注意：对于总线上的每个器件，常数 MY_ADD 必须是唯一的。

本例的源程序清单从第 28 页开始。

**表 1. SMBus 状态码和状态**

| 方式 | 状态码 | SMBus 状态 | 典型操作 |
|---|---|---|---|
| MT/MR | 0x08 | 起始条件已发出 | 将从地址+R/W 装入到 SMB0DAT |
| | 0x10 | 重复起始条件已发出 | 将从地址+R/W 装入到 SMB0DAT |
| 主发送器 | 0x18 | 从地址+W 已发出。收到 ACK。 | 将要发送的数据装入到 SMB0DAT。清 STA |
| | 0x20 | 从地址+W 已发出。收到 NACK。 | 应答查询重试。置位 STO+STA |
| | 0x28 | 数据字节已发出。收到 ACK。 | 1） 将下一字节装入到 SMB0DAT，或<br>2） 置位 STO，或<br>3） 置位 STA 以发送重复起始条件 |
| | 0x30 | 数据字节已发出。收到 NACK。 | 1） 重试传输或<br>2） 置位 STO |
| | 0x38 | 竞争失败 | 保存当前数据 |
| 主接收器 | 0x40 | 从地址+R 已发出。收到 ACK。 | 清 STA。等待接收数据 |
| | 0x48 | 从地址+R 已发出。收到 NACK。 | 应答查询重试。置位 STO+STA |
| | 0x50 | 数据字节收到。ACK 已发出 | 读 SMB0DAT。等待下一字节。<br>如果下一字节是最后字节，清除 AA |
| | 0x58 | 数据字节收到。NACK 已发出 | 置位 STO |

**表 1. SMBus 状态码和状态**

| 方式 | 状态码 | SMBus 状态 | 典型操作 |
|---|---|---|---|
| 从接收器 | 0x60 | 收到自己的从地址+W。ACK 已发出。 | 等待数据 |
| | 0x68 | 在作为主器件发送 SLA+R/W 时竞争失败。收到自身地址+W。ACK 已发出。 | 保存当前数据以备总线空闲时重试。 |
| | 0x70 | 受到通用呼叫地址。ACK 已发出。 | 等待数据 |
| | 0x78 | 在作为主器件发送 SLA+R/W 时竞争失败。收到通用呼叫地址+W。ACK 已发出。 | 保存当前数据以备总线空闲时重试。 |
| | 0x80 | 收到数据字节。ACK 已发出。 | 读 SMB0DAT。等待下一字节或停止条件。 |
| | 0x88 | 收到数据字节。NACK 已发出。 | 置位 STO 以复位 SMBus |
| | 0x90 | 在通用呼叫地址之后收到数据字节。ACK 已发出。 | 读 SMB0DAT。等待下一字节或停止条件。 |
| | 0x98 | 在通用呼叫地址之后收到数据字节。NACK 已发出。 | 置位 STO 以复位 SMBus |
| | 0xA0 | 收到停止条件或重复起始条件。 | 无操作 |
| 从发送器 | 0xA8 | 收到自己的从地址+R。ACK 已发出。 | 将要发送的数据装入到 SMB0DAT。 |
| | 0xB0 | 在作为主器件发送 SLA+R/W 时竞争失败。收到自身地址+R。ACK 已发出。 | 保存当前数据以备总线空闲时重试。将要发送的数据装入到 SMB0DAT。 |
| | 0xB8 | 数据字节已发送。ACK 收到。 | 将要发送的数据装入到 SMB0DAT。 |
| | 0xC0 | 数据字节已收到。NACK 已发出 | 等待停止条件 |
| | 0xC8 | 最后字节已发送。ACK 收到。 | 置位 STO 以复位 SMBus |
| 从器件 | 0xD0 | SCL 时钟高电平定时器每 SMB0CR 超时 | 置位 STO 以复位 SMBus |
| 全部 | 0x00 | 总线错误（非法起始条件或停止条件） | 置位 STO 以复位 SMBus |
| | 0xF8 | 等待状态 | 不置位 SI |

```
;-----------------------------------------------------------------------
;
; Copyright 2001 Cygnal Integrated Products, Inc.
;
; 程序      : SMBus_EX1.asm
; 编写日期  : 2/21/01
; 作者      : JS
;
; 通过 SMBus 实现 C8051F0xx 与一个 256 字节的 EEPROM 的接口
; 假设将一个 EEPROM 连到 SDA 和 SCL 线上，从地址为 1010000，总线上没有其它主器件。
;
; SEND 子程序执行一次向 EEPROM 的单字节写操作。这包括：
; (1)START，(2)从地址 + W，(3)写存储器地址字节，(4)写数据字节
;
; 写 EEPROM 的步骤：
;   1) 将从地址装入到 SLA_ADD
;   2) 将存储器地址装入到 MEM_ADD
;   3) 将数据字节装入到 TRANSMIT_BYTE.
;   4) 调用 SEND
;
; RECEIVE 子程序执行一次从 EEPROM 的单字节读操作。这包括：(1)START，(2)从地址+W，
; (3)写存储器地址字节，(4)重复 START，(5) 从地址+R，(6)读数据字节
;
; 接收数据的步骤：
;   1) 将从地址装入到 SLA_ADD
;   2) 将存储器地址装入到 MEM_ADD
;   3) 调用 RECEIVE
;   4) 读 RECEIVE_BYTE
;
; SMBus 状态表被分成 8 字节的状态段。允许使用 SMBus 状态码(SMB0STA)作为状态索引。
;. 注意，这样每个 SMBus 状态定义只有 8 字节的代码空间。因此，某些任务被做了变动
; 以不受状态定义长度的限制：
:
; 1) SMB_MTDBACK 状态(主发送器，数据字节已发送，ACK 已收到)被缩减为一个位测试
; 和一个转移操作。转移到状态表之外，以便能在该状态执行更长的代码。
;
; 2) 有三个字字节用于从地址存储：SLA_ADD、WRI_ADD、READ_ADD。
; 每个传输操作都预装地址值，而不使用位操作。由于一次接收过程包括写和读两次传输，
; 所以需要两个地址字节 WRI_ADD 和 READ_ADD。SLA_ADD 用做函数调用前的从器件选择。
;
; 注意 SLA_ADD 与 WRI_ADD 等价，因为 WRI_ADD = SLA_ADD + W (W=0)。
; 把它们分别命名是为了叙述清晰。
;
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
; 等价定义
;-----------------------------------------------------------------------

   $MOD8F000                            ; SFR 声明
```

```
    WRITE               EQU     00h         ; SMBus 写命令
    READ                EQU     01h         ; SMBus 读命令

    CHIP_A              EQU     0A0h        ; EEPROM 从地址

    ; SMBus 状态
    SMB_BUS_ERROR       EQU     00h         ; (所有方式)总线错误
    SMB_START           EQU     08h         ; (MT & MR) 起始条件已发送
    SMB_RP_START        EQU     10h         ; (MT & MR) 重复起始条件
    SMB_MTADDACK        EQU     18h         ; (MT) 从地址+W 已发送；ACK 收到
    SMB_MTADDNACK       EQU     20h         ; (MT) 从地址+W 已发送；NACK 收到
    SMB_MTDBACK         EQU     28h         ; (MT) 数据字节已发送；ACK 收到
    SMB_MTDBNACK        EQU     30h         ; (MT) 数据字节已发送；NACK 收到
    SMB_MTARBLOST       EQU     38h         ; (MT) 竞争失败
    SMB_MRADDACK        EQU     40h         ; (MR) 从地址+R 已发送；ACK 收到
    SMB_MRADDNACK       EQU     48h         ; (MR) 从地址+R 已发送；NACK 收到
    SMB_MRDBACK         EQU     50h         ; (MR) 数据字节收到；ACK 已发送
    SMB_MRDBNACK        EQU     58h         ; (MR) 数据字节收到；NACK 已发送

;-----------------------------------------------------------------------
; 变量
;-----------------------------------------------------------------------

DSEG

    org     30h

    TRANSMIT_BYTE:      DS      1           ; 保存 SMBus 待发送的字节
    RECEIVE_BYTE:       DS      1           ; 保存 SMBus 刚收到的字节
    SLA_ADD:            DS      1           ; 保存从地址
    WRI_ADD:            DS      1           ; 保存从地址+ WRITE
    READ_ADD:           DS      1           ; 保存从地址+ READ
    MEM_ADD:            DS      1           ; 要访问的 EEPROM 存储器地址


    ; 用于测试的变量
    TEST_COUNT:         DS      1       ; Test counter variable
