AN736

An I²C™ Network Protocol for Environmental Monitoring

Authors: Stephen Bowling, Richard L. Fischer
Microchip Technology Incorporated

INTRODUCTION

Communication network systems are rapidly growing in size and complexity. These systems have many high speed integrated circuits with critical operating parameters and must provide extremely reliable service with zero down time. To maintain the performance of these systems, adequate environmental monitoring must be performed, so a failure or a data trend leading to a potential failure can be rapidly identified. Furthermore, this monitoring must be performed cheaply to keep system costs low.

To minimize system down time and increase flexibility, these communication network systems feature modular, hot-swappable components. Each component in the system typically contains multiple sub-systems that require monitoring. These sub-systems might include DC/DC regulators, high speed microprocessors, FPGAs, and cooling fans. Some of the monitored system parameters include power supply output voltage, power supply current, device temperature, ambient temperature, and fan speed.

A network is required so all sensor data is collected and fed to a central computer for monitoring and analysis. Because many of the sensors are located in close proximity to each other, the I²C bus offers a solution that can be implemented with minimal hardware cost. Furthermore, low cost microcontrollers (MCUs) with a wide range of peripherals and an I²C interface are widely available.

For the I²C bus to be an effective solution for networked environmental sensors, a suitable bus protocol is required that prevents system bus errors from affecting sensor data. The purpose of this application note is to define such a network protocol, which may be easily adapted to most any networked application. The bus protocol must be immune to adverse network conditions, such as hot-swapping, or a malfunctioning network node.

THE I²C BUS SPECIFICATION

Although a complete discussion of the I²C bus specification is outside the scope of this application note, some of the basics will be covered here. For more information on the I²C bus specification, refer to sources indicated in the *References* section on page 15. A *Glossary of Terms* is also located on page 15.

The Inter-Integrated Circuit, or I²C bus specification, was originally developed by Philips Semiconductors for the transfer of data between ICs at the PCB level. The physical interface for the bus consists of two open drain lines; one for the clock (SCL) and one for data (SDA). The SDA and SCL lines are pulled high by resistors connected to the V_{DD} rail. The bus may have a one master/many slave configuration or may have multiple master devices. The master device is responsible for generating the clock source for the linked slave devices.

The I²C protocol supports either a 7-bit addressing mode, or a 10-bit addressing mode, permitting 128 or 1024 physical devices to be on the bus, respectively. In practice, the bus specification reserves certain addresses so slightly fewer usable addresses are available. For example, the 7-bit addressing mode allows 112 usable addresses.

All data transfers on the bus are initiated by the master device and are done eight bits at a time, MSb first. There is no limit to the amount of data that can be sent in one transfer.

The I²C protocol includes a handshaking mechanism. After each 8-bit transfer, a 9th clock pulse is sent by the master. At this time, the transmitting device on the bus releases the SDA line and the receiving device on the bus acknowledges the data sent by the transmitting device. An ACK (SDA held low) is sent if the data was received successfully, or a NACK (SDA left high) is sent if it was not received successfully. A NACK is also used to terminate a data transfer after the last byte is received.

According to the I²C specification, all changes on the SDA line must occur while the SCL line is low. This restriction allows two unique conditions to be detected on the bus; a START sequence (**S**) and a STOP sequence (**P**). A START sequence occurs when the master device pulls the SDA line low, while the SCL line is high. The START sequence tells all slave devices on the bus that address bytes are about to be sent. The STOP sequence occurs when the SDA line goes high, while the SCL line is high and it terminates the transmission. Slave devices on the bus should reset their receive logic after the STOP sequence has been detected.

The I²C protocol also permits a Repeated START condition (**Rs**), which allows the master device to execute a START sequence without preceding it with a STOP sequence. Repeated START is useful, for example, when the master device changes from a write operation to a read operation and does not release control of the bus.

A typical I²C write transmission would proceed as shown in Figure 1. In this example, the master device will write two bytes to a slave device. The transmission is started when the master initiates a START condition on the bus. Next, the master sends an address byte to the slave. The upper seven bits of the address byte contain the slave address. The LSb of the address byte specifies whether the I²C operation will be a read (LSb = 1), or a write (LSb = 0). On the ninth clock pulse, the master releases the SDA line so the slave can acknowledge the reception. If the address byte was received by the slave and was the correct address, the slave responds with an ACK by holding the SDA line low. Assuming an ACK was received, the master sends out the data bytes. On the ninth clock pulse after each data byte, the slave responds with an ACK. After the last data byte, a NACK is sent by the slave to the master to indicate that no more bytes should be sent. After the NACK pulse, the master initiates the STOP condition to free the bus.

A read operation is performed similar to the write operation and is shown in Figure 2. In this case, the R/W bit in the address byte is set to indicate a read operation. After the address byte is received, the slave device sends an ACK pulse and holds the SCL line low. By holding the SCL line, the slave can take as much time as needed to prepare the data to be sent back to the master. When the slave is ready, it releases SCL and the master device clocks the data from the slave buffer. On the ninth clock pulse, the slave releases the SDA line and latches the value of the ACK bit received from the master. If an ACK pulse was received, the slave must prepare the next byte of data to be transmitted. If a NACK was received, the data transmission is complete. In this case, the slave resets its I²C receive logic and waits for the next START condition.

For many I²C peripherals, such as non-volatile EEPROM memory, an I²C write operation and a read operation are done in succession. For example, the write operation specifies the address to be read and the read operation gets the byte of data. Since the master device does not release the bus after the memory address is written to the device, a Repeated START sequence is performed to read the contents of the memory address.

DEFINING NETWORK PROTOCOL

Now that the basics of the I²C bus have been covered, let's examine the needs of the sensor network. In this system, a single master device is on the bus and will periodically initiate communications with slave devices. The protocol must allow the master device to read or write data from a particular slave device. The type and length of data read from, or written to, the slave will depend, of course, on the specific function of the slave. For this reason, it would be efficient for the network protocol to support a variable data length dependent on the sensor node. The protocol should also allow a data address to be specified. Using a data address and data length, the master node can request any or all of the data available from the slave node.

There must be a method in the network protocol to ensure that data was transmitted or received successfully. Using checksums, the master and slave devices in the system verify that the data received was valid. If the data is not valid, the data should be retransmitted. Furthermore, the network protocol must handle bus errors gracefully. The sources of error include glitches due to hot-swapping, multiple devices responding to the same address (bus collisions), and no-response conditions from devices on the bus.

FIGURE 1: TYPICAL I²C WRITE TRANSMISSION (7-BIT ADDRESS)

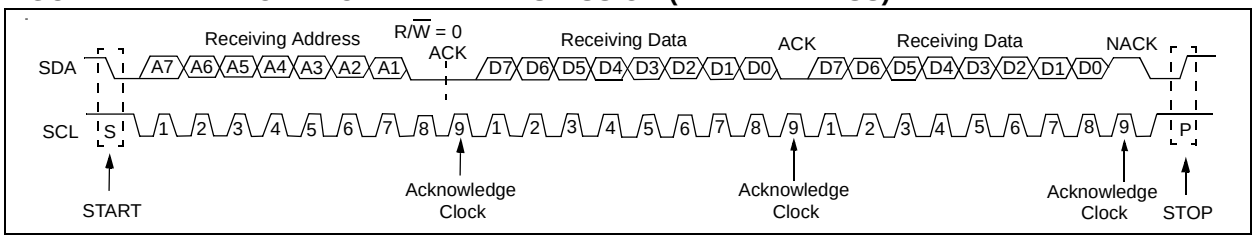
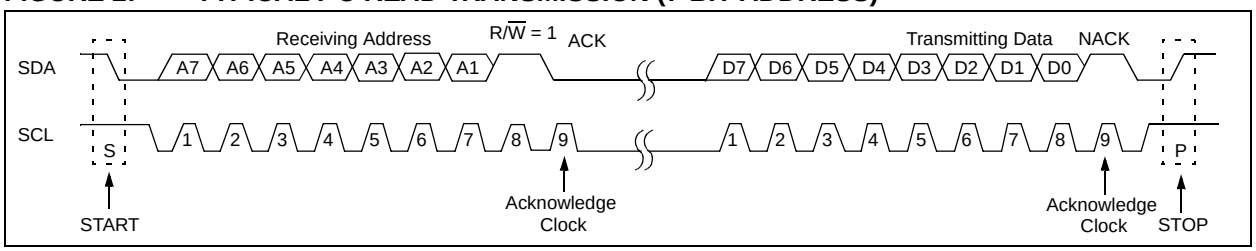


FIGURE 2: TYPICAL I²C READ TRANSMISSION (7-BIT ADDRESS)



Master Device Message Formats

Since all communication on the I^2C bus is initiated by the master device, a description of the protocol implemented by the master is required. In this application, the master device may initiate one of two message types; a *data write* message, or a *data request* message.

Data Write Message Format

The format for a data write message is shown in Figure 3. The data write message begins with the master initiating a START condition. When the START condition completes, the master device sends the I^2C address of the slave node with the R/W bit cleared to indicate data will be written to the slave device.

The next byte sent provides the byte count information. For this discussion, this byte will be referred to as the DATA_LEN byte. The DATA_LEN byte serves two purposes. First, the lower seven LSb's indicate the number of data bytes to be written to the slave device. Second, the MSb indicates whether data will be written to, or read from the slave. In this case, the MSb is cleared to indicate that a data write will be performed. The MSb of the DATA_LEN byte performs a similar function for the network protocol as the R/W bit in the I^2C address byte, but the two should not be confused.

The next byte sent by the master indicates the starting address in the slave node data buffer that will be written to, or read from. This byte will be referred to as the DATA_OFFS byte. Each slave device on the network maintains a range of data memory for received data and data to be transmitted.

In a data write message, the number of data bytes specified by the DATA_LEN byte will follow the DATA_OFFS byte. When the last byte of data has been sent, the master sends an 8-bit, two's complement checksum of all data previously sent, including the I^2C slave node address byte. Finally, the master device terminates the data write message by initiating a STOP condition.

Data Request Message Format

The format for a data request message is shown in Figure 4. Following the START condition, the master device sends the address of the slave node with the R/W bit cleared to indicate a data write to the I^2C slave device. Next, the DATA_LEN byte is sent. The seven LSb's of this byte indicate the number of data bytes to be read from the slave. Because a data read from the slave should be performed, the MSb is set. The DATA_OFFS byte follows the data length byte and indicates the starting address in the slave node data memory from which data will be read. Next, the master device sends an 8-bit, two's complement checksum of the slave address, data length byte, and data offset byte that were sent in the data request message.

FIGURE 3: DATA WRITE MESSAGE FORMAT

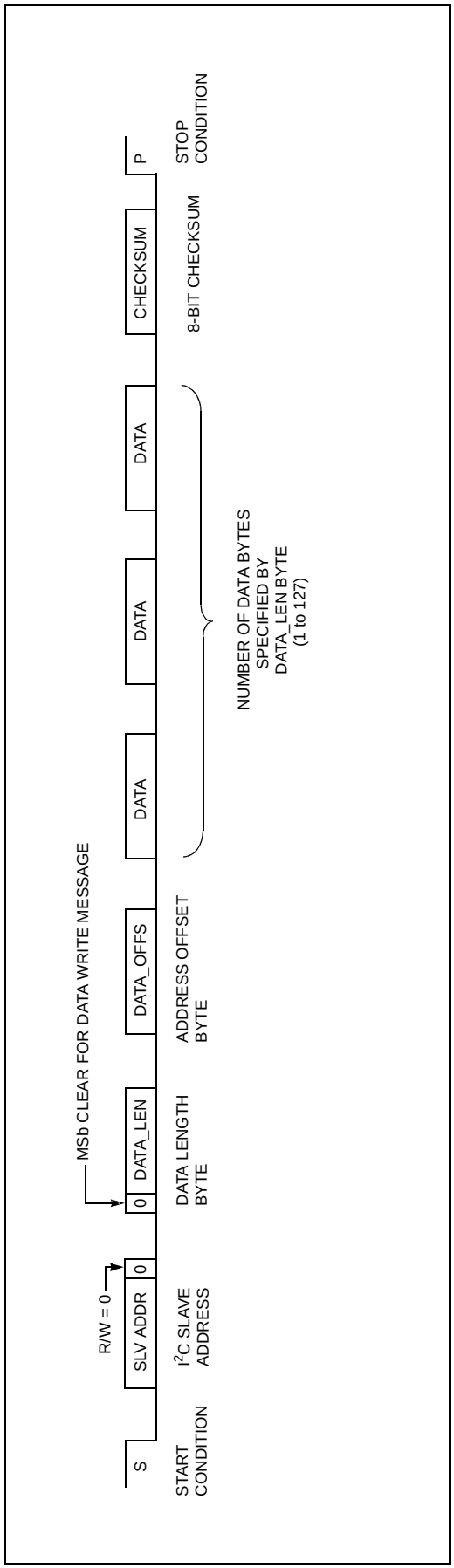
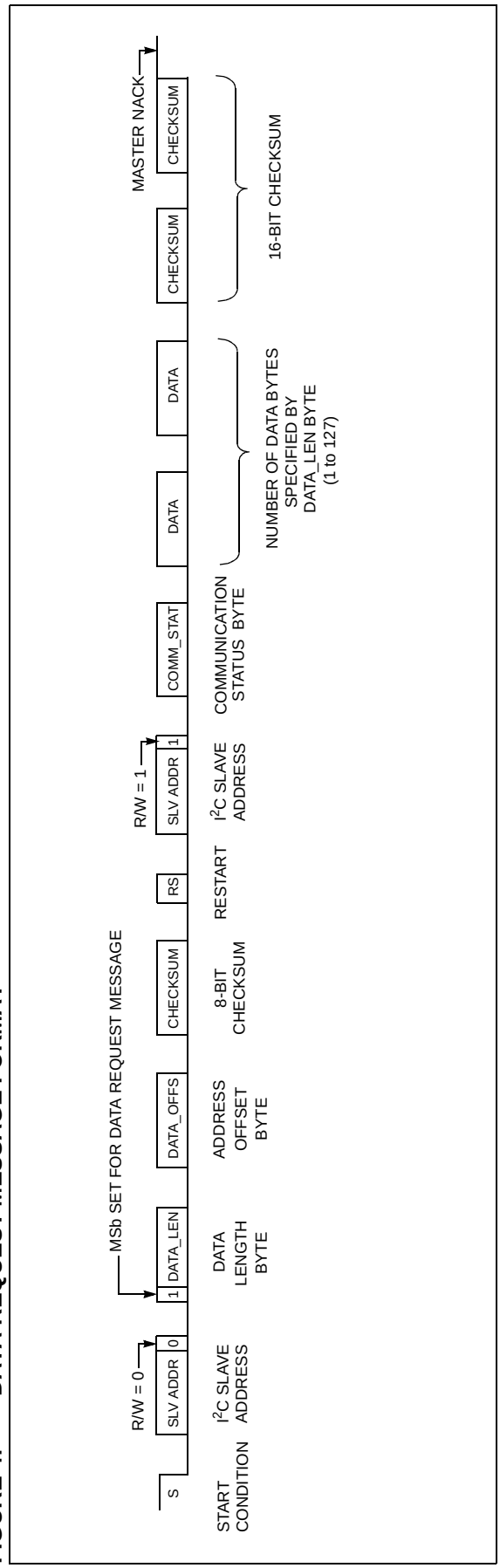


FIGURE 4: DATA REQUEST MESSAGE FORMAT



Slave Node Message Processing

In general, the master device may read data from the slave after a data write or data request message, by initiating a Restart condition on the I^2C bus and sending the slave address with the R/W bit set. The type of message that was previously sent by the master and its validity determines what data will be returned by the slave.

Each slave node maintains several status bits to indicate the validity of messages sent by the master device. These status bits are stored in the communication status (COMM_STAT) byte and Table 1 indicates the significance of each bit.

The COMM_STAT byte is always the first data byte to be returned in any data transfer from the slave node to the master node. This allows the master to verify that the previously sent message was processed correctly by the slave node. If, for example, the master sent a data write message, the value of the COMM_STAT byte would be 00h, if the data was successfully received by the slave. If a data request message was previously sent by the master, the value of the COMM_STAT byte would be 80h. If the master receives any other values for the COMM_STAT byte, some type of error has occurred and the master should send the data write or data request message again.

If a data write message was previously sent to the slave node, the master does not need to receive any more bytes from the slave node, after the COMM_STAT byte is read. For a data request message, the master should read number of data bytes specified by DATA_LEN.

A two's complement, 16-bit checksum is calculated for the data returned to the master. The checksum value includes the COMM_STAT byte, plus all data bytes that were returned. The master device should receive the two checksum bytes after the data bytes. If the master determines that a checksum error occurred while receiving the data bytes, it should try to read the data from the slave again.

TABLE 1: COMM_STAT BIT DEFINITIONS

Bit	Bit Name	Description
Bit 0	comm_stat.chkfail	Indicates a checksum failure occurred for the last message sent.
Bit 1	comm_stat.rxerror	Indicates the slave node did not interpret the last master message correctly.
Bit 2	comm_stat.ovflw	Indicates the master device has requested to read/write one or more bytes of data from the slave node, outside the valid range of addresses for that particular slave.
Bit 3	comm_stat.sspov	Indicates an overflow has occurred in the SSP module for a given slave address, because the slave device was not able to process incoming I^2C data quickly enough.
Bit 4		Unused
Bit 5		Unused
Bit 6		Unused
Bit 7	comm_stat.r_w	Indicates whether the last message from the master was a data request message (R/W = 1), or a data write message (R/W = 0).
Note: The bit structure 'comm_stat' is used in the C source code to access bits in the COMM_STAT byte.		

PUTTING IT ALL TOGETHER

Now that the basic implementation of the network protocol is defined, the functional operation of the master and slave controllers is presented.

The Master Node

General Overview

For this application, a PICmicro® PIC16F873 is implemented as the Master I^2C bus controller. This 28-pin FLASH based PICmicro device provides both the MSSP and USART modules for I^2C and USART communications, respectively.

The firmware code for this application is written in C, using the Hi-Tech PIC C Compiler™ and is included in Appendix B. Table 2 provides a brief description of the system files.

In addition to these C source files, some generic assembly I^2C master read and write routines were developed and are included in Appendix E. Table 3 provides a brief description of these files.

In this application, the master performs three basic tasks:

1. I^2C slave reads.
2. I^2C slave writes.
3. Transmission of received I^2C slave data and bus status to the PC.

For the most part, these tasks occur on an interrupt basis.

There are four types of interrupts that are implemented:

1. I^2C Event Completion Interrupt. This I^2C event interrupt indicates that an I^2C event has completed. I^2C events include START, STOP, Restart, Acknowledge, Read and Write. The hardware peripheral SSPIF bit (PIR1<6>) is asserted upon an event completion.
2. Bus Collision Interrupt. This interrupt is used for handling the detection of a bus collision. Typically, in a single master system (as described in this application), a bus collision is unlikely.
3. Timer1 Overflow Interrupt. This interrupt is used to generate a 100 ms time tick for initiating I^2C communications. When the master completes a current round of I^2C communications, Timer1 is restarted. When Timer1 overflows (100 ms later), the next round of I^2C communications begins.
4. USART Transmission Interrupt. This interrupt is used to send out 10 data bytes to the PC. After the master communicates with each slave device, a data packet is composed. The packet consists of the data read from the slave and the I^2C bus status. Each byte is transmitted to the PC on an interrupt basis at 19200 baud.

For the Master I^2C implementation, the MSSP module on the PICmicro MCU is used. The functional operation of this module is not covered within this document. For more information, consult AN735, "Using the PICmicro® MSSP Module for Master I^2C™ Communications", or refer to the specific PICmicro data sheet.

TABLE 2: MASTER I^2C 'C' SOURCE CODE FILES

File Name	Description
mstri2c.c	Main code loop and interrupt control functions.
mstri2c.h	Variable declarations & definitions.
i2c_comm.c	Routines for communicating with the I^2C slave device(s).
i2c_comm.h	Variable declarations & definitions.
init.c	Routines for initializing the PICmicro peripherals and ports.
cnfig87x.h	Configuration bit definitions for the PICmicro PIC16F87X.
pic.h	Required by compiler for SFR declarations (Hi-Tech file).
delay.h	Delay function prototypes (Hi-Tech file).

TABLE 3: MASTER I^2C 'ASM' SOURCE CODE FILES

File Name	Description
mastri2c.asm	Main code loop and interrupt control functions.
mastri2c.inc	Variable declarations & definitions.
i2ccomm1.inc	Reference linkage for variables used in i2ccomm.asm file.
i2ccomm.asm	Routines for communicating with the I^2C slave device.
i2ccomm.inc	Variable declarations & definitions.
flags.inc	Common flag definitions used within the mastri2c.asm and i2ccomm.asm files.
init.asm	Routines for initializing the PICmicro peripherals and ports.
p16f873.inc	PICmicro SFR definition file.
16f873.lkr	Modified linker script file.

Master Implementation

The master device, upon completion of the internal power-up cycle, performs some basic peripheral and key variable initialization. The functions used for peripheral initialization are listed:

- `Init_Usart()`
- `Init_Ports()`
- `Init_Timer1()`
- `Init_Ssp()`

These functions are located within the `init.c` file. Within the `Init_Ssp()` function, the MSSP module is initialized for Master I^2C mode, 400 kHz baud rate and slew rate is enabled. Once the peripheral initialization is completed, peripheral and global interrupts are enabled and the main code execution loop is entered (see Figure A-1).

In the main loop, the application firmware (F/W) tests the state of two flags:

- `sflag.event.read_i2c`
- `sflag.event.i2c_event`

These flags are initially asserted high in the Timer1 Interrupt Service Routine (ISR). The Timer1 interrupt starts the I^2C communication process and repeats every 100 ms. In the Timer1 ISR, the timer is shut off, the respective interrupt is disabled and the referenced event flags are set (see Figure A-2):

When the main loop program execution resumes, the F/W tests the state of these two flags. If both are a logic '1', the function `Service_I2CSlave()` is called. If one or both of the flags are negated (logic '0'), a loop comprised of a `CLRWDT` instruction and the flag test process repeats.

When the `Service_I2CSlave()` function is called, several operational code states are tested and executed, if true (see Figure A-3 through Figure A-4):

- Test if a new round of slave communications is to start. If so, initialize key variables and flags. This test is true every Timer1 rollover event.
- Test if the previous I^2C bus state was an I^2C write state. If so, test for Acknowledge error. If error exists, then issue bus STOP condition.
- Test if there was a I^2C bus or Acknowledge error. If true, compose error status for transmission to PC. If false, clear same error status.
- Test if the I^2C master should communicate with the next slave device. If true, then perform the following:
  - Initialize key variables and flags.
  - Call `Compose_Buffer()` function. In this function, a test is made to determine if the data packet read from the slave is valid. If valid, start transmission of data packet to PC. If invalid, perform an I^2C communication retry with same slave (see Figure A-5 and Figure A-6).
  - Test if a single data value received from the slave is out of range. Perform I^2C master write to the slave (see Figure A-7). The range limit test value is set by the `#define limit 0x80` macro (see the `i2c_comm.c` file).
- Test if the master has communicated with all slave devices. If true, return to the main code loop and wait for the next 100 ms time tick to expire. If false, initiate the next I^2C bus state, which may be a START, STOP, Restart, Read, Write, Send ACK or Send NACK (see Figure A-8, Figure A-9, and Figure A-10).

As mentioned, each new round of I^2C communications starts 100 ms from the completion of the previous round. This cycle is somewhat arbitrary, since the slave data is not used other than for display on a PC, and a data collection rate of 100 ms is adequate for this application. The I^2C communication cycle with each slave takes approximately 5 ms. Following this, 10 bytes are transmitted to the PC at 19200 baud, which equates to approximately 5.3 ms. The data is transmitted to the PC on an interrupt basis within the interrupt function, `interrupt_piv()` located in the `mastri2c.c` file.

In this application, the master I^2C device communicates with twelve slave devices. It is possible to increase the number of slaves, but PICmicro resources must be considered. For example, a slave device, upon request, may transmit up to 127 data bytes to the master. The data read from the slave must fit into contiguous memory, since an array variable is used to hold the data. This may, or may not be possible, based on the total master I^2C device resource requirements. In addition, a RAM array variable is defined and initialized with a data length byte, address offset byte, and 8-bit checksum for each slave (see Figure 4 for the message format). For twelve slaves, the array size totals 36 bytes. One can see that the size of the array depends on the number of slaves. Although this array is placed in RAM, it could have been placed in program memory, but then a dynamic update to the array would not be possible.

In short, this application may be modified to allow for more slave I^2C devices with minor code changes, but additional PICmicro resources may be required.

During each slave communication cycle, the master reads slave data and status while monitoring and recording errors, such as bus collision and Acknowledge errors (NACK). While bus collisions are more typical in a 'multi-master' environment, a bus collision may still occur in a single master system. For example, a slave device may experience a malfunction (firmware and/or hardware), and as a result, the SDA and SCL bus levels are driven low during a transmission. The later error condition may result in a permanent bus fault until corrective action is taken. In any case, the master I^2C device should monitor for this condition and take the appropriate action. When a bus collision is detected, a status bit will be set to a logic '1' for that particular slave. When the bus collision error is corrected, the same status bit will be set to a logic zero. This status information is part of the data packet sent to the PC. In addition, the master will attempt at least one I^2C communication retry. Additional retries are attempted by changing the substitution text in the macro defined in file `i2c_comm.h`. For example, one communication retry is implemented for:

```
#define MaxSlaveRetry 1
```

Two communication retries are made for:

```
#define MaxSlaveRetry 2
```

Another error condition the master I^2C device should monitor for is the Not Acknowledge (NACK) condition. If, for any reason, the slave issues a Not Acknowledge (does not drive SDA low during the ninth clock pulse of a write), the master should detect this and take the appropriate action. As with the bus collision error, a status bit will be asserted according to the error state. For this condition, the master issues a STOP condition after detecting a NACK. This action differs from the bus collision, in that as a result of a bus collision, the MSSP module goes into an IDLE state. The next valid I^2C state should be a START condition. As a result of a NACK condition, the module does not go into an IDLE state. An I^2C bus Restart or STOP/START combination should be executed, depending on the desired action.

For this application, (see Figure 4) the master reads five bytes of information from each slave, three bytes of data, and two bytes for the checksum. The data, along with the slave ID, bus and communication error status is transmitted to the PC for display. While the USART transmission is in progress, the master may also execute an I^2C write sequence to the slave. The write sequence is automatic per each slave, but the data written depends on the value of the second byte read from the slave. The `Write_I2CSlave()` function performs this write sequence and is called from within the `Compose_Buffer()` function. This write sequence is concurrent with the USART communications. For this application, the `Write_I2CSlave()` function provides the slave I^2C device with a response from the master, based upon the limit evaluation of this second byte (see Figure A-7). This function executes as a control loop using the I^2C event completion interrupt.

