
SERIAL COMMUNICATION WITH THE SMBUS

Relevant Devices

This application note applies to the following devices:

C8051F000, C8051F001, C8051F002, C8051F005, C8051F006, C8051F010, C8051F011, C8051F012, C8051F020, C8051F021, C8051F022, and C8051F023.

Introduction

C8051F0xx devices are equipped with an SMBus serial I/O device that is compliant with the System Management Bus Specification version 1.1, as well as the I²C serial bus. The SMBus is a bi-directional, 2-wire interface capable of communication with multiple devices. SMBus is a trademark of Intel; I²C is a trademark of Philips Semiconductor.

This application note describes configuration and operation of the SMBus. Example assembly and C code is given: (1) Interfacing a single EEPROM with 1-byte address space, in assembly; (2) Interfacing multiple EEPROMs with 2-byte address space, in C; and (3) Peer-to-peer communication between two C8051F0xx devices, in C.

SMBus Specification

This section presents a description of the SMBus protocol. The SMBus discussion begins in the next section--Using the SMBus.

SMBus Structure

An SMBus system is a 2-wire network, where each device has a unique address and may be addressed by any other device on the network. All transfers are initiated by a master device; if a device recog-

nizes its own address and responds, it becomes the slave device for that transfer. It is important to note that assigning one specified master device is not necessary. Any device may assume the role of master or slave for any particular transfer. In the case that two devices attempt to initiate a transfer simultaneously, an arbitration scheme forces one device to give up the bus. This arbitration scheme is non-destructive (one device wins and no information is lost). Arbitration is discussed in depth in the arbitration section.

Two wires are used in SMBus communication: SDA (serial data), and SCL (serial clock). Each line is bi-directional, with direction depending on what modes the devices are in. The master always supplies SCL; either device may transmit on SDA. Both lines should be connected to a positive power supply through a pull-up circuit. All devices on the SMBus line should have an open-drain or open collector output, so that the lines may remain high when the bus is free. The line is pulled low if one or more devices attempts to output a LOW signal. All devices must output a HIGH for the line to stay high. A typical SMBus configuration is shown in Figure 1 on page 2.

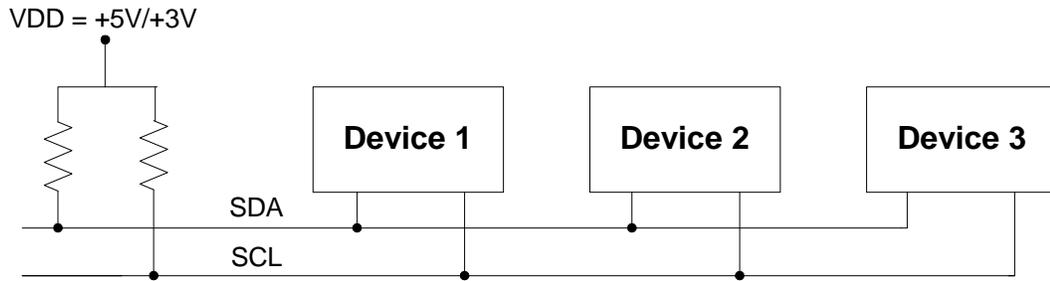


Figure 1. Typical SMBus Configuration

Handshaking

SMBus employs various line conditions as handshaking between devices. Note that during a data transfer, SDA is only allowed to change levels while SCL is low. Changes on SDA while SCL is high represent START and STOP signals, as follows:

START: This initiates a transfer. It consists of a falling edge on SDA while SCL is high.

STOP: This ends a transfer. It consists of a rising edge on SDA while SCL is high.

ACKNOWLEDGE: Also referred to as an ACK, this is transmitted by a receiving device as a confirmation. For example, after device_X receives a byte, it transmits an ACK to confirm the transfer. An ACK consists of a low level on SDA sampled when SCL is high.

NOT_ACKNOWLEDGE: Also referred to as a NACK, this is a high SDA while SCL is high.

When a receiving device fails to ACK, the sending device sees a NACK. In typical transfers, a received NACK indicates that the addressed slave is not ready for transfer, or is not present on the bus. A receiving master may transmit a NACK to indicate the last byte of a transfer. Both of these situations are discussed further in the next section. Figure 2 illustrates the handshaking signals.

Transfer Modes

Two types of transfers are possible: a WRITE (transfer from master to slave) and a READ (transfer from slave to master). During a transfer, any device may assume one of four roles. These four roles are explained below. Note that ‘slave address + R/W’ refers to an 8 bit transfer (7 address, 1 R/W).

1) **Master Transmitter:** In this mode, the device transmits serial data on SDA and drives the clock on SCL. The device initiates the transfer with a START condition, sends the slave address + W, and waits for an ACK from the slave. After the ACK,

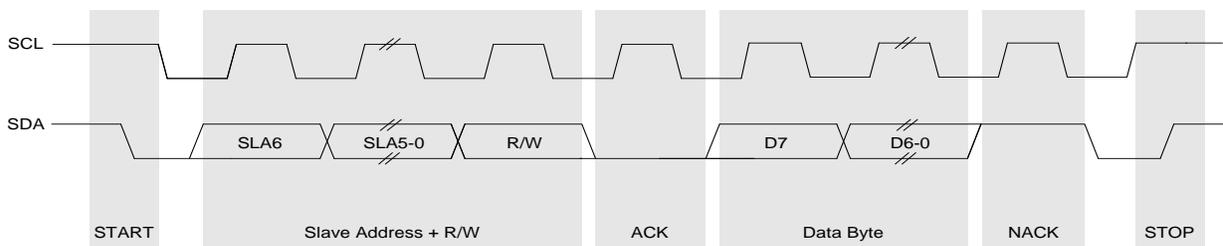


Figure 2. SMBus Timing

the device transmits one or more bytes of data, with each byte ACK'ed by the slave. After the last byte, the device transmits a STOP.

2) Master Receiver: In this role, the device receives serial data on SDA while driving the clock on SCL. The device initiates the transfer with a START followed by the slave address + R. After the slave ACK's the address, the device will output the clock on SCL, and receive data on SDA. After the last byte, the device will issue a NACK followed by a STOP.

3) Slave Transmitter: In this role, a device outputs serial data on SDA and receives the clock on SCL. The device receives a START followed by its own slave address + R, then ACK's, and enters slave transmitter mode. The device transmits serial data on SDA and receives an ACK after each byte. After

the last byte, the master will issue a NACK followed by a STOP.

4) Slave Receiver: In this role, a device receives a START followed by its own slave address + W from a master device. The device sends an ACK and enters slave receiver mode. The device now receives serial data on SDA and the clock on SCL. The device ACK's after each byte is received, and exits slave mode after the master issues a STOP. Figure 3 shows the typical WRITE scenarios. (1) shows a successful transfer.

In (2), the master receives a NACK after sending the slave address + W. This occurs when a slave is 'offline', meaning it is not responding to its own address. In this case, the master should issue a STOP or repeated START. To retry the transfer, the master follows the STOP with a START and the slave address + W again. The master will repeat the

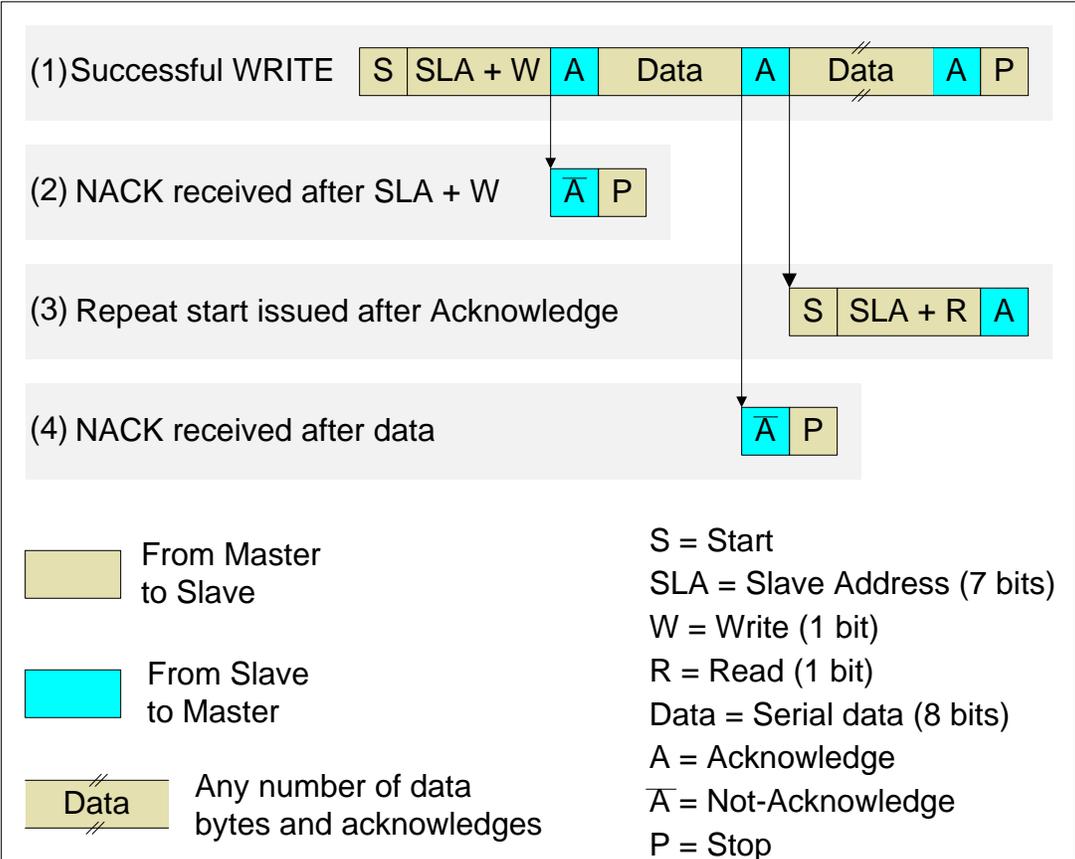


Figure 3. Typical WRITE Transfer Scenarios

AN113

cycle until it receives an ACK. This is referred to as “acknowledge polling”.

In (3), the master issues a repeated START after an ACK. This process allows the master to initiate a new transfer without giving up the bus (to switch from a WRITE to a READ, for example). The repeated START is commonly used in EEPROM memory access applications, where a memory READ must be directly preceded by a WRITE of the desired memory location. The repeated START is demonstrated in all three code examples.

In (4), a NACK is received after a data byte. In typical SMBus systems, this is how the receiving device indicates an error. The master sends a STOP, and retries the transfer as in (2), or gives up the transfer. Note that the use of NACKs is not restricted to error situations; the acknowledge level is a user-definable characteristic, and may vary in different applications.

Figure 4 shows the typical READ scenarios. (1) shows a successful READ operation. In (2), the

master receives a NACK after sending the slave address + R. This situation is handled in the same fashion as in (2) of the WRITE discussion. The master can use acknowledge polling to retry the transfer, or it can give up the transfer. (3) Shows the master sending a repeated START after sending a byte of data. This is the same repeated START state as in the WRITE discussion. A master may send a repeated START after any data byte, and may initiate a READ or a WRITE following the repeated START. Generally a repeated START is used to change direction (R/W) or to change addresses (slave devices).

Note that the READ and WRITE diagrams show only the typical scenarios. Bus errors, time outs, and arbitration are also possible occurrences. Time-outs are used to detect when a transfer has stalled or when the bus is free. Often a device may hold SCL low until it is ready to continue a transfer. This process allows a slower slave device to communicate with a faster master, since stalling the bus effectively reduces the SCL frequency. The SMBus protocol specifies that all devices on the SMBus

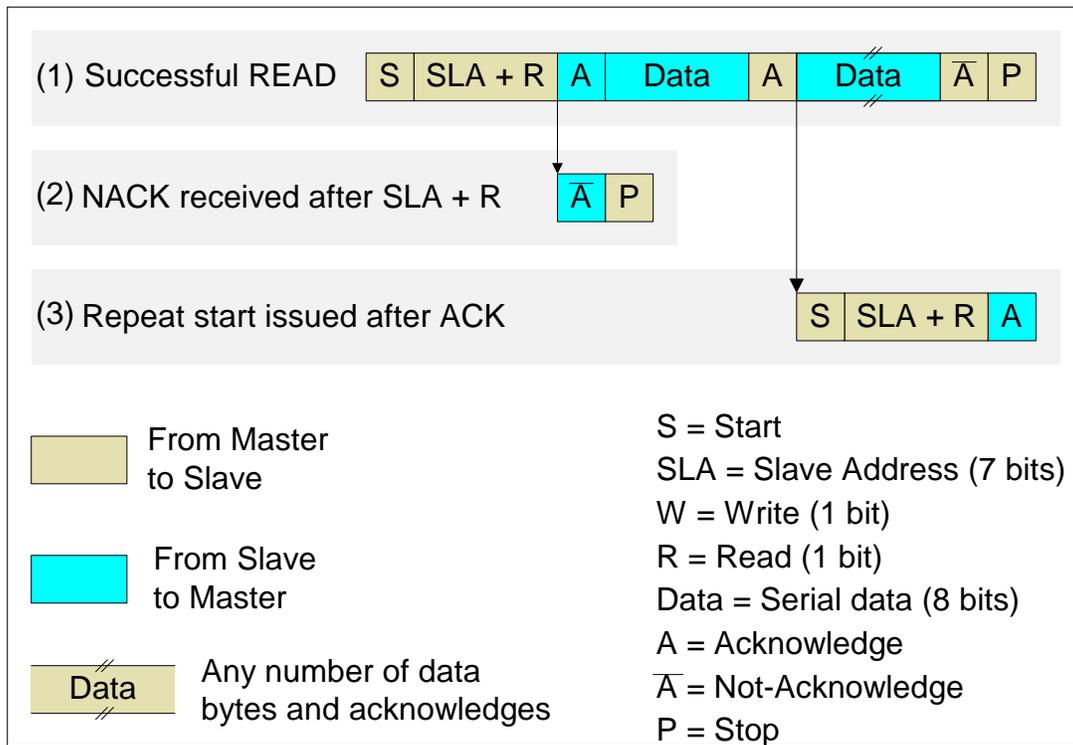


Figure 4. Typical Read Scenarios

must declare any SCL signal held low for more than 25 ms a “timeout”. In this case, all devices on the bus must reset communication. A high SCL timeout may also occur. If both SDA and SCL remain high for more than 50 μsec, the bus is designated as free.

