SILICON LABORATORIES

AN189

MMC DATA LOGGER EXAMPLE

Relevant Devices

This application note applies to the following devices:
C8051F320

1. Introduction

This application note describes a data logging system using the C8051F320, that stores temperature samples in a Multimedia Card (MMC). The software system is presented in two pieces:

- At the bottom level is the interface to the MMC which provides transparent Flash access for a user application such as the data logger.
- At the top level is the data logger which handles sampling, log table maintenance, and the user interface.

The main focus of this document is the interface between the Silicon Laboratories device and the MMC; the data logging portion is simply an application of this interface.

1.1. MMC Overview

The Multimedia Card provides a large amount of Flash memory that is accessed through a serial interface. There is a complicated command structure that must be followed when communicating with the card. The MMC interface section of this system encapsulates the complex command structure and provides access to the Flash as a large linear address space using simple read, write, and erase commands.

1.2. Data Logger Overview

The Data Logger section of the system contains a temperature sampling engine which generates a temperature log entry once every second. It uses the simple memory access functions provided by the MMC interface software to maintain a table of temperature log entries on the MMC. This section also provides a PC user interface through the UART, which allows the user to control several system options as well as to display and manipulate the log.

2. MMC Details

Multimedia cards are accessed serially using a low level command protocol that is defined in the MMC specification (http://mmca.org). The goal of the MMC interface provided in this system is to make the MMC command protocol transparent to the user. This is accomplished by providing simple access functions that handle the necessary MMC communication. This section of the application note describes the MMC SPI communication protocol as well as the MMC interface functions. However, before covering these topics, a short discussion of the MMC memory structure is necessary. Note that in order to fully understand the MMC operation, the reader must become familiar with the MMC specification.

2.1. MMC Memory Structure

MMC memory is divided into 512-byte sectors, much like a hard drive. These sectors are in turn organized into erase groups of 16 sectors each. The following list briefly describes SPI data operations in terms of the MMC structure:

- Block Read operations can be performed at any length ranging from 1 byte to the card specific maximum block size. The default block size is 512 bytes.
- Block Write operations have a minimum size requirement of one sector or 512 bytes and must be sector aligned.
- Erase operations can be performed at the sector level, the erase group level, or nearly any combination of the two. Erases are performed by tagging a start sector or group, tagging a stop sector or group, and then issuing the erase command. Note that any number of start tag/stop tag pairs may be selected before issuing the erase.

2.2. MMC Commands

There are two modes of communication that can be used with an MMC: “MMC mode” uses a unique serial protocol that has been defined specifically for MMC’s,

and “SPI mode” uses generic SPI transfers. The SPI protocol is ideal for this system because of the hardware SPI interface available on most Silicon Laboratories devices (See Figure 1 for the necessary SPI connections).

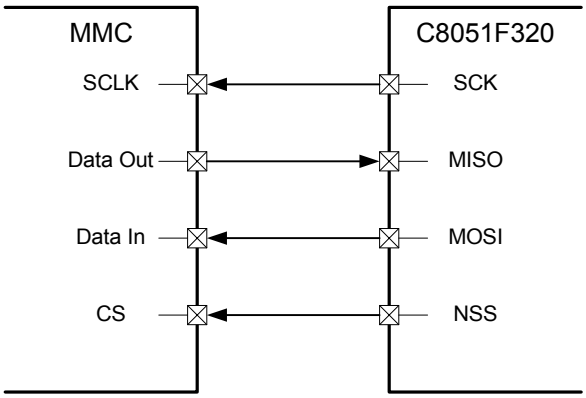

Figure 1. SPI Hardware Connection

2.2.1. Communication Format

SPI communication with the MMC takes the form of a command followed by a card response and any necessary data. This is shown in Figure 2. MMC Commands in SPI mode all have the same format. Each command has a total length of six bytes and contains three fields:

1. The first field is an 8-bit opcode that signals to the MMC what command is being issued.
2. The second field is the argument. It is a 32-bit field that contains a parameter such as an address, data length, or register value. If a command does not require an argument, this field contains all logic 1’s.
3. The last field is the CRC byte for the previous data fields. In SPI mode, CRC checking may be turned off, so for this system, the CRC field will consist of all 1’s.

After receiving a command, the MMC must send a response. This response has four possible formats depending on the command that was issued:

1. The “R1” response is a single byte consisting of basic error flags and a card state flag.
2. The “R1b” response is identical to the “R1” response, with the addition of a busy signal. The busy signal takes the form of one-byte tokens. These tokens have a zero value as long as the card is busy. When a non-zero value is returned, the card is ready to accept new commands.
3. The “R2” response is a two-byte value that contains the “R1” response plus some additional operational flags.
4. The “R3” response is a five-byte response that consists of the “R1” response plus the value of a four byte register (OCR) that describes the current operating conditions of the MMC.

Any necessary data transfer will occur after the card response. Data transfers consist of a Start token followed by a block of data. If the operation is a write, the MMC will send a data response after the data block. The data response indicates whether the write was successful or not. Read operations do not have a data response, however, if a read is unsuccessful an error token is returned in place of the data.

2.2.2. MMC Commands

The entire command set for the MMC is described in the MMC specification. This spec is available through the MMC Association web site (www.mmca.org) or from MMC manufacturers such as Sandisk or Hitachi.

A list of the commands used in this system as well as a brief description of each is shown in Table 1 .

2.3. MMC Interface Low Level Operation

There are two levels of abstraction applied in the MMC interface of this system. At the high level are simple memory access functions such as read, write, and erase that provide transparent MMC access. These functions break the memory accesses into a series of MMC command calls. The low level takes these command calls and generates the necessary SPI traffic to execute them on the MMC. The low level implementation consists of a single command execution function that accepts a MMC command and argument as parameters. This function uses a command description table to determine specific behavior for each command. Each entry in the command description table contains a command and information such as what the

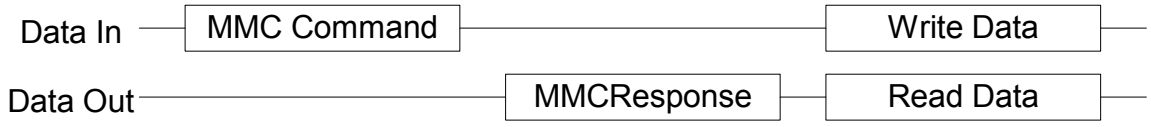

Figure 2. Communication Format

command type is (read, write, control), which card response is generated by the command, and whether the command requires an argument. The command execution state diagram is show in Figure 3.