Finally, for each I^2C communication state with a slave, excluding the `Write_I2CSlave()` function, the master generates each I^2C bus state within the `I2CBusState()` function. This function is based upon switch/case control statements. Upon entering this function, the F/W performs a table lookup for the next I^2C state. The states for each sequence are predefined in the `const unsigned char` array, `ReadFSlaveI2CStates` declared in the file `i2c_comm.h`. This implementation allows simple addition or deletion of I^2C bus states. When the next I^2C state has been obtained, a switch statement evaluates the state variable `i2cstate` and the correct case statement initiates the next bus state. The F/W then returns to the main code loop and waits for the next I^2C event completion interrupt.

The Slave Node

The slave node firmware is provided in Appendix D and was written for a PIC16C72A device using the Hi-Tech PICC compiler. The PIC16C72A device was chosen for the sensor node, because it is a low cost device that has the SSP module required for I^2C communications. The slave firmware contains the following primary C functions:

- `Setup()`
- `ISR_Handler()`
- `SSP_Handler()`
- `AD_Handler()`
- `CCP2_Handler()`

The `Setup()` function initializes all of the Special Function Registers (SFR) in the PIC16C72A and all of the program variables.

Interrupts

The slave node firmware is primarily interrupt-driven. The SSP module, CCP2 module, and A/D module are the sources of interrupts. The `ISR_Handler()` function polls the interrupt flag bits and calls the appropriate module handler function.

Event Timing

The CCP2 module is used in the Compare mode as an event timer for the firmware and provides an interrupt every 1 msec. The `CCP2_Handler()` function is called when a CCP2 interrupt occurs. In addition to the 1 msec interrupt, `CCP2_Handler()` also maintains 10 msec, 100 msec, and 1000 msec timing flags for scheduling other events.

Slave Node Data Buffers

Three data buffers are used in the slave node application. The first of these data buffers is SensorBuf, which is 12 bytes in length and holds all sensor data to be sent to the master node. The SensorBuf buffer is implemented as a union that allows this data space to be addressed, both as bit fields and as bytes. The first byte of SensorBuf holds the communication status (COMM_STAT) byte, which has status bits indicating the success or failure of an operation by the master device. The next two bytes in SensorBuf hold status bits reserved for indicating out-of-range conditions for each sensor channel in the slave node. These bits could be read by the master device to get a quick 'go/no-go' response for all of the parameters the slave node is monitoring. The remaining nine bytes in SensorBuf hold 8-bit data values for each of the slave node sensor measurements. Constants are defined at the beginning of the source code for the index values to SensorBuf.

The next buffer is RXBuffer, which holds bytes sent by the master device during data request and data write messages. The length of this buffer is defined to be eight bytes in the firmware. This buffer has to be large enough to hold the slave address byte (SLAVE_ADDR), the data length byte (DATA_LEN), the data offset byte (DATA_OFFS), the transmit checksum, plus the total number of data bytes the master may write to the slave.

The third buffer used in the firmware is CmdBuf, which holds data bytes written to the slave device. For this application, up to four bytes may be written to a particular slave node. The four data bytes are copied from RXBuffer to CmdBuf, when a valid data write message from the master has been received. If the data write message is invalid, the data bytes in RXBuffer are discarded.

Sensor Data

The firmware for the PIC16C72A performs the following measurements as a remote sensor node:

- Analog Voltage/Current, 4 channels
- Fan Tachometer, 4 channels
- Temperature, 1 channel

This particular combination of sensor inputs was arbitrarily chosen, based on parameters commonly measured in an environmental monitoring application. In fact, the master firmware in this application only requests three of the nine available sensor data values. In practice, you may want to modify the firmware to accommodate a different combination of input channels. Furthermore, the firmware will operate on most any PICmicro device that has a SSP or MSSP module, with minor modifications. For example, you may want to select another device if you need more I/O pins, more A/D channels, non-volatile EEPROM data memory, or a higher resolution A/D converter.

A/D Conversions

A new A/D conversion is started in main() each time the 10 msec timing flag is detected. The AD_Handler() function is called from the Interrupt Service Routine each time an A/D interrupt occurs. The AD_Handler() function determines the presently selected A/D channel and stores the result in the correct location in SensorBuf. The A/D input multiplexer is then set to the next channel to be read. Each A/D input channel is sampled every 50 msec, which is adequate for most applications.

A thermistor is connected to CH4, which requires linearization to provide correct temperature readings. The A/D result from CH4 is used as an index to a temperature lookup table that provides the correct temperature in degrees Fahrenheit. The values in the temperature lookup table will depend on the thermistor and external circuit chosen for your design.

Fan Tachometer Data

I/O pins RB7:RB4 are used for fan tachometer inputs. These four pins have the weak pull-up feature and minimize the amount of hardware required in the design. Every 1 msec, the tachometer inputs are sampled and compared with their values from the previous sample. A count variable is maintained for each tachometer input. If a change has occurred on an input pin since the last sample, the count variable for that input is incremented. Each time a 1000 msec timing flag is detected in main(), the number of counts accumulated in the count variables are stored in the appropriate locations of SensorBuf and the count variables are cleared so that a new speed sample can be acquired.

The characteristics of the tachometer output depends on the particular fan that is used. Some brushless DC cooling fans, for example, have an open collector tachometer option that provides between 1 and 4 pulses per revolution. A small DC cooling fan with the following specifications was selected to provide design data for calculations:

- Voltage: 12 VDC
- Speed: 3000 RPM
- Tach: open collector square wave output, 2 pulses per revolution, 50% duty cycle

Based on these specifications, the fan will provide a tachometer output frequency of 100 Hz at its rated speed and the tachometer count variable will advance at the rate of 200 counts per second at the maximum fan speed. The I/O pin must be sampled at a frequency greater than 200 Hz to avoid signal aliasing and the accumulation time must be adjusted to scale the maximum fan speed data value. In this case, unsigned integers are used to hold the tachometer values, which allows a maximum data value of 255. If a 1000 msec accumulation time is used, the tachometer reading will be 200 at the rated fan speed. This choice of accumulation time allows some overhead to prevent overflow of the accumulated tachometer data.

SSP Event Handling

I²C bus events are processed in the SSP_Handler() function, which is the heart of the I²C network protocol. If you need more general information on using the SSP module as an I²C slave device, please refer to AN734, "Using the PICmicro® SSP for Slave I²C™ Communication".

The SSP module is configured for I²C Slave mode, 7-bit addressing. When a SSP interrupt occurs, the SSP_Handler() function must identify the I²C event that just occurred on the bus and take the appropriate action. For the purposes of explanation, it is helpful to identify all possible states of SSP module after an I²C event and discuss each one individually.

The following five states are recognized and handled in the SSP_Handler() function by testing bits in the SSPSTAT register:

- State 1: I²C write operation, last byte received was an address, buffer is full
- State 2: I²C write operation, last byte received was data, buffer is full
- State 3: I²C read operation, last byte received was an address, buffer is empty
- State 4: I²C read operation, last byte received was data, buffer is empty
- State 5: I²C logic reset by NACK from master device

Flow charts for the SSP_Handler() function are given in Appendix C.

State 1

State 1 occurs after a valid START condition has occurred on the bus and an address was transmitted that caused an address match in the SSP module of the slave device. The LSb of the address byte is '0', which indicates a I²C write operation. This condition indicates that the master device is about to send the bytes for a new data write, or data request message. Since this is the beginning of a new transaction on the bus, a status flag is set in software to disable clearing of the Watchdog Timer in the main program loop. If the transaction takes longer than expected, due to a problem with the slave device, or an error on the bus, then the Watchdog Timer will reset the slave device and SSP module. In addition, the COMM_STAT byte is initialized with the comm_stat.rxerror bit set. This bit will not be cleared until all bytes in the data write or data request message have been received and a valid transmission has been verified. The RXBufferIndex variable is set to '0' and RXBuffer is cleared. The address byte that is currently in SSPBUF is stored in RXBuffer and is also used to initialize the value of RXChecksum.

State 2

In the second state recognized by SSP_Handler(), the bytes for a data write or data request message are stored in RXBuffer and RXBufferIndex is incremented after each byte received, to point to the next empty buffer location. The value of RXBufferIndex is checked against the length of RXBuffer to ensure that a buffer overflow does not occur. If a RXBuffer overflow occurs, the value of RXBufferIndex is set to the last location in the buffer and the comm_stat.ovflw bit is set in the COMM_STAT byte to indicate that the overflow occurred. If a SSP module overflow has occurred, the comm_stat.sspov bit is set in COMM_STAT. After each data byte is received, its value is added to RXChecksum and RXBufferIndex is compared against constant index values to determine the significance of the present byte in SSPBUF.

If the byte just received is the DATA_LEN byte (byte #1), the MSb is checked to see if a data write or a data request is to be performed and the comm_stat.r_w bit in the COMM_STAT byte is set to indicate the status of the message. If the MSb of the DATA_LEN byte is set, indicating a data request message, this bit is masked to '0' so that it will not affect future calculations using the data length value stored in the 7 LSbs. The DATA_LEN value is used to determine the value of RXByteCount, which holds the expected number of bytes to be received for the message. For a data request message, RXByteCount is always set to '3', because the number of expected bytes is fixed. For a data write message, RXByteCount is set to '3', plus the number of bytes indicated by the DATA_LEN byte.

If the byte just received is the DATA_OFFS byte (byte #2), a check is performed to see if the data request message or data write message will exceed the size of SensorBuf or CmdBuf. If the message exceeds the size of the buffer, the comm_stat.ovflw status bit is set.

If the number of bytes received is equal to RXByteCount, the end of the message has been reached. If the value of RXChecksum is not '0', the comm_stat.chkfail status bit in the COMM_STAT byte is set. If a data write message was sent and RXChecksum is '0', then the data contained in the message is considered valid and is transferred from RXBuffer into CmdBuf.

State 3

State 3 occurs after a valid START condition has occurred on the bus and an address was transmitted that caused an address match in the SSP module of the slave device. The LSb of the address byte is '1', which indicates a I²C read operation. This condition indicates that the master device wishes to read bytes from the slave device.

As mentioned, the COMM_STAT byte will always be the first byte returned during a read from the slave. This byte is written to SSPBUF and the value of TXChecksum is initialized. The value of SensBufIndex is set to '0' for future read operations.

State 4

In State 4, the slave node will send data bytes in `SensorBuf` to the master based on the values of the DATA_LEN and DATA_OFFS bytes. Each byte that is sent is added to the value of `TXChecksum`. If the number of bytes specified in the DATA_LEN byte have been sent, then the 16-bit value of `TXChecksum` is returned. If there are no more bytes to be returned to the master, then the slave simply returns dummy data.

State 5

The final state detected in `SSP_Handler()` is caused by a NACK from the master device. This action indicates to the slave device that the master does not wish to receive any more data. The NACK event is used as a signal in this protocol to indicate the completion of a transaction on the I^2C bus. Consequently, the `stat.wdtdis` flag is cleared in the slave firmware to re-enable clearing of the Watchdog Timer.

Design Calculations for the I^2C Bus

When designing an I^2C network, the number of devices on the bus, physical characteristics of the bus wiring, and the length of the bus must be considered. These variables determine the total amount of capacitive load on the bus, which the I^2C specification limits to 400pF. The value of the bus pull-up resistors are chosen based on the bus capacitance.

If the electrical characteristics of the wiring used for the I^2C bus are known, then it is easy to determine the total bus capacitance. All that is required is to figure out the capacitance contribution of each device on the bus. If the capacitance of each device is not known, then 10pF per device is a good estimate.

Another way to find the total bus capacitance is to pick preliminary values for the pull-up resistors and analyze the rise time on the bus, using a digital storage oscilloscope. For most applications, 2000Ω would be a good starting value for the pull-up resistors. The rise time is the time that the signal takes to go from 10% to 90% of the final value. Then, the total bus capacitance can be determined using Equation 1.

EQUATION 1: BUS CAPACITANCE CALCULATION

$$C_{BUS} = \frac{t_R}{2.2 \bullet R}$$

Next, the rise time specification for the I^2C bus must be known, which is dependent on the bus frequency. For high speed mode (400kHz), the maximum rise time is 300nS. For standard mode (100kHz), the maximum rise time is 1µs. Equation 1 can be rearranged to find the required value of the pull-up resistors as shown in Equation 2.

EQUATION 2: PULL-UP RESISTANCE CALCULATION

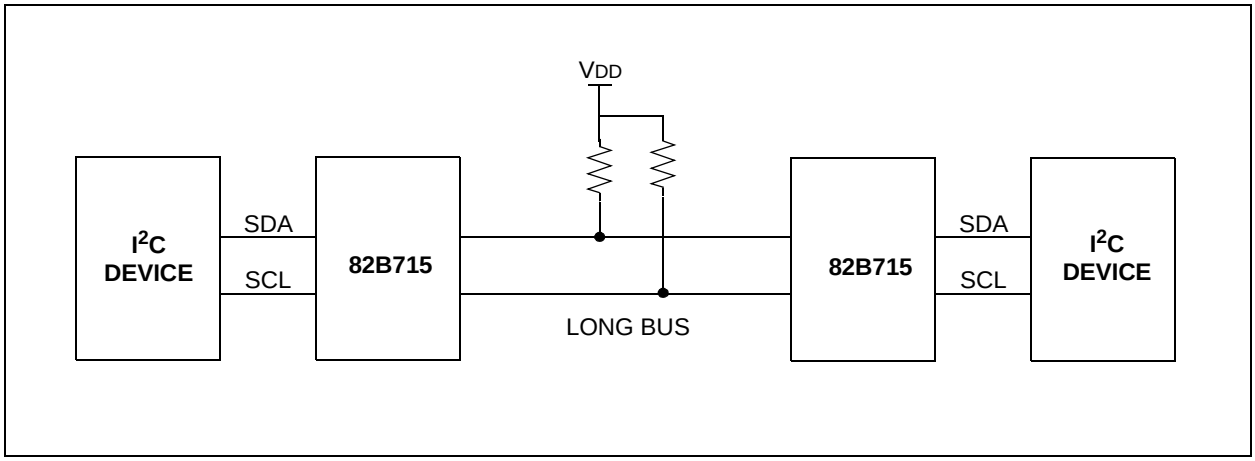
$$R_{PULLUP} = \frac{t_R}{2.2 \bullet C_{BUS}}$$

The I^2C specification limits the amount of current on the bus to 3mA, which indirectly places a limit on the value of the pull-up resistors. So for a 5V bus, the minimum pull-up resistance that could be used is 5V/3mA, or approximately 1600Ω.

Driving Longer Distances

If the bus length in the application exceeds a few feet, selection of pull-up resistor values that satisfy the I^2C specifications is a bit harder. In this case, bus extender IC's are available that allow you to use a longer bus in your design. One such IC, the Philips 82B715, provides a 10x current gain. This IC allows the total bus capacitance to increase to 4000pF and the maximum current on the bus to 30mA. Figure 4 shows how the bus extender IC's are connected. It may be possible to eliminate the need for the bus extenders since PICmicro I/O pins can sink or source greater than 3mA. Refer to the appropriate device data sheet for further details.

FIGURE 5: I^2C BUS EXTENSION BLOCK DIAGRAM



Example Design Calculations

As a design example, the characteristics of the wire that was used to test the application firmware provided in this application note, will be used in the calculations that follow. A 24 ft. length of wire was used to connect two PIC16F873 devices with 200Ω pull-up resistors on the SDA and SCL lines. The SCL line was observed on an oscilloscope and the rise time was determined to be 464ns. The wiring capacitance, per foot, is calculated in Example 1.

EXAMPLE 1: WIRING CAPACITANCE CALCULATION

$$C_{WIRE} = \frac{464\ ns}{(2.2)(200\ \Omega)(24\ ft)} = \frac{44\ pF}{ft}$$

The maximum bus length that could be used with this wire, without bus extenders, is calculated in Example 2.

EXAMPLE 2: MAXIMUM BUS LENGTH CALCULATION

$$L_{MAX} = \frac{400\ pF}{44\ pF/ft} = 9.1\ ft$$

Note that this length calculation also excludes the effects of device capacitance and would be reduced slightly in practice. Using the bus extenders, a theoretical bus length of 90 feet can be realized, using this wire.

For further calculations, assume that the bus length is specified to be 3 feet. Using the wire chosen for this design example, would set the bus capacitance to 3 x 44pF/ft. or 132pF. Now, we need to choose the bus frequency, which is arbitrarily selected to be 100kHz. Using the maximum rise time specification for a 100kHz bus frequency, the value of the pull-up resistors is calculated in Example 3.

EXAMPLE 3: PULL-UP RESISTOR CALCULATION

$$R_{PULLUP} = \frac{1\ \mu s}{(2.2)(132\ pF)} \approx 3400\ \Omega$$

A pull-up resistor value of 3400Ω will provide approximately 1.5mA on the bus, which does not violate the maximum current limit.

Table 4 shows the maximum bus length based on the bus frequency, bus current limits, use of bus extenders, and the characteristics of our wire. Although you will need to calculate the maximum bus length for your specific application, this data table will give an approximate idea of what can be achieved.

Slew Rate Control

PICmicro devices with the MSSP module have a slew rate control feature. The slew rate control limits the slope of the falling edge of the SCL and SDA lines to lower EMI. Slew rate control is enabled in the MSSP module by clearing the SSPSTAT <7> bit (SMP). If a clock frequency greater than 400kHz is used, then the slew rate control should be disabled. Otherwise, the maximum fall-time specifications may be violated.

Additional SCL and SDA pin characteristics for the MSSP module are listed in Table 5.

TABLE 4: MAXIMUM BUS LENGTHS FOR EXAMPLE DATA

Bus Capacitance = 44 pF/ft	Pull-up Resistance	Bus Frequency = 100kHz	Bus Frequency = 400kHz
		Maximum Bus Length	Maximum Bus Length
No bus extender	1600Ω	6 feet	1.8 feet
82B715 extender IC	160Ω	60 feet	18 feet

Note: Bus length is limited by the choice of pull-up resistor values that do not exceed the maximum bus current in the I^2C specification.

TABLE 5: PICMICRO DEVICES WITH MSSP MODULE

Device	I^2C Pin Characteristics			
	Slew Rate Control(1)	Glitch Filter(1) on Inputs	Open Drain Pin Driver(2,3)	SMbus Compatible Input Levels(4)
PIC16C717	Yes	Yes	No	No
PIC16C770	Yes	Yes	No	No
PIC16C771	Yes	Yes	No	No
PIC16C773	Yes	Yes	No	No
PIC16C774	Yes	Yes	No	No
PIC16F872	Yes	Yes	No	Yes
PIC16F873	Yes	Yes	No	Yes
PIC16F874	Yes	Yes	No	Yes
PIC16F876	Yes	Yes	No	Yes
PIC16F877	Yes	Yes	No	Yes
PIC17C752	Yes	Yes	Yes	No
PIC17C756A	Yes	Yes	Yes	No
PIC17C762	Yes	Yes	Yes	No
PIC17C766	Yes	Yes	Yes	No
PIC18C242	Yes	Yes	No	No
PIC18C252	Yes	Yes	No	No
PIC18C442	Yes	Yes	No	No
PIC18C452	Yes	Yes	No	No

Note 1: A “glitch” filter is on the SCL and SDA pins when the pin is an input. The filter operates in both the 100 kHz and 400 kHz modes. In the 100 kHz mode, when these pins are an output, there is a slew rate control of the pin that is independent of device frequency

2: P-Channel driver disabled for PIC16C/FXXX and PIC18CXXX devices.

3: ESD/EOS protection diode to V_{DD} rail on PIC16C/FXXX and PIC18CXXX devices.

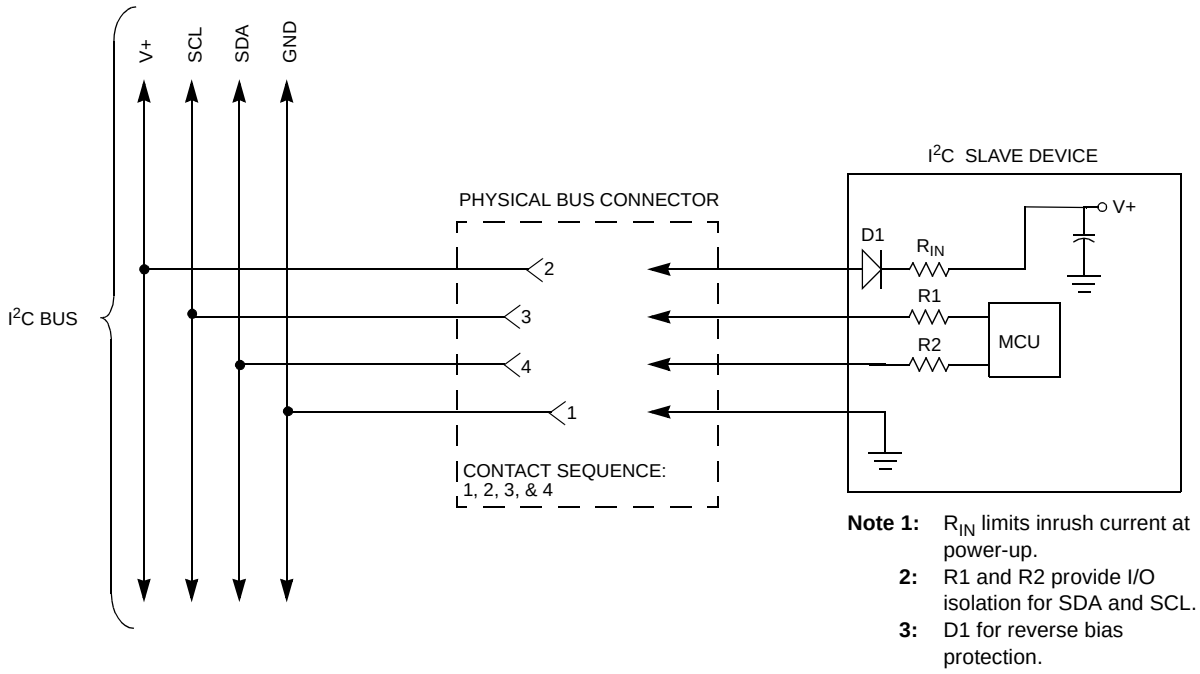
4: SMbus input levels are not available on all PICmicro devices. Consult the respective data sheet for electrical specifications.

Hardware Faults

In a distributed environmental monitoring system, slave devices may be 'hot-swapped' on the bus to replace faulty systems, or for regular maintenance and testing. The application hardware will vary depending on the system requirements, but certain hardware features can be implemented in every system to ensure that minimal errors are introduced on the I^2C bus, when a new device is inserted or removed. The connector hardware chosen must properly sequence the power supply and data signal connections to the host system. As a slave node is connected to the I^2C bus, the first physical connection made should be the ground lead, so any residual potential is discharged into the system ground. The second connection should be the power to the slave node.

To avoid brown-out conditions on the system bus, the total amount of capacitance on the power supply rails should be considered and series current limiting resistors should be installed to limit the amount of inrush current. The SDA and SCL lines should be the last connection made through the connector. It is a good idea to install small resistors in series with the SDA and SCL lines. These resistors limit the amount of current that may flow through the I/O pins of the MCU during power-up. Figure 6 shows a sample block diagram of the physical bus connection.

FIGURE 6: PHYSICAL I^2C BUS CONNECTION DETAILS



Note 1: R_{IN} limits inrush current at power-up.
2: R1 and R2 provide I/O isolation for SDA and SCL.
3: D1 for reverse bias protection.

CONCLUSION

There are several established synchronous protocols available for implementation into any design requiring such. Each protocol will have its pros and cons and should be weighed accordingly, relative to the application requirements.

For this application note, the communications network is based on the I^2C protocol. Some features of the I^2C bus include:

- Only two bus lines are required: a serial data line (SDA) and a serial clock line (SCL).
- Minimal physical bus requirements; only two pull-up resistors required.
- Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers.
- It is a true multi-master bus including collision detection and arbitration to prevent data corruption, if two or more masters simultaneously initiate data transfer.
- On-chip filtering spikes on the bus data line to preserve data integrity.

From the Slave I^2C device to the Master I^2C device, Microchip Technology offers several PICmicro devices which support these functional features. I^2C based communication network systems implementing the PICmicro device are cost effective and easy to implement.

Note:	Information contained in the application note regarding device applications and the like, is intended through suggestion only and may be superseded by updates. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated, with respect to the accuracy or use of such information, or infringement of patents, or other intellectual property rights arising from such use or otherwise.

WHAT'S IN THE APPENDIX

Flow charts and C source code for the master node application have been included in Appendix A and Appendix B, respectively. Flow charts and C source code for the slave node application have been included in Appendix C and Appendix D.

Appendix E and Appendix F contain generic I^2C code written in assembly language. The assembly code does not implement the network protocol described in this application note, but you can use the routines as a starting point for your own application. The source code for the master device transmits a string of characters to the slave device and then reads the string back. The slave device stores the character string in a data memory buffer until a new string is written.