Arbitration

If multiple masters are configured on the same SMBus system, it is possible that two will attempt to initiate a transfer at the same time. If this happens, an arbitration scheme is employed to force one device to give up the bus.

What the scheme is: both masters continue to transmit until one attempts a HIGH while the other attempts a LOW. Due to the open-drain bus, the device attempting a LOW will win the bus. The HIGH device gives up the bus, and the other device continues its transfer. Note that the collision is non-destructive: one device always wins.

How it works: Assume device_X and device_Y contend for the bus. The winner, device_X, is not affected at all by the arbitration. Since data is shifted into the SMBus data register as it is shifted out, device_Y does not miss any data. Figure 5 shows an example output sequence between two devices during arbitration. Note that Device_Y begins receiving data after it gives up the bus.

Using the SMBus

The SMBus can operate in both master and slave modes. The hardware provides timing and shifting control for the serial transfers; byte-wise control is user-defined. The SMBus hardware performs the following application-independent tasks:

Timing Control: In master mode, the hardware generates the clock signal on SCL and synchronizes the data on SDA. Hardware also recognizes timeouts and bus errors.

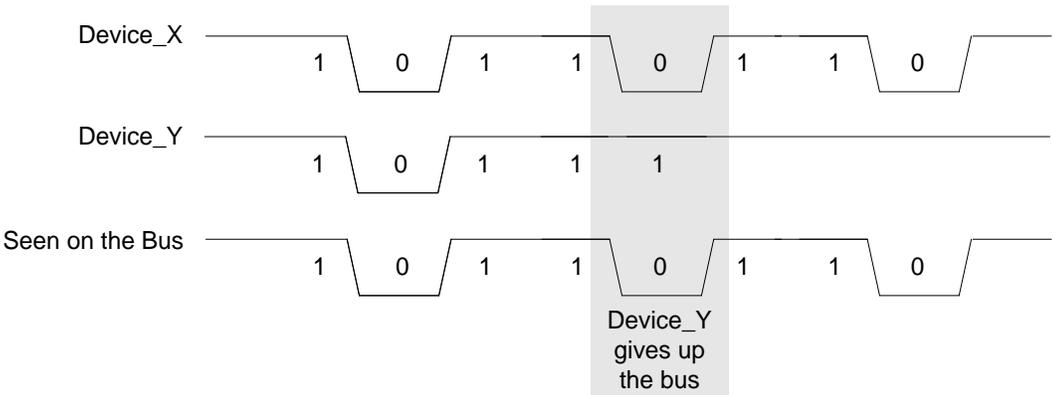
Serial Data Transfers: The hardware controls all shifting of data to and from SDA, including the acknowledge level. The acknowledge level is user-defined, as explained in the register definitions below.

Slave Address Recognition: The hardware recognizes a START from another device, and reads the following slave address. If the slave address matches the contents of the SMBus Address Register (defined below), then the hardware acknowledges the address. Note that this features is only enabled if AA (Address Acknowledge) is set.

Configuration and Control

SMBus operation is determined by the contents of the following registers.

Figure 5. Arbitration Sequence



SMB0STA. The SMBus Status Register holds an 8-bit status code for the current state of the SMBus. The contents of SMB0STA are only defined when the SI bit is set. There are 28 possible states, all of which have a unique code (the codes are multiples of 8). SMB0STA should never be written to. The 28 possible states and their descriptions are given in Table 1 on page 12.

SMB0CN. The SMBus control register is used to enable the SMBus and navigate the possible SMBus states. This register includes START and STOP control, as well as interrupt, acknowledge, and timeout control.

A transfer is initiated by setting the STA bit. The SMBus hardware will wait until the bus is free, then transmit a START. **Note that STA is not cleared by hardware.** User software must manually clear STA so that an unwanted repeated START is not generated. **User software must also manually clear STO prior to setting STA.**

A transfer is ended by setting the STO bit. In master mode, setting STO will cause a STOP condition to be generated. If STA is set when STO is set, a STOP followed by a START will be transmitted. In slave mode, setting STO will cause the hardware to act as if a STOP was received, though no STOP condition is transmitted.

The SI bit is set when any of the possible 28 SMBus states are entered (excluding the idle state). This bit is not automatically cleared by hardware. **Note that SCL is held low while SI is set.** This means that the bus is stalled until SI is cleared, synchronizing the master with the slave.

The AA bit determines the type of acknowledge returned during the acknowledge cycle. If AA=1, an ACK will be sent; if AA=0, a NACK will be sent. This means the device will respond to its slave address only if AA is set.

SCL high and low timeout detection is enabled by setting the FTE and TOE bits, respectively.

The SMBus is enabled by setting the SMBus enable bit, ENSMB.

SMB0CR. The SMBus clock register is used to control the SCL clock rate when the device is in master mode. The 8 bits held in the SMB0CR register determine the clock rate as follows:

$$\text{SMB0CR} \cong - \frac{\text{SYSCLK}}{2 \times F_{\text{SCL}}} \quad \langle 1 \rangle$$

Where SMB0CR is a 2's complement negative number. So for a SCL frequency of 100 kHz and a SYSCLK of 16 MHz, SMB0CR should be loaded with -80, or 0xB0.

SMB0CR also defines the limit for the bus free time period (high SCL timeout). The bus free time is defined by the following equation, where SMB0CR is a 2's complement negative number. Note that T_{Free} is about 5 bit periods.

$$T_{\text{Free}} = - \frac{(10 \times \text{SMB0CR}) + 1}{\text{SYSCLK}} \quad \langle 2 \rangle$$

SMB0ADR. The SMBus Address Register holds the slave address that the device will respond to in slave mode. Bits(7:1) hold the slave address; bit0 is the General Call Enable. If bit0 is set, the device will respond to the general call address (0x00).

SMB0DAT. The SMBus Data Register is used to hold data to be transmitted or data that has just been received by the SMBus. Data read from this register is only valid while SI = 1. When SI is not set, the SMBus may be in the process of shifting data in or out of SMB0DAT. Note that when transmitting, data shifted out of the most significant bit of SMB0DAT is shifted back into the least significant bit, so that after a transmit the original data is still contained in SMB0DAT.

Implementation Choices

User software controls the SMBus on a state-by-state basis. Upon each state change, the SI bit is set by hardware, and an interrupt generated if interrupts are enabled. The SMBus is then halted until user software services the state change and clears the SI bit. The SMBus operation is most easily defined in a state table; however, note that it is not necessary to define all 28 states. For example, if the SMBus is the only master in the system, the slave and arbitration states may be left undefined. If the SMBus will never operate as a master, the master states may be left undefined. If states are left undefined, a default response should be programmed to account for unexpected or error situations.

The SMBus state table lends itself to a case-switch statement definition in C. However, for simple or time-restricted systems, an assembly state decoding can be more efficient. Note that the status codes held in SMB0STA are multiples of 8. If the SMBus states are programmed in 8-byte segments, SMB0STA may be used as a software index. In this case, a status code is decoded in 3 assembly commands. However, only 8 bytes of code space are available for each state definition. For states that require more than 8 bytes, the program must branch out of the state table so that subsequent states are not disturbed.

Examples

Three examples are provided: a single EEPROM with 1-byte address space, in assembly; multiple EEPROMs with 2-byte address space, in C; and a peer-to-peer interface between two devices, in C. Each example uses interrupt-driven operation.

Single EEPROM

This is a simple interface between the SMBus and a 256-byte EEPROM. The SMBus acts as the master at all times. The transfer procedure is similar to that of any 2-wire EEPROM interface.

The Send operation is a 1-byte random WRITE. The SMBus sends a START followed by three bytes: the EEPROM's device address + W (this address is found in the EEPROM datasheet), the memory location to be written, and then the data byte. The slave should ACK after each byte. If the master receives an ACK after each byte, it sends a STOP and the transfer is over. If at any time the master receives a NACK, it will retry the transfer using acknowledge polling. It is common for an EEPROM to NACK if multiple read/write operations are performed sequentially, since most self-timed EEPROMs go offline to actually perform the memory write. Figure 6 shows SDA for the Single EEPROM send operation.

Figure 6. Single EEPROM Send Sequence



The Receive operation is a 1-byte random READ. The transfer begins, as in the WRITE function, with the master sending a START followed by the EEPROM device address + W (a WRITE is used to set the EEPROM's "current address"). After the slave ACK's, the master sends the memory location to be read. Upon receipt of an ACK, the master then issues a repeated START followed by the slave address + R. Now after the slave ACK's, it will send the data byte read from the location given in the preceding "aborted" WRITE. The master sends a NACK (since this data is the last and only byte), followed by a STOP. The repeated START is used in this case so that no other transfers may begin between the WRITE of the memory address and the READ of the data byte. Figure 7 shows SDA for a Single EEPROM Receive operation.

The software for this example was written in assembly to demonstrate the advantage of using SMB0STA as a software index. The SMBus state table written in 8-byte memory segments (8 bytes for each state). This is accomplished through the use of an 'org' statement for each state, offset from the beginning of the table by the corresponding sta-

tus code. For example, if the state table is labeled STATE_TABLE, and State_1 is 0x08, the code segment for State_1 should begin with:

```

; State_1
org STATE_TABLE + 08h
; State_1 code

```

Now when SMBOSTA holds 0x80, State_1 may be accessed with the following:

```

; Load current State
mov A, SMBOSTA;

; Point DPTR to start of table
mov DPTR, #STATE_TABLE;

; Jump to indexed state
jmp @A+DPTR;

```

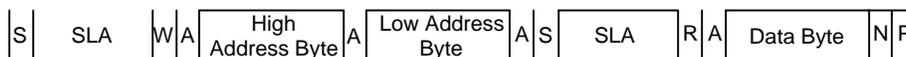
This process allows for very efficient state decoding. However, it is important to note that only 8 bytes of code space are available for each state. If a state requires more than 8 bytes, the program must jump to a segment outside of the state table, so that the next state definition is not disturbed.

To keep the states simple and understandable, the SMBus is assumed to be the only master in the system. The slave states are not defined, and the arbitration states ignore any received data. Also, the repeated START state may assume the transfer is a READ. The code listing begins on page 14.

Figure 7. Single EEPROM Receive Sequence



Figure 8. Multiple EEPROM Receive Sequence



Multiple EEPROMs

Example 2 uses multiple EEPROMs with 2-byte address space. The software is written in C. The three EEPROMs used are 8k-bytes. Note that three identical EEPROMs are used. The EEPROMs have three address selection pins, A0 - A2, that are used to set the slave address for the devices. The four high bits of the device address are set in EEPROM to “0101”; the lower three bits of the slave address are determined by the setting of the address pins (VDD for 1, GND for 0). Figure 9 shows the device configuration.

The distinction with this example is that the EEPROMs have a 2-byte address space. This means that the READ and WRITE operations must send an extra address byte for each transfer (see Figure 8) When the Interrupt Service Routine reaches the “Data Transmitted, ACK Received” state, it must know which byte was transmitted--the high address byte, the low address byte, or the data byte. This information is kept in the BYTE_NUMBER state variable.

The SMBus ISR is implemented as a case-switch statement, with the SMBus status code (SMB0STA) used as the switch variable. The code listing for this example begins on page 23.

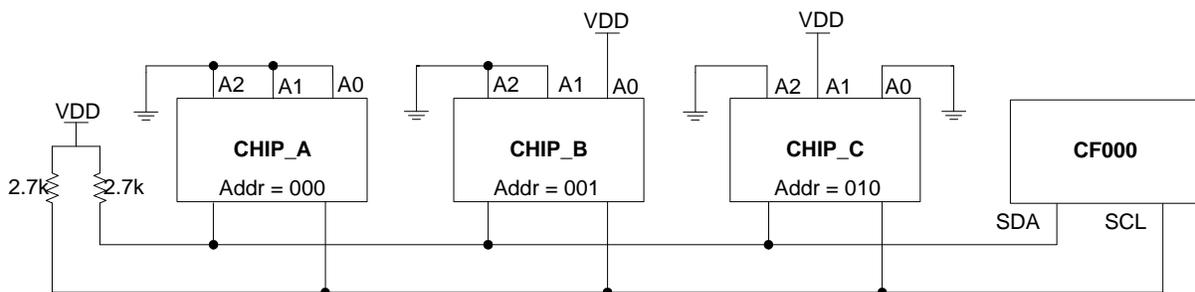


Figure 9. Multiple EEPROM Configuration

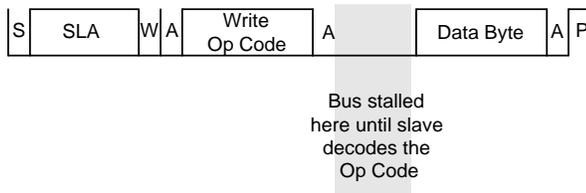
Peer-to-Peer Interface

The final example features two C8051F0xx devices configured to communicate as peers. The peer-to-peer interface uses a set of op codes to perform the set of tasks below. Either device may initiate a transfer.