    TEST_BYTE:          DS      1       ; Test data
    TEST_ADDR:          DS      1       ; Test memory location

BSEG

    org 00h

    RW:                 DBIT    1       ; R/W 命令位。 1=READ, 0=WRITE
    SM_BUSY:            DBIT    1       ; SMBus 忙标志(软件保存)
    BYTE_SENT:          DBIT    1       ; 用于指示刚发送的字节：
                                        ;   1: EEPROM 存储器地址
```

```
                                        ;   0：数据字节

;-----------------------------------------------------------------------
; 复位和中断向量
;-----------------------------------------------------------------------

CSEG

; 复位向量
    org     00h
    ljmp    Reset_ISR

; SMBus 中断向量
    org 03Bh
    ljmp    SMBus_ISR

;-----------------------------------------------------------------------
; 主程序
;-----------------------------------------------------------------------

MAIN:

    Acall   SMBus_Init              ; 初始化 SMBus
    setb    EA                      ; 允许全局中断

    mov     TEST_BYTE, #0ffh
    mov     TEST_ADDR, #00h         ; 装入初始测试值
    mov     TEST_COUNT, #0feh       ;

;                                                                    TEST
CODE-----------------------------------------------------------------------
-------


TEST:

    ; 发送 TEST_BYTE 到存储器地址 TEST_ADDR
    mov     SLA_ADD, #CHIP_A            ; 装入从地址
    mov     TRANSMIT_BYTE, TEST_BYTE    ; 将待发送数据装入 TRANSMIT_BYTE
    mov     MEM_ADD, TEST_ADDR          ; 将存储器地址装入 MEM_ADD
    acall   SEND                        ; 调用发送子程序

    ; 将从存储器地址 TEST_ADDR 读到的数据装入 RECEIVE_BYTE
    mov     SLA_ADD, #CHIP_A            ; 装入从地址
    mov     MEM_ADD, TEST_ADDR          ; 将存储器地址装入 MEM_ADD
    acall   RECEIVE                     ; 调用接收子程序

    ; 将接收到的字节与发送的字节进行比较
    mov     A, RECEIVE_BYTE             ; 将接收到的字节装入累加器
    cjne    A, TEST_BYTE, END_TEST      ; 将接收到的字节与发送的字节进行比较
                                        ; 如不相等则转到 END_TEST
```

```
    ; 改变测试变量
    dec     TEST_BYTE                   ; 如果发送字节=接收字节，改变测试变量
    inc     TEST_ADDR                   ; 并重新开始

    ; 如果 TEST_COUNTER 不为 0 则重新开始
    djnz    TEST_COUNT, TEST            ; 计数器减 1，循环到开始处
    mov     A, #99h                     ; 如果测试成功，将 99h 装入累加器

END_TEST:

    jmp     $                           ; 原地跳转
;-----------------------------------------------------------------------
; 子程序
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
; SEND 子程序。假设从地址、存储器地址和要发送的数据已经装入到它们各自的变量中。
;该子程序管理 SM_BUSY 位，设置 RW=WRITE，装入 WRI_ADD，初始化传输过程。
;
    push    ACC                         ; 保存累加器
    jb      SM_BUSY, $                  ; 等待 SMBus 空闲
    clr     RW                          ; RW = 0 (写)


    mov     A, SLA_ADD              ; 将 SLA_ADD + WRITE 保存到 WRI_ADD
    orl     A, #WRITE               ;
    mov     WRI_ADD, A              ;

    setb    SM_BUSY                 ; 占用 SMBus
    setb    STA                     ; 启动传输过程
    pop     ACC                     ; 恢复累加器

    ret

;-----------------------------------------------------------------------
; RECEIVE 字程序。假设从地址和存储器已经装入到它们各自的变量中。
;该子程序管理 SM_BUSY 位，设置 RW=READ，装入 READ_ADD 和 WRI_ADD，初始化传输过程。
;
; 注意，接收传输过程包括一个写待访问存储器地址的操作、一个重复起始条件和一个读操作。
; 因此该子程序使用 WRI_ADD 和 READ_ADD。
RECEIVE:

    push    ACC                     ; 保存累加器
    jb      SM_BUSY, $              ; 等待 SMBus 空闲
    setb    RW                      ; RW = 1 (读)

    mov     A, SLA_ADD              ; 保存 SLA_ADD + WRITE 到 WRITE_ADD
    orl     A, #WRITE               ;
    mov     WRI_ADD, A              ;
```

```
    mov     A, SLA_ADD              ; 保存 SLA_ADD + READ 到 READ_ADD
    orl     A, #READ                ;
    mov     READ_ADD, A             ;

    setb    SM_BUSY                 ; 占用 SMBus
    setb    STA                     ; 启动传输过程

    jb      SM_BUSY, $              ; 等待接收结束
    pop     ACC                     ; 恢复累加器

    ret

;------------------------------------------------------------------------
; SMBus_Init
; Smbus 初始化子程序
;

; - 配置并允许 SMBus。
; - 设置 SMBus 时钟速率
; - 允许 SMBus 中断。
; - 为第一次传输清除 SM_Busy 标志


SMBus_Init:

    mov     SMB0CN, #04h            ; 配置 SMBus 在应答周期发送 ACK
    mov     SMB0CR, #0B0h           ; 时钟速率 = 100KHz，根据 SMB0CR 公式:
                                    ; SMB0CR = -(SYSCLK)/(2*Fscl)

    orl     SMB0CN, #40h            ; 允许 SMBus

    orl     EIE1, #02h              ; 允许 SMBus 中断
    clr     SM_BUSY

    ret

;------------------------------------------------------------------------
; 中断向量
;------------------------------------------------------------------------

;------------------------------------------------------------------------
; 复位中断服务程序
;
; - 禁止看门狗定时器
; - 通过交叉开关将 SDA 和 SCL 连到通用 I/O 引脚
; - 允许交叉开关
; - 转到 MAIN

Reset_ISR:

    mov     WDTCN, #0DEh            ; 禁止看门狗定时器
    mov     WDTCN, #0ADh
```

```
orl     OSCICN, #03h            ; 将内部振荡器设置为最高频率(16 MHz)

mov     XBR0, #01h              ; 通过交叉开关将 SMBus 连到通用 I/O 引脚
mov     XBR2, #40h              ; 允许交叉开关和弱上拉

ljmp    MAIN
```

```
;-----------------------------------------------------------------------
; SMBus ISR
;
; 以状态查找表的形式实现，用 SMBus 状态寄存器作为索引值
; SMBus 状态码是 8 的整数倍；因此状态码可以用于索引 8 字节的程序段。
; 每个'org'命令指示一个新状态，从状态表的开始位置向下偏移，偏移量为其状态代码值。
;
; 注意，只有 8 字接的空间用于处理每个状态。在需要多于 8 字节的场合，程序转到状态表
; 以外的代码空间。只有在状态'SMB_MTDBACK'时才需要这样做。

SMBus_ISR:

    push   PSW                      ;
    push   ACC                      ;
    push   DPH                      ; 保护现场
    push   DPL                      ;
    push   ACC                      ;

    mov    A, SMB0STA               ; 将当前 SMBus 装入累加器
                                    ; 对于每个状态执行，状态与地址偏移量对应

    anl    A, #7Fh                  ; 屏蔽最高位，因为该位为 1 的状态没有定义

    mov    DPTR, #SMB_STATE_TABLE   ; DPTR 指向状态表的起始地址
    jmp    @A+DPTR                  ; 转移到当前状态

; SMBus 状态表-----------------------------------------------------------

SMB_STATE_TABLE:

    ; SMB_BUS_ERROR
    ; 对所有状态：总线错误
    ; 通过置 1 停止位对硬件复位
    org    SMB_STATE_TABLE + SMB_BUS_ERROR

        setb   STO
        jmp    SMB_ISR_END          ; 中断返回

    ; SMB_START
    ; 主发送器/接收器：起始条件已发送
    ; 在该状态 R/W 总是为 0(W)，因为对读和写操作来说都必须先写存储器地址。
    org SMB_STATE_TABLE + SMB_START

        mov    SMB0DAT, WRI_ADD     ; 装入从地址 + W
        clr    STA                  ; 手动清除 START 为
        jmp    SMB_ISR_END          ; 中断返回

    ; SMB_RP_START
    ; 主发送器/接收器：重复起始条件已发送
    ; 该状态只应在读操作期间出现，在存储器地址已发出并已得到确认之后。
    org    SMB_STATE_TABLE + SMB_RP_START
```

```
    mov     SMB0DAT, READ_ADD       ; 装入从地址 + R
    clr     STA                     ; 手动清除 START 位
    jmp     SMB_ISR_END

; SMB_MTADDACK
; 主发送器：从地址 + WRITE 已发送；收到 ACK
org     SMB_STATE_TABLE + SMB_MTADDACK

    mov     SMB0DAT, MEM_ADD        ; 装存储器地址
    setb    BYTE_SENT               ; BYTE_SENT=1：在下一个 ISR 调用时，
                                    ; 存储器地址刚被发送。
    jmp     SMB_ISR_END

; SMB_MTADDNACK
; 主发送器：从地址 + WRITE 已发送；收到 NACK。从器件不应答。
; 用应答查询重试。发送 STOP + START。
org     SMB_STATE_TABLE + SMB_MTADDNACK

    setb    STO
    setb    STA
    jmp SMB_ISR_END

; SMB_MTDBACK
; 主发送器：数据字节已发送；收到 ACK。该状态在读和写操作中都要用到。
; 检查 BYTE_SENT 如果为 1，说明存储器地址刚刚发出。否则，数据字节已被发出。
    org     SMB_STATE_TABLE + SMB_MTDBACK

    jbc BYTE_SENT, ADDRESS_SENT     ; 如果 BYTE_SENT=1，清除该位并转到
                                    ; ADDRESS_SENT 去执行状态表以外的
                                    ; 处理程序

    jmp DATA_SENT                   ; 如果 BYTE_SENT=0，数据刚被发出，
                                    ; 传输过程完成，转到传输结束
```

```
; SMB_MTDBNACK
; 主发送器：数据字节已发送；收到 NACK。从器件不应答。
; 用应答查询重试。发送 STOP + START 重试。
org SMB_STATE_TABLE + SMB_MTDBNACK

    setb   STO
    setb   STA
    jmp    SMB_ISR_END

; SMB_MTARBLOST
; 主发送器：竞争失败
; 不应发生。如果发生，重新开始发送。
org SMB_STATE_TABLE + SMB_MTARBLOST

    setb   STO
    setb   STA
    jmp    SMB_ISR_END

; SMB_MRADDACK
; 主接收器：从地址 + READ 已发送。收到 ACK。
; 设置为在下一次传输后发送 NACK，因为那将是最后（唯一）的字节
org SMB_STATE_TABLE + SMB_MRADDACK

    clr    AA                      ; 在应答周期发送 NACK
    jmp    SMB_ISR_END

; SMB_MRADDNACK
; 主接收器：从地址 + READ 已发送。收到 NACK。
; 从器件不应答。用应答查询重试。发送 STOP + START 重试。
org SMB_STATE_TABLE + SMB_MRADDNACK

    setb   STA
    jmp    SMB_ISR_END

; SMB_MRDBACK
; 主接收器：数据字节已收到。ACK 已发送。
; 不应出现，因为 AA 已在前一状态被清除。如果发生，发送 STOP。
org SMB_STATE_TABLE + SMB_MRDBACK

    setb   STO
    jmp    SMB_ISR_END

; SMB_MRDBNACK
; 主接收器：数据字节已收到。NACK 已发送。
; 读操作完成。读数据寄存器并发送 STOP。
```

```
    org SMB_STATE_TABLE + SMB_MRDBNACK

        mov     RECEIVE_BYTE, SMB0DAT
        setb    STO
        setb    AA                      ; 为下一次传输置位 AA
        clr     SM_BUSY
        jmp     SMB_ISR_END
; 状态表结束-------------------------------------------------------------------
;-------------------------------------------------------------------------------
; 需要大于 8 字节程序空间的处理 SMBus 状态的程序段

; 地址字节刚发出。检查 RW。如果为 R(1)，转到 RW_READ。
; 如果为 W，将待发送数据装入 SMB0DAT。
ADDRESS_SENT:

    jb      RW, RW_READ
    mov     SMB0DAT, TRANSMIT_BYTE      ; 装入数据
    jmp     SMB_ISR_END                 ; 中断返回

; 这是一个读操作，地址字节刚被发出。发送重复 START 启动存储器读
RW_READ:

    setb    STA                         ; 发送重复 START
    jmp     SMB_ISR_END                 ; 中断返回

; 这是一个写操作，数据字节刚被发出。传输过程结束。发送 STOP，释放总线，中断返回
DATA_SENT:

    setb    STO                         ; 发送 STOP 后中断返回
    clr SM_BUSY                         ; 释放 SMBus
    jmp     SMB_ISR_END                 ; 中断返回
;-------------------------------------------------------------------------------
; SMBus ISR exit.