GLOSSARY OF TERMS

ACK - Acknowledge

BRG - Baud Rate Generator

BSSP - Basic Synchronous Serial Port

EEPROM - Electrically Erasable Programmable Read Only Memory

F/W - Firmware

I^2C - Inter-Integrated Circuit

ISR - Interrupt Service Routine

MCU - Microcontroller Unit

MSSP - Master Synchronous Serial Port

NACK - Not Acknowledge

SDA - Serial Data Line

SCL - Serial Clock Line

SSP - Synchronous Serial Port

REFERENCES

The I^2C-Bus Specification, Philips Semiconductor, Version 2.1, 2000, http://www-us.semiconductors.com/i2c/

PICmicro™ Mid-Range MCU Reference Manual, Microchip Technology Inc., Document Number DS33023

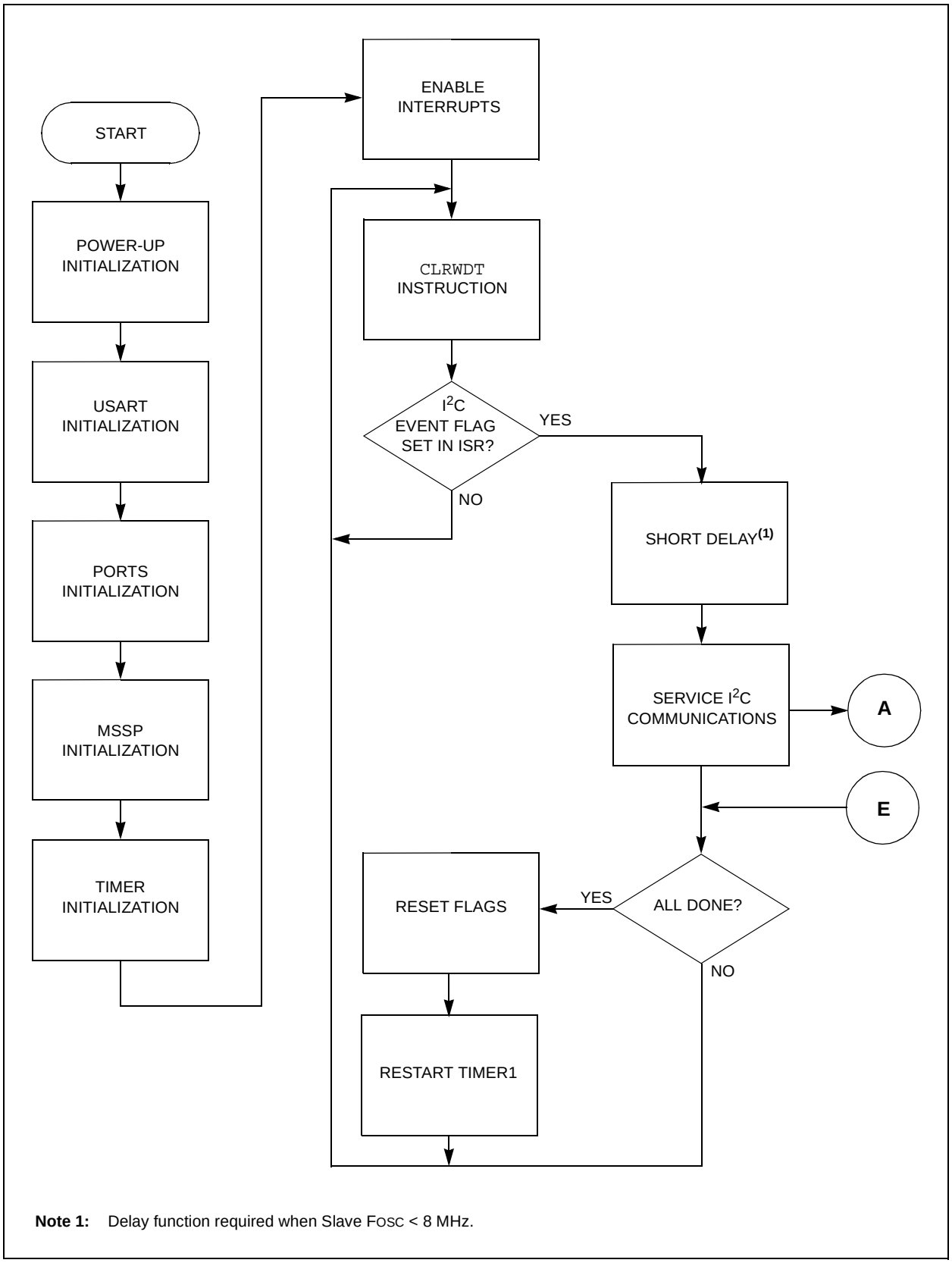
PIC16F87X Data Sheet, Microchip Technology Inc., Document Number DS30292

AN735, "Using the PICmicro® MSSP Module for Master I^2C™ Communications", Microchip Technology Inc., Document Number DS00735

AN734, "Using the PICmicro® SSP for Slave I^2C™ Communication", Microchip Technology Inc., Document Number DS00734

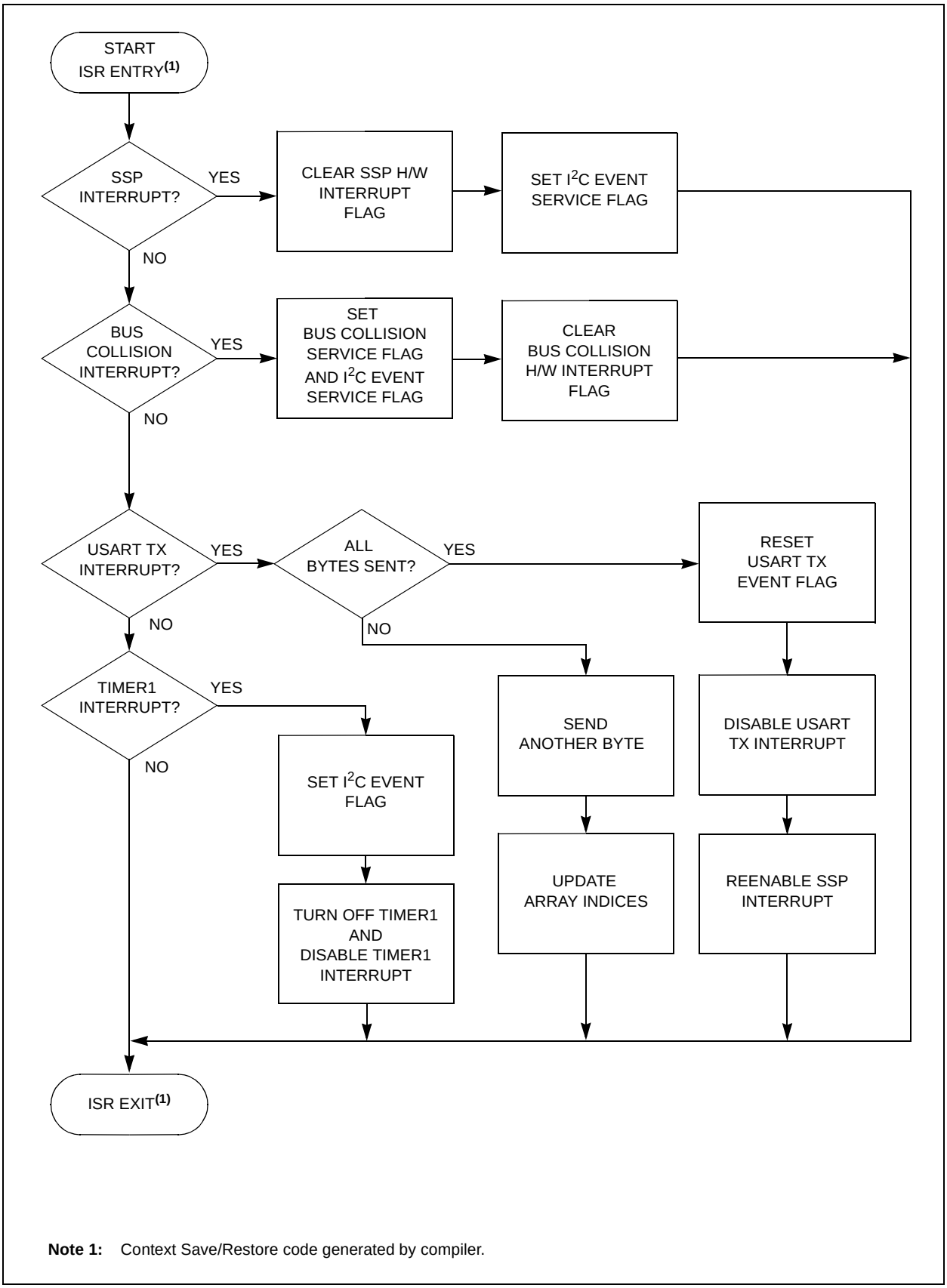
APPENDIX A: MASTER I²C CODE FLOW CHARTS

FIGURE A-1: INITIALIZATION AND MAIN CODE LOOP FLOW



Note 1: Delay function required when Slave FOSC < 8 MHz.

FIGURE A-2: INTERRUPT SERVICE ROUTINE CODE FLOW



Note 1: Context Save/Restore code generated by compiler.

FIGURE A-3: SERVICE I²C SUBROUTINE CODE FLOW (1 OF 2)

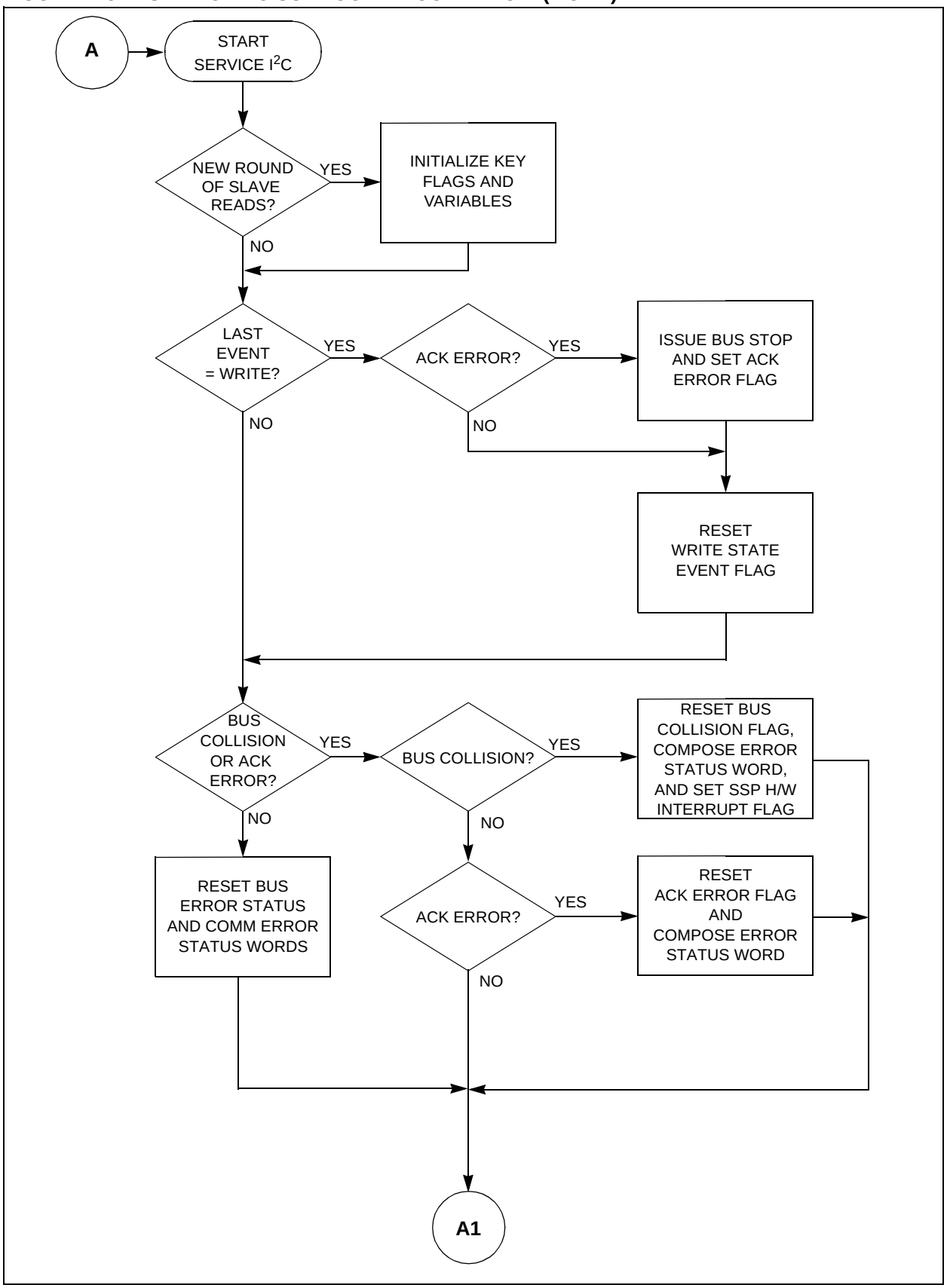


FIGURE A-4: SERVICE I²C SUBROUTINE CODE FLOW (2 OF 2)

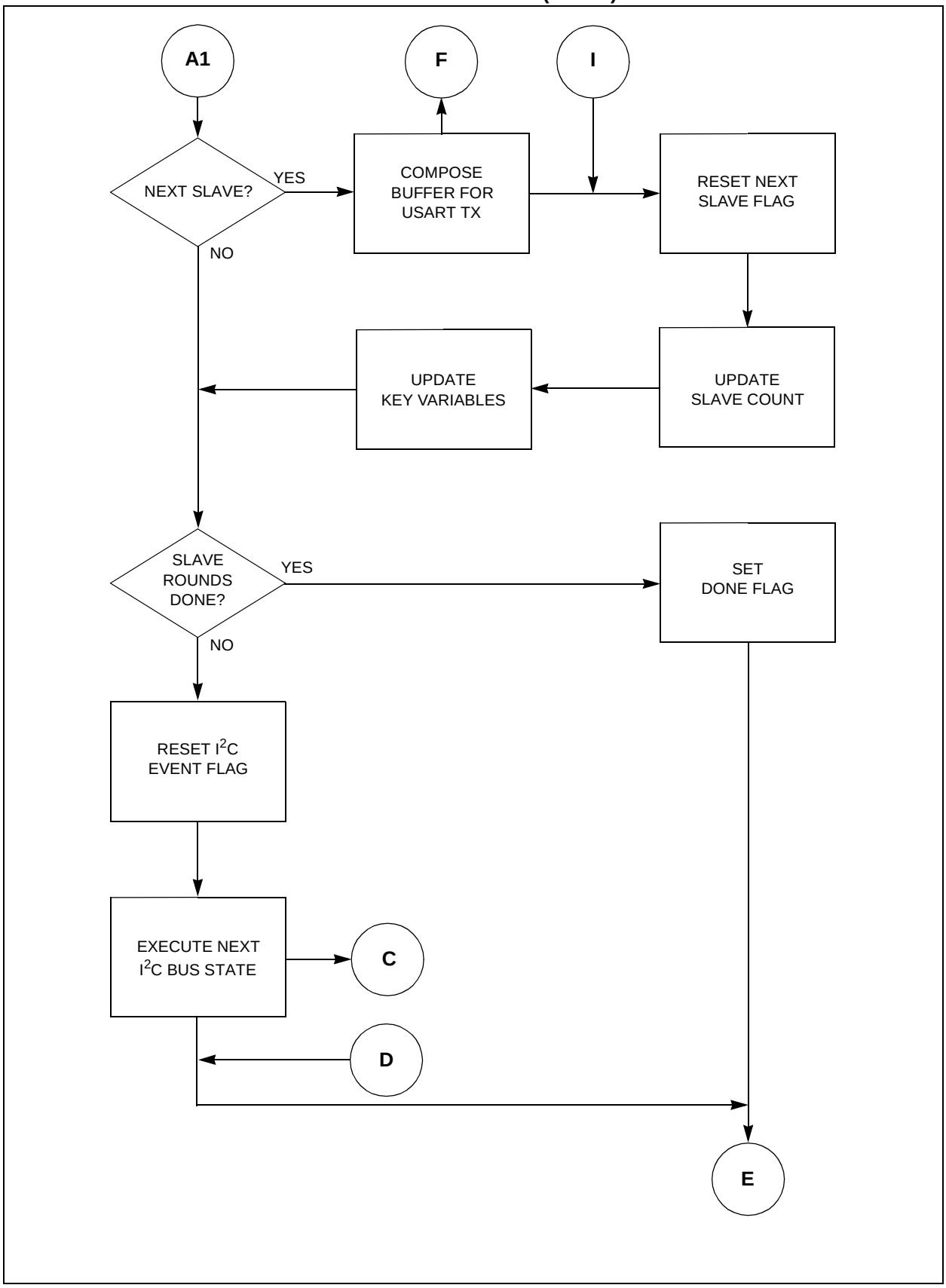
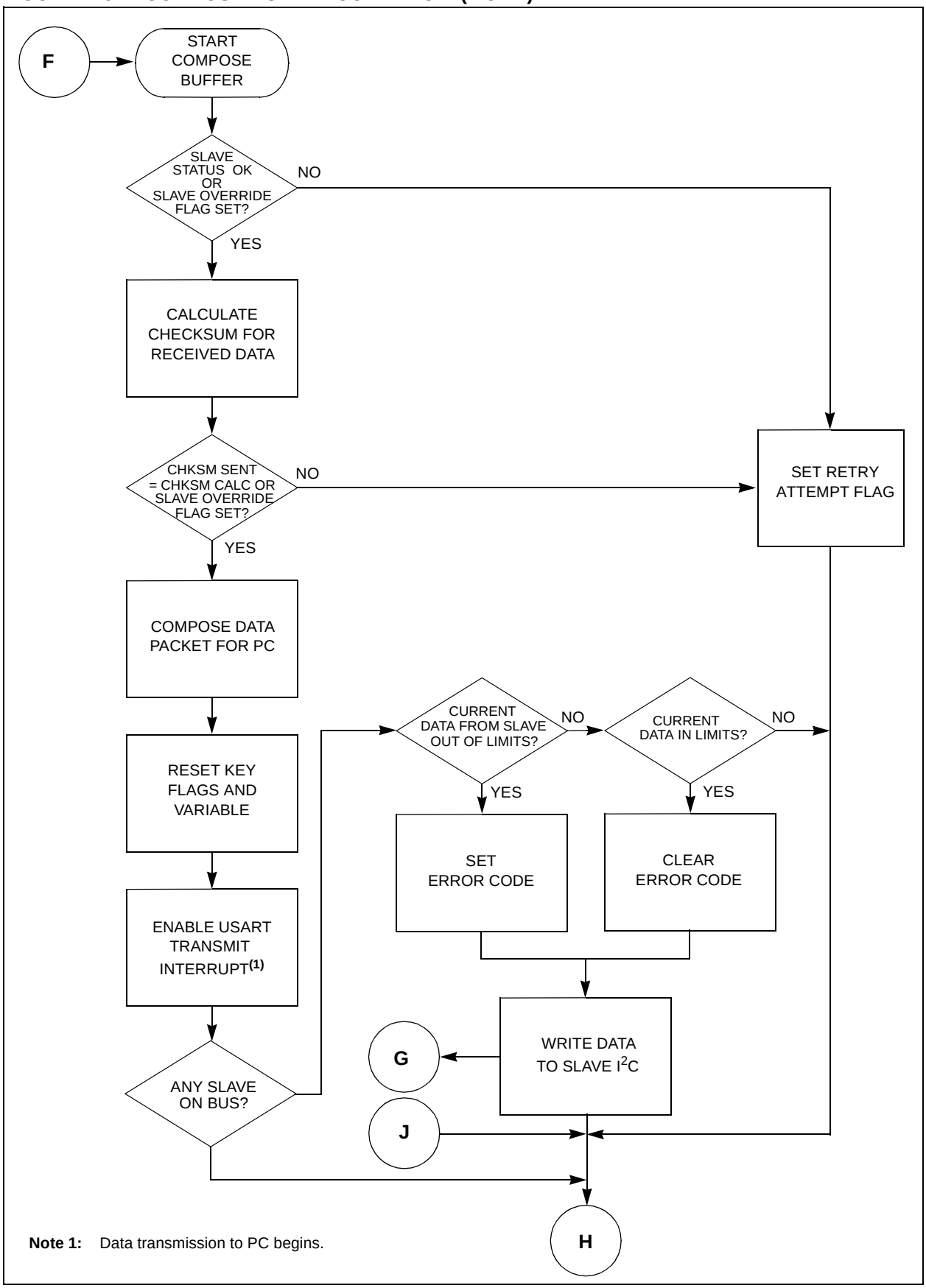


FIGURE A-5: COMPOSE BUFFER CODE FLOW (1 OF 2)



Note 1: Data transmission to PC begins.

FIGURE A-6: COMPOSE BUFFER CODE FLOW (2 OF 2)

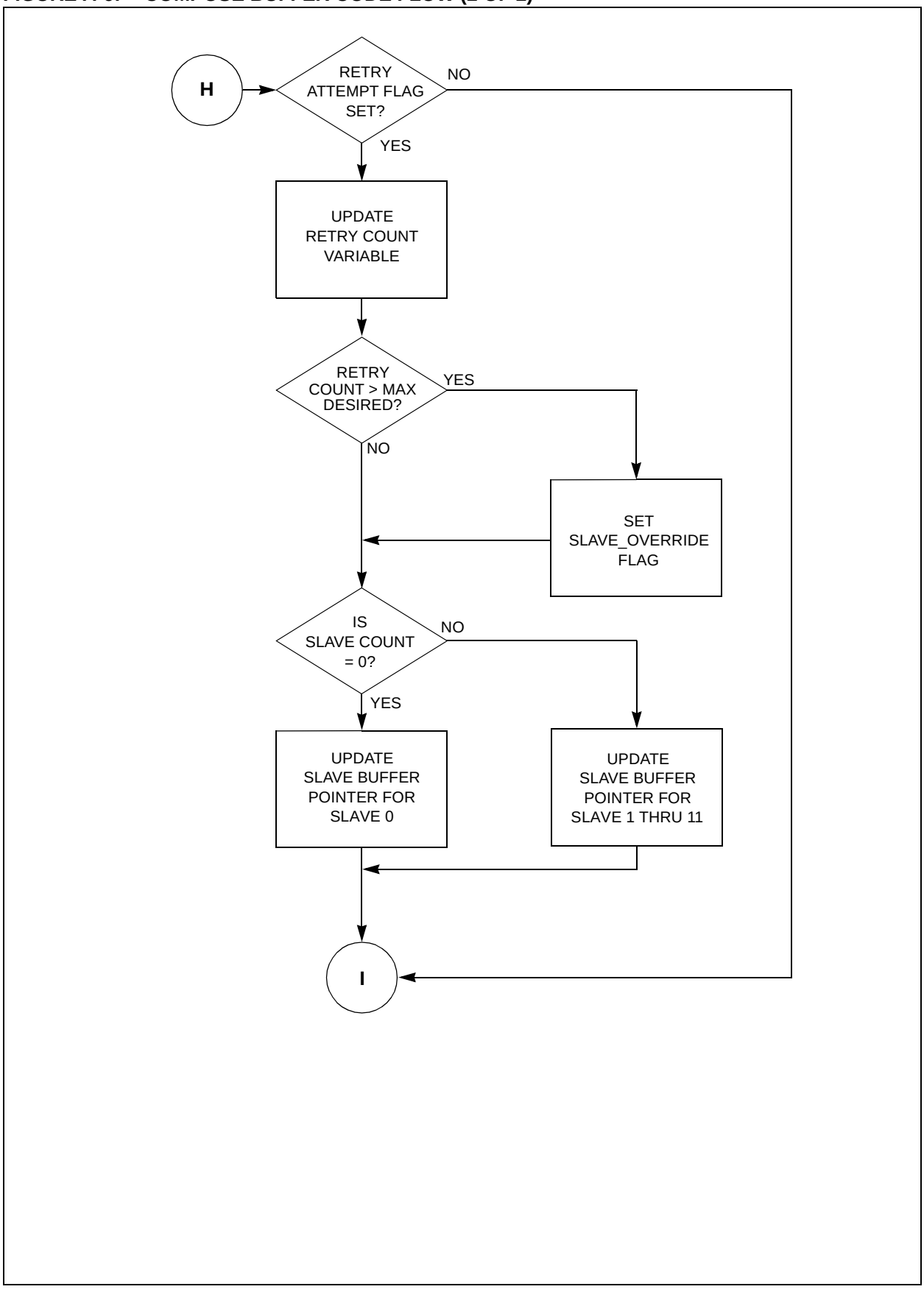
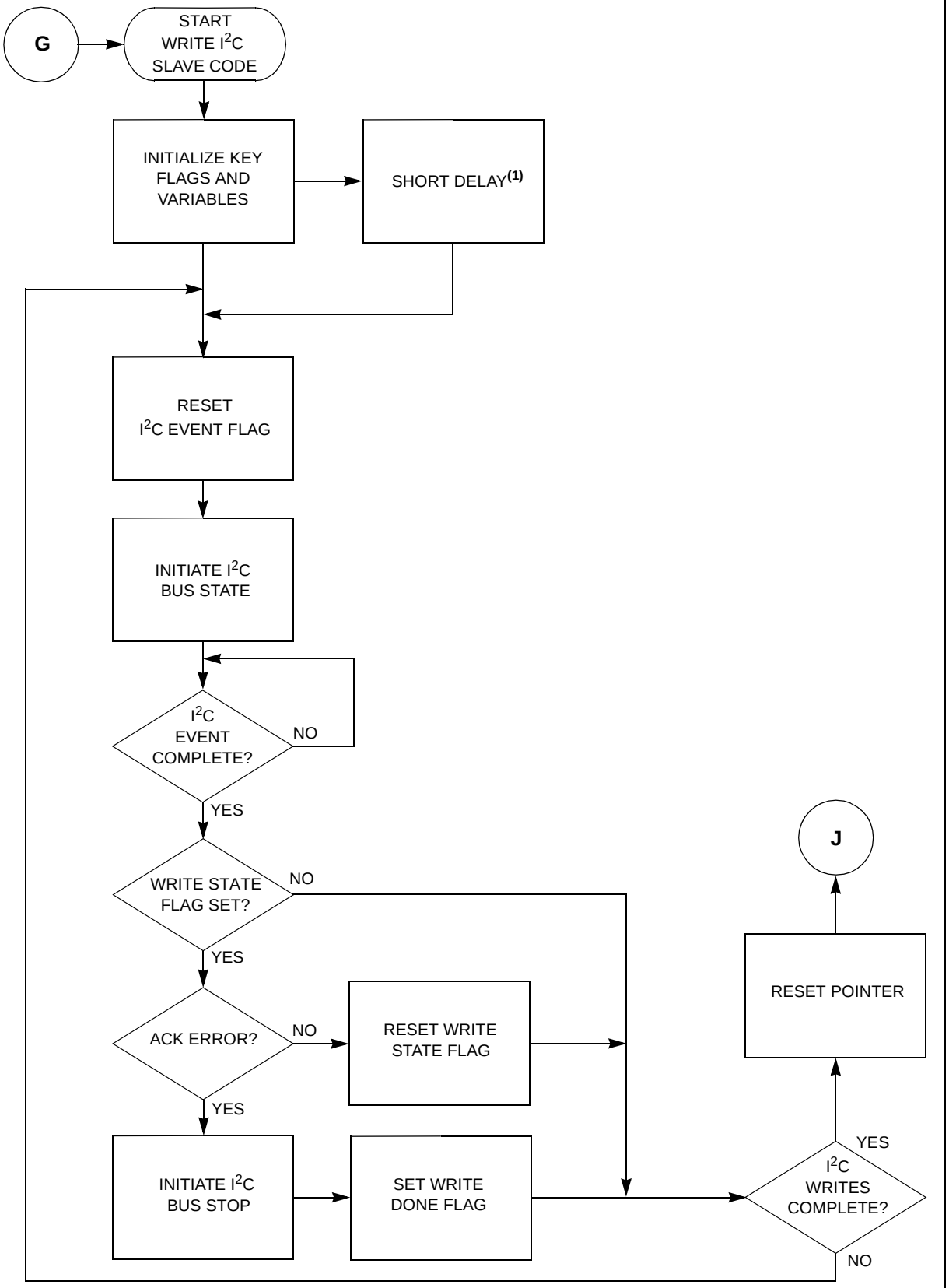


FIGURE A-7: I²C WRITE TO SLAVE CODE FLOW



Note 1: Delay function required when Slave FOSC < 8 MHz.