Write to slave DAC: The master device sends a WRITE_DAC op code followed by a byte of data. Upon receipt, the slave device writes the data to its DAC0 port.

Write to buffer: The master device sends a WRITE_BUF op code, followed by a byte of data for the receiving device to store in a buffer. The upper 4 bits of the WRITE_BUF op code hold the buffer index. Figure 10 shows a peer-to-peer WRITE sequence (same for both DAC and buffer writes).

Figure 10. Peer-to-Peer Write Sequence



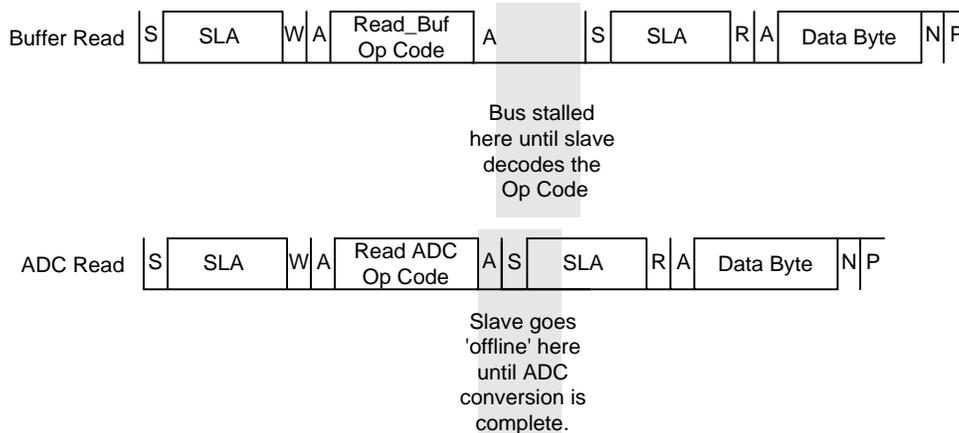
Read ADC: The master device sends a READ_ADC op code followed by a repeated

START. The slave reads its ADC input, and places the data in its SMB0DAT register. In this case, the slave clears AA to go 'offline' during the ADC conversion. While the slave is offline, the master receives a NACK after the repeated START and slave address. The master continues acknowledge polling until the slave responds. This technique is useful if the slave's operation is time-consuming, since other devices may use the bus while the slave is offline. The slave sets AA=1 when it is ready, and the transfer continues. The master requests a READ after the slave acknowledges. See Figure 11 for the transmission sequence.

Read buffer: The master sends a READ_BUF op code followed by a repeated START. The upper 4 bits of the op code hold the buffer index. In this case the slave holds the SCL line low while it decodes the op code. While SCL is held low, the master cannot attempt to continue the transfer. Additionally, no other masters on the bus may attempt a transfer. This bus stalling technique is useful when the slave's delay is short. The slave releases SCL when it has finished decoding the op code and is ready to transmit the data. The master issues the repeated START and the slave address + R. See Figure 11.

The SMBus operation in this example is defined as a case-switch statement in the SMBus ISR. All possible states are defined, including the arbitration states. If arbitration occurs, the losing device stores

Figure 11. Peer-to-Peer Read Sequence



its current transfer data (target slave address, op code, relevant data) and responds to the received op code. After the transfer is finished, the losing device retries the transfer by reverting to the saved transfer data.

An `OP_CODE_HANDLER` function runs in polled mode to process received data. When the device receives a valid op code, the `OP_CODE_HANDLER` decodes it and reacts appropriately.

To test the bus, comment out the `OP_CODE_HANDLER` call in the code for `CHIP_A`. This will allow `CHIP_A` to run the provided test code. Note that the constant `MY_ADD` must be unique to each device on the bus.

The code listing for this example begins on page 29.

Table 1. SMBus Status Codes and States

Mode	Status Code	SMBus State	Typical Action
MT/ MR	0x08	START condition transmitted.	Load SMB0DAT with Slave Address + R/W
	0x10	Repeated START condition transmitted.	Load SMB0DAT with Slave Address + R/W
Master Transmitter	0x18	Slave Address + W transmitted. ACK received.	Load SMB0DAT with data to be transmitted. Clear STA
	0x20	Slave Address + W transmitted. NACK received.	Acknowledge poll to retry. Set STO + STA
	0x28	Data byte transmitted. ACK received.	1) Load SMB0DAT with next byte, OR 2) Set STO, OR 3) Clear STO, then set STA for repeated START
	0x30	Data byte transmitted. NACK received.	1) Retry transfer OR 2) Set STO
	0x38	Arbitration Lost.	Save current data
Master Receiver	0x40	Slave Address + R transmitted. ACK received.	Clear STA. Wait for received data.
	0x48	Slave Address + R transmitted. NACK received.	Acknowledge poll to retry. Set STO + STA
	0x50	Data byte received. ACK transmitted.	Read SMB0DAT. Wait for next byte. If next byte is last byte, clear AA
	0x58	Data byte received. NACK transmitted.	Set STO

Table 1. SMBus Status Codes and States

Mode	Status Code	SMBus State	Typical Action
Slave Receiver	0x60	Own slave address + W received. ACK transmitted.	Wait for data
	0x68	Arbitration lost in sending SLA + R/W as master. Own address + W received. ACK transmitted.	Save current data for retry when bus is free. Wait for data
	0x70	General call address received. ACK transmitted.	Wait for data
	0x78	Arbitration lost in sending SLA + R/W as master. General call address received. ACK transmitted.	Save current data for retry when bus is free.
	0x80	Data byte received. ACK transmitted.	Read SMB0DAT. Wait for next byte or STOP
	0x88	Data byte received. NACK transmitted.	Set STO to reset SMBus
	0x90	Data byte received after general call address. ACK transmitted.	Read SMB0DAT. Wait for next byte or STOP
	0x98	Data byte received after general call address. NACK transmitted.	Set STO to reset SMBus
	0xA0	STOP or repeated START received.	No action necessary
Slave Transmitter	0xA8	Own address + R received. ACK transmitted.	Load SMB0DAT with data to transmit.
	0xB0	Arbitration lost in transmitting SLA + R/W as master. Own address + R received. ACK transmitted.	Save current data for retry when bus is free. Load SMB0DAT with data to transmit.
	0xB8	Data byte transmitted. ACK received.	Load SMB0DAT with data to transmit.
	0xC0	Data byte transmitted. NACK received.	Wait for STOP
	0xC8	Last data byte transmitted (AA=0). ACK received.	Set STO to reset SMBus
Slave	0xD0	SCL Clock High Timer per SMB0CR timed out	Set STO to reset SMBus
All	0x00	Bus Error (illegal START or STOP)	Set STO to reset SMBus
	0xF8	Idle	State does not set SI

Software Examples for the C8051F00x and C8051F01x series

```
-----  
;  
;  
; Copyright 2001 Cygnal Integrated Products, Inc.  
;  
; Program: SMBus_EX1.asm  
; Created on: 2/21/01  
; Last mod  : 27 AUG 03 -- BW  
; Created by: JS  
;  
; Example code to interface a single 256-byte EEPROM to a C8051F00x via the SMBus  
; Code assumes a single EEPROM with slave address 1010000 is connected on  
; the SDA and SCL lines, and no other masters are on the bus.  
;  
; The SEND routine performs a 1-byte write to the EEPROM. This consists of (1) START,  
; (2) slave address + W, (3) memory location byte write, and (4) a data byte write.  
;  
; STEPS FOR WRITING TO EEPROM:  
; 1) Load slave address into SLA_ADD  
; 2) Load memory address into MEM_ADD  
; 3) Load data byte into TRANSMIT_BYTE.  
; 4) Call SEND  
;  
; The RECEIVE routine performs a 1-byte read from the EEPROM. This consists of (1)  
; START, (2) slave address + W, (3) memory location byte write, (4) repeated START,  
; (5) slave address + R, (6) data byte read.  
;  
; STEPS FOR RECEIVING DATA:  
; 1) Load slave address into SLA_ADD  
; 2) Load memory address into MEM_ADD  
; 3) Call RECEIVE  
; 4) Read RECEIVE_BYTE  
;  
; The SMBus state table is broken into 8-byte state segments, allowing the SMBus  
; status code (SMB0STA) to be used as a state index. Note that this leaves only  
; 8 bytes of code space per SMBus state definition. As a result, certain tasks  
; have been altered to limit state definition lengths:  
;  
; 1) The SMB_MTDBACK state (Master transmitter, data byte sent, ACK received) is  
; reduced to a bit-check and branch operation. The branch is outside of the state  
; table, so that a larger code segment may be executed for this state.  
;  
; 2) Three data bytes are used for slave address storage: SLA_ADD, WRI_ADD, READ_ADD.  
; Rather than using bit-wise operations in the SMBus states, each transfer routine  
; pre-loads the address values. Since a RECEIVE includes both a WRITE and READ  
; transfer, two address bytes are necessary - WRI_ADD and READ_ADD. SLA_ADD is used  
; as a generic slave chip select before a function call.  
;  
; Note that SLA_ADD is equivalent to WRI_ADD, since WRI_ADD = SLA_ADD + W (W=0).  
; The two are left separate to clarify the demonstration.  
;  
-----
```

```

;-----
; EQUATES
;-----

#include (c8051f000.inc)      ; Include register definition file.

WRITE      EQU    00h        ; SMBus WRITE command
READ       EQU    01h        ; SMBus READ command

CHIP_A     EQU    0A0h       ; EEPROM slave address

; SMBus States
SMB_BUS_ERROR EQU 00h      ; (all modes) BUS ERROR
SMB_START    EQU 08h      ; (MT & MR) START transmitted
SMB_RP_START EQU 10h      ; (MT & MR) repeated START
SMB_MTADDACK EQU 18h      ; (MT) Slave address + W transmitted;
                        ; ACK received
SMB_MTADDNACK EQU 20h     ; (MT) Slave address + W transmitted;
                        ; NACK received
SMB_MTDBACK  EQU 28h      ; (MT) data byte transmitted; ACK rec'vd
SMB_MTDNACK  EQU 30h      ; (MT) data byte transmitted; NACK rec'vd
SMB_MTBLOST  EQU 38h      ; (MT) arbitration lost
SMB_MRADDACK EQU 40h      ; (MR) Slave address + R transmitted;
                        ; ACK received
SMB_MRADDNACK EQU 48h     ; (MR) Slave address + R transmitted;
                        ; NACK received
SMB_MRDBACK  EQU 50h      ; (MR) data byte rec'vd; ACK transmitted
SMB_MRDNACK  EQU 58h      ; (MR) data byte rec'vd; NACK transmitted

;-----
; VARIABLES
;-----

MYDATA      SEGMENT DATA      ; declare DATA segment
            RSEG MYDATA        ; select DATA segment

TRANSMIT_BYTE: DS 1           ; Holds a byte to be transmitted by the SMBus
RECEIVE_BYTE: DS 1           ; Holds a byte just received by the SMBus
SLA_ADD:     DS 1             ; Holds the slave address
WRI_ADD:     DS 1             ; Holds the slave address + WRITE
READ_ADD:    DS 1             ; Holds the slave address + READ
MEM_ADD:     DS 1             ; EEPROM memory location to be accessed

; Variables used for testing.
TEST_COUNT:  DS 1             ; Test counter variable
TEST_BYTE:   DS 1             ; Test data
TEST_ADDR:   DS 1             ; Test memory location

MYBITS      SEGMENT BIT
            RSEG MYBITS

RW:          DBIT 1           ; R/W command bit. 1=READ, 0=WRITE
SM_BUSY:     DBIT 1           ; SMBus Busy flag (kept in software)
BYTE_SENT:   DBIT 1           ; Used to indicate what byte was just sent:
                        ; 1: EEPROM memory address sent
                        ; 0: Data byte sent