; 恢复寄存器，清除 SI 位，从中断返回
SMB_ISR_END:

    clr     SI
    pop     ACC
    pop     DPL
    pop     DPH
    pop     ACC
    pop     PSW

    reti

END
```

```
//-----------------------------------------------------------------------
//
// Copyright 2001 Cygnal Integrated Products, Inc.
//
// 文件名       : SMB_Ex2.c
// 目标器件     : C8051F000
// 编写日期     : 2/20/01
// 作者         : JS
//
//
// C8051F0xx 通过 SMBus 与三个 EEPROM 接口的示例代码。
// 该程序假设三个具有 16 位地址空间的 EEPROM 连在 SCL 和 SDA 线上，
// 被配置为具有如下从地址：
// CHIP_A = 1010000
// CHIP_B = 1010001
// CHIP_C = 1010010
//
// 从状态和竞争状态没有定义。假设 CF000 为系统中唯一的主器件。
// 功能：SM_Send 执行向指定 EEPROM 的单字节写操作
// SM_Receive 执行从指定 EEPROM 地址读一个字节的操作（两者都用到存储器地址）
//
// 包含测试代码部分。

//-----------------------------------------------------------------------
// 包含文件
//-----------------------------------------------------------------------
#include <c8051f000.h>                  // SFR 声明

//-----------------------------------------------------------------------
// 全局常量
//-----------------------------------------------------------------------

#define   WRITE      0x00               // SMBus 写命令
#define   READ       0x01               // SMBus 读命令

// 器件地址（7 位，最低位没使用）
#define   CHIP_A     0xA0               // 芯片 A 的器件地址
#define   CHIP_B     0xA2               // 芯片 B 的器件地址
#define   CHIP_C     0xA4               // 芯片 C 的器件地址

// SMBus 状态
// MT = 主发送器
// MR = 主接收器

#define   SMB_BUS_ERROR    0x00         // （对所有方式）总线错误
#define   SMB_START        0x08         // (MT & MR)起始条件已发送
#define   SMB_RP_START     0x10         // (MT & MR)重复起始条件
#define   SMB_MTADDACK     0x18         // (MT) 从地址 + W 已发送；收到 ACK
#define   SMB_MTADDNACK    0x20         // (MT) 从地址 + W 已发送；收到 NACK
#define   SMB_MTDBACK      0x28         // (MT) 数据字节已发送；收到 ACK
```

```
#define   SMB_MTDBNACK     0x30       // (MT) 数据字节已发送；收到 NACK

#define   SMB_MTARBLOST    0x38       // (MT) 竞争失败
#define   SMB_MRADDACK     0x40       // (MR) 从地址 + R 已发送；收到 ACK
#define   SMB_MRADDNACK 0x48          // (MR) 从地址 + W 已发送；收到 NACK
#define   SMB_MRDBACK      0x50       // (MR) 收到数据字节；ACK 已发送
#define   SMB_MRDBNACK  0x58          // (MR) 收到数据字节；NACK 已发送

//-----------------------------------------------------------------------
//全局变量
//-----------------------------------------------------------------------
char COMMAND;                         // 在 SMBus 中断服务程序中用于
                                      // 保存从地址 + R/W 位。

char WORD;                            // 保持 SMBus 要发送的数据字节
                                      // 或刚收到的数据

char BYTE_NUMBER;                     // 在中用于检查发送的是什么数据
                                      // 高地址字节、低地址字节或数据字节

unsigned char HIGH_ADD, LOW_ADD;      // EEPROM 存储器地址的高、低字节

bit SM_BUSY;                          // 该位在发送或接收开始时被置 1，
                                      // 操作结束后由中断服务程序清 0

//-----------------------------------------------------------------------
// 函数原型
//-----------------------------------------------------------------------

void SMBus_ISR (void);
void SM_Send (char chip_select, unsigned int byte_address, char out_byte);
char SM_Receive (char chip_select, unsigned int byte_address);
```

```
//-----------------------------------------------------------------------
// 主程序
//-----------------------------------------------------------------------
//
// 主程序配置交叉开关和 SMBus，并测试 SMBus 与三个 EEPROM 之间的接口
void main (void)
{
   unsigned char check;               // 用于测试目的

   WDTCN = 0xde;                      // 禁止看门狗定时器
   WDTCN = 0xad;

   OSCICN |= 0x03;                    // 设置内部振荡器为最高频率(16 MHz)

   XBR0 = 0x01;                       // 通过交叉开关将 SMBus 连到通用 I/O 引脚
   XBR2 = 0x40;                       // 允许交叉开关和弱上拉

   SMB0CN = 0x44;                     // 允许 SMBus 在应答周期发送 ACK
   SMB0CR = -80;                      // SMBus 时钟频率 = 100kHz.

   EIE1 |= 2;                         // SMBus 中断允许
   EA = 1;                            // 全局中断允许

   SM_BUSY = 0;                       // 为第一次传输释放 SMBus。


// 测试代码-------------------------------------------------------------
   SM_Send(CHIP_A, 0x0088, 0x53);     //发送 0x53（数据）到 CHIP_A 的地址 0x88
   SM_Send(CHIP_B, 0x0001, 0x66);     //发送 0x66（数据）到 CHIP_B 的地址 0x01
   SM_Send(CHIP_C, 0x0010, 0x77);
   SM_Send(CHIP_B, 0x0333, 0xF0);
   SM_Send(CHIP_A, 0x0242, 0xF0);

   check = SM_Receive(CHIP_A, 0x0088);  // 读 CHIP_A 的地址 0x88
   check = SM_Receive(CHIP_B, 0x0001);  // 读 CHIP_B 的地址 0x01
   check = SM_Receive(CHIP_C, 0x0010);
   check = SM_Receive(CHIP_B, 0x0333);
   check = SM_Receive(CHIP_A, 0x0242);
// 擦拭代码结束---------------------------------------------------------

}
```

```
// SMBus 字节写函数-----------------------------------------------------
// 向给定存储器地址写一个字节
//
// out_byte = 待写数据
// byte_address = 待写存储器地址（2 字节）
// chip_select  = 待写 EEPROM 芯片的器件地址
void SM_Send (char chip_select, unsigned int byte_address, char out_byte)
{
    while (SM_BUSY);                        // 等待 SMBus 空闲
    SM_BUSY = 1;                            // 占用 SMBus（设置为忙）
    SMB0CN = 0x44;                          // SMBus 允许，应答周期发 ACK

    BYTE_NUMBER = 2;                        // 2 地址字节
    COMMAND = (chip_select | WRITE);        // 片选 + WRITE

    HIGH_ADD = ((byte_address >> 8) & 0x00FF); // 高 8 位地址
    LOW_ADD = (byte_address & 0x00FF);          // 低 8 位地址

    WORD = out_byte;                            // 待写数据

    STA = 1;                                    // 启动传输过程
}

// SMBus 随机读函数-----------------------------------------------------
// 从给定存储器地址读一个字节
//
// byte_address = 要读取的存储器地址
// chip_select = 待读 EEPROM 的器件地址
char SM_Receive (char chip_select, unsigned int byte_address)
{
    while (SM_BUSY);                            // 等待总线空闲
    SM_BUSY = 1;                                //占用 SMBus（设置为忙）
    SMB0CN = 0x44;                              // 允许 SMBus，应答周期发 ACK

    BYTE_NUMBER = 2;                            // 2 地址字节
    COMMAND = (chip_select | READ);             // 片选 + READ

    HIGH_ADD = ((byte_address >> 8) & 0x00FF); // 高 8 位地址
    LOW_ADD = (byte_address & 0x00FF);          // 低 8 位地址

    STA = 1;                                    // 启动传输过程
    while (SM_BUSY);                            // 等待传输结束
    return WORD;
}
//-----------------------------------------------------------------------
// 中断服务程序
//-----------------------------------------------------------------------

// SMBus 中断服务程序

void SMBUS_ISR (void) interrupt 7
```

```
{
   switch (SMB0STA){                        // SMBus 状态码（SMB0STA 寄存器）

      // 主发送器/接收器：起始条件已发送
      // 在该状态发送的 COMMAND 字的 R/W 位总是为 0(W)，
      // 因为对于读和写操作来说都必须先写存储器地址。
      case SMB_START:
         SMB0DAT = (COMMAND & 0xFE);    // 装入要访问的从器件的地址
         STA = 0;                       // 手动清除 START 位
         break;

      //主发送器/接收器：重复起始条件已发送。
      // 该状态只应在读操作期间出现，在存储器地址已发送并得到确认之后
      case SMB_RP_START:
         SMB0DAT = COMMAND;             // COMMAND 中应保持从地址 + R.