The following steps describe the command execution function flow:

1. The command description table is accessed to retrieve specific operation data.
2. The one-byte command OpCode is transmitted over SPI. The OpCode value is found in the command table.
3. The next four transmitted bytes hold the command argument, or logic 1's if no argument is required. The command table specifies whether an argument is necessary.
4. The next byte carries the CRC data. CRC checking is disabled in SPI mode, so usually this byte can be logic 1's. However, after a reset, CMD0 is used to force the MMC into SPI mode, and therefore requires a valid CRC byte which is always equal to 0x95 for CMD0.
5. The MMC must generate a response for each command. The response format will depend on the command. The command table indicates which response is generated by each command.
6. At this point, any necessary data transfers must be performed. In a read operation, the MMC will send either a Start token followed by a data block or an Error token. In a write operation the MMC receives a Start token and a data block, followed by a data response byte. Some commands are control commands and do not require reading or writing. These commands are complete after the initial command / response phase.

Table 1. Command List

CMD INDEX	Abbreviation	Arg	Resp	Command Description
CMD0	GO_IDLE_STATE	None	R1	Reset MMC
CMD1	SEND_OP_COND	None	R1	Activate MMC initialization process
CMD9	SEND_CSD	None	R1	Get card-specific data
CMD10	SEND_CID	None	R1	Get card identification data
CMD13	SEND_STATUS	None	R2	Get card status register
CMD16	SET_BLOCKLEN	Block Length	R1	Set block length for data operations
CMD17	READ_SINGLE_BLOCK	Address	R1	Read a single data block
CMD24	WRITE_BLOCK	Address	R1	Write a single data block
CMD32	TAG_SECTOR_START	Address	R1	Set first sector for erase
CMD33	TAG_SECTOR_END	Address	R1	Set last sector for erase
CMD35	TAG_ERASE_GROUP_START	Address	R1	Set first group for erase
CMD36	TAG_ERASE_GROUP_END	Address	R1	Set last group for erase
CMD38	ERASE	None	R1b	Erase all selected groups and sectors

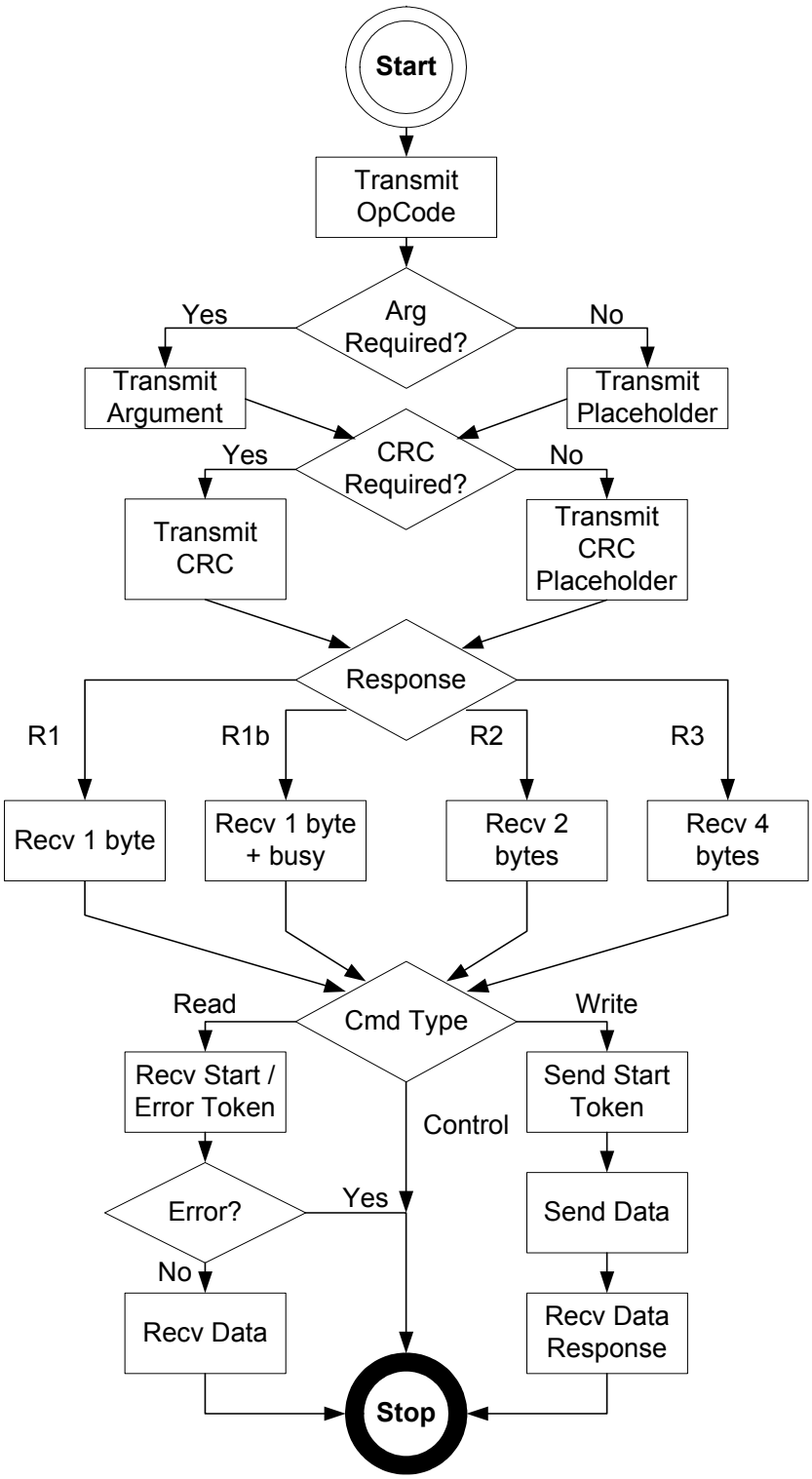

Figure 3. Command Execution Flowchart

See the comments of the command execution function *MMC_Command_Exec()* in the code for specific implementation details.