FIGURE A-8: I^2C BUS STATE EXECUTION CODE FLOW (1 OF 3)

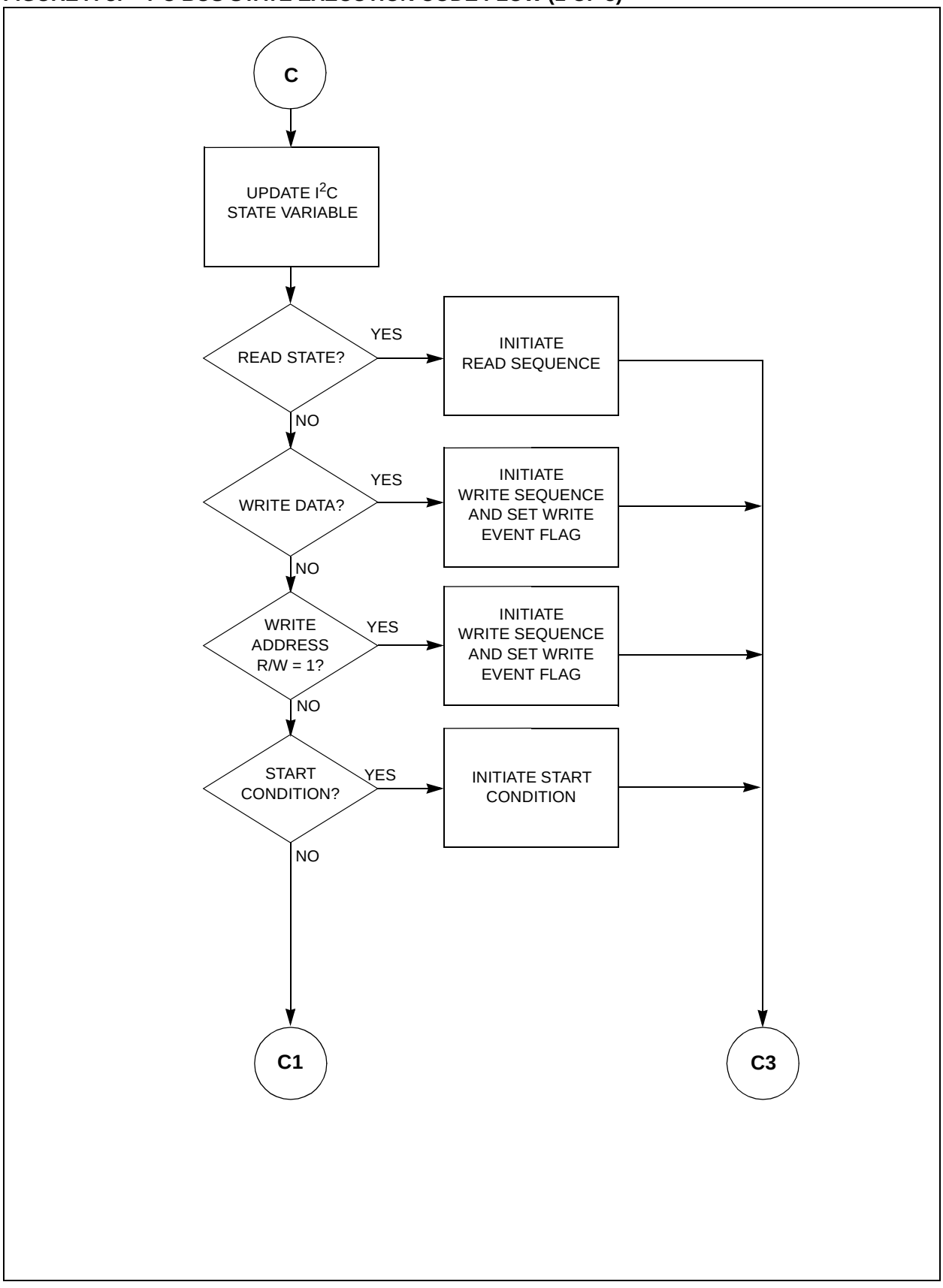


FIGURE A-9: I²C BUS STATE EXECUTION CODE FLOW (2 OF 3)

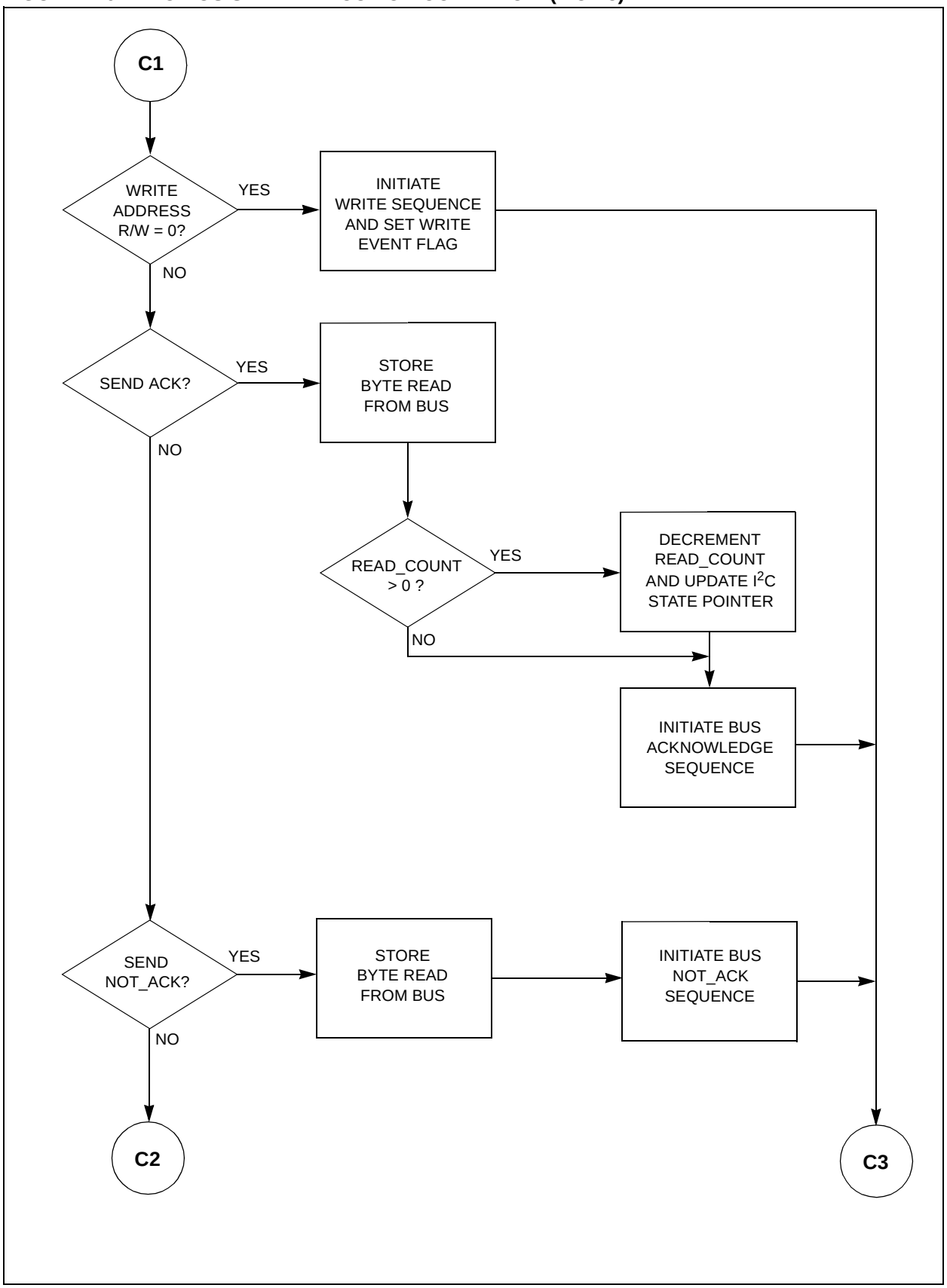
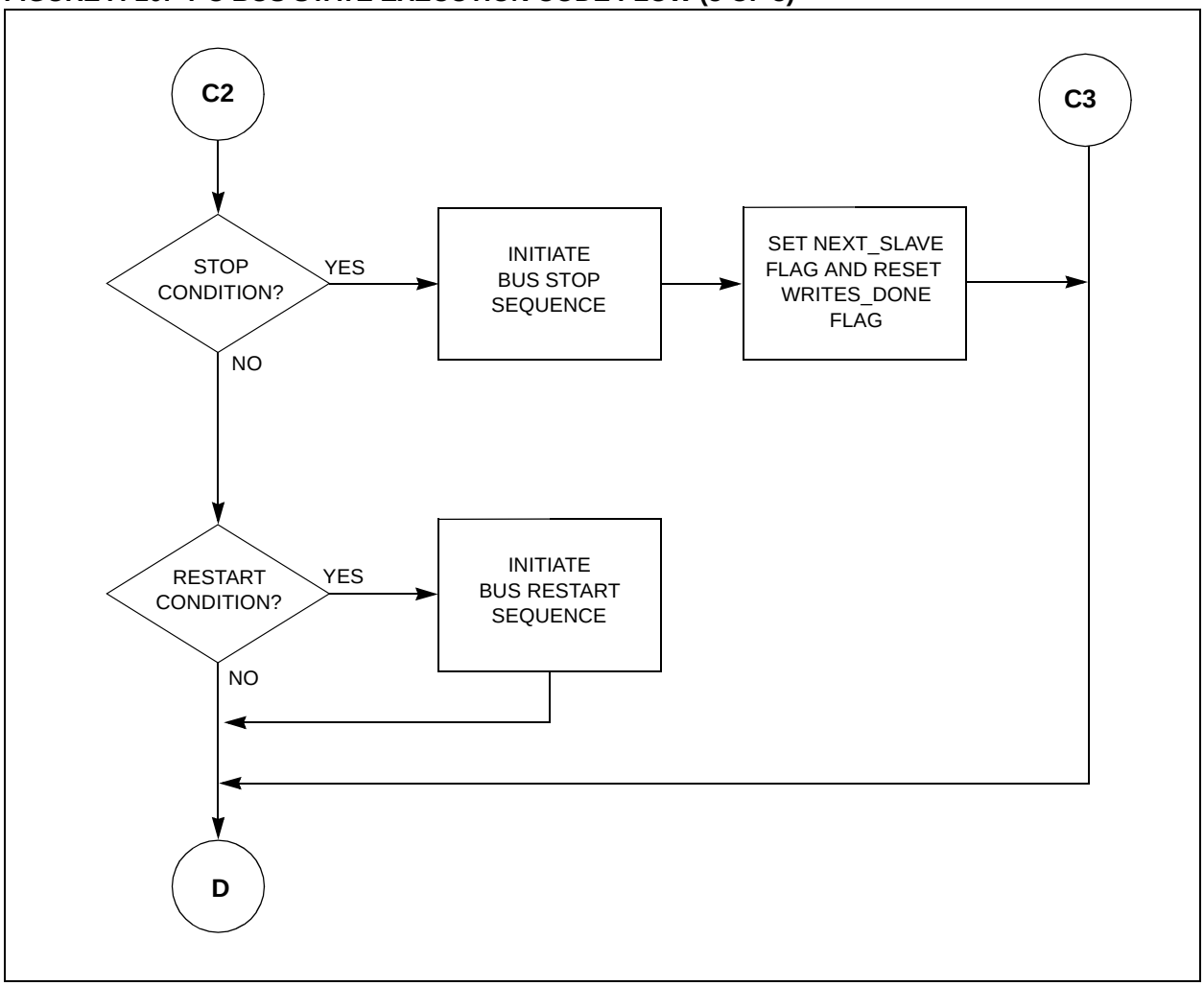


FIGURE A-10: I²C BUS STATE EXECUTION CODE FLOW (3 OF 3)