         STA = 0;
         break;

      // 主发送器：从地址 + WRITE 已发送，收到 ACK。
      case SMB_MTADDACK:
         SMB0DAT = HIGH_ADD;            // 装入待写存储器地址的高字节
         break;

      // 主发送器：从地址 + WRITE 已发送，收到 NACK。
      // 从器件不应答，发送 STOP + START 重试
      case SMB_MTADDNACK:
         STO = 1;
         STA = 1;
         break;

      // 主发送器：数据字节已发送，收到 ACK。
      // 该状态在写和读操作中都要用到。BYTE_NUMBER 看存储器地址状态 – 如果
      // 只发送了 HIGH_ADD，则装入 LOW_ADD。如果 LOW_ADD 已发送，检查 COMMAND
      // 中的 R/W 值以决定下一状态。
      case SMB_MTDBACK:
         switch (BYTE_NUMBER){
            case 2:                     // 如果 BYTE_NUMBER=2，
               SMB0DAT = LOW_ADD;       // 只发送了 HIGH_ADD。
               BYTE_NUMBER--;           // 减 1，为下一轮作准备
               break;
            case 1:                     // 如果 BYTE_NUMBER=1，LOW_ADD 已发送。
               if (COMMAND & 0x01)      // 如果 R/W=READ，发送重复起始条件
                  STA = 1;

               else{
                  SMB0DAT = WORD;       // 如果 R/W=WRITE，装入待写字节
                  BYTE_NUMBER--;}
               break;
            default:                    // 如果 BYTE_NUMBER=0，传输结束
               STO = 1;
               SM_BUSY = 0;             // 释放 SMBus
```

```
            }
        break;


    // 主发送器：数据字节已发送，收到 NACK。
    // 从器件不应答，发送 STOP + START 重试
    case SMB_MTDBNACK:
        STO = 1;
        STA = 1;
        break;

    // 主发送器：竞争失败
    // 不应出现。如果出现，重新开始传输过程
    case SMB_MTARBLOST:
        STO = 1;
        STA = 1;
        break;

    // 主接收器：从地址 + READ 已发送。收到 ACK。
    // 设置为在下一次传输后发送 NACK，因为那将是最后一个字节（唯一）。
    case SMB_MRADDACK:
        AA = 0;                         // 在应答周期 NACK。
        break;

    // 主接收器：从地址 + READ 已发送。收到 NACK。
    // 从器件不应答，发送重复起始条件重试
    case SMB_MRADDNACK:
        STA = 1;
        break;

    // 收到数据字节。ACK 已发送。
    // 该状态不应出现，因为 AA 已在前一状态被清 0。如果出现，发送停止条件。

    case SMB_MRDBACK:
        STO = 1;
        SM_BUSY = 0;
        break;

    // 收到数据字节。NACK 已发送。
    // 读操作已完成。读数据寄存器后发送停止条件。
    case SMB_MRDBNACK:
        WORD = SMB0DAT;
        STO = 1;
        SM_BUSY = 0;                // 释放 SMBus
        break;

    // 在本应用中，所有其它状态码没有意义。通信复位。
    default:
        STO = 1;                    // 通信复位。
        SM_BUSY = 0;
        break;
    }

SI=0;                               // 清除中断标志
```

```
}



//-----------------------------------------------------------------------
//
// Copyright 2001 Cygnal Integrated Products, Inc.
//
// 文件名       : SMB_Ex3.c
// 目标器件     : C8051F000
// 编写日期     : 2/20/01
// 作者         : JS
//
// 两个 C8051F0xx 器件之间用 SMBus 接口的示例代码。
// 这两个器件以点对点方式工作。
//
// 该例演示每个器件如何使用操作码命令另一个器件完成：
//
// 1） 向 DAC0 写一个字节
// 2） 写一个字节到一个数据缓冲区
// 3） 进行一次 ADC 转换
// 4） 从一个数据缓冲区读一个字节
//
// 如果每个器件都将 DAC0 连到 AIN0，则最这些操作码的测试是很容易的。
// 在这种配置下，READ_ADC 命令可以用于测试 WRITE_DAC 的输出。
//
// 本程序假设两个 CF0xx 器件通过 SCL 和 SDA 连在一起，从地址（在寄存器 SMB0ADR 中）为：
// CHIP_A = 1111000
// CHIP_B = 1110000
//
// 测试代码已包含在内。为了达到测试目的，一个器件中的测试代码应被去掉，而运行另一个
// 器件的测试代码。这可以通过注释掉假定的主器件中测试代码之前的 OP_CODE_HANDLER()
// 调用来完成。
//
// 请注意，常数 MY_ADD 必须与当前器件对应，在向 CHIP_B 下载代码时应将其改为 CHIP_B。
//
//-----------------------------------------------------------------------


//-----------------------------------------------------------------------
// 包含文件
//-----------------------------------------------------------------------
#include <c8051f000.h>                  // 声明
//-----------------------------------------------------------------------
// 全局常量
//-----------------------------------------------------------------------

#define   WRITE          0x00       // 写方向位
#define   READ           0x01       // 读方向位
```

```
// 器件地址
#define   CHIP_A        0xF0
#define   CHIP_B        0xE0
#define   MY_ADD        CHIP_A     // 对应当前被编程的器件

// 点对点操作码
#define   READ_ADC      0x01       // OP_CODE 读从 ADC
#define   WRITE_DAC     0x02       // OP_CODE 写从 DAC
#define   WRITE_BUF     0x03       // OP_CODE 写从缓冲区
#define   READ_BUF      0x04       // OP_CODE 读从缓冲区

//SMBus 状态:
// MT = 主发送器
// MR = 主接收器
// ST = 从发送器
// SR = 从接收器

#define   SMB_BUS_ERROR    0x00    // （对所有方式）总线错误
#define   SMB_START        0x08    // (MT & MR) 起始条件已发送
#define   SMB_RP_START     0x10    // (MT & MR) 重复起始条件
#define   SMB_MTADDACK     0x18    // (MT) 从地址 + W 已发送；收到 ACK
#define   SMB_MTADDNACK    0x20    // (MT) 从地址 + W 已发送；收到 NACK
#define   SMB_MTDBACK      0x28    // (MT) 数据字节已发送；收到 ACK
#define   SMB_MTDBNACK     0x30    // (MT) 数据字节已发送；收到 NACK
#define   SMB_MTARBLOST    0x38    // (MT) 竞争失败
#define   SMB_MRADDACK     0x40    // (MR) 从地址 + R 已发送；收到 ACK
#define   SMB_MRADDNACK    0x48    // (MR) 从地址 + R 已发送；收到 NACK
#define   SMB_MRDBACK      0x50    // (MR) 收到数据字节；ACK 已发送
#define   SMB_MRDBNACK     0x58    // (MR) 收到数据字节；NACK 已发送
#define   SMB_SROADACK     0x60    // (SR) 收到自己的从地址+W；ACK 已发送

#define   SMB_SROARBLOST   0x68    // (SR) 收到自己的从地址+W；竞争失败
#define   SMB_SRGADACK     0x70    // (SR) 收到通用呼叫地址+W；ACK 已发送
#define   SMB_SRGARBLOST   0x78    // (SR) 在作为主器件发送从地址 + R/W
                                   // 时竞争失败。收到通用呼叫地址；
                                   // ACK 已发送
#define   SMB_SRODBACK     0x80    // (SR) 收到属于自己从地址的数据字节；
                                   // ACK 已发送
#define   SMB_SRODBNACK    0x88    // (SR) 收到属于自己从地址的数据字节；