This lower level of the MMC interface provides a contact point between software and hardware. It presents a way for the high level MMC interface functions to issue commands to the MMC without concern over command format or SPI communication. The high level functions use a series of MMC command issues to perform memory accesses such as reads, writes, and erases.

2.4. MMC Interface High Level Operation

The high level of the MMC interface provides the user with simple memory functions to access the MMC. These functions take the user's memory request and break it into a series of MMC commands. These commands are then executed through the low level command execution function. Following is a list of memory operations along with a description of their implementation in this system. Note that read and write data lengths are limited to a 512 byte maximum in this implementation. Also note that operations on data spanning more than one data block require additional overhead. Specifically, data blocks must be operated on one at a time, so operations that span two blocks of data must be broken into two separate, sequential operations. Figure 4 shows desired data contained in one block vs. data that spans two blocks; this figure supplements the text and flow charts that describe the high level Flash routines that follow.

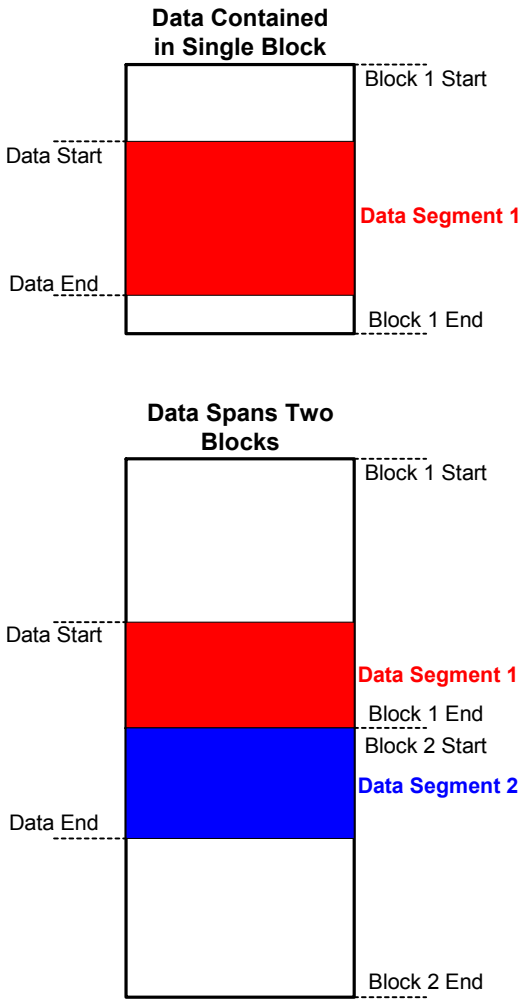

Figure 4. Data Alignment

2.4.1. Flash Initialization

Before the MMC can be used, it must be properly powered on and initialized. It must also be configured to SPI mode. Figure 5 and the following steps show the initialization sequence:

1. After receiving power, transmit at least 74 SPI clock cycles so that all internal start up operations can complete.
2. Drive the CS pin low.
3. Transmit a CMD0 to switch the card into SPI mode.
4. Transmit 8 SPI clock cycles.
5. Transmit a CMD1.
6. If the response to CMD1 indicates that the card is busy, go back to step 4. Otherwise, the card initialization is successful and the card can now receive and respond to commands.
7. Once the card is initialized, the size must be determined. This is done by retrieving the card specific data register (CSD) using CMD9 (SEND_CSD). The card size fields are located within this register. See the comments of the Flash initialization function on *MMC_FLASH_Init()* for specific implementation details.

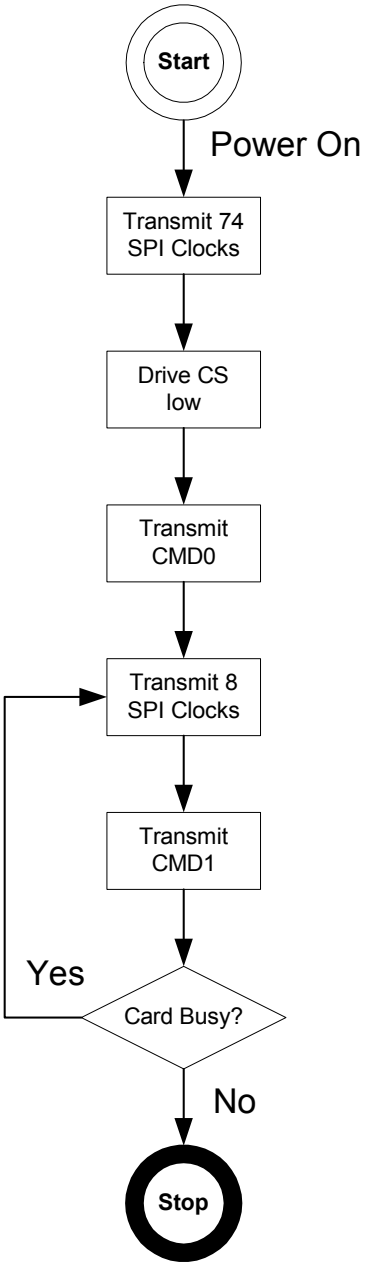

Figure 5. Flash Initialization

2.4.2. Flash Read

Reading data on the MMC is a simple operation. If the data to be read is contained within a single MMC block, then the only necessary actions are to set the MMC read block length and perform the read. If the data spans two MMC blocks, the read must be broken into two parts. The following steps and Figure 6 show how reads are implemented in this system.

1. Test the requested read length to see if it is greater than 512 bytes.
2. If data is contained within one MMC block, issue a set block length command followed by a read block command. If data spans two blocks, set the block length to the length of data segment 1 (see Figure 4), and then issue a read. Next, set the block length to the length of data segment 2, and then issue a second read.

See the comments of the Flash Read function *MMC_FLASH_Read()* for specific implementation details.