```

AN113

```
;-----  
; STACK  
  
STACK      SEGMENT IDATA          ; declare STACK segment  
           RSEG STACK  
           DS 80h                  ; reserve 128 bytes for stack  
  
;-----  
; RESET and INTERRUPT VECTORS  
;-----  
  
CSEG  
  
; Reset Vector  
  org 00h  
  ljmp Reset_Vector  
  
; SMBus Interrupt Vector  
  org 03Bh  
  ljmp SMBus_ISR  
  
MYCODE     SEGMENT CODE  
           RSEG MYCODE  
           USING 0  
  
;-----  
; Reset Vector  
;  
; - Disables Watchdog Timer  
; - Routes SDA and SCL to GPIO pins via the crossbar  
; - Enables crossbar  
; - Jumps to MAIN  
  
Reset_Vector:  
  
  mov WDTCN, #0DEh                ; Disable Watchdog Timer  
  mov WDTCN, #0ADh  
  
  mov SP, #STACK                  ; Initialize Stack Pointer  
  
  orl OSCICN, #03h                ; Set internal oscillator to highest setting  
                                   ; (16 MHz)  
  
  mov XBR0, #01h                  ; Route SMBus to GPIO pins through crossbar  
  mov XBR2, #40h                  ; Enable crossbar and weak pull-ups  
  
  ljmp MAIN  
  
;-----  
; MAIN PROGRAM  
;-----  
  
MAIN:  
  acall SMBus_Init                ; Initialize SMBus  
  setb EA                          ; Enable global interrupts  
  
  mov TEST_BYTE, #0ffh            ;
```

```

mov    TEST_ADDR, #00h           ; Load initial test values
mov    TEST_COUNT, #0feh        ;

; TEST CODE-----
TEST:

    ; Send TEST_BYTE to memory location TEST_ADDR
mov    SLA_ADD, #CHIP_A         ; Load slave address
mov    TRANSMIT_BYTE, TEST_BYTE ; Load transmit data into TRANSMIT_BYTE
mov    MEM_ADD, TEST_ADDR       ; Load memory address into MEM_ADD
acall  SEND                     ; Call send routine

    ; Read memory location TEST_ADDR into RECEIVE_BYTE
mov    SLA_ADD, #CHIP_A         ; Load slave address
mov    MEM_ADD, TEST_ADDR       ; Load memory address into MEM_ADD
acall  RECEIVE                  ; Call receive routine

    ; Compare byte received to byte sent
mov    A, RECEIVE_BYTE         ; Load received byte into accumulator
cjne  A, TEST_BYTE, END_TEST   ; Compare sent byte to received byte
                                        ; Jump to END_TEST if not equal

    ; Change test variables
dec    TEST_BYTE               ; If sent=received, change test variables
inc    TEST_ADDR               ; and cycle through again.

    ; Cycle through again if TEST_COUNTER not zero
djnz  TEST_COUNT, TEST         ; Decrement counter, loop back to beginning
mov    A, #99h                 ; Load accumulator with 99h if test successful.

END_TEST:

    jmp    $                    ; Spin
;-----
; SUBROUTINES
;-----

;-----
; SEND subroutine. Assumes that the slave address, memory location, and transmit
; data have all been loaded into their associated variables. This routine manages
; the SM_BUSY bit, sets RW=WRITE, loads the WRI_ADD, and initiates the transfer.
;
SEND:

    push  ACC                   ; Preserve accumulator
    jb    SM_BUSY, $            ; Wait for SMBus to be free
    clr  RW                     ; RW = 0 (WRITE)

    mov  A, SLA_ADD              ; Store SLA_ADD + WRITE
    orl  A, #WRITE              ; in WRI_ADD
    mov  WRI_ADD, A             ;

    setb SM_BUSY                ; Occupy SMBus
    setb STA                    ; Initiate Transfer
    pop  ACC                    ; Restore accumulator

```

AN113

```
ret

;-----
; RECEIVE subroutine. Assumes that the slave address and memory location have been
; loaded into their associated variables. This routine manages the SM_BUSY bit, sets
; RW=READ, loads the READ_ADD and WRI_ADD, and initiates the transfer.
;
; Note that the RECEIVE transfer consists of a WRITE of the memory location to be accessed,
; followed by a repeated START and a READ operation. Therefore, both WRI_ADD
; and READ_ADD are used by this routine.
RECEIVE:

    push ACC                ; Preserve accumulator
    jb SM_BUSY, $           ; Wait for SMBus to be free
    setb RW                 ; RW = 1 (READ)

    mov A, SLA_ADD          ; Store SLA_ADD + WRITE
    orl A, #WRITE           ; in WRITE_ADD
    mov WRI_ADD, A         ;

    mov A, SLA_ADD          ; Store SLA_ADD + READ
    orl A, #READ            ; in READ_ADD
    mov READ_ADD, A        ;

    setb SM_BUSY            ; Occupy SMBus
    setb STA                ; Initiate Transfer

    jb SM_BUSY, $           ; Wait for receive to finish
    pop ACC                 ; Restore accumulator

    ret

;-----
; SMBus_Init
; SMBus initialization routine
;
; - Configures and enables the SMBus.
; - Sets SMBus clock rate.
; - Enables SMBus interrupt.
; - Clears SM_Busy flag for first transfer.

SMBus_Init:

    mov SMB0CN, #04h        ; Configure SMBus to send ACKs on acknowledge cycle
    mov SMB0CR, #0B0h       ; SMBus clock rate = 100KHz, per SMB0CR equation:
                            ; SMB0CR = -(SYSCLK)/(2*Fsc1)

    orl SMB0CN, #40h        ; Enable SMBus

    orl EIE1, #02h          ; Enable SMBus interrupts
    clr SM_BUSY

    ret

;-----
; INTERRUPT VECTORS
;-----
```

```

;-----
; SMBus ISR
;
; Implemented as a state table lookup, with the SMBus status register as the index.
; SMBus status codes are multiples of 8; thus the status code can be used to index
; program segments that are spaced by 8 bytes. Each 'org' command indicates
; a new state, offset from the beginning of the table by its status code value.
;
; Note that only 8 bytes are available to process each state. In the cases where
; more than 8 bytes are necessary, the code jumps to a program location outside
; of the state table. This is only necessary in the state 'SMB_MTDBACK'.

SMBus_ISR:

    push    PSW                ;
    push    ACC                ;
    push    DPH                ; Resource preservation
    push    DPL                ;
    push    ACC                ;

    mov     A, SMBOSTA         ; Load accumulator with current SMBus state.
                                ; State corresponds to the address offset
                                ; for each state execution

    anl    A, #7Fh            ; Mask out upper bit, since any states that
                                ; set this bit are not defined in this code.

    mov     DPTR, #SMB_STATE_TABLE ; Point DPTR to the beginning of the state table
    jmp     @A+DPTR           ; Jump to the current state

; SMBus State Table-----
SMB_STATE_TABLE:

    ; SMB_BUS_ERROR
    ; All Modes: Bus Error
    ; Reset hardware by setting STOP bit
    org     SMB_STATE_TABLE + SMB_BUS_ERROR

        setb  STO
        jmp   SMB_ISR_END        ; Jump to exit ISR

    ; SMB_START
    ; Master Transmitter/Receiver: START transmitted.
    ; The R/W bit will always be a zero (W) in this state because
    ; for both write and read, the memory address must first be written.
    org SMB_STATE_TABLE + SMB_START

        mov   SMB0DAT, WRI_ADD    ; Load slave address + W
        clr   STA                ; Manually clear START bit
        jmp   SMB_ISR_END        ; Jump to exit ISR

    ; SMB_RP_START
    ; Master Transmitter/Receiver: Repeated START transmitted.
    ; This state should only occur during a read, after the memory
    ; address has been sent and acknowledged.
    org     SMB_STATE_TABLE + SMB_RP_START

```

AN113

```
    mov    SMB0DAT, READ_ADD          ; Load slave address + R
    clr    STA                        ; Manually clear START bit
    jmp    SMB_ISR_END

; SMB_MTADDACK
; Master Transmitter: Slave address + WRITE transmitted.
; ACK received
org    SMB_STATE_TABLE + SMB_MTADDACK

    mov    SMB0DAT, MEM_ADD           ; Load memory address
    setb   BYTE_SENT                 ; BYTE_SENT=1: In the next ISR call,
                                    ; the memory address will have just been
                                    ; sent.

    jmp    SMB_ISR_END

; SMB_MTADDNACK
; Master Transmitter: Slave address + WRITE transmitted.
; NACK received. The slave is not responding. Try again with
; acknowledge polling. Send STOP + START.
org    SMB_STATE_TABLE + SMB_MTADDNACK

    setb   STO
    setb   STA
    jmp    SMB_ISR_END

; SMB_MTDBACK
; Master Transmitter: Data byte transmitted. ACK received.
; This state is used in both read and write operations.
; Check BYTE_SENT; if 1, memory address has just been sent. Else,
; data has been sent.
org    SMB_STATE_TABLE + SMB_MTDBACK

    jbc    BYTE_SENT, ADDRESS_SENT    ; If BYTE_SENT=1, clear bit and
                                    ; jump to ADDRESS_SENT to process
                                    ; outside of state table.

    jmp    DATA_SENT                 ; If BYTE_SENT=0, data has just been sent,
                                    ; transfer is finished.
                                    ; jump to end transfer

; SMB_MTDBNACK
; Master Transmitter: Data byte transmitted. NACK received.
; Slave not responding. Send STOP followed by START to try again.
org    SMB_STATE_TABLE + SMB_MTDBNACK

    setb   STO
    setb   STA
    jmp    SMB_ISR_END

; SMB_MTARBLOST
; Master Transmitter: Arbitration Lost.
; Should not occur. If so, restart transfer.
org    SMB_STATE_TABLE + SMB_MTARBLOST

    setb   STO
    setb   STA
    jmp    SMB_ISR_END
```

```

; SMB_MRADDACK
; Master Receiver: Slave address + READ transmitted. ACK received.
; Set to transmit NACK after next transfer since it will be the
; last (only) byte.
org SMB_STATE_TABLE + SMB_MRADDACK

    clr    AA                ; NACK sent on acknowledge cycle
    jmp    SMB_ISR_END

; SMB_MRADDNACK
; Master Receiver: Slave address + READ transmitted. NACK received.
; Slave not responding. Send repeated START to try again.
org SMB_STATE_TABLE + SMB_MRADDNACK

    clr    STO
    setb   STA
    jmp    SMB_ISR_END

; SMB_MRDBACK
; Master Receiver: Data byte received. ACK transmitted.
; Should not occur because AA is cleared in previous state.
; Send STOP if state does occur.
org SMB_STATE_TABLE + SMB_MRDBACK

    setb   STO
    jmp    SMB_ISR_END

; SMB_MRDBNACK
; Master Receiver: Data byte received. NACK transmitted.
; Read operation completed. Read data register and send STOP
org SMB_STATE_TABLE + SMB_MRDBNACK

    mov    RECEIVE_BYTE, SMB0DAT
    setb   STO
    setb   AA                ; Set AA for next transfer
    clr    SM_BUSY
    jmp    SMB_ISR_END

; End of State Table-----

;-----
; Program segment to handle SMBus states that require more than 8 bytes of program
; space.

; Address byte has just been sent. Check RW. If R (1), jump to RW_READ.
; If W, load data to transmit into SMB0DAT.
ADDRESS_SENT:

    jb     RW, RW_READ
    mov    SMB0DAT, TRANSMIT_BYTE    ; Load data
    jmp    SMB_ISR_END              ; Jump to exit ISR

; Operation is a READ, and the address byte has just been sent. Send
; repeated START to initiate memory read.
RW_READ:

    clr    STO
    setb   STA                ; Send repeated START
    jmp    SMB_ISR_END        ; Jump to exit ISR

```

AN113

```
; Operation is a WRITE, and the data byte has just been sent. Transfer
; is finished. Send STOP, free the bus, and exit the ISR.
DATA_SENT:
```

```
    setb  STO                ; Send STOP and exit ISR.
    clr   SM_BUSY           ; Free SMBus
    jmp   SMB_ISR_END       ; Jump to exit ISR
```

```
;-----
```

```
; SMBus ISR exit.
; Restore registers, clear SI bit, and return from interrupt.
SMB_ISR_END:
```

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

```
    reti
```

```
END
```

```

//-----
//
// Copyright 2001 Cygnal Integrated Products, Inc.
//
// FILE NAME      : SMB_Ex2.c
// TARGET DEVICE  : C8051F000
// CREATED ON     : 2/20/01
// CREATED BY    : JS
//
//
// Example code for interfacing a C8051F0xx to three EEPROMs via the SMBus.
// Code assumes that three 16-bit address space EEPROMs are connected
// on the SCL and SDA lines, and configured so that their slave addresses
// are as follows:
// CHIP_A = 1010000
// CHIP_B = 1010001
// CHIP_C = 1010010
//
// Slave and arbitration states are not defined. Assume the CF000 is the only
// master in the system.
// Functions: SM_Send performs a 1-byte write to the specified EEPROM
// SM_Receive performs a 1-byte read of the specified EEPROM address (both include
// memory address references).
//
// Includes test code section.

//-----
// Includes
//-----
#include <c8051f000.h>           // SFR declarations

//-----
// Global CONSTANTS
//-----

#define WRITE 0x00              // SMBus WRITE command
#define READ  0x01             // SMBus READ command

// Device addresses (7 bits, lsb is a don't care)
#define CHIP_A 0xA0            // Device address for chip A
#define CHIP_B 0xA2            // Device address for chip B
#define CHIP_C 0xA4            // Device address for chip C

// SMBus states:
// MT = Master Transmitter
// MR = Master Receiver
#define SMB_BUS_ERROR 0x00     // (all modes) BUS ERROR
#define SMB_START     0x08     // (MT & MR) START transmitted
#define SMB_RP_START  0x10     // (MT & MR) repeated START
#define SMB_MTADDACK   0x18     // (MT) Slave address + W transmitted;
// ACK received
#define SMB_MTADDNACK 0x20     // (MT) Slave address + W transmitted;
// NACK received
#define SMB_MTDBACK    0x28     // (MT) data byte transmitted; ACK rec'vd
#define SMB_MTDNACK    0x30     // (MT) data byte transmitted; NACK rec'vd
#define SMB_MTBLOST    0x38     // (MT) arbitration lost
#define SMB_MRADDACK   0x40     // (MR) Slave address + R transmitted;
// ACK received
#define SMB_MRADDNACK 0x48     // (MR) Slave address + R transmitted;