                                   // NACK 已发送
#define   SMB_SRGDBACK     0x90    // (SR) 收到通用呼叫地址的数据字节；
                                   // ACK 已发送
#define   SMB_SRGDBNACK    0x98    // (SR) 收到通用呼叫地址的数据字节；
                                   // NACK 已发送
#define   SMB_SRSTOP       0xa0    // (SR) 在作为从其间被访问时
                                   // 收到停止条件或重复起始条件
#define   SMB_STOADACK     0xa8    // (ST) 收到自己的从地址+R；ACK 已发送
```

```
#define   SMB_STOARBLOST   0xb0   // (ST) 在作为主器件发送从地址 + R/W
                                  // 时竞争失败。收到自己的从地址;
                                  // ACK 已发送
#define   SMB_STDBACK      0xb8   // (ST) 数据字节已发送;收到 ACK
#define   SMB_STDBNACK     0xc0   // (ST) 数据字节已发送;收到 NACK
#define   SMB_STDBLAST     0xc8   // (ST) 最后数据字节已发送(AA=0);收到 ACK
#define   SMB_SCLHIGHTO    0xd0   // (ST & SR) SCL 时钟高电平定时器根据
                                  //  SMB0CR 已超时(FTE=1)
#define   SMB_IDLE         0xf8   // (对所有方式)等待


//-----------------------------------------------------------------------
//全局变量
//-----------------------------------------------------------------------

char COMMAND;                     // 在 SMBus ISR 中保存从地址 + R/W 位。

char WORD;                        // 保存 SMBus 待发送字节或刚收到的字节

char OP_CODE;                     // 保存待发送或刚收到的操作码

char LOST_COMMAND, LOST_WORD, LOST_CODE;    // 用于在竞争失败后保存相关数据

char DATA_BUF[16];                // 由 OP_CODE_HANDLER 访问的数据缓冲区

bit LOST;                         // 竞争失败标志,在主方式下竞争失败时置位。
                                  // 用于恢复失败的传输过程

bit SM_BUSY;                      // 该位在发送或接收开始时置 1。
                                  // 操作结束后由 ISR 清 0

bit VALID_OP;                     // 用于确定作为从器件收到的字节是操作码
                                  // 还是数据。

bit DATA_READY;                   // 由 OP_CODE 处理程序使用,用于指示
                                  // 从主器件接收的数据何时有效

//-----------------------------------------------------------------------
// 函数原型
//-----------------------------------------------------------------------

void SMBUS_ISR (void);
char SLA_READ(char chip_select, char out_op);
void SLA_SEND(char chip_select, char out_op, char out_data);
void OP_CODE_HANDLER(void);

//-----------------------------------------------------------------------
// 主程序
//-----------------------------------------------------------------------

void MAIN (void)
```

```
{
    char i, check_1, check_2;       // 只用于测试目的的变量

    WDTCN = 0xde;                   // 禁止看门狗定时器
    WDTCN = 0xad;

    XBR0 = 0x01;                    // 通过交叉开关将 SMBus 连到通用 I/O 引脚
    XBR2 = 0x40;                    // 允许交叉开关和弱上拉

    SMB0CN = 0x44;                  // 允许 SMBus，应答为低电平(AA = 1)
    SMB0CR = -80;                   // SMBus 时钟频率 = 100 kHz
    SMB0ADR = MY_ADD;               // 设置自己的从地址

    ADC0CN = 0x80;                  // 允许 ADC，用写 ADBUSY 启动转换

    ADC0CN |= 0x01;                 // ADC 数据寄存器左对齐

    DAC0CN = 0x84;                  // 允许 DAC0，数据寄存器左对齐

    REF0CN = 0x03;                  // 允许电压基准

    EIE1 |= 2;                      // SMBus 中断允许
    EA = 1;                         // 全局中断允许

    SM_BUSY = 0;                    // 为第一次传输释放总线
    SI = 0;                         //

//  OP_CODE_HANDLER();              // 该行只能在两个点对点器件中的一个里
                                    // 被注释掉。只用于测试目的。
                                    // 在正常设置下 OP_CODE_HANDLER 将一直运行，
                                    // 以便根据发送给该器件的操作码执行相应操作

// 测试代码-------------------------------------------------------------
// 本代码只用于测试两个器件间的接口。如果上面的 OP_CODE_HANDLER 行被去掉注释标记，
// 该器件被假设为主器件。另一个器件应一直运行 OP_CODE_HANDLER，响应下面的操作码：


    SLA_SEND(CHIP_B, (0x40 | WRITE_BUF), 0x24);     // 写到数据缓冲区下标 4
    SLA_SEND(CHIP_B, (0x60 | WRITE_BUF), 0x25);     // 写到下标 6
    SLA_SEND(CHIP_B, (0x80 | WRITE_BUF), 0x26);     // 写到下标 8
    SLA_SEND(CHIP_B, (0x10 | WRITE_BUF), 0x27);     // 写到下标 1

    check_1 = SLA_READ(CHIP_B, (0x40 | READ_BUF)); // 从缓冲区读下标 4
    check_1 = SLA_READ(CHIP_B, (0x60 | READ_BUF));  // 读下标 6
    check_1 = SLA_READ(CHIP_B, (0x80 | READ_BUF));  // 读下标 8
    check_1 = SLA_READ(CHIP_B, (0x10 | READ_BUF));  // 读下标 1


// 在循环内连续增加 CHIP_B 的 DAC 输出并在每次循环读 ADC。CHIP_B 的 DAC 输出应斜升。
```

```
    for (i=0;i<50;i++){
        SLA_SEND(CHIP_B, WRITE_DAC, 2*i);        // 写 2* i 到 CHIP_B 的 DAC0
        check_1 = SLA_READ(CHIP_B, READ_ADC);    // 读 CHIP_B 的 AIN0
        check_2 = 2*i;}                          // check_1 应与 check_2
                                                 // 基本一致
// 测试代码结束-------------------------------------------------------------

}

//---------------------------------------------------------------------------
// 函数
//---------------------------------------------------------------------------

//向从器件发送
// 发送函数向从器件发送两个字节：一个操作码和一个数据字节。有两个操作码用于发送数据：
// WRITE_DAC 和 WRITE_BUF。如果操作码为 WRITE_BUF，则操作码的高 4 位应含有缓冲区
// 的下表。例如，为了写数据缓冲区下标 2 对应的地址，操作码参数应为(0x20 | WRITE_BUF)。
//
// chip_select = 从器件地址。
// out_op = 要发送的操作码。
// out_data = 要发送的数据。
void SLA_SEND(char chip_select, char out_op, char out_data){

    while(SM_BUSY);                     // 在 SMBus 忙时等待
    SM_BUSY = 1;                        // SMBus 忙标志置位
    SMB0CN = 0x44;                      // 允许 SMBus，ACK 为低电平
    COMMAND = (chip_select | WRITE);    // COMMAND = 7 个地址位 + WRITE.