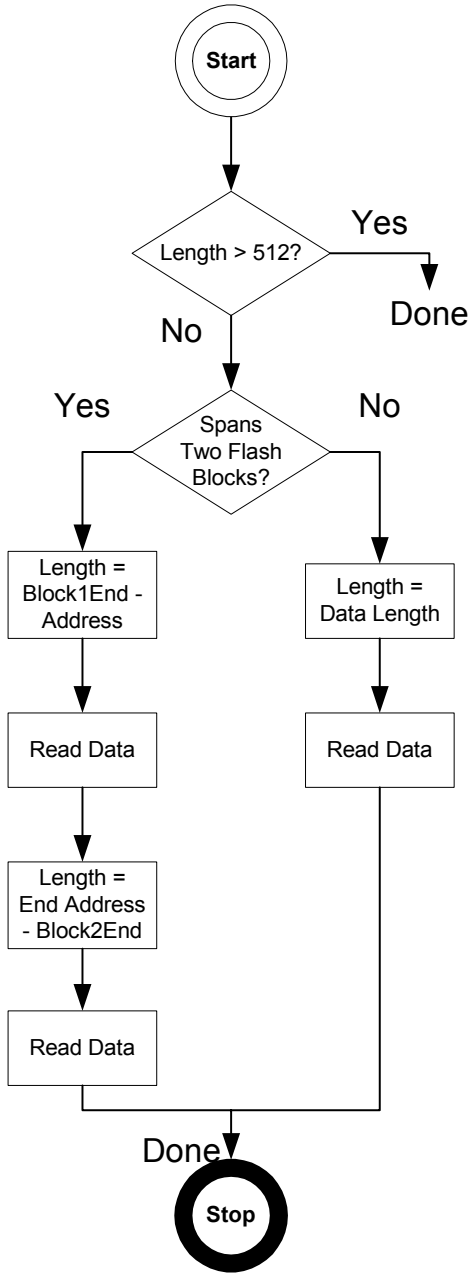

Figure 6. Flash Read Operation

2.4.3. Flash Clear

The smallest unit that can be erased by the MMC is a single 512 byte sector. As a result, erasing smaller arbitrary lengths of data requires some software manipulation. Figure 7 and the following steps illustrate the process:

1. Test the required clear length to see if it is less than 512 bytes.
2. If data is all contained in one block, read that block into local memory. If data spans two blocks, skip to step 6.
3. Clear the desired data in the local copy of the data block.
4. Tag and erase the block on the MMC.
5. Write the local copy of the block back to the MMC. The clear operation is complete at this point.
6. If data spans two blocks, the above steps 3 through 5 must be performed for each block. The only difference in execution will be step 3. For the first block, data must be cleared from the starting clear address to the end of the block. For the second block, data must be cleared from the start of the block to the end of the data (start address + length) to be cleared.

See the comments of the Flash Clear function *MMC_FLASH_Clear()* for specific implementation details.

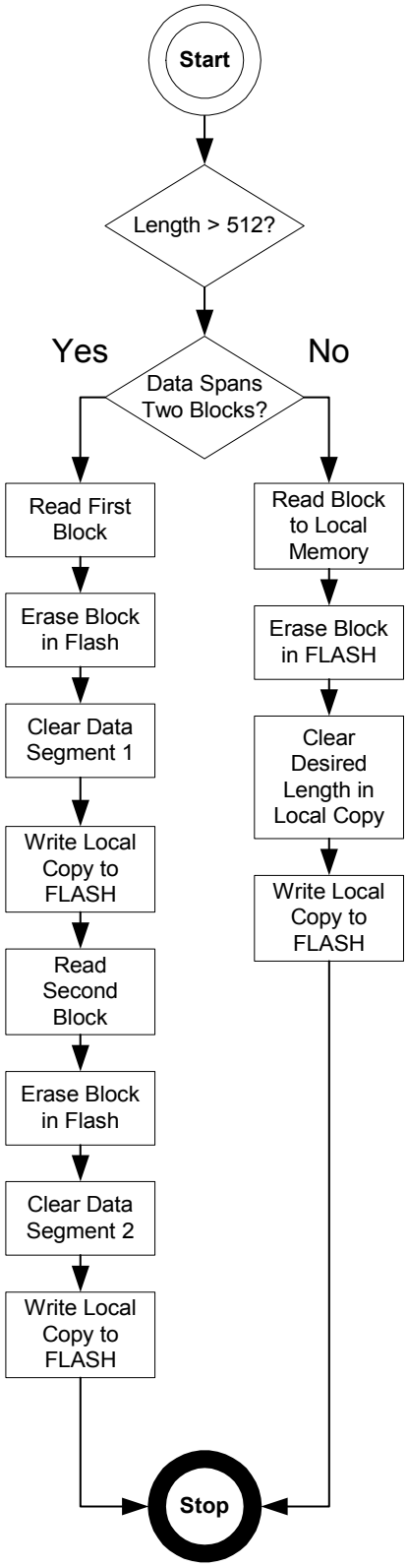

Figure 7. Flash Clear Operation

2.4.4. Flash Write

Like erases, MMC write operations have a minimum block size of 512 bytes. As a result, writes of an arbitrary length require some software overhead. Figure 8 and the following steps illustrate the write process:

1. Test the required write length to see if it is less than 512 bytes.
2. Clear the memory to be written using the Flash Clear operation.
3. If data is all contained within one block, read that block into local memory. If data spans two blocks, skip to step 6.
4. Perform the desired write on the local copy of the data block.
5. Write the local block back to the MMC. The write operation is complete at this point.
6. If the data spans two blocks, then steps 3, 4, and 5 must be modified slightly and executed for each block. For the first block, data should be written from the starting address to the end of the block. For the second block, data should be written from the beginning of the block to the ending address (start address + length).

See the comments of the Flash Write function *MMC_FLASH_Write()* for specific implementation details.