```

AN113

```

// NACK received
#define SMB_MRDBACK 0x50 // (MR) data byte rec'vd; ACK transmitted
#define SMB_MRDBNACK 0x58 // (MR) data byte rec'vd; NACK transmitted

//-----
//Global VARIABLES
//-----
char COMMAND; // Holds the slave address + R/W bit for
               // use in the SMBus ISR.

char WORD; // Holds data to be transmitted by the SMBus
            // OR data that has just been received.

char BYTE_NUMBER; // Used by ISR to check what data has just been
                  // sent - High address byte, Low byte, or data
                  // byte

unsigned char HIGH_ADD, LOW_ADD; // High & Low byte for EEPROM memory address

bit SM_BUSY; // This bit is set when a send or receive
              // is started. It is cleared by the
              // ISR when the operation is finished.

//-----
// Function PROTOTYPES
//-----

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);

//-----
// MAIN Routine
//-----
//
// Main routine configures the crossbar and SMBus, and tests
// the SMBus interface between the three EEPROMs
void main (void)
{
    unsigned char check; // Used for testing purposes

    WDTCN = 0xde; // disable watchdog timer
    WDTCN = 0xad;

    OSCICN |= 0x03; // Set internal oscillator to highest setting
                  // (16 MHz)

    XBR0 = 0x01; // Route SMBus to GPIO pins through crossbar
    XBR2 = 0x40; // Enable crossbar and weak pull-ups

    SMB0CN = 0x44; // Enable SMBus with ACKs on acknowledge
                  // cycle
    SMB0CR = -80; // SMBus clock rate = 100kHz.

    EIE1 |= 2; // SMBus interrupt enable
    EA = 1; // Global interrupt enable

```

```

SM_BUSY = 0; // Free SMBus for first transfer.

// TEST CODE-----
SM_Send(CHIP_A, 0x0088, 0x53); // Send 0x53(data) to address 0x88 on CHIP_A
SM_Send(CHIP_B, 0x0001, 0x66); // Send 0x66(data) to address 0x01 on CHIP_B
SM_Send(CHIP_C, 0x0010, 0x77);
SM_Send(CHIP_B, 0x0333, 0xF0);
SM_Send(CHIP_A, 0x0242, 0xF0);

check = SM_Receive(CHIP_A, 0x0088); // Read address 0x88 on CHIP_A
check = SM_Receive(CHIP_B, 0x0001); // Read address 0x01 on CHIP_B
check = SM_Receive(CHIP_C, 0x0010);
check = SM_Receive(CHIP_B, 0x0333);
check = SM_Receive(CHIP_A, 0x0242);
// END TEST CODE-----
}

// SMBus byte write function-----
// Writes a single byte at the specified memory location.
//
// out_byte = data byte to be written
// byte_address = memory location to be written into (2 bytes)
// chip_select = device address of EEPROM chip to be written to
void SM_Send (char chip_select, unsigned int byte_address, char out_byte)
{
    while (SM_BUSY); // Wait for SMBus to be free.
    SM_BUSY = 1; // Occupy SMBus (set to busy)
    SMB0CN = 0x44; // SMBus enabled,
    // ACK on acknowledge cycle

    BYTE_NUMBER = 2; // 2 address bytes.
    COMMAND = (chip_select | WRITE); // Chip select + WRITE

    HIGH_ADD = ((byte_address >> 8) & 0x00FF); // Upper 8 address bits
    LOW_ADD = (byte_address & 0x00FF); // Lower 8 address bits

    WORD = out_byte; // Data to be written

    STO = 0;
    STA = 1; // Start transfer
}

// SMBus random read function-----
// Reads 1 byte from the specified memory location.
//
// byte_address = memory address of byte to read
// chip_select = device address of EEPROM to be read from
char SM_Receive (char chip_select, unsigned int byte_address)
{
    while (SM_BUSY); // Wait for bus to be free.
    SM_BUSY = 1; // Occupy SMBus (set to busy)
    SMB0CN = 0x44; // SMBus enabled, ACK on acknowledge cycle

    BYTE_NUMBER = 2; // 2 address bytes
    COMMAND = (chip_select | READ); // Chip select + READ
}

```

```
HIGH_ADD = ((byte_address >> 8) & 0x00FF); // Upper 8 address bits
LOW_ADD = (byte_address & 0x00FF); // Lower 8 address bits

STO = 0;
STA = 1; // Start transfer
while (SM_BUSY); // Wait for transfer to finish
return WORD;
}

//-----
// Interrupt Service Routine
//-----

// SMBus interrupt service routine:

void SMBUS_ISR (void) interrupt 7
{
    switch (SMBOSTA){ // Status code for the SMBus (SMBOSTA register)

        // Master Transmitter/Receiver: START condition transmitted.
        // The R/W bit of the COMMAND word sent after this state will
        // always be a zero (W) because for both read and write,
        // the memory address must be written first.
        case SMB_START:
            SMBODAT = (COMMAND & 0xFE); // Load address of the slave to be accessed.
            STA = 0; // Manually clear START bit
            break;

        // Master Transmitter/Receiver: Repeated START condition transmitted.
        // This state should only occur during a read, after the memory address has been
        // sent and acknowledged.
        case SMB_RP_START:
            SMBODAT = COMMAND; // COMMAND should hold slave address + R.
            STA = 0;
            break;

        // Master Transmitter: Slave address + WRITE transmitted. ACK received.
        case SMB_MTADDACK:
            SMBODAT = HIGH_ADD; // Load high byte of memory address
            // to be written.

            break;

        // Master Transmitter: Slave address + WRITE transmitted. NACK received.
        // The slave is not responding. Send a STOP followed by a START to try again.
        case SMB_MTADDNACK:
            STO = 1;
            STA = 1;
            break;

        // Master Transmitter: Data byte transmitted. ACK received.
        // This state is used in both READ and WRITE operations. Check BYTE_NUMBER
        // for memory address status - if only HIGH_ADD has been sent, load LOW_ADD.
        // If LOW_ADD has been sent, check COMMAND for R/W value to determine
        // next state.
        case SMB_MTDBACK:
            switch (BYTE_NUMBER){
```

```

    case 2:                // If BYTE_NUMBER=2, only HIGH_ADD
        SMB0DAT = LOW_ADD; // has been sent.
        BYTE_NUMBER--;    // Decrement for next time around.
        break;
    case 1:                // If BYTE_NUMBER=1, LOW_ADD was just sent.
        if (COMMAND & 0x01){ // If R/W=READ, sent repeated START.
            STO = 0;
            STA = 1;

            } else {
                SMB0DAT = WORD; // If R/W=WRITE, load byte to write.
                BYTE_NUMBER--;
            }
        break;
    default:               // If BYTE_NUMBER=0, transfer is finished.
        STO = 1;
        SM_BUSY = 0;      // Free SMBus
    }
break;

// Master Transmitter: Data byte transmitted. NACK received.
// Slave not responding. Send STOP followed by START to try again.
case SMB_MTDNACK:
    STO = 1;
    STA = 1;
    break;

// Master Transmitter: Arbitration lost.
// Should not occur. If so, restart transfer.
case SMB_MTARBLST:
    STO = 1;
    STA = 1;
    break;

// Master Receiver: Slave address + READ transmitted. ACK received.
// Set to transmit NACK after next transfer since it will be the last (only)
// byte.
case SMB_MRADDACK:
    AA = 0;                // NACK sent on acknowledge cycle.
    break;

// Master Receiver: Slave address + READ transmitted. NACK received.
// Slave not responding. Send repeated start to try again.
case SMB_MRADDNACK:
    STO = 0;
    STA = 1;
    break;

// Data byte received. ACK transmitted.
// State should not occur because AA is set to zero in previous state.
// Send STOP if state does occur.
case SMB_MRDBACK:
    STO = 1;
    SM_BUSY = 0;
    break;

// Data byte received. NACK transmitted.
// Read operation has completed. Read data register and send STOP.

```

AN113

```
case SMB_MRDBNACK:
    WORD = SMB0DAT;
    STO = 1;
    SM_BUSY = 0;           // Free SMBus
    break;

// All other status codes meaningless in this application. Reset communication.
default:
    STO = 1;               // Reset communication.
    SM_BUSY = 0;
    break;
}

SI=0;                     // clear interrupt flag
}

//-----
--
//
// Copyright 2001 Cygnal Integrated Products, Inc.
//
// FILE NAME       : SMB_Ex3.c
// TARGET DEVICE  : C8051F000
// CREATED ON     : 2/20/01
// CREATED BY    : JS
//
// Example code to demonstrate the use of the SMBus interface between two CF000
// devices.
// The devices operate in a peer-to-peer configuration.
//
// Demonstration includes use of op codes for each device to command the other to:
//
// 1) Write a byte to DAC0
// 2) Write a byte to a data buffer
// 3) Perform an ADC conversion
// 4) Read a byte from a data buffer
//
// These op codes are can be tested easily if each chip has DAC0 routed to AIN0.
// With this configuration, a READ_ADC command can be used to test the output
// of a WRITE_DAC command.
//
// Code assumes that two CF0xx devices are connected via SCL and SDA, with
// slave addresses (held by register SMB0ADR)
// CHIP_A = 1111000
// CHIP_B = 1110000
//
// Test code is included. For testing purposes, the test code should be omitted
// in one device, and run in the other. This can be accomplished by commenting
// the OP_CODE_HANDLER() call before the test code in the device that will assume
// the master role.
//
// PLEASE NOTE that the constant MY_ADD must correspond with the
// current device - change it to CHIP_B when downloading code to CHIP_B.
//
```

```

//-----
--

//-----
--
// Includes
//-----
--
#include <c8051f000.h>                // SFR declarations
//-----
--
// Global CONSTANTS
//-----
--

#define WRITE          0x00           // WRITE direction bit
#define READ           0x01           // READ direction bit

// Device addresses
#define CHIP_A         0xF0
#define CHIP_B         0xE0
#define MY_ADD         CHIP_A        // Corresponds to the chip currently
                                     // being programmed.

// Peer-to-Peer OP_CODES
#define READ_ADC       0x01           // OP_CODE to read from slave ADC
#define WRITE_DAC      0x02           // OP_CODE to write to slave DAC
#define WRITE_BUF      0x03           // OP_CODE to write to slave buffer
#define READ_BUF       0x04           // OP_CODE to read from slave buffer

//SMBus states:
// MT = Master Transmitter
// MR = Master Receiver
// ST = Slave Transmitter
// SR = Slave Receiver

#define SMB_BUS_ERROR  0x00           // (all modes) BUS ERROR
#define SMB_START      0x08           // (MT & MR) START transmitted
#define SMB_RP_START   0x10           // (MT & MR) repeated START
#define SMB_MTADDACK   0x18           // (MT) Slave address + W transmitted;
                                     // ACK received
#define SMB_MTADDNACK  0x20           // (MT) Slave address + W transmitted;
                                     // NACK received
#define SMB_MTDBACK    0x28           // (MT) data byte transmitted; ACK rec'vd
#define SMB_MTDNACK    0x30           // (MT) data byte transmitted; NACK rec'vd
#define SMB_MTARBLST   0x38           // (MT) arbitration lost
#define SMB_MRADDACK   0x40           // (MR) Slave address + R transmitted;
                                     // ACK received
#define SMB_MRADDNACK  0x48           // (MR) Slave address + R transmitted;
                                     // NACK received
#define SMB_MRDBACK    0x50           // (MR) data byte rec'vd; ACK transmitted
#define SMB_MRDBNACK   0x58           // (MR) data byte rec'vd; NACK transmitted

```

AN113

```
#define SMB_SROADACK 0x60 // (SR) SMB's own slave address + W
rec'vd; // ACK transmitted

#define SMB_SROARBLOST 0x68 // (SR) SMB's own slave address + W
rec'vd; // arbitration lost

#define SMB_SRGADACK 0x70 // (SR) general call address rec'vd;
// ACK transmitted

#define SMB_SRGARBLIST 0x78 // (SR) arbitration lost when transmitting
// slave addr + R/W as master; general
// call address rec'vd; ACK transmitted

#define SMB_SRODBACK 0x80 // (SR) data byte received under own slave
// address; ACK returned

#define SMB_SRODBNACK 0x88 // (SR) data byte received under own slave
// address; NACK returned

#define SMB_SRGDBACK 0x90 // (SR) data byte received under general
// call address; ACK returned

#define SMB_SRGDBNACK 0x98 // (SR) data byte received under general
// call address; NACK returned

#define SMB_SRSTOP 0xa0 // (SR) STOP or repeated START received
// while addressed as a slave

#define SMB_STOADACK 0xa8 // (ST) SMB's own slave address + R
rec'vd; // ACK transmitted

#define SMB_STOARBLOST 0xb0 // (ST) arbitration lost in transmitting
// slave address + R/W as master; own
// slave address rec'vd; ACK transmitted

#define SMB_STDBACK 0xb8 // (ST) data byte transmitted; ACK rec'ed
#define SMB_STDBNACK 0xc0 // (ST) data byte transmitted; NACK rec'ed
#define SMB_STDBLAST 0xc8 // (ST) last data byte transmitted (AA=0);
// ACK received

#define SMB_SCLHIGHTO 0xd0 // (ST & SR) SCL clock high timer per
// SMB0CR timed out (FTE=1)

#define SMB_IDLE 0xf8 // (all modes) Idle

//-----
-
//Global VARIABLES
//-----
-

char COMMAND; // Holds the slave address + R/W bit for
// use in the SMBus ISR.

char WORD; // Holds data to be transmitted by the
SMBus // OR data that has just been received.

char OP_CODE; // Holds an op code to be sent or one
// that has just been received.

char LOST_COMMAND, LOST_WORD, LOST_CODE; // Used to hold relevant data after a
// lost arbitration.
```

```

char DATA_BUF[16]; // Data buffer accessed by OP_CODE_HANDLER

bit LOST; // Arbitration lost flag, set when
mode. // arbitration is lost while in master

// Used to resume a failed transfer.

bit SM_BUSY; // This bit is set when a send or receive
// is started. It is cleared by the
// ISR when the operation is finished.

bit VALID_OP; // Flag used to determine if byte received
// as a slave is an OP_CODE or data.

bit DATA_READY; // Used by OP_CODE handler to flag when
// valid data has been received from the
// master

//-----
--
// Function PROTOTYPES
//-----
--

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);

//-----
--
// MAIN Routine
//-----
--

void MAIN (void)
{
    char i, check_1, check_2; // Variables used for testing purposes
only.

    WDTCN = 0xde; // disable watchdog timer
    WDTCN = 0xad;

    XBR0 = 0x01; // Route SMBus to GPIO pins through
crossbar
    XBR2 = 0x40; // Enable crossbar and weak pull-ups

    SMB0CN = 0x44; // Enable SMBus with acknowledge low (AA =
1)
    SMB0CR = -80; // SMBus clock rate = 100 kHz
    SMB0ADR = MY_ADD; // Set own slave address.

    ADC0CN = 0x80; // Enable ADC, conversions to start with