APPENDIX B: MASTER I²C SOURCE CODE (C LANGUAGE)

```
/************************************************************************
*                                                                       *
*           I2C Master and Slave Network using the PICmicro             *
*                                                                       *
************************************************************************
*                                                                       *
*   Filename:       mstri2c.c                                           *
*   Date:           06/09/2000                                          *
*   Revision:       1.00                                                *
*                                                                       *
*   Tools:          MPLAB 5.00.00                                       *
*                   Hi-Tech PIC C Compiler V7.85                        *
*                                                                       *
*   Author:         Richard L. Fischer                                  *
*   Company:        Microchip Technology Incorporated                   *
*                                                                       *
************************************************************************
*                                                                       *
*   System files required:                                              *
*                                                                       *
*                   mstri2c.c                                           *
*                   i2c_comm.c                                          *
*                   init.c                                              *
*                   delay.c      (Hi-Tech file)                         *
*                                                                       *
*                   pic.h        (Hi-Tech file)                         *
*                   delay.h      (Hi-Tech file)                         *
*                   mstri2c.h                                           *
*                   i2c_comm.h                                          *
*                   cnfig87x.h                                          *
*                                                                       *
************************************************************************
*                                                                       *
*   Notes:                                                              *
*                                                                       *
*   Device Fosc -> 16.00MHz                                             *
*   WDT -> on                                                           *
*   Brownout -> on                                                      *
*   Powerup timer -> on                                                 *
*   Code Protect -> off                                                 *
*                                                                       *
*   Interrupt sources -                                                 *
*                       1. USART based transmissions                    *
*                       2. I2C events (valid events)                    *
*                       3. I2C Bus Collision                            *
*                       4. Timer1 - 100mS intervals                     *
```

```
*                                                                     *
*                                                                     *
*   Memory Usage Map:                                                 *
*                                                                     *
*   Program ROM   $0000 - $00BC  $00BD ( 189) words                   *
*   Program ROM   $0587 - $07FF  $0279 ( 633) words                   *
*   Program ROM   $2007 - $2007  $0001 (   1) words                   *
*                                $0337 ( 823) words total Program ROM*
*                                                                     *
*   Bank 0 RAM    $0020 - $0075  $0056 (  86) bytes                   *
*   Bank 0 RAM    $007F - $007F  $0001 (   1) bytes                   *
*                                $0057 (  87) bytes total Bank 0 RAM *
*                                                                     *
*   Bank 1 RAM    $00A0 - $00AA  $000B (  11) bytes                   *
*   Bank 1 RAM    $00FF - $00FF  $0001 (   1) bytes                   *
*                                $000C (  12) bytes total Bank 1 RAM *
*                                                                     *
**********************************************************************/

   #include <pic.h>                    // processor if/def file
   #include "cnfig87x.h"               // configuration word definitions
   #include "mstri2c.h"
   #include "c:\ht-pic\samples\delay.h"


   __CONFIG    ( CONBLANK & CP_OFF & DEBUG_OFF & WRT_ENABLE_OFF & CPD_OFF &
                 LVP_OFF & BODEN_ON & PWRTE_ON & WDT_ON & HS_OSC );
```

```
/*********************************************************************
  MAIN PROGRAM BEGINS HERE
*********************************************************************/

void main(void)
{
/* Initialization is done here */
   Init_Usart();                        // initialize USART peripheral
   Init_Ports();                        // initialize Ports
   Init_Ssp();                          // initialize SSP module
   Init_Timer1();                       // initialize TMR1 peripheral

/* Interrupts are enabled here */
   TMR1IE = 1;                          // enable TMR1 Overflow interrupt
   BCLIE = 1;                           // enable bus collision interrupt
   SSPIE = 1;                           // enable I2C based interrupts

   sflag.status = 0x0000;               // ensure all event flags are reset
   eflag.status = 0x00;                 // ensure all event flags are reset

   PEIE = 1;                            // enable peripheral interrupts
   ei();                                // enable global interrupts

   for ( ;; )                           // infinite loop
   {
     CLRWDT();                          // reset WDT

//---------------------------------------------------------------------
//   Will execute these statements if Master wants to read from Slave I2C
//---------------------------------------------------------------------
     if ( sflag.event.read_i2c && sflag.event.i2c ) // test if read I2C is active
    {
      if ( !sflag.event.reads_done )    // test if more reads are needed
      {
        DelayUs( 400 );                 // short delay between events
                                        // to allow for slow FOSC on Slave
        Service_I2CSlave();             // Service I2C slave device(s)
      }

      if ( sflag.event.reads_done )     // test if that was last read
      {
        sflag.status &= 0x00F0;         // reset all I2C event flags
        eflag.status = 0x00;            // reset all error event flags
        TMR1ON = 1;                     // turn on Timer1 module
        TMR1IE = 1;                     // re-enable Timer1 interrupt
      }
    }
  }
}

//---------------------------------------------------------------------
//   Will evaluate these conditional statements on an interrupt basis
//---------------------------------------------------------------------
void interrupt piv( void )
{
  if ( SSPIE && SSPIF )                 // test for I2C event completion
  {
    SSPIF = 0;                          // reset I2C based interrupt flag
    sflag.event.i2c = 1;                // set I2C event service flag
    PORTB ^= 0b00000001;                // ***** test purposes only *****
```



```
  }

  else if ( BCLIE && BCLIF )           // test for bus collision
  {
    eflag.i2c.bus_coll_error = 1;      // set bus collision flag error
    sflag.event.i2c = 1;               // set I2C event service flag
    BCLIF = 0;                         // reset bus collision interrupt flag
    PORTB ^= 0b00000010;               // ***** test purposes only *****
  }

  else if ( TXIE && TXIF )             // test if USART based transmit interrupt
  {
    if ( index < MaxLength2PC )        // is all data sent ?
    {
      TXREG = ReadStatBufFromSlave[index];  // send another byte out
      index++;                         // increment array index
    }
    else
    {
      sflag.event.usart_tx = 0;        // reset USART TX in progress flag
      TXIE = 0;                        // disable transmit interrupt
      SSPIE = 1;                       // enable I2C based interrupts
      index = 0x00;                    // reset array index
    }
  }

  else if ( TMR1IE && TMR1IF )         // test for valid TMR1 interrupt
  {
    sflag.event.read_i2c = 1;          // set 100mS event service flag
    sflag.event.i2c = 1;               // set I2C event service flag
    PORTB &= 0b00001110;               // ***** test purposes only *****
    PORTB ^= 0b00001000;               // ***** test purposes only *****
    TMR1ON = 0;                        // turn off Timer1 module
    TMR1IF = 0;                        // reset TMR1 rollover interrupt flag
    TMR1IE = 0;                        // disable TMR1 based interrupt
    TMR1L += 0x60;                     // re-initialize TMR1 for
    TMR1H = 0x3C;                      // 100mS intervals
  }
}
```

```
/************************************************************************
*                                                                       *
*           I2C Master and Slave Network using the PICmicro             *
*                                                                       *
************************************************************************
*                                                                       *
*   Filename:       i2c_comm.c                                          *
*   Date:           06/09/2000                                          *
*   Revision:       1.00                                                *
*                                                                       *
*   Tools:          MPLAB 5.00.00                                       *
*                   Hi-Tech PIC C Compiler V7.85                        *
*                                                                       *
*   Author:         Richard L. Fischer                                  *
*   Company:        Microchip Technology Incorporated                   *
*                                                                       *
************************************************************************
*                                                                       *
*   Files required:                                                     *
*                                                                       *
*                   pic.h         (Hi-Tech file)                        *
*                   delay.h       (Hi-Tech file)                        *
*                   i2c_comm.h                                          *
*                                                                       *
************************************************************************
*                                                                       *
*   Notes: The routines within this file are for communicating          *
*          with the I2C Slave device(s).                                *
*                                                                       *
************************************************************************/

   #include <pic.h>                     // processor if/def file
   #include "i2c_comm.h"
   #include "c:\ht-pic\samples\delay.h"


#define LIMIT  0x80                     // limit value for slave data compare
```



```
/*********************************************************************
  MAIN PROGRAM BEGINS HERE
*********************************************************************/
void Service_I2CSlave( void )
{
//---------------------------------------------------------------------
//   Will execute these statements once per each round of slave reads.
//---------------------------------------------------------------------
 if ( !sflag.event.read_start )                 // execute once per entire rounds of
                                                // slave reads
 {
   sflag.event.read_start = 1;                  // set reads start flag
   index = 0x00;
   slave_count = 0x00;                          // reset running slave counter
   address_hold = SlaveAddress[slave_count];    // initialize address hold buffer
                                                // (1st slave)
   Write2Slave_Ptr = &WriteStatBuf2Slave[0];    // (bytecount, functional, checksum)
   read_count = OperChannelsPerSlave + 1;       // set byte read count
                                                // (bytes + 1/2checksum)
   ReadFSlave_Ptr = &ReadStatBufFromSlave[0];   // set up pointer for data read from
                                                // Slave
   I2CState_Ptr = &ReadFSlaveI2CStates[0];      // initialize I2C state pointer
 }

//---------------------------------------------------------------------
//  Will execute these statements if last I2C bus state was a WRITE
//---------------------------------------------------------------------
 if ( sflag.event.write_state )      // test if previous I2C state was a write
 {
   if ( ACKSTAT )                    // was NOT ACK received?
   {
     PEN = 1;                        // generate bus stop condition
     eflag.i2c.ack_error = 1;        // set acknowledge error flag
   }
   sflag.event.write_state = 0;      // reset write state flag
 }


//---------------------------------------------------------------------
// Will execute these statements if a bus collision or an acknowledge error
//---------------------------------------------------------------------
 if ( eflag.i2c.bus_coll_error || eflag.i2c.ack_error  )
 {
   sflag.event.read_loop = 0;         // reset read loop flag for any error
   sflag.event.next_i2cslave = 1;     // set flag indicating next slave

   temp.error = error_mask << slave_count; // compose error status word

  if ( eflag.i2c.bus_coll_error )    // test for bus collision error
  {
    eflag.i2c.bus_coll_error = 0;    // reset bus collision error flag
    bus.error_word |= temp.error;    // compose bus error status word
    SSPIF = 1;                       // set false interrupt to restart comm
  }
  if ( eflag.i2c.ack_error )         // test for acknowledge error
  {
    eflag.i2c.ack_error = 0;         // reset acknowledge error flag
    comm.error_word |= temp.error;   // compose communication error status word
  }
 }
```

```
 else                                  // else no error for this slave
 {
   temp.error = error_mask << slave_count; // compose error status word
   bus.error_word &= ~temp.error;      // reset bus error bit for this slave
   comm.error_word &= ~temp.error;     // reset comm error bit for this slave
 }

//---------------------------------------------------------------------
//  Will execute these statements for each new slave device after the first
//---------------------------------------------------------------------
 if ( sflag.event.next_i2cslave )      // if next slave is being requested
 {
   ComposeBuffer();                    // compose buffer for sending to PC
   sflag.event.next_i2cslave = 0;      // reset next slave status flag

   if ( sflag.event.usart_tx )         // test if USART TX still in progress
   {
     slave_count ++;                   // increment slave counter
     SSPIE = 0;                        // disable SSP based interrupt
     if ( !sflag.event.usart_tx )      // test if interrupt occurred while here
     {
       SSPIE = 1;                      // re-enable SSP based interrupt
     }
   }

  address_hold = SlaveAddress[slave_count]; // obtain slave address (repeat or next)
  read_count = OperChannelsPerSlave + 1;    // set byte read count (bytes, 1/2checksum)
  ReadFSlave_Ptr = &ReadStatBufFromSlave[0];// set up pointer for data read from Slave
  I2CState_Ptr = &ReadFSlaveI2CStates[0];   // re-initialize I2C state pointer
}

//---------------------------------------------------------------------
//  Test if all slaves have been communicated with or continue with next bus state
//---------------------------------------------------------------------
 if ( slave_count < OperNumberI2CSlaves )  // test if all slaves have not been accessed
 {
   sflag.event.i2c = 0;                    // reset I2C state event flag
   I2CBusState();                          // execute next I2C state
 }

 else                                      // else
 {
   sflag.event.reads_done = 1;             // set flag indicating all slaves are read
 }
}


//---------------------------------------------------------------------
//  Will execute this switch/case evaluation for next I2C bus state
//---------------------------------------------------------------------
void I2CBusState ( void )
{
  i2cstate = *I2CState_Ptr++;        // retrieve next I2C state
  switch ( i2cstate )                // evaluate which I2C state to execute
  {
    case ( READ ):                   // test for I2C read
      RCEN = 1;                      // initiate i2C read state
      break;
```



```
    case ( WRITE_DATA ):                // test for I2C write (DATA)
      SSPBUF = *Write2Slave_Ptr++;      // initiate I2C write state
      sflag.event.write_state = 1;      // set flag indicating write event in action
      break;

    case ( WRITE_ADDRESS1 ):            // test for I2C address (R/W=1)
      SSPBUF = address_hold + 1;        // initiate I2C address write state
      sflag.event.write_state = 1;      // set flag indicating write event in action
      break;

    case ( START ):                     // test for I2C start state
      SEN = 1;                          // initiate I2C bus start state
      break;

    case ( WRITE_ADDRESS0 ):            // test for I2C address (R/W=0)
      SSPBUF = address_hold;            // initiate I2C address write state
      sflag.event.write_state = 1;      // set flag indicating write event in action
     break;

    case ( SEND_ACK ):                  // test for send acknowledge state
      *ReadFSlave_Ptr++ = SSPBUF;       // save off byte
      if ( read_count > 0)              // test if still in read loop
      {
        read_count -= 1;                // reduce read count
        I2CState_Ptr -= 2;              // update state pointer
      }
      ACKDT = 0;                        // set acknowledge data state (true)
      ACKEN = 1;                        // initiate acknowledge state
      break;

    case ( SEND_NACK ):                 // test if sending NOT acknowledge state
      *ReadFSlave_Ptr = SSPBUF;         // save off byte
      ACKDT = 1;                        // set acknowledge data state (false)
      ACKEN = 1;                        // initiate acknowledge sequence
      break;

    case ( STOP ):                      // test for stop state
      PEN = 1;                          // initiate I2C bus stop state
      sflag.event.next_i2cslave = 1;    // set flag indicating next slave
      sflag.event.writes_done = 1;      // reset flag, write is done
      break;

    case ( RESTART ):                   // test for restart state
      RSEN = 1;                         // initiate I2C bus restart state
      break;

    default:                            //
      break;
  }
}


//---------------------------------------------------------------------
//  Compose Buffer to transmit to PC and Slave I2C (slave I2C if overlimit)
//---------------------------------------------------------------------
void ComposeBuffer( void )
{
  if ( ( ReadStatBufFromSlave[0] & 0x80 )  ||  ( eflag.i2c.slave_override ) )
  {
    checksum.word = Calc_Checksum( &ReadStatBufFromSlave[0], 4 );
```

```
    temp.hold.lobyte = ReadStatBufFromSlave[4];
    temp.hold.hibyte = ReadStatBufFromSlave[5];

    if ( ( ( checksum.word + temp.checksum ) == 0 )  ||  ( eflag.i2c.slave_override ) )
    {
      ReadStatBufFromSlave[6] = bus.error.hibyte;      //
      ReadStatBufFromSlave[7] = bus.error.lobyte;      //
      ReadStatBufFromSlave[8] = comm.error.hibyte;     //
      ReadStatBufFromSlave[9] = comm.error.lobyte;     //

      if ( eflag.i2c.slave_override )   // test if comm failed with Slave
      {
        ReadStatBufFromSlave[5] = 0x00; // null out voltage data
        ReadStatBufFromSlave[4] = 0x00; // null out rpm data
        ReadStatBufFromSlave[3] = 0x00; // null out temperature data
      }
      else                              // else comm with Slave OK
      {
        ReadStatBufFromSlave[5] = ReadStatBufFromSlave[3];    // voltage data
        ReadStatBufFromSlave[4] = ReadStatBufFromSlave[2];    // rpm data
        ReadStatBufFromSlave[3] = ReadStatBufFromSlave[1];    // temperature data
      }

      ReadStatBufFromSlave[2] = ( slave_count + 1 );          // slave ID
      ReadStatBufFromSlave[1] = 0x55;  // start sync character 2
      ReadStatBufFromSlave[0] = 0xAA;  // start sync character 1

      sflag.event.usart_tx = 1;        // set flag indicating USART TX in progress
      TXIE = 1;                        // enable USART TX interrupts

     if ( comm.error_word & 0x0FFF )   // test if any slave is on the bus
     {
       if ( (ReadStatBufFromSlave[5] >= LIMIT)  &&  ( !eflag.i2c.slave_override ) )
       {
         WriteData2Slave[3] = 0x01;    // out of limits indicator to slave
         Write_I2CSlave();             // write "error" code to slave
       }
       else if ( (ReadStatBufFromSlave[5] < LIMIT)  &&  ( !eflag.i2c.slave_override ) )
       {
         WriteData2Slave[3] = 0x00;    // in limits indicator to slave
         Write_I2CSlave();             // write "valid" code to slave
       }
     }

     eflag.i2c.slave_override = 0;     // reset slave override flag
     read_retry = 0x00;                // reset retry count
     eflag.i2c.retry_attempt = 0;      // reset retry communication flag
    }
    else
    {
      eflag.i2c.retry_attempt = 1;     // set retry communications flag
    }
  }
  else
  {
    eflag.i2c.retry_attempt = 1;       // set retry communications flag
  }

  if ( eflag.i2c.retry_attempt )       // test if there was a retry request
  {
```



```
    read_retry ++;                          // update retry counter
    if ( read_retry > MaxSlaveRetry -1 )    // test if all retries have been attempted
    {
        eflag.i2c.slave_override = 1;       // set flag to process next packet no matter
                                            // what
    }
    if ( slave_count == 0 )                 // test for first slave
    {
      Write2Slave_Ptr = &WriteStatBuf2Slave[0];                 // reinitialize pointer
    }
    else                                    // else slave 1 -> X
    {
       Write2Slave_Ptr = &WriteStatBuf2Slave[slave_count * 3];  // reinitialize pointer
    }
  }
}


//---------------------------------------------------------------------------
//  Will execute these statements when requiring to write to a Slave I2C device
//---------------------------------------------------------------------------
void Write_I2CSlave( void )
{
   unsigned char temp_ptr;              // define auto variable
   sflag.event.writes_done = 0;         // ensure flag is reset
   temp_ptr = Write2Slave_Ptr;          // save off current write pointer
   I2CState_Ptr = Write2SlaveStates;    // initialize I2C state pointer for writes
   ReadFSlave_Ptr = &ReadStatBufFromSlave[0];

   WriteData2Slave[0] = address_hold;   // obtain slave address
   WriteData2Slave[1] = 0x01;           // byte number request
   WriteData2Slave[2] = 0x00;           // functional offset
   checksum.word = Calc_Checksum( &WriteData2Slave[0], 4 );
   checksum.word = ~checksum.word + 1;
   WriteData2Slave[4] = (unsigned char)checksum.word; // save off checksum to array
   Write2Slave_Ptr = &WriteData2Slave[1];             // initialize pointer

   do
   {
     DelayUs( 400 );                    // delay between events
     sflag.event.i2c = 0;               // reset I2C state event flag
     I2CBusState();                     // execute next I2C state
     while ( !sflag.event.i2c );        // wait here until event completes
     if ( sflag.event.write_state )     // test if previous I2C state was a write
     {
       if ( ACKSTAT )                   // was NOT ACK received?
       {
         PEN = 1;                       // generate bus stop condition
         sflag.event.writes_done = 1;   // set write done flag do to error
       }
       sflag.event.write_state = 0;     // reset write state flag
     }
   } while( !sflag.event.writes_done );  // stay in loop until error or done

   PORTB ^= 0b00000100;                 // ***** test purposes only *****
   Write2Slave_Ptr = temp_ptr ;         // restore pointer contents
}
```

```
//------------------------------------------------------------------------
//  Generic checksum calculation routine
//------------------------------------------------------------------------
unsigned int Calc_Checksum( unsigned char *ptr, unsigned char length )
{
   unsigned int checksum;               // define auto type variable
   checksum = 0x0000;                   // reset checksum word
   while ( length )                     // while data string length != 0
   {
     checksum += *ptr++;                // generate checksum
     length --;                         // decrement data string length
   }
   return ( checksum );                 // return additive checksum
}
```



```
/*********************************************************************
*                                                                    *
*           I2C Master and Slave Network using the PICmicro          *
*                                                                    *
*********************************************************************
*                                                                    *
*   Filename:       init.c                                           *
*   Date:           06/09/2000                                       *
*   Revision:       1.00                                             *
*                                                                    *
*   Tools:          MPLAB 5.00.00                                    *
*                   Hi-Tech PIC C Compiler V7.85                     *
*                                                                    *
*   Author:         Richard L. Fischer                               *
*   Company:        Microchip Technology Incorporated                *
*                                                                    *
*********************************************************************
*                                                                    *
*   Files required:                                                  *
*                                                                    *
*                   pic.h                                            *
*                                                                    *
*********************************************************************
*                                                                    *
*   Notes: The routines within this file are required for            *
*          initializing the PICmicro peripherals and Ports.          *
*                                                                    *
*                                                                    *
*********************************************************************/

   #include <pic.h>                    // processor if/def file


 #define FOSC          (16000000L)     // define external clock frequency
 #define baud          19200           // define USART baud rate
 #define i2c_bus_rate (400000L)        // define I2C bus rate


/*********************************************************************
*********************************************************************/
void Init_Ports( void )
{
  OPTION &= 0b01111110;                //
  PORTA = 0b000000;                    // set default pin drive states
  PORTB = 0b00000000;                  // set default pin drive states
  PORTC = 0b00000000;                  // set default pin drive states

  ADCON1 = 0b00000110;                 // ensure PortA is digital
  TRISA = 0b000000;                    // set PORTA as outputs
  TRISB = 0b11110000;                  // RB0-RB3 outputs, RB4-RB7 inputs
  TRISC = 0b11011000;                  //
}

void Init_Usart( void )
{
   unsigned long temp;                 // define auto type long variable
/* calculate and set baud rate register for asynchronous mode */
   temp = 16UL * baud;
```

```
   TRISC |= 0b11000000;                // ensure Rx and Tx are inputs (default)
   SPBRG = (int)(FOSC/temp) - 1;       // 9600 baud @ 16MHz
   TXSTA = 0b00100100;                 // enable Transmitter, BRGH = 1
   RCREG;                              // dummy read
   RCSTA = 0b10010000;                 // continuous receive, serial port enabled
}

void Init_Ssp( void )
{
   TRISC |= 0b00011000;                 // ensure SDI and SD0 are inputs
   SSPIF = 0;                           // reset I2C based interrupt flag
   SSPCON2 = 0b00000000;                // ensure all state bits are reset
   SSPSTAT = 0b00000000;                //
   SSPADD = (( FOSC / (4 * i2c_bus_rate) )) - 1;  // initialize i2c bus rate
   SSPCON = 0b00111000;                 // Master I2C mode
}

void Init_Timer1( void )                // set for 100mS intervals
{
   T1CON = 0b00110000;                  // 1:8 Prescale, T1OSCEN shut-off
   TMR1L = 0x60;                        // initialize TMR1 for
   TMR1H = 0x3C;                        // 100 mS intervals
   TMR1IF = 0;                          // reset Timer1 overflow flag
   TMR1ON = 1;                          // turn on Timer1 module
}
```



```
/************************************************************************
*                                                                       *
*   Filename:       mstri2c.h                                           *
*   Date:           06/09/2000                                          *
*   Revision:       1.00                                                *
*                                                                       *
*   Tools:          MPLAB 5.00.00                                       *
*                   Hi-Tech PIC C Compiler V7.85                        *
*                                                                       *
************************************************************************/



 #define MaxNumberI2CSlaves    12      // maximum number of I2C slave devices
 #define OperNumberI2CSlaves   12      // operational number of I2C slaves

 #define MaxChannelsPerSlave   12      // maximum channels of data per slave
 #define OperChannelsPerSlave   3      // operational number of channels of data

 #define MaxLength2PC  10

// FUNCTION PROTOTYPES

/* Functions defined in init.c file  */
extern void Init_Ports( void );
extern void Init_Ssp( void );
extern void Init_Usart( void );
extern void Init_Timer1( void );

/* Functions defined in i2c_comm.c file  */
extern void Service_I2CSlave( void );


// VARIABLES ( DECLARED HERE )

/* Variables defined in i2c_comm.c file  */
extern unsigned char read_count;
extern unsigned char index;

extern bank1 unsigned char ReadStatBufFromSlave[OperChannelsPerSlave+8];
extern const unsigned char *I2CState_Ptr;


/* Variables defined in i2c_comm.c file  */
extern unsigned char slave_count;



// VARIABLES ( DEFINED HERE )

union events {
unsigned int status;                    // entire status word
struct bit_events {                     // structure with 8 bits
         unsigned int usart_tx      :1;  // flag indicating USART transmit event
         unsigned int               :1;  //
         unsigned int i2c           :1;  // flag indicating I2C event
         unsigned int               :1;  //
         unsigned int               :1;  //
         unsigned int write_state   :1;  // flag indicating write state entered
```

```
        unsigned int writes_done   :1;  // flag indicating that write state done
        unsigned int               :1;  //
        unsigned int               :1;  //
        unsigned int               :1;  //
        unsigned int next_i2cslave :1;  // flag indicating service next slave
        unsigned int read_loop     :1;  // flag indicating read loop in progress
        unsigned int read_start    :1;  // flag indicating read state entered
        unsigned int reads_done    :1;  // flag indicating that read state is done
        unsigned int read_i2c      :1;  // flag indicating I2C read state
        unsigned int               :1;  //
    } event;
} sflag;


union i2c_error_events {
    unsigned char status;
    struct error_events {                // structure with 16 bits
        unsigned int slave_override :1;  // flag indicating
        unsigned int retry_attempt  :1;  // flag indicating to retry I2C comm
        unsigned int                :1;  //
        unsigned int                :1;  //
        unsigned int                :1;  //
        unsigned int overlimit      :1;  // flag indicating byte is out-of-range
        unsigned int ack_error      :1;  // flag indicating acknowledge error
        unsigned int bus_coll_error :1;  // flag indicating bus collision error
    } i2c;
} eflag;
```



```
/************************************************************************
*                                                                       *
*   Filename:       i2c_comm.h                                          *
*   Date:           06/09/2000                                          *
*   Revision:       1.00                                                *
*                                                                       *
*   Tools:          MPLAB 5.00.00                                       *
*                   Hi-Tech PIC C Compiler V7.85                        *
*                                                                       *
************************************************************************/


 #define MaxNumberI2CSlaves    12     // maximum number of I2C slave devices
 #define OperNumberI2CSlaves   12     // operational number of I2C slaves

 #define MaxChannelsPerSlave   10     // maximum channels of data per slave
 #define OperChannelsPerSlave   3     // operational number of channels of data

 #define  MaxSlaveRetry   1           // default is one retry

#define  error_mask  0b0000000000000001

#define COMM_STAT  0
#define STATUS0    1
#define STATUS1    2

#define TEMP0      3
#define ADRES0     4
#define ADRES1     5
#define ADRES2     6
#define ADRES3     7

#define TACH0      8
#define TACH1      9
#define TACH2     10
#define TACH3     11
#define MAX_CHNNL TACH3



// FUNCTION PROTOTYPES

unsigned int Calc_Checksum( unsigned char *ptr, unsigned char length );
void I2CBusState( void );
void Write_I2CSlave( void );
void ComposeBuffer( void );
void I2CBusState ( void );


// VARIABLES ( DEFINED HERE )

unsigned char slave_count, read_count;
unsigned char address_hold, address_offset;
unsigned char read_retry;
unsigned char write_count;

unsigned char I2CWriteState;
unsigned char index;
```

```
unsigned char i2cstate;
unsigned char *Write2Slave_Ptr;

bank1 unsigned char ReadStatBufFromSlave[OperChannelsPerSlave+8];
bank1 unsigned char *ReadFSlave_Ptr;

unsigned char WriteData2Slave[MaxChannelsPerSlave];


// FORMAT -> byte request count, functional code, checksum (repeats per each slave)
unsigned char WriteStatBuf2Slave[MaxNumberI2CSlaves * 3] =
                       { 0x83,TEMP0,0x78, 0x83,TEMP0,0x76, 0x83,TEMP0,0x74,
                         0x83,TEMP0,0x72, 0x83,TEMP0,0x70, 0x83,TEMP0,0x6E,
                         0x83,TEMP0,0x6C, 0x83,TEMP0,0x6A, 0x83,TEMP0,0x68,
                         0x83,TEMP0,0x66, 0x83,TEMP0,0x64, 0x83,TEMP0,0x62 };

union {
    unsigned int error;
    unsigned int checksum;
    struct {
       unsigned char lobyte;
       unsigned char hibyte;
    } hold;
} temp;


union {
    unsigned int error_word;
    struct {
       unsigned char lobyte;
       unsigned char hibyte;
    } error;
} bus;


union {
    unsigned int error_word;
    struct {
       unsigned char lobyte;
       unsigned char hibyte;
    } error;
} comm;


union {
    unsigned int word;
    struct {
       unsigned char low;
       unsigned char high;
    } byte;
} checksum;


// VARIABLES ( DECLARED HERE / REFERENCE LINKAGE )

extern union events {
    unsigned int status;              // entire status word
    struct bit_events {               // structure with 8 bits
       unsigned int usart_tx     :1;  // flag indicating USART transmit event
       unsigned int              :1;  //
```



```
       unsigned int i2c           :1;  // flag indicating I2C event
       unsigned int               :1;  //
       unsigned int               :1;  //
       unsigned int write_state   :1;  // flag indicating write state entered
       unsigned int writes_done   :1;  // flag indicating that write state done
       unsigned int               :1;  //
       unsigned int               :1;  //
       unsigned int               :1;  //
       unsigned int next_i2cslave :1;  // flag indicating service next slave
       unsigned int read_loop     :1;  // flag indicating read loop in progress
       unsigned int read_start    :1;  // flag indicating read state entered
       unsigned int reads_done    :1;  // flag indicating that read state is done
       unsigned int read_i2c      :1;  // flag indicating I2C read state
       unsigned int               :1;  //
     } event;
} sflag;


extern union i2c_error_events {
     unsigned char status;
     struct error_events {               // structure with 16 bits
         unsigned int slave_override :1; // flag indicating
         unsigned int retry_attempt  :1; // flag indicating to retry I2C comm
         unsigned int                :1; //
         unsigned int                :1; //
         unsigned int                :1; //
         unsigned int overlimit      :1; // flag indicating byte is out-of-range
         unsigned int ack_error      :1; // flag indicating acknowledge error
         unsigned int bus_coll_error :1; // flag indicating bus collision error
    } i2c;
} eflag;


// define I2C bus states
enum i2c_bus_states{ START =1, RESTART =2, STOP =3, SEND_ACK =4, SEND_NACK =5,
                     GEN_CALL =6, READ =7, WRITE_DATA =8, WRITE_ADDRESS1 =9,
                     WRITE_ADDRESS0 =10 };

const unsigned char ReadFSlaveI2CStates[] = {1,10,8,8,8,2,9,7,4,7,5,3,0};
                                             //Pad with null state
const unsigned char Write2SlaveStates[] = {1,10,8,8,8,8,2,9,7,5,3,0};
                                             //Pad with null state

const unsigned char *I2CState_Ptr;        // define pointer for accessing I2C states


// Slave Address defined here ( base  >   1,2,3,4, 5, 6, 7, 8, 9,10,11,12
const unsigned char SlaveAddress[MaxNumberI2CSlaves+1] =
                                          {2,4,6,8,10,12,14,16,18,20,22,24,0};
```

```
/************************************************************************
*                                                                       *
*   Filename:       cnfig87x.h                                          *
*   Date:           06/09/2000                                          *
*   File Version:   1.00                                                *
*                                                                       *
*   Compiler:       Hi-Tech PIC C Compiler V7.85                        *
*                                                                       *
************************************************************************/

/*****CONFIGURATION BIT DEFINITIONS FOR PIC16F87X PICmicro  *****/


 #define    CONBLANK        0x3FFF

#define     CP_ALL          0x0FCF
#define     CP_HALF         0x1FDF
#define     CP_UPPER_256    0x2FEF
#define     CP_OFF          0x3FFF
#define     DEBUG_ON        0x37FF
#define     DEBUG_OFF       0x3FFF
#define     WRT_ENABLE_ON   0x3FFF
#define     WRT_ENABLE_OFF  0x3DFF
#define     CPD_ON          0x3EFF
#define     CPD_OFF         0x3FFF
#define     LVP_ON          0x3FFF
#define     LVP_OFF         0x3F7F
#define     BODEN_ON        0x3FFF
#define     BODEN_OFF       0x3FBF
#define     PWRTE_OFF       0x3FFF
#define     PWRTE_ON        0x3FF7
#define     WDT_ON          0x3FFF
#define     WDT_OFF         0x3FFB
#define     LP_OSC          0x3FFC
#define     XT_OSC          0x3FFD
#define     HS_OSC          0x3FFE
#define     RC_OSC          0x3FFF
```


APPENDIX C: SLAVE I²C FLOW CHARTS

FIGURE C-1: `SSP_HANDLER()` FLOW CHART, CASE DETECTION

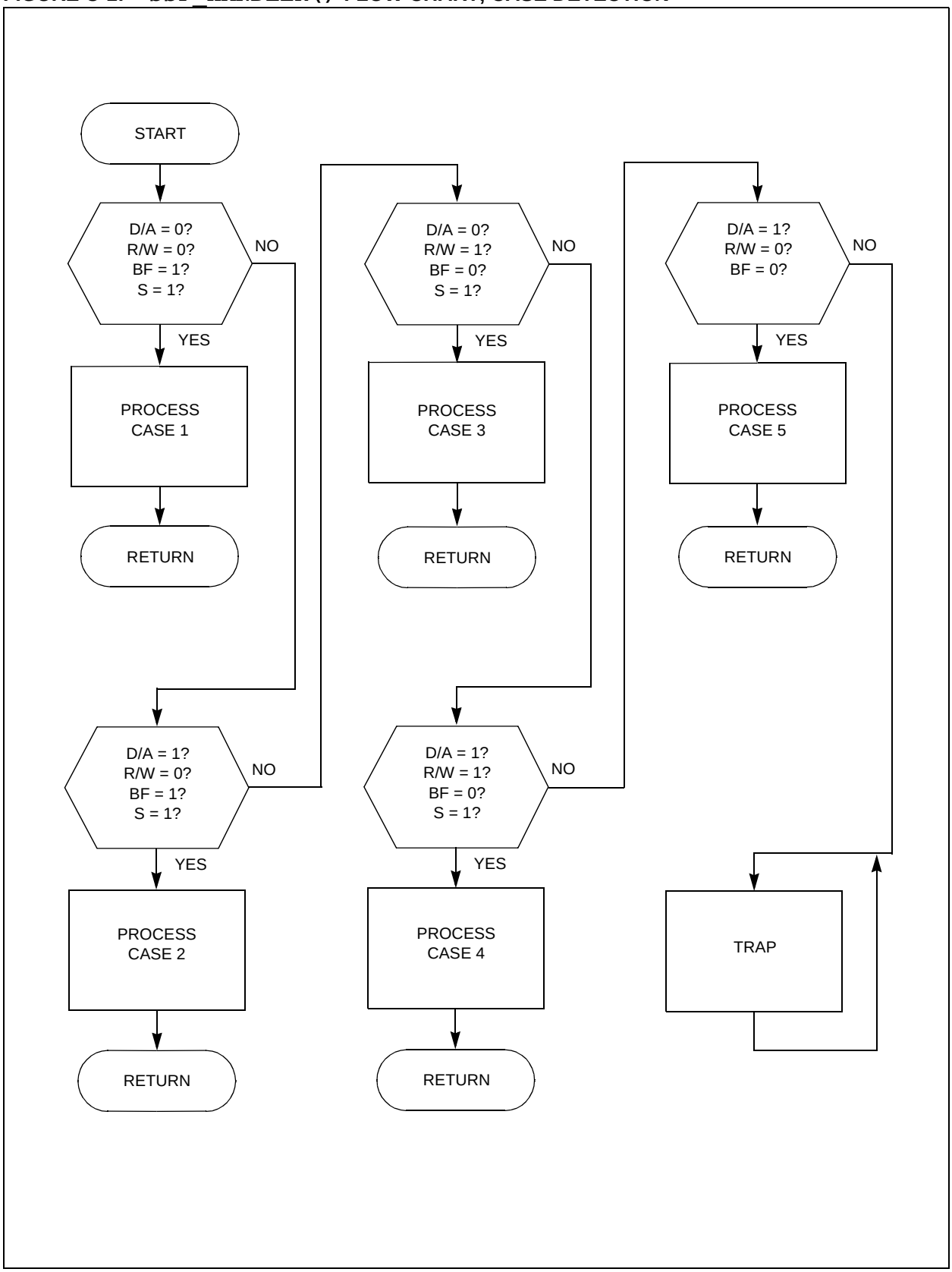


FIGURE C-2: `SSP_HANDLER()` – CASE 1

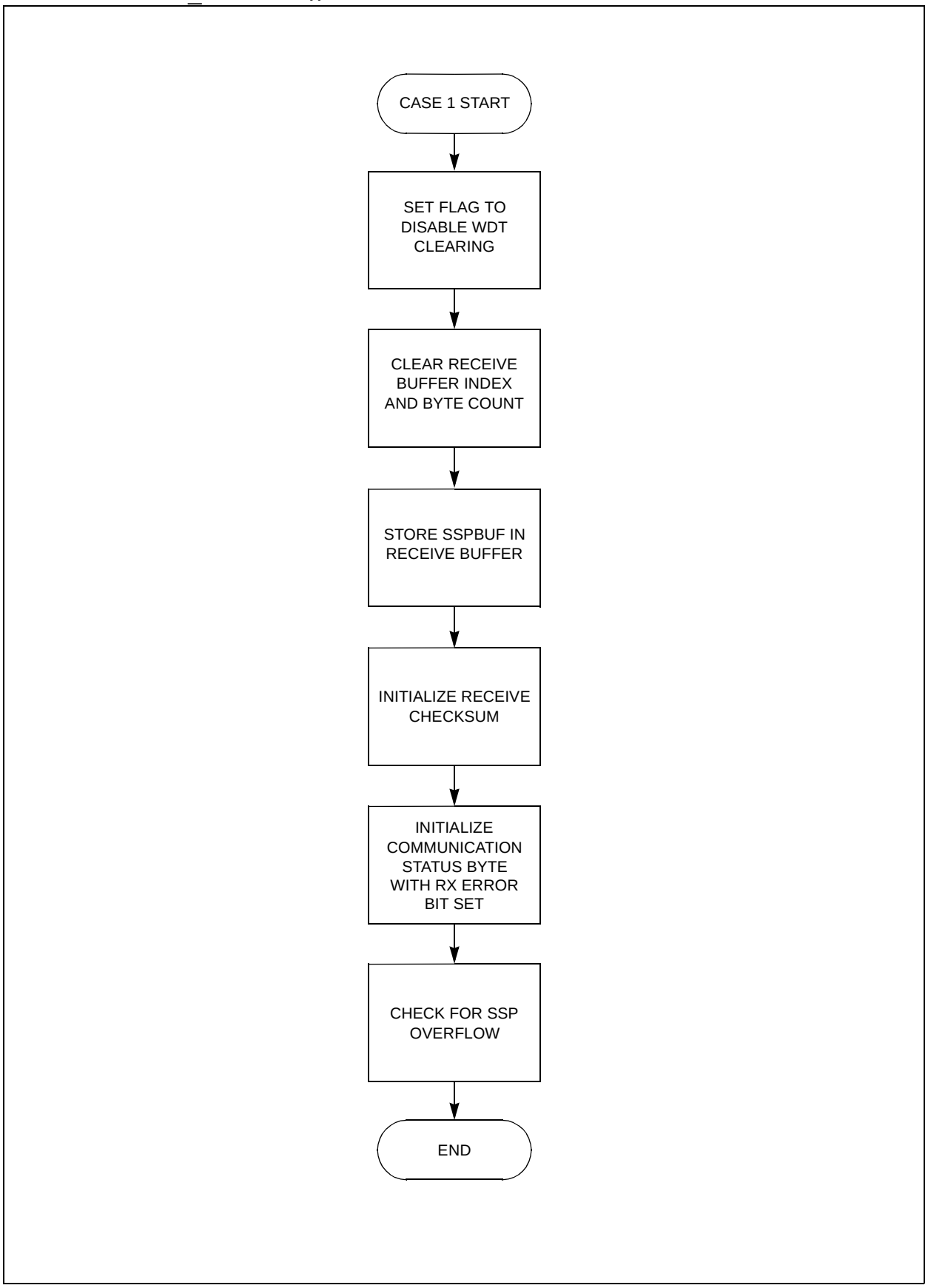


FIGURE C-3: `SSP_HANDLER()` – CASE 2 (SHEET 1 OF 2)

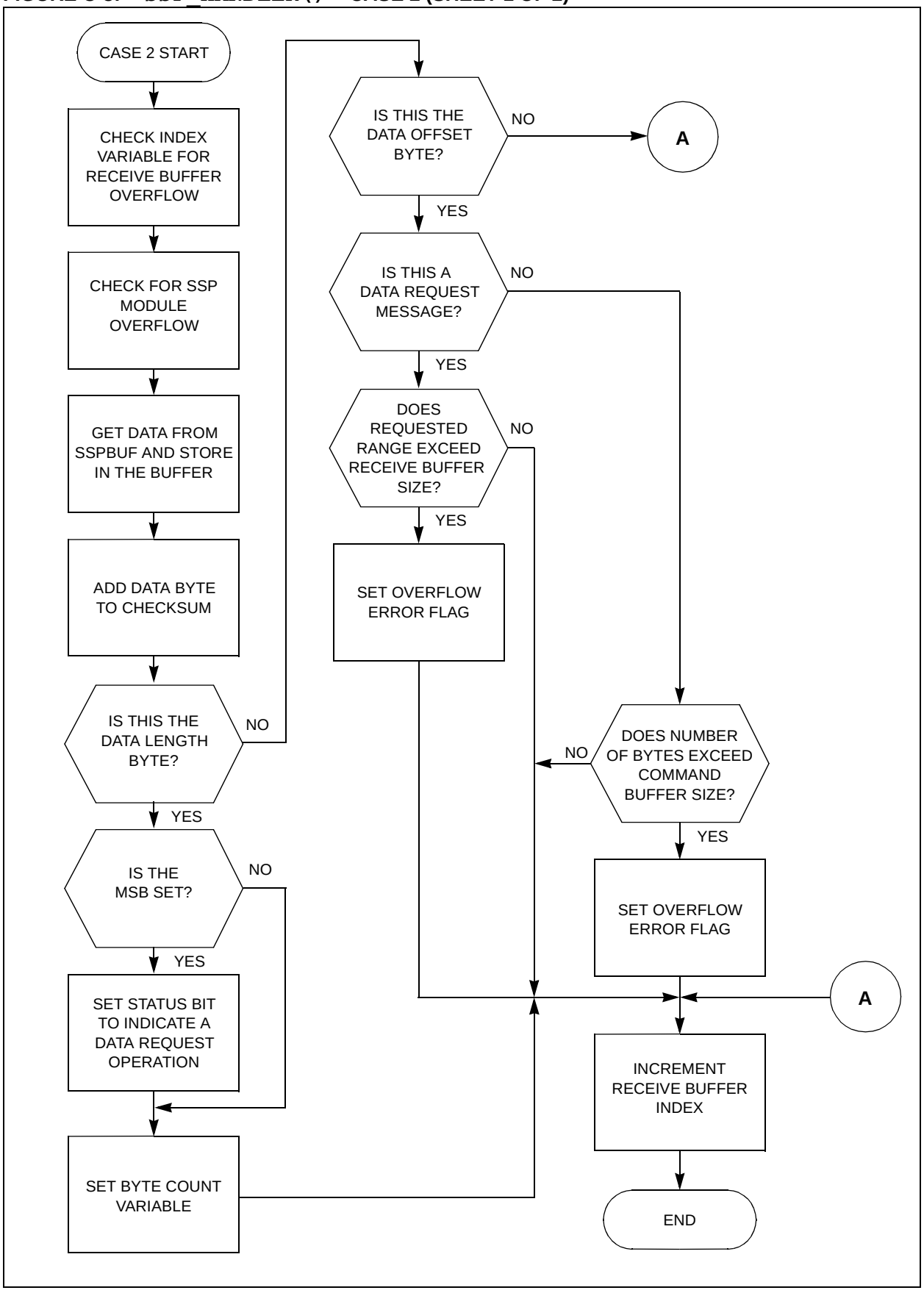


FIGURE C-4: `SSP_HANDLER()` – CASE 2 (SHEET 2 OF 2)