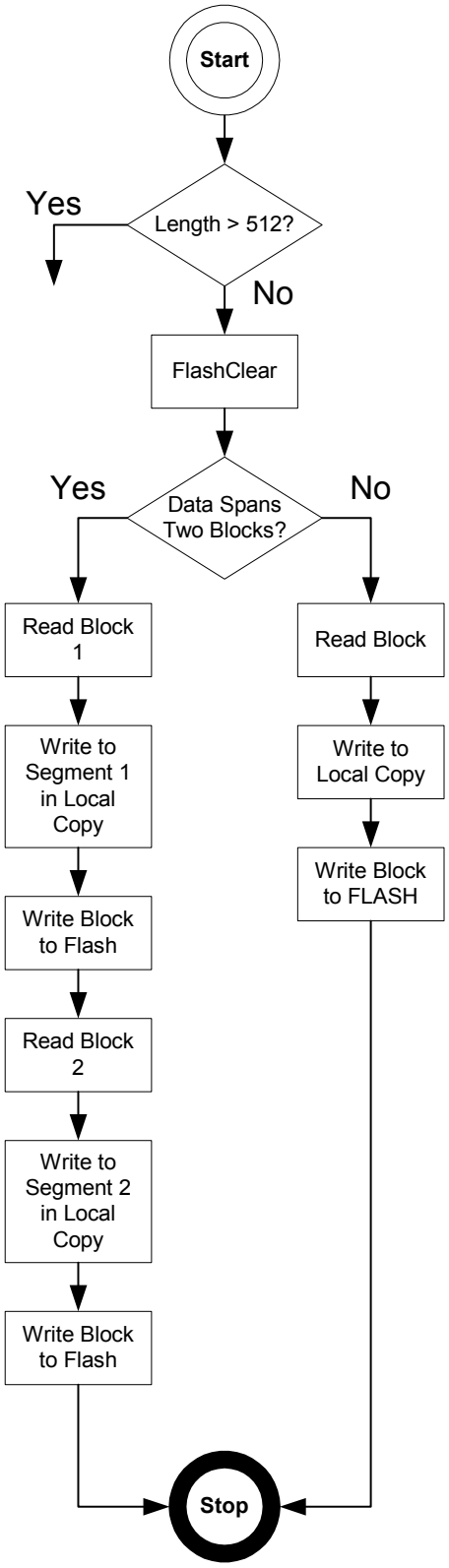

Figure 8. Flash Write Operation

2.4.5. Flash Mass Erase

It is often desirable to erase large portions of memory. This is best accomplished by using the erase group structure available in the MMC. Erase groups consist of 16 sectors each. An erase operation occurs by tagging sectors and groups for erase and then issuing an erase command (ERASE). The tagging process consists of a tag start sector (or group) command (TAG_SECTOR_START) and a tag stop sector command. All sectors between and including the Start and Stop tags will be erased on an Erase command. It is important to note that Start and Stop sector tags may not cross a group boundary. If sectors spanning two groups need be erased, the operation must be performed in two parts. Figure 9 and the steps below should illustrate the mass erase process:

1. Determine the starting and ending sector and group.
2. If the starting and ending group are the same, the erase operation will be wholly sector-based. Tag the Start and End sectors.
3. If the erase will span multiple groups and is not group aligned, the partial groups must be erased using sector tags. Tag the first sector to be erased and the last sector of the partial group.
4. Tag all whole groups to be erased.
5. Tag the first sector of the last partial group and the last sector to be erased.
6. Issue the Erase command.

See the comments of the Flash Mass Erase function *FLASH_Mass_Erase()* for specific implementation details.

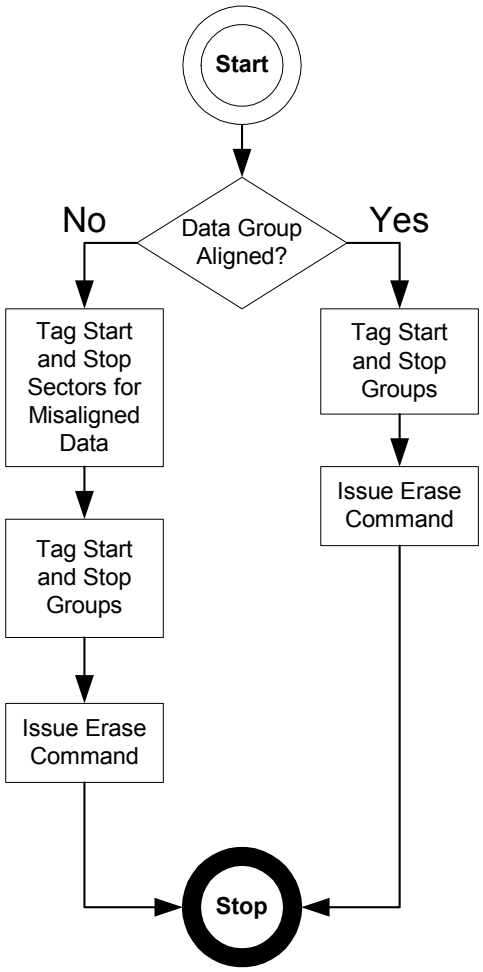

Figure 9. Flash Mass Erase Operation

3. Data Logger Details

The data logger portion of this system handles temperature sampling, log creation and maintenance, and the PC user interface. This section of the application note gives a brief introduction to the high-level data logger operation before describing the individual components in more detail.

3.1. Data Logger Operation

The data logger takes temperature samples and creates a record that holds the temperature and the time (days, hours, minutes, seconds) that the sample was taken. This record is stored in a table in Flash using the MMC interface routines. The PC interface allows the user to start and stop storing samples, display the log, erase the log, and initialize the clock.

3.2. Temperature Sampling

Temperature is measured using the on-chip temperature sensor. While temperature samples are only necessary once every second, the 10-bit ADC actually accumulates samples at 4096 Hz. Given that an oversampling by 4x yields an additional 1-bit of resolution, oversampling by 4096x will yield 6 additional bits. Therefore, when 4096 temperature samples have been taken, the accumulator is averaged to a value with 16-bits of effective resolution. This provides a temperature resolution to hundredths of a degree Celsius. The temperature sampling process builds a 7 byte structure that stores the temperature as well as the time stamp. These structures are stored in a table in Flash during log updates.

3.3. Logging Routines

There are several software routines used to maintain the Flash data log. The processes necessary for logging include initialization, size checking, erasing, printing, and updating. In this system, each of these processes is executed in a separate software routine. These routines are described below. Specific implementation details are described more thoroughly in the comments of the software.

3.3.1. Log Initialization

Each time a temperature sample is taken, it is stored in a structure with a timestamp. This structure is built in RAM before being buffered and stored in Flash. The log initialization routine simply clears out the locations in RAM that will be used for this structure. See the comments of the log initialization function *LogInit()* for specific implementation details.