```

```

// write to ADBUSY.

ADC0CN |= 0x01; // ADC data registers left-justified.

DAC0CN = 0x84; // enable DAC0, with left justified data
// registers.

REF0CN = 0x03; // reference voltage enabled.

EIE1 |= 2; // SMBus interrupt enable
EA = 1; // Global interrupt enable

SM_BUSY = 0; // Free bus for first transfer.
SI = 0; //

// OP_CODE_HANDLER(); // This line should be commented in only
// one of the two peer devices. It is
// for testing purposes only.
// In a normal setup, the OP_CODE_HANDLER
// would be running at all times in order
// to react to OP_CODES being sent to the
// device.

// TEST CODE-----
--
// This code is used only to test the interface between the two devices. If
// the above OP_CODE_HANDLER line is commented out, this device assumes the master
// role. The other device should be running the OP_CODE_HANDLER at all times, to
// respond to the OP_CODES below.

SLA_SEND(CHIP_B, (0x40 | WRITE_BUF), 0x24); // Write to index 4
// in the data buffer
SLA_SEND(CHIP_B, (0x60 | WRITE_BUF), 0x25); // Write to index 6
SLA_SEND(CHIP_B, (0x80 | WRITE_BUF), 0x26); // Write to index 8
SLA_SEND(CHIP_B, (0x10 | WRITE_BUF), 0x27); // Write to index 1

check_1 = SLA_READ(CHIP_B, (0x40 | READ_BUF)); // Read index 4 from the buffer
check_1 = SLA_READ(CHIP_B, (0x60 | READ_BUF)); // Read index 6
check_1 = SLA_READ(CHIP_B, (0x80 | READ_BUF)); // Read index 8
check_1 = SLA_READ(CHIP_B, (0x10 | READ_BUF)); // Read index 1

// Loop to continuously increase the DAC output on CHIP_B, and read its
// ADC each round. DAC output on CHIP_B should ramp.

for (i=0;i<50;i++){
    SLA_SEND(CHIP_B, WRITE_DAC, 2*i); // Write 2*i to DAC0 on CHIP_B
    check_1 = SLA_READ(CHIP_B, READ_ADC); // Read AIN0 on CHIP_B
    check_2 = 2*i;} // check_1 should be
approximately // the same as check_2.

// END TEST CODE-----
--
```

```

}

//-----
--
// Functions
//-----
--

// Send to slave.
// The send function transmits two bytes to the slave device: an op code, and a data
// byte. There are two op code choices for sending data: WRITE_DAC and WRITE_BUF.
// If the op code is WRITE_BUF, then the upper 4 bits of the op code should contain
// the buffer index. For example, to write to index 2 of the data buffer, the
// op_code parameter should be (0x20 | WRITE_BUF).
//
// chip_select = address of slave device.
// out_op = OP_CODE to be sent.
// out_data = data byte to be sent.
void SLA_SEND(char chip_select, char out_op, char out_data){

    while(SM_BUSY);                // Wait while SMBus is busy.
    SM_BUSY = 1;                   // SMBus busy flag set.
    SMB0CN = 0x44;                 // SMBus enabled, ACK low.
    COMMAND = (chip_select | WRITE); // COMMAND = 7 address bits + WRITE.
    OP_CODE = out_op;              // WORD = OP_CODE to be transmitted.
    WORD = out_data;               // DATA = data to be transmitted.
    STO = 0;
    STA = 1;                       // Start transfer.
}

// Read from slave.
// The read function transmits a 1-byte op code, then issues a repeated start
// to request a 1-byte read. The two op code choices are READ_ADC and READ_BUF.
// If the op code is READ_BUF, then the upper 4 bits of the op code should
// contain the buffer index. For example, to read index 5 of the data buffer,
// the op code should be (0x50 | READ_BUF).
//
// chip_select = address of slave device.
// out_op = OP_CODE to be sent.
char SLA_READ(char chip_select, char out_op){

    while(SM_BUSY);                // Wait while SMBus is busy.
    SM_BUSY = 1;                   // Set busy flag.
    SMB0CN = 0x44;                 // Enable SMBus, ACK low.
    COMMAND = (chip_select | READ); // COMMAND = 7 address bits + READ
    OP_CODE = out_op;
    STO = 0;
    STA = 1;                       // Start transfer.
    while(SM_BUSY);                // Wait for transfer to finish.
    return WORD;                   // Return received word.
}

```

```
// OP_CODE handler.
// Decodes incoming op codes and performs tasks according to those op codes.
// A call to this function runs forever.
//
// The VALID_OP bit flags when a valid op code has been received. Upon receipt,
// the handler decodes the op code, performs the task, then clears
// VALID_OP to wait for another code.
void OP_CODE_HANDLER(void){

    char index;                                // data buffer index
    while (1){                                  // run forever
        VALID_OP = 0;                          // Wait for a valid OP_CODE
        while (!VALID_OP);                    //

        // The lower 4 bits of the OP_CODE are used to determine the action, while the
        // upper 4 bits are used to index the DATA_BUF array when the READ_BUF or
        // WRITE_BUF OP_CODES are received. Note that the SMBus is stalled until the
        // OP_CODE is decoded.
        switch (OP_CODE & 0x0F){              // Decode OP_CODE

            // OP_CODE = READ_ADC - Perform an ADC conversion, and place data in
            // output buffer.
            // Read only ADC high byte.
            case READ_ADC:
                SI = 0;                        // Free the bus
                AA = 0;                        // Take slave 'offline'
                ADCINT = 0;                    // Clear ADC interrupt flag.
                ADBUSY = 1;                   // Start conversion.
                while (!ADCINT);              // Wait for conversion to finish.
                WORD = ADC0H;                  // Put data in output buffer.
                AA = 1;                        // Put slave back 'online'
                VALID_OP = 0;                 // Look for a new OP_CODE
                break;

            // OP_CODE = WRITE_DAC - Wait for a valid data byte, and write it to high
            // byte of DAC0.
            case WRITE_DAC:
                SI = 0;                        // Free the bus
                DATA_READY = 0;              // Wait for valid data.
                while (!DATA_READY);          //
                DAC0L = 0;                    // DAC low byte
                DAC0H = WORD;                 // DAC high byte
                VALID_OP = 0;                 // Look for new OP_CODE
                SI = 0;                       // Free bus when finished.
                break;

            // OP_CODE = WRITE_BUF - Wait for valid data byte, then place data in
            // DATA_BUF array. Index data according to upper 4 bits of OP_CODE.
            case WRITE_BUF:
                SI = 0;                        // Free the bus
                index = (OP_CODE & 0xF0);     // Use upper 4 bits as array index.
                DATA_READY = 0;              // Wait for valid data.
                while (!DATA_READY);          //
        }
    }
}
```

```

        DATA_BUF[index] = WORD;           // Store data in array.
        VALID_OP = 0;                     // Look for new OP_CODE
        SI = 0;                           // Free the bus when finished.
        break;

// OP_CODE = READ_BUF - Read DATA_BUF array and place byte in output
buffer.
// Array index determined by upper 4 bits of OP_CODE.
case READ_BUF:
    index = (OP_CODE & 0xF0);           // Use upper 4 bits as array index.
    WORD = DATA_BUF[index];           // Place indexed data in output buffer.
    VALID_OP = 0;                       // Look for new OP_CODE
    SI = 0;                              // Free the bus when finished.
    break;
}

if (LOST){                               // If LOST is set, the device has recently
    COMMAND = LOST_COMMAND;            // lost an arbitration. Load saved values
    WORD = LOST_WORD;                  // back into transfer variables, and retry
    OP_CODE = LOST_CODE;               // transfer.
    LOST = 0;
    STO = 0;
    STA = 1;
}
}

//-----
--
// SMBus Interrupt Service Routine
//-----
--

void SMBUS_ISR (void) interrupt 7
{
    switch (SMB0STA){                   // Status code for the SMBus
                                        // (SMB0STA register)

        // Master Transmitter/Receiver: START condition transmitted.
        // Load SMB0DAT with slave device address. Mask out R/W bit since all
transfers
        // start with an OP_CODE write.
        case SMB_START:
            SMB0DAT = (COMMAND & 0xFE); // Load address of the slave to be
accessed.
                                        // Mask out R/W bit because first transfer
                                        // will always be a write of the OP_CODE.

            STA = 0;                     // Manually clear STA bit
            SI = 0;                       // Clear interrupt flag
            break;

        // Master Transmitter/Receiver: Repeated START condition transmitted.

```

AN113

```
// This state only occurs during a READ, after the OP_CODE has been sent.
Load
// device address + READ into SMB0DAT.
case SMB_RP_START:
    SMB0DAT = COMMAND;
    STA = 0; // Manually clear START bit.
    SI = 0;
    break;

// Master Transmitter: Slave address + WRITE transmitted. ACK received.
// Load OP_CODE into SMB0DAT.
case SMB_MTADDACK:
    SMB0DAT = OP_CODE;
    SI = 0; // Clear interrupt flag
    break;

// Master Transmitter: Slave address + WRITE transmitted. NACK received.
// The slave is not responding. Use ACK polling to retry.
case SMB_MTADDNACK:
    STO = 1;
    STA = 1;
    SI = 0; // Clear interrupt flag
    break;

// Master Transmitter: Data byte transmitted. ACK received.
// Check OP_CODE - If it is a READ code, send repeated START to begin
// read. If it is a WRITE code, load WORD into SMB0DAT for transfer.
// If it is not a valid code, then either 1) data has been transmitted
// and the transfer is finished, or 2) there is an error. In either case,
// send STOP and end transfer.
case SMB_MTDBACK:
    switch (OP_CODE & 0x0F){ // Check only lower 4 bits.

        // OP_CODE is a READ. Send repeated START.
        case READ_BUF:
        case READ_ADC:
            OP_CODE = 0; // Current OP_CODE no longer useful
            STO = 0;
            STA = 1;
            break;

        // OP_CODE is a WRITE. Load output data into SMB0DAT.
        case WRITE_BUF:
        case WRITE_DAC:
            SMB0DAT = WORD;
            OP_CODE = 0; // Clear OP_CODE so transfer ends the next
            break; // time this state occurs
            // (after data is sent).

        default: // No valid OP_CODE. End transfer.
            STO = 1;
            SM_BUSY = 0;
            break;
    }
}
```

```

    SI = 0;
    break;

// Master Transmitter: Data byte transmitter. NACK received.
// Use ACK polling to retry transfer.
case SMB_MTDENACK:
    STO = 1;
    STA = 1;
    SI = 0; // Clear interrupt flag
    break;

// Master Transmitter: Arbitration lost.
case SMB_MTARBLOST:
    LOST_COMMAND = COMMAND; //
    LOST_WORD = WORD; // Store variables for use when bus is
free.
    LOST_CODE = OP_CODE; //

    LOST = 1; // Set flag to retry transfer
// when bus is free.

    SI = 0; // Clear interrupt flag
    break;

// Master Receiver: Slave address + READ transmitted. ACK received.
// Set to transmit NACK after next transfer since it will be the
// last (only) byte.
case SMB_MRADDACK:
    AA = 0; // NACK sent during acknowledge cycle.
    SI = 0;
    break;

// Master Receiver: Slave address + READ transmitted. NACK received.
// Slave not responding. Send repeated START to try again.
case SMB_MRADDNACK:
    STO = 0;
    STA = 1;
    SI = 0;
    break;

// Master Receiver: Data byte received. ACK transmitted.
// State should not occur because AA is cleared in previous state.
// Send STOP if state does occur.
case SMB_MRDBACK:
    STO = 1;
    SM_BUSY = 0;
    SI = 0;
    break;

// Master Receiver: Data byte received. NACK transmitted.
// Read operation has completed. Read data register and send STOP.
case SMB_MRDBNACK:
    WORD = SMB0DAT;
    STO = 1;
    SM_BUSY = 0;