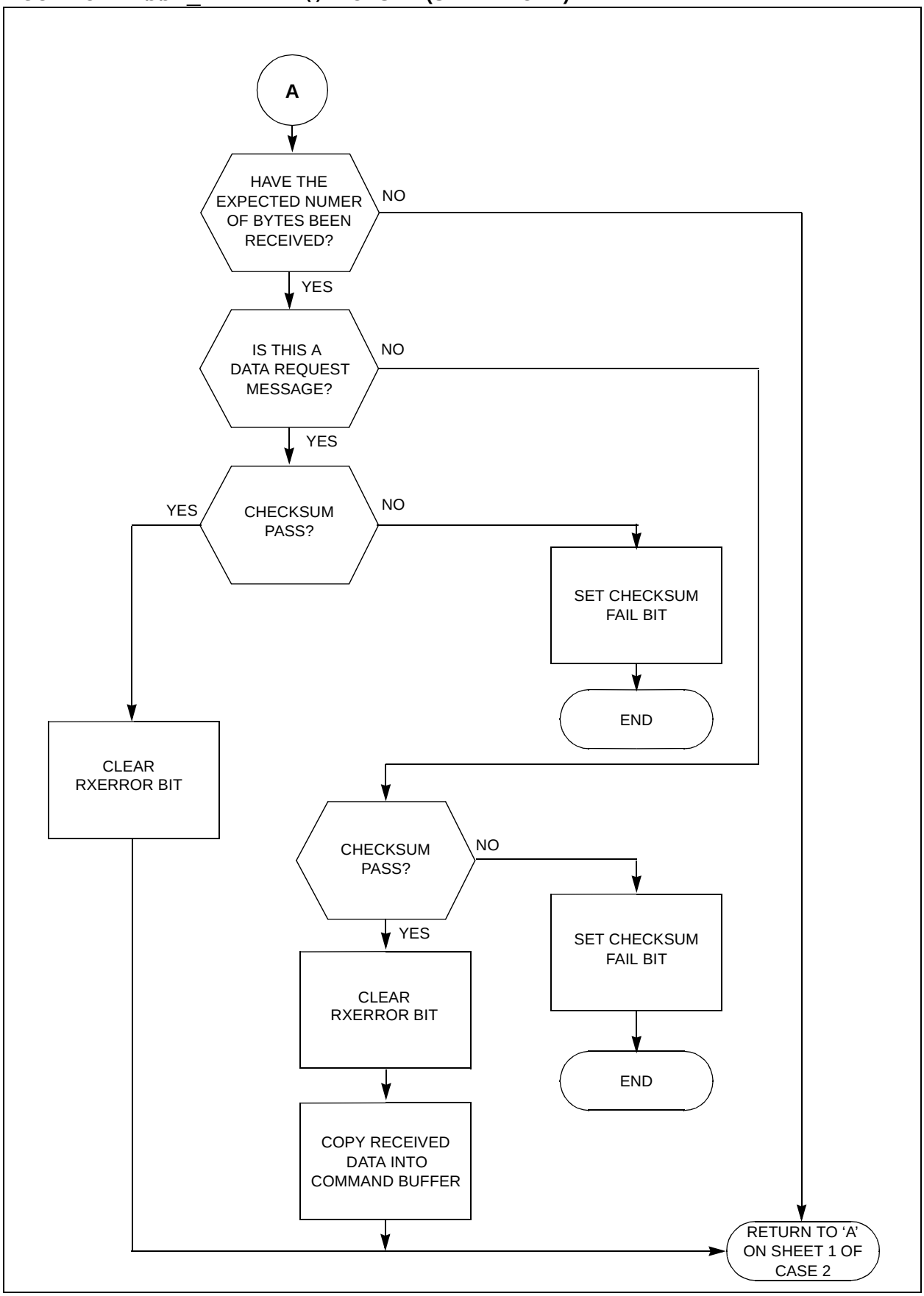


FIGURE C-5: `SSP_HANDLER()` – CASE 3

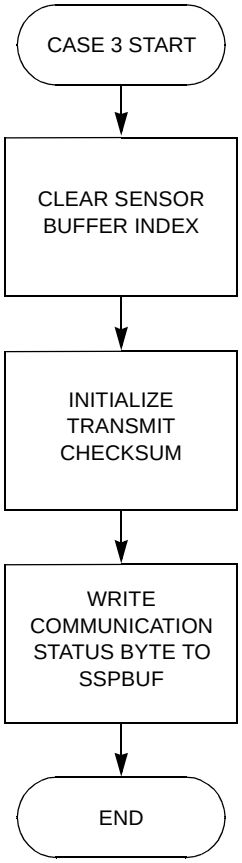


FIGURE C-6: `SSP_HANDLER()` – CASE 4

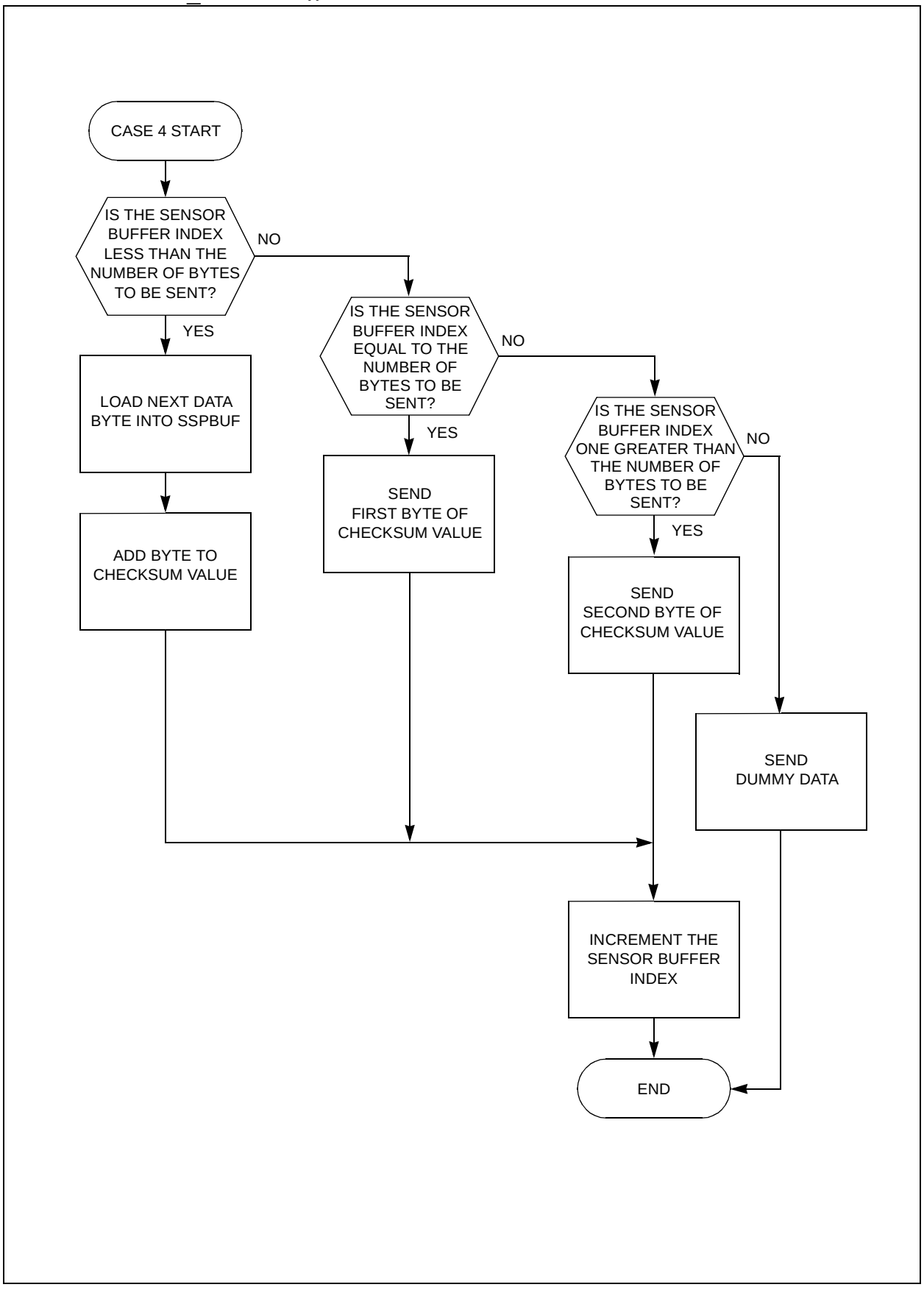
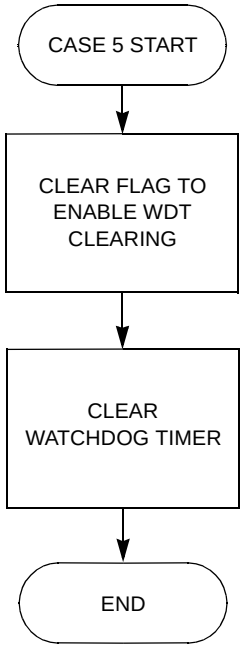


FIGURE C-7: `SSP_HANDLER()` – CASE 5



APPENDIX D: SLAVE I²C SOURCE CODE

```
//
//-----------------------------------------------------------------------
// File:        slavnode.c
//
// Written By:  Stephen Bowling, Microchip Technology
//
// Version:     1.00
//
// Compiled using HiTech PICC Compiler, V. 7.85
//
// This code implements the slave node network protocol for an I2C slave
// device with the SSP module.
//
// The following files should be included in the MPLAB project:
//
//     slavnode.c -- Main source code file
//
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
//Constant Definitions
//-----------------------------------------------------------------------

#define  CCP_HBYTE     0x03          // Set Compare timeout to 1msec
#define  CCP_LBYTE     0xe8
#define  COUNT_10MSEC  10            // Number of Compare timeouts for 10ms
#define  COUNT_100MSEC 10            // Number of Compare timeouts for 100ms
#define  COUNT_1000MSEC10            // Number of Compare timeouts for 1000ms

#define  TEMP_OFFSET   58            // Offset value for temperature table

#define  NODE_ADDR     0x18          // I2C address of this node

#define  ADRES         ADRESH        // Redefine for 10-bit A/D

#define  ON            1
#define  TRUE          1
#define  OFF           0
#define  FALSE         0

//-----------------------------------------------------------------------
// Buffer Length Definitions
//-----------------------------------------------------------------------

#define  RX_BUF_LEN     8            // Length of receive buffer
#define  SENS_BUF_LEN   12           // Length of buffer for sensor data.
#define  CMD_BUF_LEN    4            // Length of buffer for command data.

//-----------------------------------------------------------------------
// Receive Buffer Index Values
//-----------------------------------------------------------------------

#define  SLAVE_ADDR     0            //
#define  DATA_OFFS      2            //
#define  DATA_LEN       1            //
#define  RX_DATA        3            //

//-----------------------------------------------------------------------
// Sensor Buffer Index Values
```



```
//-----------------------------------------------------------------------

#define  COMM_STAT      0            // Communication status byte
#define  SENSOR_DATA    3            // Start index for sensor data

#define  STATUS0        1            // Sensor out-of-range status bits
#define  STATUS1        2            //            "
#define  TEMP0          3            // Temperature (A/D CH4)
#define  TACH0          4            // Fan tachometer #1
#define  ADRES0         5            // A/D CH0
#define  ADRES1         6            // A/D CH1
#define  ADRES2         7            // A/D CH2
#define  ADRES3         8            // A/D CH3
#define  TACH1          9            // Fan tachometer #2
#define  TACH2          10           // Fan tachometer #3
#define  TACH3          11           // Fan tachometer #4

//-----------------------------------------------------------------------
// Command Buffer Index Values
//-----------------------------------------------------------------------

#define  CMD_BYTE0      0
#define  CMD_BYTE1      1
#define  CMD_BYTE2      2
#define  CMD_BYTE3      3

//-----------------------------------------------------------------------
//  Pin Definitions
//-----------------------------------------------------------------------

#define  TACH_IN0       0x10         // Mask values for fan tach
#define  TACH_IN1       0x20         // input pins
#define  TACH_IN2       0x40
#define  TACH_IN3       0x80

#define  LED_0          RB0          // Pin definitions for general
#define  LED_1          RB1          // purpose I/O pins
#define  FAN_CNTRL      RC2
```

```
//-----------------------------------------------------------------------
// Include Files
//-----------------------------------------------------------------------

#include <pic.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

//-----------------------------------------------------------------------
// Function Prototypes
//-----------------------------------------------------------------------
void Setup(void);
interrupt void ISR_Handler(void);
void WriteI2C(char data);
char ReadI2C(void);
void SSP_Handler(void);
void AD_Handler(void);
void CCP2_Handler(void);

//-----------------------------------------------------------------------
// Variable declarations
//-----------------------------------------------------------------------

unsigned char Count_10m,              // Holds number of compare timeouts
              Count_100m,             // Holds number of compare timeouts
              Count_1000m,            //           "
              Count_Tach0,            // Holds number of accumulated pulses
              Count_Tach1,            // for fan speed measurements.
              Count_Tach2,            //
              Count_Tach3;            //           "

char RXBuffer[RX_BUF_LEN];            // Holds incoming bytes from master
                                      // device.
char CmdBuf[CMD_BUF_LEN];             //

char RXChecksum;                      //

unsigned char
       RXBufferIndex,                 // Index to received bytes.
       RXByteCount,                   // Number of bytes received
       SensBufIndex,                  // Index to sensor data table
       CmdBufIndex,                   //
       PORTBold,                      // Holds previous value of PORTB
       temp;

union INTVAL
{
char b[2];
int i;
}

union INTVAL TXChecksum;              // Holds checksum of bytes sent to
                                      // master
union SENSORBUF                       // Holds sensor data and other bytes
{                                     // to be sent to master.
struct{
   unsigned     chkfail:1;
```



```
   unsigned     rxerror:1;
   unsigned     ovflw:1;
   unsigned     sspov:1;
   unsigned     bit4:1;
   unsigned     bit5:1;
   unsigned     bit6:1;
   unsigned     r_w:1;

      } comm_stat ;

unsigned charb[SENS_BUF_LEN];
}  SensorBuf ;

struct{                              // Flags for program
   unsigned     msec10:1;            // 10msec time flag
   unsigned     msec100:1;           // 100msec time flag
   unsigned     msec1000:1;          // 1000msec time flag
   unsigned     bit3:1;
   unsigned     wdtdis:1;            // Watchdog Timer disable flag
   unsigned     bit5:1;
   unsigned     bit6:1;
   unsigned     bit7:1;
      } stat ;

const char temperature[] = {32,32,32,32,33,33,34,34,35,35,35,36,
                            36,37,37,37,38,38,39,39,40,41,41,42,
                            43,43,44,44,45,45,46,46,47,48,49,50,
                            51,52,53,54,55,55,56,57,58,59,59,60,
                            61,61,62,63,63,63,64,65,66,67,68,68,
                            69,70,71,71,72,72,73,73,74,75,76,77,
                            78,79,80,81,81,82,82,83,84,84,85,86,
                            87,88,89,90,91,91,92,93,94,95,96,97,
                            98,99,99,99  };
```

```
//-----------------------------------------------------------------------
// Interrupt Code
//-----------------------------------------------------------------------

interrupt void ISR_Handler(void)
{
if(SSPIF)
    {
    LED_0 = ON;                     // Turn on LED to indicate I2C activity.
    SSPIF = 0;                      // Clear the interrupt flag.
    SSP_Handler();                  // Do I2C communication
    }
if(CCP2IF)
    {
    CCP2IF = 0;                     // Clear the interrupt flag.
    CCP2_Handler();                 // Do timekeeping and sample tach inputs.
    }
if(ADIF)
    {
    ADIF = 0;                       // Clear the interrupt flag.
    AD_Handler();                   // Get A/D data ready and change channel.
    }
}

//-----------------------------------------------------------------------
// void SSP_Handler(void)
//-----------------------------------------------------------------------

void SSP_Handler(void)
{
unsigned char i,j;

//-----------------------------------------------------------------------
// STATE 1: If this is a WRITE operation and last byte was an ADDRESS
//-----------------------------------------------------------------------


if(!STAT_DA && !STAT_RW && STAT_BF && STAT_S)
    {
    // Clear WDT and disable clearing in the main program loop.  The
    // WDT will be used to reset the device if an I2C message exceeds
    // the expected amount of time.

    CLRWDT();
    stat.wdtdis = 1;

    // Since the address byte was the last byte received, clear
    // the receive buffer and the index.  Put the received data
    // in the first buffer location.

    RXBufferIndex = SLAVE_ADDR;
    RXByteCount = 0;

    RXBuffer[RXBufferIndex] = ReadI2C();

    // Initialize the receive checksum.
    RXChecksum = RXBuffer[RXBufferIndex];

    // Increment the buffer index
    RXBufferIndex++;
```



```
    // Reset the communication status byte.  The rxerror bit remains
    // set until a valid data request has taken place.

    SensorBuf.b[COMM_STAT] = 0x02;

    // Check to make sure an SSP overflow has not occurred.
    if(SSPOV)
        {
        SensorBuf.comm_stat.sspov = 1;
        SSPOV = 0;
        }

    }

//-----------------------------------------------------------------------
// STATE 2:  If this is a WRITE operation and the last byte was DATA
//-----------------------------------------------------------------------

else if(STAT_DA && !STAT_RW && STAT_BF)
    {
    // Check the number of data bytes received.

    if(RXBufferIndex == RX_BUF_LEN)
        {
        SensorBuf.comm_stat.ovflw = 1;
        RXBufferIndex = RX_BUF_LEN - 1;
        }

    // Check to see if SSP overflow occurred.

    if(SSPOV)
        {
        SensorBuf.comm_stat.sspov = 1;
        SSPOV = 0;
        }

    // Get the incoming byte of data.

    RXBuffer[RXBufferIndex] = ReadI2C();

    // Add the received value to the checksum.

    RXChecksum += RXBuffer[RXBufferIndex];

    // Check to see if the current byte is the DATA_LEN byte.  If it is,
    // check the MSb to see if this is a data write or data request.

    if(RXBufferIndex == DATA_LEN)
        {
        if(RXBuffer[DATA_LEN] & 0x80)
            {
            // This will be a data request, so the master should send
            // a total of 4 bytes:  SLAVE_ADDR, DATA_LEN, DATA_OFFS,
            // and an 8 bit checksum value.

            // Mask out the R/W bit in the DATA_LEN byte to simplify
            // further calculations.

            RXBuffer[DATA_LEN] &= 0x7f;
```

```
            // Set the R/W bit in COMM_STAT byte to indicate a data
            // request.

            SensorBuf.comm_stat.r_w = 1;
            RXByteCount = 3;
            }
        else
            {
            // This will be a data write, so the master should send the
            // four bytes used for a data request, plus the number of
            // bytes specified by the DATA_LEN byte.  If the total
            // number of bytes to be written exceeds the slave receive
            // buffer, the error flag needs to be set.

            SensorBuf.comm_stat.r_w = 0;
            RXByteCount = RXBuffer[DATA_LEN] + 3;

            if(RXByteCount > RX_BUF_LEN)
                {
                SensorBuf.comm_stat.rxerror = 1;
                SensorBuf.comm_stat.ovflw = 1;
                }
            }
        }

    // If not the DATA_LEN byte, check to see if the current byte
    // is the DATA_OFFS byte.

    else if(RXBufferIndex == DATA_OFFS)
        {

        // If this is a data request command.

        if(SensorBuf.comm_stat.r_w)
            {
            // Is the range of sensor data requested within the limits of the
            // sensor data buffer?  If so, set the appropriate flags.

            if
            (RXBuffer[DATA_LEN] + RXBuffer[DATA_OFFS] > SENS_BUF_LEN - 1)
                {
                SensorBuf.comm_stat.rxerror = 1;
                SensorBuf.comm_stat.ovflw = 1;
                }
            else
                {

                SensorBuf.comm_stat.rxerror = 0;
                SensorBuf.comm_stat.ovflw = 0;
                }
            }

        // Otherwise, this is a data write command.

        else
            {

            // Is the master requesting to write more bytes than are available
            // in the command buffer?
```



```
        if(RXBuffer[DATA_LEN] + RXBuffer[DATA_OFFS] > CMD_BUF_LEN - 1)
            {
            SensorBuf.comm_stat.rxerror = 1;
            SensorBuf.comm_stat.ovflw = 1;
            }
        else
            {
            SensorBuf.comm_stat.rxerror = 0;
            SensorBuf.comm_stat.ovflw = 0;
            }

        }
    }


// If the master is doing a data write to the slave, we must check
// for the end of the data string so we can do the checksum.

else if(RXBufferIndex == RXByteCount)
    {

    //  Is this a data request?

    if(SensorBuf.comm_stat.r_w)
        {
        if(RXChecksum)
            SensorBuf.comm_stat.chkfail = 1;

        else
            SensorBuf.comm_stat.chkfail = 0;
        }

    // Was this a data write?

    else
        {
        if(RXChecksum)
            SensorBuf.comm_stat.chkfail = 1;

        else
            {
            // Checksum was OK, so copy the data in receive buffer
            // into the command buffer.

            for(i=RXBuffer[DATA_OFFS]+3, j = 0;
                i < (RXBuffer[DATA_LEN] + RXBuffer[DATA_OFFS] + 3);
                i++,j++)
                {
                if(j == CMD_BUF_LEN) j--;
                CmdBuf[j] = RXBuffer[i];
                }

            SensorBuf.comm_stat.chkfail = 0;
            }

        }
    }

else;
```

```
    // Increment the receive buffer index.
    RXBufferIndex++;
    }

//-----------------------------------------------------------------------
// STATE 3:  If this is a READ operation and last byte was an ADDRESS
//-----------------------------------------------------------------------

else if(!STAT_DA && STAT_RW && !STAT_BF && STAT_S)
    {
    // Clear the buffer index to the sensor data.
    SensBufIndex = 0;

    // Initialize the transmit checksum

    TXChecksum.i = (int)SensorBuf.b[COMM_STAT];

    // Send the communication status byte.

    WriteI2C(SensorBuf.b[COMM_STAT]);
    }

//-----------------------------------------------------------------------
// STATE 4:  If this is a READ operation and the last byte was DATA
//-----------------------------------------------------------------------

else if(STAT_DA && STAT_RW && !STAT_BF)
    {
    // If we haven't transmitted all the required data yet,
    // get the next byte out of the TXBuffer and increment
    // the buffer index.  Also, add the transmitted byte to
    // the checksum

    if(SensBufIndex < RXBuffer[DATA_LEN])
        {
        WriteI2C(SensorBuf.b[SensBufIndex + RXBuffer[DATA_OFFS]]);
        TXChecksum.i += (int)ReadI2C();
        SensBufIndex++;
        }

    // If all the data bytes have been sent, invert the checksum
    // value and send the first byte.

    else
    if(SensBufIndex == RXBuffer[DATA_LEN])
        {
        TXChecksum.i = ~TXChecksum.i;
        TXChecksum.i++;
        WriteI2C(TXChecksum.b[0]);
        SensBufIndex++;
        }
```



```
   // Send the second byte of the checksum value.

   else
   if(SensBufIndex == (RXBuffer[DATA_LEN] + 1))
       {
       WriteI2C(TXChecksum.b[1]);
       SensBufIndex++;
       }

   // Otherwise, just send dummy data back to the master.

   else
       {
       WriteI2C(0x55);
       }
   }

//-----------------------------------------------------------------------
// STATE 5:  A NACK from the master device is used to indicate that a
// complete transmission has occurred.  The clearing of the
// WDT is reenabled in the main loop at this time.
//-----------------------------------------------------------------------

else if(STAT_DA && !STAT_RW && !STAT_BF)
   {
   stat.wdtdis = 0;
   CLRWDT();
   }

else;
}


//-----------------------------------------------------------------------
// void CCP2_Handler(void)
//
// At each CCP2 interrupt, the tachometer inputs are sampled to see
// if a pin change occurred since the last interrupt.  If so, the count
// value for that tach input is incremented.  Count values are also
// maintained to determine when 10ms, 100msec, and 1000msec have
// elapsed.
//-----------------------------------------------------------------------

void CCP2_Handler(void)
{
TMR1L = 0;                              // Clear Timer1
TMR1H = 0;

temp = PORTB;                           // Get present PORTB value
PORTBold ^= temp;                       // XOR to get pin changes

if(PORTBold & TACH_IN3) Count_Tach3++;  // Test each input to see if pin
if(PORTBold & TACH_IN2) Count_Tach2++;  // changed.
if(PORTBold & TACH_IN1) Count_Tach1++;
if(PORTBold & TACH_IN0) Count_Tach0++;

PORTBold = temp;                        // Store present PORTB value for
// next sample time.
Count_10m++;                            // Increment 10msec count.
```

```
if(Count_10m == COUNT_10MSEC)        // Set flag and zero count if
    {                                // 10msec have elapsed.
    Count_10m = 0;
    Count_100m++;
    stat.msec10 = 1;
    }
if(Count_100m == COUNT_100MSEC)      // Set flag and zero count if
    {                                // 100msec have elapsed.
    Count_100m = 0;
    Count_1000m++;
    stat.msec100 = 1;
    }
if(Count_1000m == COUNT_1000MSEC)    // Set flag and zero count if
    {                                // 1000msec have elapsed.
    Count_1000m = 0;
    stat.msec1000 = 1;
    }

}

//-----------------------------------------------------------------------
// void AD_Handler(void)
//
// This routine gets the data that is ready in the ADRES register and
// changes the A/D channel to the next source.
//-----------------------------------------------------------------------

void AD_Handler(void)
{
switch(ADCON0 & 0x38)                // Get current A/D channel
    {
    case 0x00:    SensorBuf.b[ADRES0] = ADRES;
                  CHS0 = 1;          // Change to CH1
                  CHS1 = 0;
                  CHS2 = 0;
                  break;

    case 0x08:    SensorBuf.b[ADRES1] = ADRES;
                  CHS0 = 0;          // Change to CH2
                  CHS1 = 1;
                  CHS2 = 0;
                  break;

    case 0x10:    SensorBuf.b[ADRES2] = ADRES;
                  CHS0 = 1;          // Change to CH3
                  CHS1 = 1;
                  CHS2 = 0;
                  break;

    case 0x18:    SensorBuf.b[ADRES3] = ADRES;
                  CHS0 = 0;          // Change to CH4
                  CHS1 = 0;
                  CHS2 = 1;
                  break;

    case 0x20:    if(ADRES < TEMP_OFFSET || ADRES > (100 + TEMP_OFFSET))
                     SensorBuf.b[TEMP0] = 0;
                  else
                     SensorBuf.b[TEMP0] = temperature[ADRES - TEMP_OFFSET];
```



```
                CHS0 = 0;               // Change to CH0
                CHS1 = 0;
                CHS2 = 0;
                break;

    default:    CHS0 = 0;               // Change to CH0
                CHS1 = 0;
                CHS2 = 0;
                break;
    }
}

//-----------------------------------------------------------------------
// void WriteI2C(char data)
//-----------------------------------------------------------------------

void WriteI2C(char data)
{
do
    {
    WCOL = 0;
    SSPBUF = data;
    } while(WCOL);

// Release the clock.

CKP = 1;
}

//-----------------------------------------------------------------------
// char ReadI2C(void)
//-----------------------------------------------------------------------

char ReadI2C(void)
{
return(SSPBUF);
}

//-----------------------------------------------------------------------
// void main(void)
//-----------------------------------------------------------------------

void main(void)
{
Setup();

while(1)
    {
    // Check WDT software flag to see if we need to clear the WDT.  The
    // clearing of the WDT is disabled by this flag during I2C events to
    // increase reliablility of the slave I2C function.  In the event that
    // a sequence on the I2C bus takes longer than expected, the WDT will
    // reset the device (and SSP module).

    if(!stat.wdtdis)
        CLRWDT();
```

```
    // The 10msec flag is used to start a new A/D conversion.  When the
    // conversion is complete, the AD_Handler() function called from the
    // ISR will get the conversion results and select the next A/D channel.
    // Therefore, each A/D result will get updated every 10msec x (number of
    // channels used).

    if(stat.msec10)
        {
        // Start the next A/D conversion.
        ADGO = 1;

        // Clear the 10 msec time flag
        stat.msec10 = 0;
        }


    // The 100msec time flag is used to update new values that have been
    // written to the command buffer.

    if(stat.msec100)
        {
        if(CmdBuf[0]) FAN_CNTRL = ON;

        else        FAN_CNTRL = OFF;

        // Clear the activity LEDs
        LED_0 = OFF;
        LED_1 = OFF;

        // Clear the 100msec time flag
        stat.msec100 = 0;
        }

    // The 1000msec time flag is used to update the tachometer values in the
    // SensorBuf array.

    if(stat.msec1000)
        {
        SensorBuf.b[TACH0] = Count_Tach0;
        Count_Tach0 = 0;
        SensorBuf.b[TACH1] = Count_Tach1;
        Count_Tach1 = 0;
        SensorBuf.b[TACH2] = Count_Tach2;
        Count_Tach2 = 0;
        SensorBuf.b[TACH3] = Count_Tach3;
        Count_Tach3 = 0;

        // Clear the 1000msec time flag
        stat.msec1000 = 0;
        }

    } // end while(1);
}
```



```
//-----------------------------------------------------------------------
// void Setup(void)
//
// Initializes program variables and peripheral registers.
//-----------------------------------------------------------------------

void Setup(void)
{
stat.msec10 = 0;                        // Clear the software status bits.
stat.msec100 = 0;
stat.msec1000 = 0;
stat.wdtdis = 0;
stat.button = 0;
stat.b_latch = 0;

RXBufferIndex = 0;                      // Clear software variables
SensBufIndex = 0;
CmdBufIndex = 0;
TXChecksum.i = 0;
RXChecksum = 0;

Count_10m = 0;
Count_100m = 0;
Count_Tach0 = 0;
Count_Tach1 = 0;
Count_Tach2 = 0;
Count_Tach3 = 0;

CmdBuf[0] = 0;

PORTA = 0xff;
TRISA = 0xff;
TRISB = 0xf0;
TRISC = 0x18;

OPTION = 0x78;                          // Weak pullups on, WDT prescaler 2:1

SSPADD = NODE_ADDR;                     // Configure SSP module
SSPSTAT = 0;
SSPCON = 0;
SSPCON = 0x36;

CCPR2L = CCP_LBYTE;                     // Setup CCP2 for event timing
CCPR2H = CCP_HBYTE;
CCP2CON = 0x0a;                         // Compare mode, no output

TMR1L = 0;                              // Timer1 is CCP1 timebase
TMR1H = 0;
T1CON = 0x01;

ADCON1 = 0x02;                          // Setup A/D converter
ADCON0 = 0x81;

if(!TO) LED_1 = 1;                      // Set status LED to indicate WDT
                                        // timeout has occured.
else
   {
   PORTB = 0;                           // Don't clear port values on WDT
   PORTC = 0;                           //
   }
```

```
CLRWDT();

CCP2IF = 0;
CCP2IE = 1;
ADIF = 0;
ADIE = 1;
SSPIF = 0;
SSPIE = 1;
PEIE = 1;
GIE = 1;
}
```