3.3.2. Log Size Checking

Before any logging can occur, the current size of the log must be determined. The log size checking routine returns the number of table entries in Flash by reading them one by one and maintaining a count until the end of the table is reached. The current size is stored in a globally available variable for use by the other logging functions. See the comments of the log count finding function *LogFindCount()* for specific implementation details.

3.3.3. Log Erase

The log erase routine clears the entire Flash log table using the Flash mass erase routine. See the comments of the log erase function *LogErase()* for specific implementation details.

3.3.4. Log Print

The log print routine prints the entire log to the PC display through UART. The Flash read routine is used to read individual table entries from the log table until the number of entries read matches the globally available *PHYSICAL_SIZE* variable (returned by the log size checking routine.) See the comments of the log print function *LogPrint()* for specific implementation details.

3.3.5. Log Update

The *LogUpdate()* routine builds the table of temperature entries in Flash. Due to the complexities of writing to Flash memory, this is the most complicated of the logging routines. Although the Flash write routine available in the MMC interface is capable of handling single byte writes, it is much more efficient to write data in larger blocks. For this reason, the log update routine buffers 32 log entries (224 bytes) in XRAM memory before actually calling the Flash write function. While writing larger data blocks is more efficient, the buffering scheme introduces some issues that must be addressed.

Buffered data must be stored to Flash either when the buffer is full, or when the user stops the logging process. The process is slightly different for each situation. During normal operation, the log update routine converts a 16-bit ADC value to a temperature, adds the temperature entry to the buffer, prints the current sample data through UART, and writes the buffer to Flash if it is full. If the user has stopped the logging operation, only the buffer write takes place. A global state machine determines if the log update routine will follow normal operation or if the user has stopped operation and only a buffer write should occur. See the comments of the log update function *LogUpdate()* for specific implementation details.

4. Reduced RAM Implementation

Some applications may benefit from an MMC interface that requires less RAM. A reduced RAM implementation is available and included at the end of this document. This implementation maintains the temperature buffer as well as MMC page swap space in an off-chip EEPROM instead of on-chip RAM. The modifications to the software are described in the following paragraphs.

4.1. Temperature Buffer

As temperature entries are created, they are stored in an EEPROM buffer. When this buffer is full, it is written to the MMC.

4.2. MMC Data Operations

All MMC data operations access EEPROM space rather than the on-chip memory. MMC reads retrieve data from the MMC and store it in the EEPROM. The

microcontroller can then read any necessary data from the EEPROM. MMC writes store data from an EEPROM buffer in the MMC. MMC writes are required to be at least 512 bytes (MMC physical page size) in length, so 512 bytes of the EEPROM are used as scratch space to read the targeted page from the MMC, modify it with the EEPROM temperature buffer data, and write it back to the MMC.

At the command interpreter level, data is shuttled in small pieces between the MMC and the EEPROM. In an MMC read, a page is retrieved from the MMC, shuttled in small pieces through the microcontroller, and reconstructed in the EEPROM. The same process is used for a write, except data travels in the opposite direction.

4.3. EEPROM Communication

EEPROM communication is performed through SMBus. Data can be transferred either in single bytes or in arrays up to the length of one physical EEPROM page. The SMBus is configured for data transfers by the following functions:

- EEPROM_WriteByte()
- EEPROM_WriteArray()
- EEPROM_ReadByte()
- EEPROM_*ReadArray*()

The software comments for these functions and the SMBus interrupt service routine explain the operational details for EEPROM communications. Figure 10 shows the necessary hardware connections for the MMC and EEPROM.

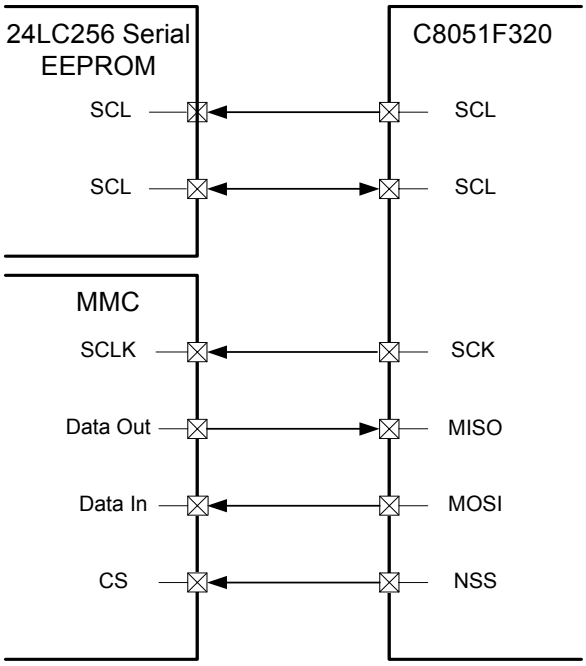

Figure 10. SPI and EEPROM Hardware Connection

APPENDIX—SOFTWARE

There are two MMC interface implementations included in this section. MMC_DataLogger.c uses on-chip ram for all buffers and scratch space while MMC_DataLogger_EEPROM.c uses an external EEPROM. Note that these software routines do not calculate the exact card capacity in bytes. For information on determining exact card capacity, see the CSD Register description in the MMC Specification. These software routines have been tested with SanDisk, Memorex, and Lexar MultiMedia Cards.