```

```
    AA = 1;                // Set AA for next transfer
    SI = 0;
    break;

// Slave Receiver: Arbitration lost, general call address received.
// Set LOST flag to retry transfer when bus is free.  Fall through.
case SMB_SRGARBLOST:

// Slave Receiver: Arbitration lost, own slave address + WRITE received.
// Set LOST flag to retry transfer when bus is free.
// Set STO bit to get out of master mode.
case SMB_SROARBLOST:
    LOST_COMMAND = COMMAND;    //
    LOST_WORD = WORD;         // Store variables for use when bus is
free.
    LOST_CODE = OP_CODE;      //
    LOST = 1;                 // Retry transfer when bus is free.
    SI = 0;
    break;

// Slave Receiver: Slave address + WRITE received.  ACK transmitted.
// Fall through.
case SMB_SROADACK:

// Slave Receiver: General call address received.  ACK transmitted.
case SMB_SRGADACK:
    SI = 0;
    break;

// Slave Receiver: Data byte received after addressed by general
// call address + WRITE.
// ACK transmitted.  Fall through.
case SMB_SRGDBACK:

// Slave Receiver: Data byte received after addressed by own
// slave address + WRITE.
// ACK transmitted.
// Take action depending on OP_CODE or data received.
case SMB_SRODBACK:
    if (!VALID_OP){          // if VALID_OP=0, this byte is an OP_CODE.
        OP_CODE = SMB0DAT;   // Store OP_CODE
        VALID_OP = 1;       // Next byte is not an OP_CODE
    } else {
        DATA_READY = 1;    // Valid data has been received.  Process
// in OP_CODE handler.
        WORD = SMB0DAT;
        SI = 0;
    }
    break;

// Slave Receiver: Data byte received while addressed as slave.
// NACK transmitted.  Should not occur since AA will not be cleared
// as slave.  Fall through to next state.
case SMB_SRODBNACK:
```

```
// Slave Receiver: Data byte received while addressed by general call.
// NACK transmitted.
// Should not occur since AA will not be cleared as slave.
case SMB_SRGNACK:
    AA = 1;
    SI = 0;
    break;

// Slave Receiver: STOP or repeated START received while addressed as slave.
case SMB_SRSTOP:
    SI = 0;
    break;

// Slave Transmitter: Own slave address + READ received. ACK transmitted.
// Load SMB0DAT with data to be output.
case SMB_STOACK:
    SMB0DAT = WORD;
    SI = 0;
    break;

// Slave Transmitter: Arbitration lost as master. Own address + READ received.
// ACK transmitted.
case SMB_STOARBLST:
    LOST_COMMAND = COMMAND;           //
    LOST_WORD = WORD;                 // Store variables for use when bus
    LOST_CODE = OP_CODE;              // is free.
    LOST = 1;                          // Retry when bus is free.

    SI = 0;
    break;

// Slave Transmitter: Data byte transmitted. ACK received. Fall through.
case SMB_STDBACK:

// Slave Transmitter: Data byte transmitted. NACK received. Fall through.
case SMB_STDBNACK:

// Slave Transmitter: Last data byte transmitted. ACK received.
// No action necessary.
case SMB_STDBLAST:
    SI = 0;
    break;

// All other status codes invalid. Reset communication.
default:
    STO = 1;
    SM_BUSY = 0;
    break;
}
}
```

Software Example for the C8051F02x series

```
//-----  
--  
//  
// Copyright 2001 Cygnal Integrated Products, Inc.  
//  
// FILE NAME      : SMB_Ex3.c  
// TARGET DEVICE  : C8051F020  
// CREATED ON     : 6/5/02  
// CREATED BY    : JS / FB  
//  
// Example code to demonstrate the use of the SMBus interface between two CF000  
// devices.  
// The devices operate in a peer-to-peer configuration.  
//  
// Demonstration includes use of op codes for each device to command the other to:  
//  
// 1) Write a byte to DAC0  
// 2) Write a byte to a data buffer  
// 3) Perform an ADC conversion  
// 4) Read a byte from a data buffer  
//  
// These op codes are can be tested easily if each chip has DAC0 routed to AIN0.  
// With this configuration, a READ_ADC command can be used to test the output  
// of a WRITE_DAC command.  
//  
// Code assumes that two CF0xx devices are connected via SCL and SDA, with  
// slave addresses (held by register SMB0ADR)  
// CHIP_A = 1111000  
// CHIP_B = 1110000  
//  
// Test code is included.  For testing purposes, the test code should be omitted  
// in one device, and run in the other.  This can be accomplished by commenting  
// the OP_CODE_HANDLER() call before the test code in the device that will assume  
// the master role.  
//  
// PLEASE NOTE that the constant MY_ADD must correspond with the  
// current device - change it to CHIP_B when downloading code to CHIP_B.  
//  
//-----  
--  
  
//-----  
// Includes  
//-----  
#include <c8051f020.h>                // SFR declarations  
  
//-----  
// 16-bit SFR Definitions for 'F02x  
//-----  
sfr16 DP      = 0x82;                // data pointer
```

```

sfr16 TMR3RL    = 0x92;           // Timer3 reload value
sfr16 TMR3      = 0x94;           // Timer3 counter
sfr16 ADC0      = 0xbe;           // ADC0 data
sfr16 ADC0GT    = 0xc4;           // ADC0 greater than window
sfr16 ADC0LT    = 0xc6;           // ADC0 less than window
sfr16 RCAP2     = 0xca;           // Timer2 capture/reload
sfr16 T2        = 0xcc;           // Timer2
sfr16 RCAP4     = 0xe4;           // Timer4 capture/reload
sfr16 T4        = 0xf4;           // Timer4
sfr16 DAC0      = 0xd2;           // DAC0 data
sfr16 DAC1      = 0xd5;           // DAC1 data

//-----
--
// Global CONSTANTS
//-----
--

#define WRITE    0x00             // WRITE direction bit
#define READ     0x01             // READ direction bit

// Device addresses
#define CHIP_A   0xF0
#define CHIP_B   0xE0
#define MY_ADD   CHIP_A          // Corresponds to the chip currently
                                // being programmed.

// Peer-to-Peer OP_CODES
#define READ_ADC 0x01             // OP_CODE to read from slave ADC
#define WRITE_DAC 0x02           // OP_CODE to write to slave DAC
#define WRITE_BUF 0x03           // OP_CODE to write to slave buffer
#define READ_BUF  0x04           // OP_CODE to read from slave buffer

//SMBus states:
// MT = Master Transmitter
// MR = Master Receiver
// ST = Slave Transmitter
// SR = Slave Receiver

#define SMB_BUS_ERROR 0x00        // (all modes) BUS ERROR
#define SMB_START     0x08        // (MT & MR) START transmitted
#define SMB_RP_START  0x10        // (MT & MR) repeated START
#define SMB_MTADDACK   0x18        // (MT) Slave address + W transmitted;
                                // ACK received
#define SMB_MTADDNACK  0x20        // (MT) Slave address + W transmitted;
                                // NACK received
#define SMB_MTDBACK    0x28        // (MT) data byte transmitted; ACK rec'vd
#define SMB_MTDNACK    0x30        // (MT) data byte transmitted; NACK rec'vd
#define SMB_MTARBLST   0x38        // (MT) arbitration lost
#define SMB_MRADDACK   0x40        // (MR) Slave address + R transmitted;
                                // ACK received
#define SMB_MRADDNACK  0x48        // (MR) Slave address + R transmitted;
                                // NACK received
#define SMB_MRDBACK    0x50        // (MR) data byte rec'vd; ACK transmitted

```

AN113

```
#define SMB_MRDBNACK 0x58 // (MR) data byte rec'vd; NACK transmitted
#define SMB_SROADACK 0x60 // (SR) SMB's own slave address + W
rec'vd; // ACK transmitted

#define SMB_SROARBLOST 0x68 // (SR) SMB's own slave address + W
rec'vd; // arbitration lost

#define SMB_SRGADACK 0x70 // (SR) general call address rec'vd;
// ACK transmitted
#define SMB_SRGARBLIST 0x78 // (SR) arbitration lost when transmitting
// slave addr + R/W as master; general
// call address rec'vd; ACK transmitted
#define SMB_SRODBACK 0x80 // (SR) data byte received under own slave
// address; ACK returned
#define SMB_SRODBNACK 0x88 // (SR) data byte received under own slave
// address; NACK returned
#define SMB_SRGDBACK 0x90 // (SR) data byte received under general
// call address; ACK returned
#define SMB_SRGDBNACK 0x98 // (SR) data byte received under general
// call address; NACK returned
#define SMB_SRSTOP 0xa0 // (SR) STOP or repeated START received
// while addressed as a slave
#define SMB_STOADACK 0xa8 // (ST) SMB's own slave address + R
rec'vd; // ACK transmitted
#define SMB_STOARBLOST 0xb0 // (ST) arbitration lost in transmitting
// slave address + R/W as master; own
// slave address rec'vd; ACK transmitted
#define SMB_STDBACK 0xb8 // (ST) data byte transmitted; ACK rec'ed
#define SMB_STDBNACK 0xc0 // (ST) data byte transmitted; NACK rec'ed
#define SMB_STDBLAST 0xc8 // (ST) last data byte transmitted (AA=0);
// ACK received
#define SMB_SCLHIGHTO 0xd0 // (ST & SR) SCL clock high timer per
// SMB0CR timed out (FTE=1)
#define SMB_IDLE 0xf8 // (all modes) Idle

//-----
-
//Global VARIABLES
//-----
-

char COMMAND; // Holds the slave address + R/W bit for
// use in the SMBus ISR.

char WORD; // Holds data to be transmitted by the
SMBus // OR data that has just been received.

char OP_CODE; // Holds an op code to be sent or one
// that has just been received.

char LOST_COMMAND, LOST_WORD, LOST_CODE; // Used to hold relevant data after a
```

```

// lost arbitration.

char DATA_BUF[16]; // Data buffer accessed by OP_CODE_HANDLER

bit LOST; // Arbitration lost flag, set when
mode. // arbitration is lost while in master

// Used to resume a failed transfer.

bit SM_BUSY; // This bit is set when a send or receive
// is started. It is cleared by the
// ISR when the operation is finished.

bit VALID_OP; // Flag used to determine if byte received
// as a slave is an OP_CODE or data.

bit DATA_READY; // Used by OP_CODE handler to flag when
// valid data has been received from the
// master

//-----
--
// Function PROTOTYPES
//-----
--

void SYSCLK_Init (void);

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);

//-----
--
// MAIN Routine
//-----
--

void MAIN (void)
{
    char i, check_1, check_2; // Variables used for testing purposes
only.

    WDTCN = 0xde; // disable watchdog timer
    WDTCN = 0xad;

    SYSCLK_Init(); // turn on the external oscillator

    XBR0 = 0x01; // Route SMBus to GPIO pins through
crossbar
    XBR2 = 0x40; // Enable crossbar and weak pull-ups

```

AN113

```
SMB0CN = 0x44; // Enable SMBus with acknowledge low (AA =
1)
SMB0CR = -80; // SMBus clock rate = 100 kHz
SMB0ADR = MY_ADD; // Set own slave address.

ADC0CN = 0x80; // Enable ADC, conversions to start with
// write to AD0BUSY.

ADC0CN |= 0x01; // ADC data registers left-justified.

DAC0CN = 0x84; // enable DAC0, with left justified data
// registers.

REF0CN = 0x03; // reference voltage enabled.

EIE1 |= 2; // SMBus interrupt enable
EA = 1; // Global interrupt enable

SM_BUSY = 0; // Free bus for first transfer.
SI = 0; //

// OP_CODE_HANDLER(); // This line should be commented in only
// one of the two peer devices. It is
// for testing purposes only.
// In a normal setup, the OP_CODE_HANDLER
// would be running at all times in order
// to react to OP_CODES being sent to the
// device.

// TEST CODE-----
--
// This code is used only to test the interface between the two devices. If
// the above OP_CODE_HANDLER line is commented out, this device assumes the master
// role. The other device should be running the OP_CODE_HANDLER at all times, to
// respond to the OP_CODES below.

SLA_SEND(CHIP_B, (0x40 | WRITE_BUF), 0x24); // Write to index 4
// in the data buffer
SLA_SEND(CHIP_B, (0x60 | WRITE_BUF), 0x25); // Write to index 6
SLA_SEND(CHIP_B, (0x80 | WRITE_BUF), 0x26); // Write to index 8
SLA_SEND(CHIP_B, (0x10 | WRITE_BUF), 0x27); // Write to index 1

check_1 = SLA_READ(CHIP_B, (0x40 | READ_BUF)); // Read index 4 from the buffer
check_1 = SLA_READ(CHIP_B, (0x60 | READ_BUF)); // Read index 6
check_1 = SLA_READ(CHIP_B, (0x80 | READ_BUF)); // Read index 8
check_1 = SLA_READ(CHIP_B, (0x10 | READ_BUF)); // Read index 1

// Loop to continuously increase the DAC output on CHIP_B, and read its
// ADC each round. DAC output on CHIP_B should ramp.

for (i=0;i<50;i++){
    SLA_SEND(CHIP_B, WRITE_DAC, 2*i); // Write 2*i to DAC0 on CHIP_B
    check_1 = SLA_READ(CHIP_B, READ_ADC); // Read AIN0 on CHIP_B
```

```

        check_2 = 2*i;                // check_1 should be
approximately                          // the same as check_2.
    }

// END TEST CODE-----
--

}

//-----
--
// Initialization Routines
//-----
--

//-----
// SYSCLK_Init
//-----
//
// This routine initializes the system clock to use an 22.1184MHz crystal
// as its clock source.
//
void SYSCLK_Init (void)
{
    int i;                            // delay counter

    OSCXCN = 0x67;                    // start external oscillator with
                                        // 22.1184MHz crystal

    for (i=0; i < 256; i++) ;        // XTLVLD blanking interval (>1ms)

    while (!(OSCXCN & 0x80)) ;        // Wait for crystal osc. to settle

    OSCICN = 0x88;                    // select external oscillator as SYSCLK
                                        // source and enable missing clock
                                        // detector
}

//-----
--
// Functions
//-----
--

// Send to slave.
// The send function transmits two bytes to the slave device: an op code, and a data
// byte. There are two op code choices for sending data: WRITE_DAC and WRITE_BUF.
// If the op code is WRITE_BUF, then the upper 4 bits of the op code should contain
// the buffer index. For example, to write to index 2 of the data buffer, the
// op_code parameter should be (0x20 | WRITE_BUF).
//
// chip_select = address of slave device.
// out_op = OP_CODE to be sent.
// out_data = data byte to be sent.