APPENDIX E: GENERIC I²C MASTER READ AND WRITE ROUTINES (ASSEMBLY)

```
;*****************************************************************
;     Implementing Master I2C with the MSSP module on a PICmicro     *
;                                                                    *
;*****************************************************************
;                                                                    *
;   Filename:       mastri2c.asm                                     *
;   Date:           07/18/2000                                       *
;   Revision:       1.00                                             *
;                                                                    *
;   Tools:          MPLAB   5.11.00                                  *
;                   MPLINK  2.10.00                                  *
;                   MPASM   2.50.00                                  *
;                                                                    *
;  Author:          Richard L. Fischer                               *
;                                                                    *
;  Company:         Microchip Technology Incorporated                *
;                                                                    *
;*****************************************************************
;                                                                    *
;   System files required:                                           *
;                                                                    *
;                   mastri2c.asm                                     *
;                   i2ccomm.asm                                      *
;                   init.asm                                         *
;                                                                    *
;                   mastri2c.inc                                     *
;                   i2ccomm.inc                                      *
;                   i2ccomm1.inc                                     *
;                   flags.inc                                        *
;                                                                    *
;                   p16f873.inc                                      *
;                   16f873.lkr     (modified for interrupts)         *
;                                                                    *
;*****************************************************************
;                                                                    *
;   Notes:                                                           *
;                                                                    *
;   Device Fosc -> 8.00MHz                                           *
;   WDT -> on                                                        *
;   Brownout -> on                                                   *
;   Powerup timer -> on                                              *
;   Code Protect -> off                                              *
;                                                                    *
```

```
;   Interrupt sources -                                                 *
;                       1. I2C events (valid events)                    *
;                       2. I2C Bus Collision                            *
;                       3. Timer1 - 100mS intervals                     *
;                                                                       *
;                                                                       *
************************************************************************/
   list      p=16f873              ; list directive to define processor
   #include <p16f873.inc>          ; processor specific variable definitions
    __CONFIG (_CP_OFF & _WDT_ON & _BODEN_ON & _PWRTE_ON & _HS_OSC & _WRT_ENABLE_ON
            & _LVP_OFF & _CPD_OFF)

   #include  "mastri2c.inc"        ;
   #include  "i2ccomm1.inc"        ; required include file
   errorlevel -302

 #define  ADDRESS     0x01         ; Slave I2C address



;-----------------------------------------------------------------------
;   ********************* RESET VECTOR LOCATION  *********************
;-----------------------------------------------------------------------
RESET_VECTOR  CODE    0x000         ; processor reset vector
   movlw  high  start              ; load upper byte of 'start' label
   movwf  PCLATH                   ; initialize PCLATH
   goto   start                    ; go to beginning of program

;-----------------------------------------------------------------------
;  ******************* INTERRUPT VECTOR LOCATION  *******************
;-----------------------------------------------------------------------
INT_VECTOR   CODE    0x004          ; interrupt vector location
   movwf   w_temp                  ; save off current W register contents
   movf    STATUS,w                ; move status register into W register
   clrf    STATUS                  ; ensure file register bank set to 0
   movwf   status_temp             ; save off contents of STATUS register
   movf    PCLATH,w
   movwf   pclath_temp             ; save off current copy of PCLATH
   clrf    PCLATH                  ; reset PCLATH to page 0

; TEST FOR COMPLETION OF VALID I2C EVENT
   bsf     STATUS,RP0              ; select SFR bank
   btfss   PIE1,SSPIE              ; test is interrupt is enabled
   goto    test_buscoll            ; no, so test for Bus Collision Int
```



```
   bcf     STATUS,RP0              ; select SFR bank
   btfss   PIR1,SSPIF              ; test for SSP H/W flag
   goto    test_buscoll            ; no, so test for Bus Collision Int
   bcf     PIR1,SSPIF              ; clear SSP H/W flag

   pagesel service_i2c             ; select page bits for function
   call    service_i2c             ; service valid I2C event

; TEST FOR I2C BUS COLLISION EVENT
test_buscoll
   banksel PIE2                    ; select SFR bank
   btfss   PIE2,BCLIE              ; test if interrupt is enabled
   goto    test_timer1             ; no, so test for Timer1 interrupt
   bcf     STATUS,RP0              ; select SFR bank
   btfss   PIR2,BCLIF              ; test if Bus Collision occured
   goto    test_timer1             ; no, so test for Timer1 interrupt
   bcf     PIR2,BCLIF              ; clear Bus Collision H/W flag
   call    service_buscoll         ; service bus collision error

; TEST FOR TIMER1 ROLLOVER EVENT
test_timer1
   banksel PIE1                    ; select SFR bank
   btfss   PIE1,TMR1IE             ; test if interrupt is enabled
   goto    exit_isr                ; no, so exit ISR
   bcf     STATUS,RP0              ; select SFR bank
   btfss   PIR1,TMR1IF             ; test if Timer1 rollover occured
   goto    exit_isr                ; no so exit isr
   bcf     PIR1,TMR1IF             ; clear Timer1 H/W flag

   pagesel service_i2c             ; select page bits for function
   call    service_i2c             ; service valid I2C event
   banksel T1CON                   ; select SFR bank
   bcf     T1CON,TMR1ON            ; turn off Timer1 module
   movlw   0x58                    ;
   addwf   TMR1L,f                 ; reload Timer1 low
   movlw   0x9E                    ;
   movwf   TMR1H                   ; reload Timer1 high
   banksel PIE1                    ; select SFR bank
   bcf     PIE1,TMR1IE             ; disable Timer1 interrupt
   bsf     PIE1,SSPIE              ; enable SSP H/W interrupt

exit_isr
   clrf    STATUS                  ; ensure file register bank set to 0
   movf    pclath_temp,w
```

```
movwf   PCLATH                  ; restore PCLATH
movf    status_temp,w           ; retrieve copy of STATUS register
movwf   STATUS                  ; restore pre-isr STATUS register contents
swapf   w_temp,f                ;
swapf   w_temp,w                ; restore pre-isr W register contents
retfie                          ; return from interrupt
```



```
;-------------------------------------------------------------------------
;   ****************** MAIN CODE START LOCATION  ******************
;-------------------------------------------------------------------------
MAIN    CODE
start
   pagesel init_ports               ;
   call    init_ports               ; initialize Ports
   call    init_timer1              ; initialize Timer1
   pagesel init_i2c
   call    init_i2c                 ; initialize I2C module

   banksel eflag_event              ; select GPR bank
   clrf    eflag_event              ; initialize event flag variable
   clrf    sflag_event              ; initialize event flag variable
   clrf    i2cState                 ;

   call    CopyRom2Ram              ; copy ROM string to RAM
   call    init_vars                ; initialize variables

   banksel PIE2                     ; select SFR bank
   bsf     PIE2,BCLIE               ; enable interrupt
   banksel PIE1                     ; select SFR bank
   bsf     PIE1,TMR1IE              ; enable Timer1 interrupt
   bsf     INTCON,PEIE              ; enable peripheral interrupt
   bsf     INTCON,GIE               ; enable global interrupt


;*****************************************************************
;                     MAIN LOOP BEGINS HERE
;*****************************************************************
main_loop
   clrwdt                           ; reset WDT

   banksel eflag_event              ; select SFR bank
   btfsc   eflag_event,ack_error    ; test for ack error event flag
   call    service_ackerror         ; service ack error

   banksel sflag_event              ; select SFR bank
   btfss   sflag_event,rw_done      ; test if read/write cycle complete
   goto    main_loop                ; goto main loop
   call    string_compare           ; else, go compare strings

   banksel T1CON                    ; select SFR bank
   bsf     T1CON,TMR1ON             ; turn on Timer1 module
```

```
   banksel PIE1                     ; select SFR bank
   bsf     PIE1,TMR1IE              ; re-enable Timer1 interrupts

   call    init_vars                ; re-initialize variables
   goto    main_loop                ; goto main loop


;*********************************************************************
;   *************** Bus Collision Service Routine ******************
;*********************************************************************
service_buscoll
   banksel i2cState                 ; select GPR bank
   clrf    i2cState                 ; reset I2C bus state variable
   call    init_vars                ; re-initialize variables
   bsf     T1CON,TMR1ON             ; turn on Timer1 module
   banksel PIE1                     ; select SFR bank
   bsf     PIE1,TMR1IE              ; enable Timer1 interrupt
   return                           ;


;*********************************************************************
;   ************* Acknowledge Error Service Routine ***************
;*********************************************************************
service_ackerror
   banksel eflag_event              ; select SFR bank
   bcf     eflag_event,ack_error    ; reset acknowledge error event flag
   clrf    i2cState                 ; reset bus state variable
   call    init_vars                ; re-initialize variables
   bsf     T1CON,TMR1ON             ; turn on Timer1 module
   banksel PIE1                     ; select SFR bank
   bsf     PIE1,TMR1IE              ; enable Timer1 interrupt
   return                           ;


;*********************************************************************
;   *****  INITIALIZE VARIABLES USED IN SERVICE_I2C FUNCTION  ******
;*********************************************************************
init_vars
   movlw   D'21'                    ; byte count for this example
   banksel write_count              ; select GPR bank
   movwf   write_count              ; initialize write count
   movwf   read_count               ; initialize read count

   movlw   write_string             ; get write string array address
```



```
   movwf   write_ptr               ; initialize write pointer
   movlw   read_string             ; get read string placement address
   movwf   read_ptr                ; initialize read pointer

   movlw   ADDRESS                 ; get address of slave
   movwf   temp_address            ; initialize temporary address hold reg
   return                          ;


;-----------------------------------------------------------------------
;  ******************  Compare Strings   ************************
;-----------------------------------------------------------------------
;Compare the string written to and read back from the Slave
string_compare
   movlw   read_string             ;
   banksel ptr1                    ; select GPR bank
   movwf   ptr1                    ; initialize first pointer
   movlw   write_string
   movwf   ptr2                    ; initialize second pointer

loop
   movf    ptr1,w                  ; get address of first pointer
   movwf   FSR                     ; init FSR
   movf    INDF,w                  ; retrieve one byte
   banksel temp_hold               ; select GPR bank
   movwf   temp_hold               ; save off byte 1
   movf    ptr2,w                  ;
   movwf   FSR                     ; init FSR
   movf    INDF,w                  ; retrieve second byte
   subwf   temp_hold,f             ; do comparison
   btfss   STATUS,Z                ; test for valid compare
   goto    not_equal               ; bytes not equal
   iorlw   0x00                    ; test for null character
   btfsc   STATUS,Z                ;
   goto    end_string              ; end of string has been reached
   incf    ptr1,f                  ; update first pointer
   incf    ptr2,f                  ; update second pointer
   goto    loop                    ; do more comparisons

not_equal
   banksel PORTB                   ; select GPR bank
   movlw   b'00000001'
   xorwf   PORTB,f
   goto    exit
```

```
end_string
   banksel  PORTB                    ; select GPR bank
   movlw    b'00000010'              ; no error
   xorwf    PORTB,f
exit
   banksel  sflag_event              ; select SFR bank
   bcf      sflag_event,rw_done      ; reset flag
   return



;-----------------------------------------------------------------------
;  ********************  Program Memory Read   ********************
;-----------------------------------------------------------------------
;Read the message from location MessageTable
CopyRom2Ram
   movlw   write_string             ;
   movwf   FSR                      ; initialize FSR

   banksel EEADRH                   ; select SFR bank
   movlw   High (Message1)          ; point to the Message Table
   movwf   EEADRH                   ; init SFR EEADRH
   movlw   Low (Message1)           ;
   movwf   EEADR                    ; init SFR EEADR

next1
   banksel EECON1                   ; select SFR bank
   bsf     EECON1,EEPGD             ; select the program memory
   bsf     EECON1,RD                ; read word
   nop                              ;
   nop                              ;
   banksel EEDATA                   ;
   rlf     EEDATA,w                 ; get bit 7 in carry
   rlf     EEDATH,w                 ; get high byte in w

   movwf   INDF                     ; save it
   incf    FSR,f                    ;

   banksel EEDATA                   ; select SFR bank
   bcf     EEDATA,7                 ; clr bit 7
   movf    EEDATA,w                 ; get low byte and see = 0?
   btfsc   STATUS,Z                 ; end?
   return                           ;
   movwf   INDF                     ; save it
   incf    FSR,f                    ; update FSR pointer
```



```
   banksel EEADR                    ; point to address
   incf    EEADR,f                  ; inc to next location
   btfsc   STATUS,Z                 ; cross over 0xff
   incf    EEADRH,f                 ; yes then inc high
   goto    next1                    ; read next byte


;-----------------------------------------------------------------------
;-----------------------------------------------------------------------

Message1   DA      "Master and Slave I2C",0x00,0x00

   END                              ; required directive
```

```
;*****************************************************************
;                                                                *
;     Implementing Master I2C with the MSSP module on a PICmicro     *
;                                                                *
;*****************************************************************
;                                                                *
;   Filename:       i2ccomm.asm                                  *
;   Date:           07/18/2000                                   *
;   Revision:       1.00                                         *
;                                                                *
;   Tools:          MPLAB   5.11.00                              *
;                   MPLINK  2.10.00                              *
;                   MPASM   2.50.00                              *
;                                                                *
;  Author:          Richard L. Fischer                           *
;                   John E. Andrews                              *
;                                                                *
;  Company:         Microchip Technology Incorporated            *
;                                                                *
;*****************************************************************
;                                                                *
;   Files required:                                              *
;                                                                *
;                   i2ccomm.asm                                  *
;                   i2ccomm.inc                                  *
;                   flags.inc    (referenced in i2ccomm.inc file)    *
;                   i2ccomm1.inc (must be included in main file)     *
;                   p16f873.inc                                  *
;                                                                *
;*****************************************************************
;                                                                *
;   Notes:   The routines within this file are used to read from     *
;   and write to a Slave I2C device. The MSSP initialization         *
;   function is also contained within this file.                 *
;                                                                *
;*****************************************************************/

   #include <p16f873.inc>              ; processor specific definitions
   #include  "i2ccomm.inc"             ; required include file
   errorlevel -302

 #define  FOSC        D'8000000'         ; define FOSC to PICmicro
 #define  I2CClock    D'400000'          ; define I2C bite rate
 #define  ClockValue  (((FOSC/I2CClock)/4) -1) ;
```



```
;-----------------------------------------------------------------------
;   ***********************  I2C Service  ************************
;-----------------------------------------------------------------------
I2C_COMM   CODE
service_i2c

   movlw   high  I2CJump            ; fetch upper byte of jump table address
   movwf   PCLATH                   ; load into upper PC latch
   movlw   i2cSizeMask
   banksel i2cState                 ; select GPR bank
   andwf   i2cState,w               ; retrieve current I2C state
   addlw   low  (I2CJump + 1)       ; calc state machine jump addr into W
   btfsc   STATUS,C                 ; skip if carry occured
   incf    PCLATH,f                 ; otherwise add carry
I2CJump       ; address were jump table branch occurs, this addr also used in fill
   movwf   PCL                      ; index into state machine jump table
; jump to processing for each state = i2cState value for each state

   goto    WrtStart                 ; write start sequence           =  0
   goto    SendWrtAddr              ; write address, R/W=1           =  1
   goto    WrtAckTest               ; test ack,write data            =  2
   goto    WrtStop                  ; do stop if done                =  3

   goto    ReadStart                ; write start sequence           =  4
   goto    SendReadAddr             ; write address, R/W=0           =  5
   goto    ReadAckTest              ; test acknowledge after address =  6
   goto    ReadData                 ; read more data                 =  7
   goto    ReadStop                 ; generate stop sequence         =  8

I2CJumpEnd
        Fill (return),  (I2CJump-I2CJumpEnd) + i2cSizeMask

;-----------------------------------------------------------------------
;   ********************* Write data to Slave   *********************
;-----------------------------------------------------------------------
; Generate I2C bus start condition               [ I2C STATE -> 0 ]
WrtStart
   banksel  write_ptr               ; select GPR bank
   movf     write_ptr,w             ; retrieve ptr address
   movwf    FSR                     ; initialize FSR for indirect access
   incf     i2cState,f              ; update I2C state variable
   banksel  SSPCON2                 ; select SFR bank
   bsf      SSPCON2,SEN             ; initiate I2C bus start condition
   return                           ;
```

```
; Generate I2C address write (R/W=0)             [ I2C STATE -> 1 ]
SendWrtAddr
   banksel temp_address             ; select GPR bank
   bcf     STATUS,C                 ; ensure carry bit is clear
   rlf     temp_address,w           ; compose 7-bit address
   incf    i2cState,f               ; update I2C state variable
   banksel SSPBUF                   ; select SFR bank
   movwf   SSPBUF                   ; initiate I2C bus write condition
   return                           ;

; Test acknowledge after address and data write  [ I2C STATE -> 2 ]
WrtAckTest
   banksel SSPCON2                  ; select SFR bank
   btfss   SSPCON2,ACKSTAT          ; test for acknowledge from slave
   goto    WrtData                  ; go to write data module
   banksel eflag_event              ; select GPR bank
   bsf     eflag_event,ack_error    ; set acknowledge error
   clrf    i2cState                 ; reset I2C state variable
   banksel SSPCON2                  ; select SFR bank
   bsf     SSPCON2,PEN              ; initiate I2C bus stop condition
   return                           ;

; Generate I2C write data condition
WrtData
   movf    INDF,w                   ; retrieve byte into w
   banksel write_count              ; select GPR bank
   decfsz  write_count,f            ; test if all done with writes
   goto    send_byte                ; not end of string
   incf    i2cState,f               ; update I2C state variable
send_byte
   banksel SSPBUF                   ; select SFR bank
   movwf   SSPBUF                   ; initiate I2C bus write condition
   incf    FSR,f                    ; increment pointer
   return                           ;

; Generate I2C bus stop condition                [ I2C STATE -> 3 ]
WrtStop
   banksel SSPCON2                  ; select SFR bank
   btfss   SSPCON2,ACKSTAT          ; test for acknowledge from slave
   goto    no_error                 ; bypass setting error flag
   banksel eflag_event              ; select GPR bank
   bsf     eflag_event,ack_error    ; set acknowledge error
   clrf    i2cState                 ; reset I2C state variable
```



```
   goto    stop
no_error
   banksel i2cState                ; select GPR bank
   incf    i2cState,f              ; update I2C state variable for read
stop
   banksel SSPCON2                 ; select SFR bank
   bsf     SSPCON2,PEN             ; initiate I2C bus stop condition
   return                          ;


;-----------------------------------------------------------------------
;   ********************* Read data from Slave   *********************
;-----------------------------------------------------------------------
; Generate I2C start condition                  [ I2C STATE -> 4 ]
ReadStart
   banksel  read_ptr               ; select GPR bank
   movf     read_ptr,W             ; retrieve ptr address
   movwf    FSR                    ; initialize FSR for indirect access
   incf     i2cState,f             ; update I2C state variable
   banksel  SSPCON2                ; select SFR bank
   bsf      SSPCON2,SEN            ; initiate I2C bus start condition
   return                          ;

; Generate I2C address write (R/W=1)            [ I2C STATE -> 5 ]
SendReadAddr
   banksel temp_address            ; select GPR bank
   bsf     STATUS,C                ; ensure cary bit is clear
   rlf     temp_address,w          ; compose 7 bit address
   incf    i2cState,f              ; update I2C state variable
   banksel SSPBUF                  ; select SFR bank
   movwf   SSPBUF                  ; initiate I2C bus write condition
   return                          ;

; Test acknowledge after address write          [ I2C STATE -> 6 ]
ReadAckTest
   banksel SSPCON2                 ; select SFR bank
   btfss   SSPCON2,ACKSTAT         ; test for not acknowledge from slave
   goto    StartReadData           ; good ack, go issue bus read
   banksel eflag_event             ; ack error, so select GPR bank
   bsf     eflag_event,ack_error   ; set ack error flag
   clrf    i2cState                ; reset I2C state variable
   banksel SSPCON2                 ; select SFR bank
   bsf     SSPCON2,PEN             ; initiate I2C bus stop condition
   return
```

```
StartReadData
   bsf     SSPCON2,RCEN             ; generate receive condition
   banksel i2cState                 ; select GPR bank
   incf    i2cState,f               ; update I2C state variable
   return

; Read slave I2C                                [ I2C STATE -> 7 ]
ReadData
   banksel SSPBUF                   ; select SFR bank
   movf    SSPBUF,w                 ; save off byte into W
   banksel read_count               ; select GPR bank
   decfsz  read_count,f             ; test if all done with reads
   goto    SendReadAck              ; not end of string so send ACK

; Send Not Acknowledge
SendReadNack
   movwf   INDF                     ; save off null character
   incf    i2cState,f               ; update I2C state variable
   banksel SSPCON2                  ; select SFR bank
   bsf     SSPCON2,ACKDT            ; acknowledge bit state to send (not ack)
   bsf     SSPCON2,ACKEN            ; initiate acknowledge sequence
   return

; Send Acknowledge
SendReadAck
   movwf   INDF                     ; no, save off byte
   incf    FSR,f                    ; update receive pointer
   banksel SSPCON2                  ; select SFR bank
   bcf     SSPCON2,ACKDT            ; acknowledge bit state to send
   bsf     SSPCON2,ACKEN            ; initiate acknowledge sequence
   btfsc   SSPCON2,ACKEN            ; ack cycle complete?
   goto    $-1                      ; no, so loop again
   bsf     SSPCON2,RCEN             ; generate receive condition
   return                           ;

; Generate I2C stop condition                   [ I2C STATE -> 8 ]
ReadStop
   banksel SSPCON2                  ; select SFR bank
   bcf     PIE1,SSPIE               ; disable SSP interrupt
   bsf     SSPCON2,PEN              ; initiate I2C bus stop condition
   banksel i2cState                 ; select GPR bank
   clrf    i2cState                 ; reset I2C state variable
   bsf   sflag_event,rw_done        ; set read/write done flag
   return
```



```
;-----------------------------------------------------------------------
;   ******************* Generic bus idle check ***********************
;-----------------------------------------------------------------------
; test for i2c bus idle state; not implemented in this code (example only)
i2c_idle
   banksel SSPSTAT                  ; select SFR bank
   btfsc   SSPSTAT,R_W              ; test if transmit is progress
   goto    $-1                      ; module busy so wait
   banksel SSPCON2                  ; select SFR bank
   movf    SSPCON2,w                ; get copy of SSPCON2 for status bits
   andlw   0x1F                     ; mask out non-status bits
   btfss   STATUS,Z                 ; test for zero state, if Z set, bus is idle
   goto    $-3                      ; bus is busy so test again
   return                           ; return to calling routine


;-----------------------------------------------------------------------
;  ****************** INITIALIZE MSSP MODULE  ******************
;-----------------------------------------------------------------------

init_i2c
   banksel SSPADD                ; select SFR bank
   movlw   ClockValue            ; read selected baud rate
   movwf   SSPADD                ; initialize I2C baud rate
   bcf     SSPSTAT,6             ; select I2C input levels
   bcf     SSPSTAT,7             ; enable slew rate

   movlw   b'00011000'           ;
   iorwf   TRISC,f               ; ensure SDA and SCL are inputs
   bcf     STATUS,RP0            ; select SFR bank
   movlw   b'00111000'           ;
   movwf   SSPCON                ; Master mode, SSP enable
   return                        ; return from subroutine


   END                           ; required directive
```

```
;*****************************************************************
;                                                                *
;     Implementing Master I2C with the MSSP module on a PICmicro     *
;                                                                *
;*****************************************************************
;                                                                *
;   Filename:      init.asm                                      *
;   Date:          07/18/2000                                    *
;   Revision:      1.00                                          *
;                                                                *
;   Tools:         MPLAB   5.11.00                               *
;                  MPLINK  2.10.00                               *
;                  MPASM   2.50.00                               *
;                                                                *
;  Author:         Richard L. Fischer                            *
;                                                                *
;  Company:        Microchip Technology Incorporated             *
;                                                                *
;*****************************************************************
;                                                                *
;   Files required:                                              *
;                                                                *
;                   init.asm                                     *
;                                                                *
;                   p16f873.inc                                  *
;                                                                *
;                                                                *
;*****************************************************************
;                                                                *
;   Notes:                                                       *
;                                                                *
;                                                                *
;*****************************************************************/

   #include <p16f873.inc>         ; processor specific variable definitions
   errorlevel -302



   GLOBAL  init_timer1            ; make function viewable for other modules
   GLOBAL  init_ports             ; make function viewable for other modules
```



```
;-----------------------------------------------------------------------
;  ******************* INITIALIZE PORTS  ************************
;-----------------------------------------------------------------------
INIT_CODE   CODE

init_ports
   banksel PORTA                ; select SFR bank
   clrf    PORTA                ; initialize PORTS
   clrf    PORTB                ;
   clrf    PORTC                ;

   bsf     STATUS,RP0           ; select SFR bank
   movlw   b'00000110'          ;
   movwf   ADCON1               ; make PORTA digital
   clrf    TRISB                ;
   movlw   b'000000'            ;
   movwf   TRISA                ;
   movlw   b'00011000'          ;
   movwf   TRISC                ;
   return


;-----------------------------------------------------------------------
;  ******************* INITIALIZE TIMER1 MODULE  ******************
;-----------------------------------------------------------------------
init_timer1
   banksel T1CON                ; select SFR bank
   movlw   b'00110000'          ; 1:8 prescale, 100mS rollover
   movwf   T1CON                ; initialize Timer1

   movlw   0x58                 ;
   movwf   TMR1L                ; initialize Timer1 low
   movlw   0x9E                 ;
   movwf   TMR1H                ; initialize Timer1 high

   bcf     PIR1,TMR1IF          ; ensure flag is reset
   bsf     T1CON,TMR1ON         ; turn on Timer1 module
   return                       ; return from subroutine

   END                          ; required directive
```

```
;*********************************************************************
;                                                                    *
;   Filename:       mastri2c.inc                                     *
;   Date:           07/18/2000                                       *
;   Revision:       1.00                                             *
;                                                                    *
;   Tools:          MPLAB   5.11.00                                  *
;                   MPLINK  2.10.00                                  *
;                   MPASM   2.50.00                                  *
;                                                                    *
;*********************************************************************



;*******    INTERRUPT CONTEXT SAVE/RESTORE VARIABLES
INT_VAR        UDATA   0x20       ; create uninitialized data "udata" section
w_temp           RES     1        ;
status_temp      RES     1        ;
pclath_temp      RES     1


INT_VAR1       UDATA   0xA0       ; reserve location 0xA0
w_temp1          RES     1


;*******    GENERAL PURPOSE VARIABLES
GPR_DATA         UDATA
temp_hold        RES     1        ; temp variable for string compare
ptr1             RES     1        ; used as pointer in string compare
ptr2             RES     1        ; used as pointer in string compare


STRING_DATA      UDATA
write_string     RES      D'30'
read_string      RES      D'30'


   EXTERN   init_timer1           ; reference linkage for function
   EXTERN   init_ports            ; reference linkage for function
```



```
;*****************************************************************
;                                                                *
;   Filename:       i2ccomm1.inc                                 *
;   Date:           07/18/2000                                   *
;   Revision:       1.00                                         *
;                                                                *
;   Tools:          MPLAB   5.11.00                              *
;                   MPLINK  2.10.00                              *
;                   MPASM   2.50.00                              *
;                                                                *
;*****************************************************************
;                                                                *
;   Notes:                                                       *
;                                                                *
;   This file is to be included in the <main.asm> file. The      *
;   <main.asm> notation represents the file which has the        *
;   subroutine calls for the functions 'service_i2c' and 'init_i2c'. *
;                                                                *
;                                                                *
;*****************************************************************/

   #include  "flags.inc"         ; required include file


   GLOBAL   write_string         ; make variable viewable for other modules
   GLOBAL   read_string          ; make variable viewable for other modules

   EXTERN   sflag_event          ; reference linkage for variable
   EXTERN   eflag_event          ; reference linkage for variable
   EXTERN   i2cState             ; reference linkage for variable
   EXTERN   read_count           ; reference linkage for variable
   EXTERN   write_count          ; reference linkage for variable
   EXTERN   write_ptr            ; reference linkage for variable
   EXTERN   read_ptr             ; reference linkage for variable
   EXTERN   temp_address         ; reference linkage for variable

   EXTERN   init_i2c             ; reference linkage for function
   EXTERN   service_i2c          ; reference linkage for function


;*****************************************************************
;                                                                *
```

```
;   Additional notes on variable usage:                                *
;                                                                      *
;   The variables listed below are used within the function            *
;   service_i2c. These variables must be initialized with the          *
;   appropriate data from within the calling file. In this             *
;   application code the main file is 'mastri2c.asm'. This file        *
;   contains the function calls to service_i2c. It also contains       *
;   the function for initializing these variables, called 'init_vars'* 
;                                                                      *
;   To use the service_i2c function to read from and write to an       *
;   I2C slave device, information is passed to this function via       *
;   the following variables.                                           *
;                                                                      *
;                                                                      *
;   The following variables are used as function parameters:           *
;                                                                      *
;   read_count   - Initialize this variable for the number of bytes    *
;                  to read from the slave I2C device.                  *
;   write_count  - Initialize this variable for the number of bytes    *
;                  to write to the slave I2C device.                   *
;   write_ptr    - Initialize this variable with the address of the    *
;                  data string or data byte to write to the slave      *
;                  I2C device.                                         *
;   read_ptr     - Initialize this variable with the address of the    *
;                  location for storing data read from the slave I2C   *
;                  device.                                             *
;   temp_address - Initialize this variable with the address of the    *
;                  slave I2C device to communicate with.               *
;                                                                      *
;                                                                      *
;   The following variables are used as status or error events         *
;                                                                      *
;   sflag_event  - This variable is implemented for status or          *
;                  event flags. The flags are defined in the file      *
;                  'flags.inc'.                                        *
;   eflag_event  - This variable is implemented for error flags. The   *
;                  flags are defined in the file 'flags.inc'.          *
;                                                                      *
;                                                                      *
;   The following variable is used in the state machine jumnp table.   *
;                                                                      *
;   i2cState     - This variable holds the next I2C state to execute.* 
;                                                                      *
;*********************************************************************
```



```
;*****************************************************************
;                                                                *
;   Filename:       flags.inc                                    *
;   Date:           07/18/2000                                   *
;   Revision:       1.00                                         *
;                                                                *
;   Tools:          MPLAB   5.11.00                              *
;                   MPLINK  2.10.00                              *
;                   MPASM   2.50.00                              *
;                                                                *
;*****************************************************************
;                                                                *
;   Notes:                                                       *
;                                                                *
;   This file defines the flags used in the i2ccomm.asm file.    *
;                                                                *
;                                                                *
;*****************************************************************/



; bits for variable sflag_event
#define  sh1          0                   ; place holder
#define  sh2          1                   ; place holder
#define  sh3          2                   ; place holder
#define  sh4          3                   ; place holder
#define  sh5          4                   ; place holder
#define  sh6          5                   ; place holder
#define  sh7          6                   ; place holder
#define  rw_done      7                   ; flag bit


; bits for variable eflag_event
#define  ack_error    0                   ; flag bit
#define  eh1          1                   ; place holder
#define  eh2          2                   ; place holder
#define  eh3          3                   ; place holder
#define  eh4          4                   ; place holder
#define  eh5          5                   ; place holder
#define  eh6          6                   ; place holder
#define  eh7          7                   ; place holder
```

```
;*********************************************************************
;                                                                    *
;   Filename:       i2ccomm.inc                                      *
;   Date:           07/18/2000                                       *
;   Revision:       1.00                                             *
;                                                                    *
;   Tools:          MPLAB   5.11.00                                  *
;                   MPLINK  2.10.00                                  *
;                   MPASM   2.50.00                                  *
;                                                                    *
;*********************************************************************
;   Notes:                                                           *
;                                                                    *
;   This file is to be included in the i2ccomm.asm file              *
;                                                                    *
;*********************************************************************/

  #include  "flags.inc"          ; required include file

i2cSizeMask  EQU  0x0F

   GLOBAL   sflag_event          ; make variable viewable for other modules
   GLOBAL   eflag_event          ; make variable viewable for other modules
   GLOBAL   i2cState             ; make variable viewable for other modules
   GLOBAL   read_count           ; make variable viewable for other modules
   GLOBAL   write_count          ; make variable viewable for other modules
   GLOBAL   write_ptr            ; make variable viewable for other modules
   GLOBAL   read_ptr             ; make variable viewable for other modules
   GLOBAL   temp_address         ; make variable viewable for other modules

   GLOBAL   init_i2c             ; make function viewable for other modules
   GLOBAL   service_i2c          ; make function viewable for other modules

;*******    GENERAL PURPOSE VARIABLES
GPR_DATA   UDATA
sflag_event     RES     1        ; variable for i2c general status flags
eflag_event     RES     1        ; variable for i2c error status flags
i2cState        RES     1        ; I2C state machine variable
read_count      RES     1        ; variable used for slave read byte count
write_count     RES     1        ; variable used for slave write byte count
write_ptr       RES     1        ; variable used for pointer (writes to)
read_ptr        RES     1        ; variable used for pointer (reads from)
temp_address    RES     1        ; variable used for passing address to functions
```



```
;***********************************************************************
;                                                                      *
;   Additional notes on variable usage:                                *
;                                                                      *
;   The variables listed below are used within the function            *
;   service_i2c. These variables must be initialized with the          *
;   appropriate data from within the calling file. In this             *
;   application code the main file is 'mastri2c.asm'. This file        *
;   contains the function calls to service_i2c. It also contains       *
;   the function for initializing these variables, called 'init_vars'* 
;                                                                      *
;   To use the service_i2c function to read from and write to an       *
;   I2C slave device, information is passed to this function via       *
;   the following variables.                                           *
;                                                                      *
;   The following variables are used as function parameters:           *
;                                                                      *
;   read_count   - Initialize this variable for the number of bytes    *
;                  to read from the slave I2C device.                  *
;   write_count  - Initialize this variable for the number of bytes    *
;                  to write to the slave I2C device.                   *
;   write_ptr    - Initialize this variable with the address of the    *
;                  data string or data byte to write to the slave      *
;                  I2C device.                                         *
;   read_ptr     - Initialize this variable with the address of the    *
;                  location for storing data read from the slave I2C   *
;                  device.                                             *
;   temp_address - Initialize this variable with the address of the    *
;                  slave I2C device to communicate with.               *
;                                                                      *
;   The following variables are used as status or error events         *
;                                                                      *
;   sflag_event  - This variable is implemented for status or          *
;                  event flags. The flags are defined in the file      *
;                  'flags.inc'.                                        *
;   eflag_event  - This variable is implemented for error flags. The   *
;                  flags are defined in the file 'flags.inc'.          *
;                                                                      *
;   The following variable is used in the state machine jumnp table.   *
;                                                                      *
;   i2cState     - This variable holds the next I2C state to execute.* 
;                                                                      *
;***********************************************************************
```