4.4. MMC_DataLogger.c

```
//-----------------------------------------------------------------------------
// MMC_DataLogger.c
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories
//
// AUTH: BW / JS / GV
// DATE: 08 MAR 04
//
// This program shows an example of a data logging application that maintains
// the log on an MMC card.
//
// Control Function:
//
// The system is controlled via the hardware UART, operating at a baud rate
// determined by the constant <BAUDRATE>, using Timer1 overflows as the baud
// rate source.  The commands are as follows (not case sensitive):
//  'c' - Clear Log
//  'd' - Display Log
//  'i' - Init RTC
//  'p' - Stop Logging
//  's' - Start Logging
//  '?' - List Commands
//
// Sampling Function:
//
// The ADC is configured to sample the on-chip temperature sensor at 4.096kHz,
// using Timer0 (in 8-bit auto-reload mode) as the start-of-conversion source.
// The ADC result is accumulated and decimated by a factor of 4096, yielding
// a 16-bit resolution quantitity from the original 10-bit sample at an
// effective output word rate of about 1Hz.  This decimated value is
// stored in the global variable <result>.
//
// A note about oversampling and averaging as it applies to this temp
// sensor example:  The transfer characteristic of the temp sensor on the
// 'F320 family of devices is 2.86mV/C.  The LSB size of the ADC using the
// internal VREF (2.43V) as its voltage reference is 2.3mV/code.
// This means that adjacent ADC codes are about ~1 degrees C apart.
//
// If we desire a higher temperature resolution, we can achieve it by
// oversampling and averaging (See AN118 on the Silicon Labs website).  For
// each additional bit of resolution required, we must oversample by a power
// of 4.  For example, increasing the resolution by 4 bits requires
// oversampling by a factor of 4^4, or 256.
//
// By what factor must we oversample to achieve a temperature resolution to
```

```
// the nearest hundredth of a degree C?  In other words, "How may bits of
// resolution do we need to add?"  The difference between 1 degrees C and
// 0.01 degrees C is a factor of 100 (100 is between 2^6 and 2^7, so we need
// somewhere between 6 and 7 more bits of resolution).  Choosing '6 bits',
// we calculate our oversampling ratio to be 4^6, or 4096.
//
// A note about accuracy:  oversampling and averaging provides a method to
// increase the 'resolution' of a measurement.  The technique does nothing
// to improve a measurement's 'accuracy'.  Just because we can measure a
// 0.01 degree change in temperature does not mean that the measurements
// are accurate to 0.01 degrees.  Some possible sources of inaccuracies in
// this system are:
//  1. manufacturing tolerances in the temperature sensor itself (transfer
//     characteristic variation)
//  2. VDD or VREF tolerance
//  3. ADC offset, gain, and linearity variations
//  4. Device self-heating
//
// Temperature Clock Function:
//
// The temperature clock maintains a record of days, hours, minutes, and
// seconds.  The current time record is stored with the temperature value
// in each log entry.  Clock updates are performed in the ADC end-of-conversion
// ISR at approximately once every second.
//
// Storage Function:
//
// MMC FLASH is used for storing the log entries.  Each entry contains
// the temperature in hundredths of a degree C, the day, hour, minute, and
// second that the reading was taken.  The LogUpdate function stores log
// entries in an external memory buffer and then writes that buffer out to the
// MMC when it is full.  Communication with the MMC is performed through the
// MMC access functions.  These functions provide transparent MMC access to
// the higher level functions (logging functions).  The MMC interface is broken
// into two pieces.  The high level piece consists of the user callable MMC
// access functions (MMC_FLASH_Read, MMC_FLASH_Write, MMC_FLASH_Clear,
// MMC_FLASH_MassErase).  These functions are called by the user to execute
// data operations on the MMC.  They break down the data operations into MMC
// commands.  The low level piece consists of a single command execution
// function (MMC_Command_Exec) which is called by the MMC data manipulation
// functions.  This function is called every time a command must be sent to the
// MMC.  It handles all of the required SPI traffic between the Silicon Labs
// device and the MMC.
//
// Target: C8051F32x
// Tool chain: KEIL C51 6.03 / KEIL EVAL C51
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <c8051f320.h>                  // SFR declarations
#include <stdio.h>                      // printf() and getchar()
#include <ctype.h>                      // tolower()

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F32x
//-----------------------------------------------------------------------------
```

```
sfr16 DP       = 0x82;                 // data pointer
sfr16 TMR2RL   = 0xca;                 // Timer2 reload value
sfr16 TMR2     = 0xcc;                 // Timer2 counter
sfr16 PCA0CP1  = 0xe9;                 // PCA0 Module 1 Capture/Compare
sfr16 PCA0CP2  = 0xeb;                 // PCA0 Module 2 Capture/Compare
sfr16 PCA0     = 0xf9;                 // PCA0 counter
sfr16 PCA0CP0  = 0xfb;                 // PCA0 Module 0 Capture/Compare
sfr16 ADC0     = 0xbd;                 // ADC0 Data

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
#define VERSION      "1.0"             // version identifier
#define TRUE         1
#define FALSE        0

#define START_SYSCLK 12000000
#define SYSCLK       START_SYSCLK * 2  // SYSCLK frequency in Hz
#define BAUDRATE     115200            // Baud rate of UART in bps
#define SAMPLE_RATE  4096              // Sample frequency in Hz
#define INT_DEC      4096              // integrate and decimate ratio
#define FULL_SCALE   65536             // Full scale ADC0 value
#define PREC_FACTOR  1024              // This constant is used to preserve
                                       // precision during temperature calc;
                                       // VREF offset constant used (.01 mV)
#define VREF         243000
                                       // in conversion of ADC sample to temp
                                       // value;
                                       // Temp sensor offset constant used
#define V_OFFSET     77600
                                       // in conversion of ADC sample to temp
                                       // value (.01 mV);
#define TEMP_SLOPE   2.86              // Temp sensor slope constant used
                                       // in conversion of ADC sample to temp
                                       // value;

// Constants that define available card sizes, 8MB through 128MB