```

AN113

```
void SLA_SEND(char chip_select, char out_op, char out_data){

    while(SM_BUSY);                // Wait while SMBus is busy.
    SM_BUSY = 1;                   // SMBus busy flag set.
    SMB0CN = 0x44;                 // SMBus enabled, ACK low.
    COMMAND = (chip_select | WRITE); // COMMAND = 7 address bits + WRITE.
    OP_CODE = out_op;              // WORD = OP_CODE to be transmitted.
    WORD = out_data;               // DATA = data to be transmitted.
    STO = 0;
    STA = 1;                       // Start transfer.

}

// Read from slave.
// The read function transmits a 1-byte op code, then issues a repeated start
// to request a 1-byte read. The two op code choices are READ_ADC and READ_BUF.
// If the op code is READ_BUF, then the upper 4 bits of the op code should
// contain the buffer index. For example, to read index 5 of the data buffer,
// the op code should be (0x50 | READ_BUF).
//
// chip_select = address of slave device.
// out_op = OP_CODE to be sent.
char SLA_READ(char chip_select, char out_op){

    while(SM_BUSY);                // Wait while SMBus is busy.
    SM_BUSY = 1;                   // Set busy flag.
    SMB0CN = 0x44;                 // Enable SMBus, ACK low.
    COMMAND = (chip_select | READ); // COMMAND = 7 address bits + READ
    OP_CODE = out_op;
    STO = 0;
    STA = 1;                       // Start transfer.
    while(SM_BUSY);                // Wait for transfer to finish.
    return WORD;                   // Return received word.

}

// OP_CODE handler.
// Decodes incoming op codes and performs tasks according to those op codes.
// A call to this function runs forever.
//
// The VALID_OP bit flags when a valid op code has been received. Upon receipt,
// the handler decodes the op code, performs the task, then clears
// VALID_OP to wait for another code.
void OP_CODE_HANDLER(void){

    char index;                    // data buffer index
    while (1){                     // run forever
        VALID_OP = 0;              // Wait for a valid OP_CODE
        while (!VALID_OP);        //

        // The lower 4 bits of the OP_CODE are used to determine the action, while the
        // upper 4 bits are used to index the DATA_BUF array when the READ_BUF or
        // WRITE_BUF OP_CODES are received. Note that the SMBus is stalled until the
```

```

// OP_CODE is decoded.
switch (OP_CODE & 0x0F){           // Decode OP_CODE

    // OP_CODE = READ_ADC - Perform an ADC conversion, and place data in
    // output buffer.
    // Read only ADC high byte.
    case READ_ADC:
        SI = 0;                    // Free the bus
        AA = 0;                    // Take slave 'offline'
        AD0INT = 0;                // Clear ADC interrupt flag.
        AD0BUSY = 1;               // Start conversion.
        while (!AD0INT);           // Wait for conversion to finish.
        WORD = ADC0H;              // Put data in output buffer.
        AA = 1;                    // Put slave back 'online'
        VALID_OP = 0;              // Look for a new OP_CODE
        break;

    // OP_CODE = WRITE_DAC - Wait for a valid data byte, and write it to high
    // byte of DAC0.
    case WRITE_DAC:
        SI = 0;                    // Free the bus
        DATA_READY = 0;           // Wait for valid data.
        while (!DATA_READY);       //
        DAC0L = 0;                 // DAC low byte
        DAC0H = WORD;              // DAC high byte
        VALID_OP = 0;              // Look for new OP_CODE
        SI = 0;                    // Free bus when finished.
        break;

    // OP_CODE = WRITE_BUF - Wait for valid data byte, then place data in
    // DATA_BUF array.  Index data according to upper 4 bits of OP_CODE.
    case WRITE_BUF:
        SI = 0;                    // Free the bus
        index = (OP_CODE & 0xF0);  // Use upper 4 bits as array index.
        DATA_READY = 0;           // Wait for valid data.
        while (!DATA_READY);       //
        DATA_BUF[index] = WORD;   // Store data in array.
        VALID_OP = 0;              // Look for new OP_CODE
        SI = 0;                    // Free the bus when finished.
        break;

    // OP_CODE = READ_BUF - Read DATA_BUF array and place byte in output
buffer.
    // Array index determined by upper 4 bits of OP_CODE.
    case READ_BUF:
        index = (OP_CODE & 0xF0);  // Use upper 4 bits as array index.
        WORD = DATA_BUF[index];   // Place indexed data in output buffer.
        VALID_OP = 0;              // Look for new OP_CODE
        SI = 0;                    // Free the bus when finished.
        break;
}

if (LOST){                          // If LOST is set, the device has recently
    COMMAND = LOST_COMMAND;         // lost an arbitration.  Load saved values
}

```

AN113

```
        WORD = LOST_WORD;           // back into transfer variables, and retry
        OP_CODE = LOST_CODE;        // transfer.
        LOST = 0;
        STO = 0;
        STA = 1;
    }
}

//-----
--
// SMBus Interrupt Service Routine
//-----
--

void SMBUS_ISR (void) interrupt 7
{
    switch (SMB0STA){                // Status code for the SMBus
                                    // (SMB0STA register)

        // Master Transmitter/Receiver: START condition transmitted.
        // Load SMB0DAT with slave device address. Mask out R/W bit since all
transfers
        // start with an OP_CODE write.
        case SMB_START:
            SMB0DAT = (COMMAND & 0xFE); // Load address of the slave to be
accessed.
                                            // Mask out R/W bit because first transfer
                                            // will always be a write of the OP_CODE.

            STA = 0;                    // Manually clear STA bit
            SI = 0;                    // Clear interrupt flag
            break;

        // Master Transmitter/Receiver: Repeated START condition transmitted.
        // This state only occurs during a READ, after the OP_CODE has been sent.
Load
        // device address + READ into SMB0DAT.
        case SMB_RP_START:
            SMB0DAT = COMMAND;
            STA = 0;                    // Manually clear START bit.
            SI = 0;
            break;

        // Master Transmitter: Slave address + WRITE transmitted. ACK received.
        // Load OP_CODE into SMB0DAT.
        case SMB_MTADDACK:
            SMB0DAT = OP_CODE;
            SI = 0;                    // Clear interrupt flag
            break;

        // Master Transmitter: Slave address + WRITE transmitted. NACK received.
        // The slave is not responding. Use ACK polling to retry.
```

```

case SMB_MTADDNACK:
    STO = 1;
    STA = 1;
    SI = 0;                // Clear interrupt flag
    break;

// Master Transmitter: Data byte transmitted. ACK received.
// Check OP_CODE - If it is a READ code, send repeated START to begin
// read. If it is a WRITE code, load WORD into SMB0DAT for transfer.
// If it is not a valid code, then either 1) data has been transmitted
// and the transfer is finished, or 2) there is an error. In either case,
// send STOP and end transfer.
case SMB_MTDDBACK:
    switch (OP_CODE & 0x0F){        // Check only lower 4 bits.

        // OP_CODE is a READ. Send repeated START.
        case READ_BUF:
        case READ_ADC:
            OP_CODE = 0;            // Current OP_CODE no longer useful
            STO = 0;
            STA = 1;
            break;

        // OP_CODE is a WRITE. Load output data into SMB0DAT.
        case WRITE_BUF:
        case WRITE_DAC:
            SMB0DAT = WORD;
            OP_CODE = 0;            // Clear OP_CODE so transfer ends the next
            break;                // time this state occurs
                                    // (after data is sent).

        default:                    // No valid OP_CODE. End transfer.
            STO = 1;
            SM_BUSY = 0;
            break;
    }
    SI = 0;
    break;

// Master Transmitter: Data byte transmitter. NACK received.
// Use ACK polling to retry transfer.
case SMB_MTDDBNACK:
    STO = 1;
    STA = 1;
    SI = 0;                // Clear interrupt flag
    break;

// Master Transmitter: Arbitration lost.
case SMB_MTARBLOST:
    LOST_COMMAND = COMMAND;        //
    LOST_WORD = WORD;            // Store variables for use when bus is
free.
    LOST_CODE = OP_CODE;        //

```

AN113

```
    LOST = 1;                // Set flag to retry transfer
                            // when bus is free.
    SI = 0;                // Clear interrupt flag
    break;

// Master Receiver: Slave address + READ transmitted. ACK received.
// Set to transmit NACK after next transfer since it will be the
// last (only) byte.
case SMB_MRADDACK:
    AA = 0;                // NACK sent during acknowledge cycle.
    SI = 0;
    break;

// Master Receiver: Slave address + READ transmitted. NACK received.
// Slave not responding. Send repeated START to try again.
case SMB_MRADDNACK:
    STO = 0;
    STA = 1;
    SI = 0;
    break;

// Master Receiver: Data byte received. ACK transmitted.
// State should not occur because AA is cleared in previous state.
// Send STOP if state does occur.
case SMB_MRDBACK:
    STO = 1;
    SM_BUSY = 0;
    SI = 0;
    break;

// Master Receiver: Data byte received. NACK transmitted.
// Read operation has completed. Read data register and send STOP.
case SMB_MRDBNACK:
    WORD = SMB0DAT;
    STO = 1;
    SM_BUSY = 0;
    AA = 1;                // Set AA for next transfer
    SI = 0;
    break;

// Slave Receiver: Arbitration lost, general call address received.
// Set LOST flag to retry transfer when bus is free. Fall through.
case SMB_SRGARBLOST:

// Slave Receiver: Arbitration lost, own slave address + WRITE received.
// Set LOST flag to retry transfer when bus is free.
// Set STO bit to get out of master mode.
case SMB_SROARBLOST:
    LOST_COMMAND = COMMAND;    //
    LOST_WORD = WORD;        // Store variables for use when bus is
free.
    LOST_CODE = OP_CODE;    //
    LOST = 1;                // Retry transfer when bus is free.
    SI = 0;
```

```
break;

// Slave Receiver: Slave address + WRITE received. ACK transmitted.
// Fall through.
case SMB_SROADACK:

// Slave Receiver: General call address received. ACK transmitted.
case SMB_SRGADACK:
    SI = 0;
    break;

// Slave Receiver: Data byte received after addressed by general
// call address + WRITE.
// ACK transmitted. Fall through.
case SMB_SRGDBACK:

// Slave Receiver: Data byte received after addressed by own
// slave address + WRITE.
// ACK transmitted.
// Take action depending on OP_CODE or data received.
case SMB_SRODBACK:
    if (!VALID_OP){                // if VALID_OP=0, this byte is an OP_CODE.
        OP_CODE = SMB0DAT;        // Store OP_CODE
        VALID_OP = 1;            // Next byte is not an OP_CODE
    } else {
        DATA_READY = 1;        // Valid data has been received. Process
                                // in OP_CODE handler.

        WORD = SMB0DAT;
        SI = 0;
    }
    break;

// Slave Receiver: Data byte received while addressed as slave.
// NACK transmitted. Should not occur since AA will not be cleared
// as slave. Fall through to next state.
case SMB_SRODBNACK:

// Slave Receiver: Data byte received while addressed by general call.
// NACK transmitted.
// Should not occur since AA will not be cleared as slave.
case SMB_SRGDBNACK:
    AA = 1;
    SI = 0;
    break;

// Slave Receiver: STOP or repeated START received while addressed as slave.
case SMB_SRSTOP:
    SI = 0;
    break;

// Slave Transmitter: Own slave address + READ received. ACK transmitted.
// Load SMB0DAT with data to be output.
case SMB_STOADACK:
    SMB0DAT = WORD;
```

```
    SI = 0;
    break;

// Slave Transmitter: Arbitration lost as master. Own address + READ received.
// ACK transmitted.
case SMB_STOARBLOST:
    LOST_COMMAND = COMMAND;           //
    LOST_WORD = WORD;                 // Store variables for use when bus
    LOST_CODE = OP_CODE;              // is free.
    LOST = 1;                          // Retry when bus is free.

    SI = 0;
    break;

// Slave Transmitter: Data byte transmitted. ACK received. Fall through.
case SMB_STDBACK:

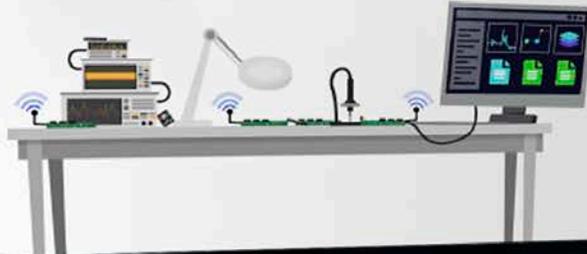
// Slave Transmitter: Data byte transmitted. NACK received. Fall through.
case SMB_STDBNACK:

// Slave Transmitter: Last data byte transmitted. ACK received.
// No action necessary.
case SMB_STDBLAST:
    SI = 0;
    break;

// All other status codes invalid. Reset communication.
default:
    STO = 1;
    SM_BUSY = 0;
    break;
}
}
```

Silicon Labs

Simplicity Studio™4



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!



IoT Portfolio
www.silabs.com/IoT



SW/HW
www.silabs.com/simplicity



Quality
www.silabs.com/quality



Support and Community
community.silabs.com

Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPPLL®, EFM®, EFM32®, EFR®, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOModem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.



Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

<http://www.silabs.com>