APPENDIX F: GENERIC I²C SLAVE READ AND WRITE ROUTINES (ASSEMBLY)

```
;-----------------------------------------------------------------------
; File:             i2cslave.asm
;
; Written By:       Stephen Bowling, Microchip Technology
;
; Version:          1.00
;
; Assembled using Microchip Assembler
;
; Functionality:
;
; This code implements the basic functions for an I2C slave device
; using the SSP module.  All I2C functions are handled in an ISR.
; Bytes written to the slave are stored in a buffer.  After a number
; of bytes have been written, the master device can then read the
; bytes back from the buffer.
;
; Variables and Constants used in the program:
;
; The start address for the receive buffer is stored in the variable
; 'RXBuffer'.  The length of the buffer is denoted by the constant
; value 'RX_BUF_LEN'.  The current buffer index is stored in the
; variable 'Index'.
;
;-----------------------------------------------------------------------
;
; The following files should be included in the MPLAB project:
;
;  i2cslave.asm     --  Main source code file
;
;  16f872.lkr       --  Linker script file
;                       change this file for the device
;                       you are using)
;
;-----------------------------------------------------------------------
;-----------------------------------------------------------------------
; Include Files
;-----------------------------------------------------------------------

#include <p16f872.inc>              ; Change to device that you are using.


;-----------------------------------------------------------------------
;Constant Definitions
;-----------------------------------------------------------------------

#define  NODE_ADDR   0x02           ; I2C address of this node
                                    ; Change this value to address that
                                    ; you wish to use.
;-----------------------------------------------------------------------
; Buffer Length Definition
;-----------------------------------------------------------------------

#define   RX_BUF_LEN 32             ; Length of receive buffer

;-----------------------------------------------------------------------
; Variable declarations
;-----------------------------------------------------------------------
```



```
   udata_shr

WREGsave    res    1
STATUSsave  res    1
FSRsave     res    1
PCLATHsave  res    1

Index       res    1                ; Index to receive buffer
Temp        res    1                ;
RXBuffer    res    RX_BUF_LEN       ; Holds rec'd bytes from master
                                    ; device.

;-----------------------------------------------------------------------
; Vectors
;-----------------------------------------------------------------------

STARTUP  code
   nop
   goto     Startup                 ;
   nop                              ; 0x0002
   nop                              ; 0x0003
   goto     ISR                     ; 0x0004

PROG code

;-----------------------------------------------------------------------
; Macros
;-----------------------------------------------------------------------

memset      macro   Buf_addr,Value,Length

            movlw   Length          ; This macro loads a range of data memory
            movwf   Temp            ; with a specified value.  The starting
            movlw   Buf_addr        ; address and number of bytes are also
            movwf   FSR             ; specified.
SetNext     movlw   Value
            movwf   INDF
            incf    FSR,F
            decfsz  Temp,F
            goto    SetNext
            endm

LFSR        macro   Address,Offset  ; This macro loads the correct value
            movlw   Address         ; into the FSR given an initial data
            movwf   FSR             ; memory address and offset value.
            movf    Offset,W
            addwf   FSR,F
            endm

;-----------------------------------------------------------------------
; Main Code
;-----------------------------------------------------------------------

Startup
       bcf      STATUS,RP1
       bsf      STATUS,RP0
       call     Setup
Main   clrwdt                       ; Clear the Watchdog Timer.
       goto     Main                ; Loop forever.
```

```
;-----------------------------------------------------------------------
; Interrupt Code
;-----------------------------------------------------------------------

ISR
       movwf   WREGsave             ; Save WREG
       movf    STATUS,W             ; Get STATUS register
       banksel STATUSsave           ; Switch banks, if needed.
       movwf   STATUSsave           ; Save the STATUS register
       movf    PCLATH,W             ;
       movwf   PCLATHsave           ; Save PCLATH
       movf    FSR,W                ;
       movwf   FSRsave              ; Save FSR

       banksel PIR1
       btfss   PIR1,SSPIF           ; Is this a SSP interrupt?
       goto    $                    ; No, just trap here.
       bcf     PIR1,SSPIF
       call    SSP_Handler          ; Yes, service SSP interrupt.

       banksel FSRsave
       movf    FSRsave,W            ;
       movwf   FSR                  ; Restore FSR
       movf    PCLATHsave,W         ;
       movwf   PCLATH               ; Restore PCLATH
       movf    STATUSsave,W         ;
       movwf   STATUS               ; Restore STATUS
       swapf   WREGsave,F           ;
       swapf   WREGsave,W           ; Restore WREG
       retfie                       ; Return from interrupt.

;-----------------------------------------------------------------------
Setup
;
; Initializes program variables and peripheral registers.
;-----------------------------------------------------------------------

       banksel PCON
       bsf     PCON,NOT_POR
       bsf     PCON,NOT_BOR
       banksel Index                ; Clear various program variables
       clrf    Index
       clrf    PORTB
       clrf    PIR1
       banksel TRISB
       clrf    TRISB

       movlw   0x36                 ; Setup SSP module for 7-bit
       banksel SSPCON
       movwf   SSPCON               ; address, slave mode
       movlw   NODE_ADDR
       banksel SSPADD
       movwf   SSPADD
       clrf    SSPSTAT
       banksel PIE1                 ; Enable interrupts
       bsf     PIE1,SSPIE
       bsf     INTCON,PEIE          ; Enable all peripheral interrupts
       bsf     INTCON,GIE           ; Enable global interrupts
```



```
       bcf     STATUS,RP0
       return

;*******************************************************************
SSP_Handler
;*******************************************************************
;  The I2C code below checks for 5 states:
;*******************************************************************
;  State 1:         I2C write operation, last byte was an address byte.
;
;  SSPSTAT bits:    S = 1, D_A = 0, R_W = 0, BF = 1
;
;  State 2:         I2C write operation, last byte was a data byte.
;
;  SSPSTAT bits:    S = 1, D_A = 1, R_W = 0, BF = 1
;
;  State 3:         I2C read operation, last byte was an address byte.
;
;  SSPSTAT bits:    S = 1, D_A = 0, R_W = 1, BF = 0
;
;  State 4:         I2C read operation, last byte was a data byte.
;
;  SSPSTAT bits:    S = 1, D_A = 1, R_W = 1, BF = 0
;
;  State 5:         Slave I2C logic reset by NACK from master.
;
;  SSPSTAT bits:    S = 1, D_A = 1, R_W = 0, BF = 0
;
; For convenience, WriteI2C and ReadI2C functions have been used.
;*******************************************************************

       banksel SSPSTAT
       movf    SSPSTAT,W             ; Get the value of SSPSTAT
       andlw   b'  00101101 '        ; Mask out unimportant bits in SSPSTAT.
       banksel Temp                  ; Put masked value in Temp
       movwf   Temp                  ; for comparision checking.

State1:                              ; Write operation, last byte was an
       movlw   b'00001001 '          ; address, buffer is full.
       xorwf   Temp,W                ;
       btfss   STATUS,Z              ; Are we in State1?
       goto    State2                ; No, check for next state.....

       memset  RXBuffer,0,RX_BUF_LEN ; Clear the receive buffer.
       clrf    Index                 ; Clear the buffer index.
       call    ReadI2C               ; Do a dummy read of the SSPBUF.
               return


State2:                              ; Write operation, last byte was data,
       movlw   b'00101001 '          ; buffer is full.
       xorwf   Temp,W
       btfss   STATUS,Z              ; Are we in State2?
       goto    State3                ; No, check for next state.....

       LFSR    RXBuffer,Index        ; Point to the buffer.
       call    ReadI2C               ; Get the byte from the SSP.
       movwf   INDF                  ; Put it in the buffer.
       incf    Index,F               ; Increment the buffer pointer.
       movf    Index,W               ; Get the current buffer index.
```

```
       sublw   RX_BUF_LEN          ; Subtract the buffer length.
       btfsc   STATUS,Z            ; Has the index exceeded the buffer length?
       clrf    Index               ; Yes, clear the buffer index.
       return

State3:                            ; Read operation, last byte was an
       movlw   b'00001100 '        ; address, buffer is empty.
       xorwf   Temp,W
       btfss   STATUS,Z            ; Are we in State3?
       goto    State4              ; No, check for next state.....

       clrf    Index               ; Clear the buffer index.
       LFSR    RXBuffer,Index      ; Point to the buffer
       movf    INDF,W              ; Get the byte from buffer.
       call    WriteI2C            ; Write the byte to SSPBUF
       incf    Index,F             ; Increment the buffer index.
       return

State4:                            ; Read operation, last byte was data,
       movlw   b'00101100 '        ; buffer is empty.
       xorwf   Temp,W
       btfss   STATUS,Z            ; Are we in State4?
       goto    State5              ; No, check for next state....

       movf    Index,W             ; Get the current buffer index.
       sublw   RX_BUF_LEN          ; Subtract the buffer length.
       btfsc   STATUS,Z            ; Has the index exceeded the buffer length?
       clrf    Index               ; Yes, clear the buffer index.
       LFSR    RXBuffer,Index      ; Point to the buffer
       movf    INDF,W              ; Get the byte
       call    WriteI2C            ; Write to SSPBUF
       incf    Index,F             ; Increment the buffer index.
       return

State5:
       movlw   b'00101000 '        ; A NACK was received when transmitting
       xorwf   Temp,W              ; data back from the master.  Slave logic
       btfss   STATUS,Z            ; is reset in this case.  R_W = 0, D_A = 1
       goto    I2CErr              ; and BF = 0
       return                      ; If we aren't in State5, then something is
                                   ; wrong.

I2CErr nop
       banksel PORTB               ; Something went wrong!  Set LED
       bsf     PORTB,7             ; and loop forever.  WDT will reset
       goto    $                   ; device, if enabled.
       return

;*********************************************************************
; WriteI2C
;*********************************************************************

WriteI2C
       banksel SSPSTAT
       btfsc   SSPSTAT,BF          ; Is the buffer full?
       goto    WriteI2C            ; Yes, keep waiting.
       banksel SSPCON              ; No, continue.
DoI2CWrite
       bcf     SSPCON,WCOL         ; Clear the WCOL flag.
       movwf   SSPBUF              ; Write the byte in WREG
```



```
        btfsc   SSPCON,WCOL             ; Was there a write collision?
        goto    DoI2CWrite
        bsf     SSPCON,CKP              ; Release the clock.
        return

;-----------------------------------------------------------------------
ReadI2C
;-----------------------------------------------------------------------

        banksel SSPBUF
        movf    SSPBUF,W                ; Get the byte and put in WREG
        return

        end                             ; End of file
```

Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable”.
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

DNV Certification, Inc. USA — ANSI-RAB QMS ACCREDITED — DNV MSC The Netherlands Accredited by the RvA — DNV — ISO 9001 / QS-9000 REGISTERED FIRM

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999. The Company’s quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs and microperipheral products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001 certified.

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com

Rocky Mountain
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-7456

Atlanta
500 Sugar Mill Road, Suite 200B
Atlanta, GA 30350
Tel: 770-640-0034 Fax: 770-640-0307

Boston
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

Chicago
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423 Fax: 972-818-2924

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Kokomo
2767 S. Albright Road
Kokomo, Indiana 46902
Tel: 765-864-8360 Fax: 765-864-8387

Los Angeles
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

New York
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 631-273-5305 Fax: 631-273-5335

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
6285 Northam Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

ASIA/PACIFIC

Australia
Microchip Technology Australia Pty Ltd
Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733 Fax: 61-2-9868-6755

China - Beijing
Microchip Technology Consulting (Shanghai) Co., Ltd., Beijing Liaison Office
Unit 915
Bei Hai Wan Tai Bldg.
No. 6 Chaoyangmen Beidajie
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Chengdu
Microchip Technology Consulting (Shanghai) Co., Ltd., Chengdu Liaison Office
Rm. 2401, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-6766200 Fax: 86-28-6766599

China - Fuzhou
Microchip Technology Consulting (Shanghai) Co., Ltd., Fuzhou Liaison Office
Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506 Fax: 86-591-7503521

China - Shanghai
Microchip Technology Consulting (Shanghai) Co., Ltd.
Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel: 86-21-6275-5700 Fax: 86-21-6275-5060

China - Shenzhen
Microchip Technology Consulting (Shanghai) Co., Ltd., Shenzhen Liaison Office
Rm. 1315, 13/F, Shenzhen Kerry Centre,
Renminnan Lu
Shenzhen 518001, China
Tel: 86-755-2350361 Fax: 86-755-2366086

Hong Kong
Microchip Technology Hongkong Ltd.
Unit 901-6, Tower 2, Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200 Fax: 852-2401-3431

India
Microchip Technology Inc.
India Liaison Office
Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaugnessey Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax: 91-80-2290062

Japan
Microchip Technology Japan K.K.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471- 6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel: 65-334-8870 Fax: 65-334-8850

Taiwan
Microchip Technology Taiwan
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Denmark
Microchip Technology Nordic ApS
Regus Business Centre
Lautrup hoj 1-3
Ballerup DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France
Microchip Technology SARL
Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Microchip Technology GmbH
Gustav-Heinemann Ring 125
D-81739 Munich, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Italy
Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-039-65791-1 Fax: 39-039-6899883

United Kingdom
Arizona Microchip Technology Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44-118 921-5820

01/18/02