    OP_CODE = out_op;                   // WORD = 要发送的操作码。
    WORD = out_data;                    // DATA = 要发送的数据。
    STA = 1;                            // 启动传输过程。

}

// 读从器件
// 读函数发出一个字接的操作码，然后发出一个重复起始条件请求读一个字节。
// 可以在两个操作码 READ_ADC 和 READ_BUF 之间选择。如果操作码为 READ_BUF，
// 则操作码的高 4 位应含有缓冲区的索引下标。例如，要读数据缓冲区下标 5 对应的地址，
// 则操作码应为(0x50 | READ_BUF)。
//
// chip_select = 从器件地址。
// out_op = 要发送的操作码。
char SLA_READ(char chip_select, char out_op){

    while(SM_BUSY);                 // 在 SMBus 忙时等待。
    SM_BUSY = 1;                    // SMBus 忙标志置位
    SMB0CN = 0x44;                  // 允许 SMBus，ACK 为低电平
    COMMAND = (chip_select | READ);   // COMMAND = 7 个地址位 + READ
    OP_CODE = out_op;
    STA = 1;                        // 启动传输过程。
```

```
    while(SM_BUSY);                 // 等待传输结束
    return WORD;                    // 返回接收字

}

// OP_CODE handler.
// 对输入操作码译码并根据操作码执行相应的任务。
// 一旦被调用，将一直运行
//
// VALID_OP 位指示收到一个有效操作码。收到后，该处理程序对操作码译码并执行相应的任务，
// 然后清除 VALID_OP 以等待下一个操作码。
void OP_CODE_HANDLER(void){

    char index;                     // 数据缓冲区索引下标
    while (1){                      // 死循环
        VALID_OP = 0;               // 等待有效的 OP_CODE
        while (!VALID_OP);          //

        // OP_CODE 的低 4 位用于确定要执行的动作，而高 4 位在收到 READ_BUF 或
        // WRITE_BUF 操作码时用于指示 DATA_BUF 数组的下标。
        // 注意 SMBus 被冻结直到 OP_CODE 被译码。
        switch (OP_CODE & 0x0F){    // 对 OP_CODE 译码

            // OP_CODE = READ_ADC - 进行一次 ADC 并将结果放到输出缓冲区。
            // 只读 ADC 的高字节。
            case READ_ADC:
                SI = 0;             // 释放总线
                AA = 0;             // 使从器件'离线'
                ADCINT = 0;         // 清 ADC 中断标志。
                ADBUSY = 1;         // 启动转换。
                while (!ADCINT);    // 等待转换结束。
                WORD = ADC0H;       // 将数据放到输出缓冲区
                AA = 1;             // 使从器件返回'在线'状态
                VALID_OP = 0;       // 等待新的 OP_CODE
                break;

            // OP_CODE=WRITE_DAC -等待一个有效数据字节，并将其写到 DAC0 的高字节。
            case WRITE_DAC:
                SI = 0;                     // 释放总线
                DATA_READY = 0;             // 等待有效数据。
                while (!DATA_READY);        //
                DAC0L = 0;                  // DAC 低字节
                DAC0H = WORD;               // DAC 高字节
                VALID_OP = 0;               // 等待新的 OP_CODE
                SI = 0;                     // 结束时释放总线
                break;

            // OP_CODE = WRITE_BUF - 等待有效数据字节，并将其存入 DATA_BUF 数组。
            // 根据 OP_CODE 的高 4 位确定数组下标。
```

```
            case WRITE_BUF:
                SI = 0;                         // 释放总线
                index = (OP_CODE & 0xF0);       // 用高 4 位作为数组下标
                DATA_READY = 0;                 // 等待有效数据。
                while (!DATA_READY);            //
                DATA_BUF[index] = WORD;         // 将数据存入数组
                VALID_OP = 0;                   // 等待新的 OP_CODE
                SI = 0;                         // 结束时释放总线
                break;

            // OP_CODE = READ_BUF - 读 DATA_BUF 数组并将字节存入输出缓冲区。
            // 数组下标由 OP_CODE 的高 4 位决定。
            case READ_BUF:
                index = (OP_CODE & 0xF0);       // 用高 4 位作为数组下标
                WORD = DATA_BUF[index];         // 将索引字节存入输出缓冲区
                VALID_OP = 0;                   // 等待新的 OP_CODE
                SI = 0;                         // 结束时释放总线
                break;
            }

        if (LOST){                              // 如果 LOST 被置位，说明器件在最近
            COMMAND = LOST_COMMAND;             // 的竞争中失败。将保存值回装到
            WORD = LOST_WORD;                   // 传输变量并重试传输过程。
            OP_CODE = LOST_CODE;
            LOST = 0;
            STA = 1;
            }
    }
}

//-----------------------------------------------------------------------
// SMBus 中断服务程序
//-----------------------------------------------------------------------

void SMBUS_ISR (void) interrupt 7
{
    switch (SMB0STA){                           // SMBus 的状态码(SMB0STA 寄存器)

        // 主发送器/接收器：起始条件已发出。
        // 将从器件地址装入 SMB0DAT。屏蔽 R/W 位，因为所有传输过程都从
        // 写 OP_CODE 开始。
        case SMB_START:
            SMB0DAT = (COMMAND & 0xFE);         // 装入待访问的从器件地址
                                                // 屏蔽 R/W 位，因为第一次传输
                                                // 总是写 OP_CODE。
            STA = 0;                            // 手动清除 STA 位
            SI = 0;                             // 清除中断标志
            break;

        // 主发送器/接收器：重复起始条件已发出。
        // 该状态只出现在读操作，在发出 OP_CODE 之后。将器件地址 + READ 装入 SMB0DAT。
```

```
case SMB_RP_START:
    SMB0DAT = COMMAND;
    STA = 0;                           // 手动清除 STA 位
    SI = 0;
    break;

// 主发送器：从地址 + WRITE 已发出。收到 ACK。
// 将 OP_CODE 装入到 SMB0DAT。
case SMB_MTADDACK:
    SMB0DAT = OP_CODE;
    SI = 0;                            // 清除中断标志
    break;

// 主发送器：从地址 + WRITE 已发出。收到 NACK。
// 从器件不响应。用 ACK 查询重试。
case SMB_MTADDNACK:
    STO = 1;
    STA = 1;
    SI = 0;                            // 清除中断标志
    break;
```

```
// 主发送器：数据字节已发出。收到 ACK。
// 检查 OP_CODE - 如果是读操作码，发出重复起始条件开始读。如果是写操作码，
// 将 WORD 装入 SMB0DAT 以待传输。如果不是有效操作码，则有两种情况：
// 1）数据已发送，传输过程结束，或 2）这是一个错误。
// 在任何一种情况，发出停止条件结束传输。
case SMB_MTDBACK:
    switch (OP_CODE & 0x0F){    // 只检查低 4 位

        // OP_CODE 为 READ。发出重复起始条件。
        case READ_BUF:
        case READ_ADC:
            OP_CODE = 0;          // 当前 OP_CODE 不再有用
            STA = 1;
            break;

        // OP_CODE 为 WRITE。将输出数据装入 SMB0DAT。
        case WRITE_BUF:
        case WRITE_DAC:
            SMB0DAT = WORD;
            OP_CODE = 0;          // 清除 OP_CODE，使传输过程在下一次
            break;                // 出现该状态时结束(在发出数据之后)。

        default:                  // 无有效 OP_CODE。结束传输
            STO = 1;
            SM_BUSY = 0;
            break;
    }
    SI = 0;
    break;

// 主发送器：数据字节已发出。收到 NACK。
// 用 ACK 查询并重试传输。
case SMB_MTDBNACK:
    STO = 1;
    STA = 1;
    SI = 0;                       // 清除中断标志
    break;

// 主发送器：竞争失败。
case SMB_MTARBLOST:
    LOST_COMMAND = COMMAND;       //
    LOST_WORD = WORD;             // 保存变量，以备总线空闲时使用
    LOST_CODE = OP_CODE;          //

    LOST = 1;                     // 设置总线空闲时的重试标志
    SI = 0;                       // 清除中断标志
    break;

// 主接收器：从地址 + READ 已发出。收到 ACK。
// 设置为在下一次传输后发送 NACK，因为这将是最后（唯一）字节。
case SMB_MRADDACK:
    AA = 0;                       // 在应答周期发送 NACK
    SI = 0;
```

```
    break;

// 主接收器：从地址 + READ 已发出。收到 NACK。
// 从器件不响应。重新发送重复起始条件。
case SMB_MRADDNACK:
    STA = 1;
    SI = 0;
    break;

// 主接收器：收到数据字节。ACK 已发出。
// 该状态不应出现，因为 AA 已在前一状态被清除。
// 如果出现，发送停止条件。
case SMB_MRDBACK:
    STO = 1;
    SM_BUSY = 0;
    SI = 0;
    break;

// 主接收器：收到数据字节。NACK 已发出。
// 读操作完成。读数据寄存器并发送停止条件。
case SMB_MRDBNACK:
    WORD = SMB0DAT;
    STO = 1;
    SM_BUSY = 0;
    AA = 1;                         // 为下一次传输设置 AA
    SI = 0;
    break;

// 从接收器：竞争失败，收到通用呼叫地址。
// 置位 LOST 标志以备总线空闲时重试。本次传输失败。
case SMB_SRGARBLOST:

// 从接收器：竞争失败，收到自己的从地址 + WRITE。
// 置位 LOST 标志以备总线空闲时重试。
// 置位 STO 位以退出主方式
case SMB_SROARBLOST:
    LOST_COMMAND = COMMAND;     //
    LOST_WORD = WORD;           // 保存变量，以备总线空闲时使用
    LOST_CODE = OP_CODE;        //
    LOST = 1;                   // 总线空闲时重试。
    SI = 0;
    break;

// 从接收器：收到自己的从地址 + WRITE。ACK 已发出。本次传输失败。
case SMB_SROADACK:

// 从接收器：收到通用呼叫地址。ACK 已发出。
case SMB_SRGADACK:
    SI = 0;
    break;

// 从接收器：在收到通用呼叫地址 + WRITE 之后收到数据字节。
// ACK 已发出。本次传输失败。
case SMB_SRGDBACK:
```

```
// 从接收器：在收到自己的从地址 + WRITE 之后收到数据字节。ACK 已发出。
// 根据所收到的 OP_CODE 或数据进行相应操作。
case SMB_SRODBACK:
    if (!VALID_OP){              // 如果 VALID_OP=0，该字节为 OP_CODE。
       OP_CODE = SMB0DAT;        // 保存 OP_CODE
       VALID_OP = 1;             // 下一字节不是 OP_CODE
       }
    else{
       DATA_READY = 1;           // 收到有效数据。
       WORD = SMB0DAT;           // 在 OP_CODE 处理程序中处理
       SI = 0;}
    break;

// 从接收器：被作为从器件访问时收到数据字节。NACK 已发出。
// 该状态不应出现，因为作为从器件 AA 不会被清除。本次失败，进入下一状态。
case SMB_SRODBNACK:

// 从接收器：被作为通用呼叫地址访问时收到数据字节。NACK 已发出。
// 该状态不应出现，因为作为从器件 AA 不会被清除。
case SMB_SRGDBNACK:
    AA = 1;
    SI = 0;
    break;

// 从接收器：被作为从器件访问时收到停止条件或重复起始条件。
case SMB_SRSTOP:
    SI = 0;
    break;

// 从发送器：收到自己的从地址 + READ。ACK 已发出。
// 将待输出数据装入到 SMB0DAT。
case SMB_STOADACK:
    SMB0DAT = WORD;
    SI = 0;
    break;

// 从发送器：作为主器件竞争失败。收到自己的从地址 + READ。ACK 已发出。
case SMB_STOARBLOST:
    LOST_COMMAND = COMMAND;      //
    LOST_WORD = WORD;            // 保存变量，以备总线空闲时使用
    LOST_CODE = OP_CODE;         //
    LOST = 1;                    // 总线空闲时重试

    SI = 0;
    break;

// 从发送器：收到数据字节。收到 ACK。失败。
case SMB_STDBACK:

// 从发送器：收到数据字节。收到 NACK。失败。
case SMB_STDBNACK:
```

```
        // 从发送器：最后数据字节已发送。收到 ACK。无需任何操作。
        case SMB_STDBLAST:
            SI = 0;
            break;

        // 所有其它状态码无效。通信复位。
        default:
            STO = 1;
            SM_BUSY = 0;
            break;
        }

}
```