#define PS_8MB       8388608L
#define PS_16MB      16777216L
#define PS_32MB      33554432L
#define PS_64MB      67108864L
#define PS_128MB     134217728L

// Physical size in bytes of one MMC FLASH sector
#define PHYSICAL_BLOCK_SIZE     512

// Erase group size = 16 MMC FLASH sectors
#define PHYSICAL_GROUP_SIZE     PHYSICAL_BLOCK_SIZE * 16

// Log table start address in MMC FLASH
#define LOG_ADDR     0x00000000

// Size in bytes for each log entry
#define LOG_ENTRY_SIZE sizeof(LOG_ENTRY)

#define BUFFER_ENTRIES 32

// Size of XRAM memory buffer that stores table entries
```

```
// before they are written to MMC
#define BUFFER_SIZE  LOG_ENTRY_SIZE * BUFFER_ENTRIES

// Command table value definitions
// Used in the MMC_Command_Exec function to
// decode and execute MMC command requests
#define     EMPTY  0
#define     YES   1
#define     NO    0
#define     CMD   0
#define     RD    1
#define     WR    2
#define     R1    0
#define     R1b   1
#define     R2    2
#define     R3    3

// Start and stop data tokens for single and multiple
// block MMC data operations
#define     START_SBR      0xFE
#define     START_MBR      0xFE
#define     START_SBW      0xFE
#define     START_MBW      0xFC
#define     STOP_MBW       0xFD

// Mask for data response token after an MMC write
#define     DATA_RESP_MASK 0x11

// Mask for busy token in R1b response
#define     BUSY_BIT       0x80

// Command Table Index Constants:
// Definitions for each table entry in the command table.
// These allow the MMC_Command_Exec function to be called with a
// meaningful parameter rather than a number.
#define     GO_IDLE_STATE            0
#define     SEND_OP_COND             1
#define     SEND_CSD                 2
#define     SEND_CID                 3
#define     STOP_TRANSMISSION        4
#define     SEND_STATUS              5
#define     SET_BLOCKLEN             6
#define     READ_SINGLE_BLOCK        7
#define     READ_MULTIPLE_BLOCK      8
#define     WRITE_BLOCK              9
#define     WRITE_MULTIPLE_BLOCK    10
#define     PROGRAM_CSD             11
#define     SET_WRITE_PROT          12
#define     CLR_WRITE_PROT          13
#define     SEND_WRITE_PROT         14
#define     TAG_SECTOR_START        15
#define     TAG_SECTOR_END          16
#define     UNTAG_SECTOR            17
#define     TAG_ERASE_GROUP_START   18
#define     TAG_ERASE_GROUP_END     19
#define     UNTAG_ERASE_GROUP       20
#define     ERASE                   21
#define     LOCK_UNLOCK             22
#define     READ_OCR                23
```

```
#define     CRC_ON_OFF              24


sbit LED = P2^2;                        // LED='1' means ON
sbit SW2 = P2^0;                        // SW2='0' means switch pressed
sbit TX0 = P0^4;                        // UART0 TX pin
sbit RX0 = P0^5;                        // UART0 RX pin


//-----------------------------------------------------------------------------
// UNIONs, STRUCTUREs, and ENUMs
//-----------------------------------------------------------------------------
typedef union LONG {                    // byte-addressable LONG
   long l;
   unsigned char b[4];
} LONG;

typedef union INT {                     // byte-addressable INT
   int i;
   unsigned char b[2];
} INT;

typedef union {                         // byte-addressable unsigned long
      unsigned long l;
      unsigned char b[4];
              } ULONG;

typedef union {                         // byte-addressable unsigned int
      unsigned int i;
      unsigned char b[2];
              } UINT;

typedef struct LOG_ENTRY {              // (7 bytes per entry)
   int wTemp;                           // temperature in hundredths of a
                                        // degree
   unsigned int uDay;                   // day of entry
   unsigned char bHour;                 // hour of entry
   unsigned char bMin;                  // minute of entry
   unsigned char bSec;                  // second of entry
   unsigned char pad;                   // dummy byte to ensure aligned access;
} LOG_ENTRY;

// The states listed below represent various phases of
// operation;
typedef enum STATE {
   RESET,                               // Device reset has occurred;
   RUNNING,                             // Data is being logged normally;
   FINISHED,                            // Logging stopped, store buffer;
   STOPPED                              // Logging completed, buffer stored;
   } STATE;

// This structure defines entries into the command table;
typedef struct {
      unsigned char command_byte;       // OpCode;
      unsigned char arg_required;       // Indicates argument requirement;
      unsigned char CRC;                // Holds CRC for command if necessary;
      unsigned char trans_type;         // Indicates command transfer type;
      unsigned char response;           // Indicates expected response;
      unsigned char var_length;         // Indicates varialble length transfer;
```

```
                } COMMAND;

// Command table for MMC.  This table contains all commands available in SPI
// mode;  Format of command entries is described above in command structure
// definition;
COMMAND code commandlist[25] = {
      { 0,NO ,0x95,CMD,R1 ,NO },    // CMD0;  GO_IDLE_STATE: reset card;
      { 1,NO ,0xFF,CMD,R1 ,NO },    // CMD1;  SEND_OP_COND: initialize card;
      { 9,NO ,0xFF,RD ,R1 ,NO },    // CMD9;  SEND_CSD: get card specific data;
      {10,NO ,0xFF,RD ,R1 ,NO },    // CMD10; SEND_CID: get card identifier;
      {12,NO ,0xFF,CMD,R1 ,NO },    // CMD12; STOP_TRANSMISSION: end read;
      {13,NO ,0xFF,CMD,R2 ,NO },    // CMD13; SEND_STATUS: read card status;
      {16,YES,0xFF,CMD,R1 ,NO },    // CMD16; SET_BLOCKLEN: set block size;
      {17,YES,0xFF,RD ,R1 ,NO },    // CMD17; READ_SINGLE_BLOCK: read 1 block;
      {18,YES,0xFF,RD ,R1 ,YES},    // CMD18; READ_MULTIPLE_BLOCK: read > 1;
      {24,YES,0xFF,WR ,R1 ,NO },    // CMD24; WRITE_BLOCK: write 1 block;
      {25,YES,0xFF,WR ,R1 ,YES},    // CMD25; WRITE_MULTIPLE_BLOCK: write > 1;
      {27,NO ,0xFF,CMD,R1 ,NO },    // CMD27; PROGRAM_CSD: program CSD;
      {28,YES,0xFF,CMD,R1b,NO },    // CMD28; SET_WRITE_PROT: set wp for group;
      {29,YES,0xFF,CMD,R1b,NO },    // CMD29; CLR_WRITE_PROT: clear group wp;
      {30,YES,0xFF,CMD,R1 ,NO },    // CMD30; SEND_WRITE_PROT: check wp status;
      {32,YES,0xFF,CMD,R1 ,NO },    // CMD32; TAG_SECTOR_START: tag 1st erase;
      {33,YES,0xFF,CMD,R1 ,NO },    // CMD33; TAG_SECTOR_END: tag end(single);
      {34,YES,0xFF,CMD,R1 ,NO },    // CMD34; UNTAG_SECTOR: deselect for erase;
      {35,YES,0xFF,CMD,R1 ,NO },    // CMD35; TAG_ERASE_GROUP_START;
      {36,YES,0xFF,CMD,R1 ,NO },    // CMD36; TAG_ERASE_GROUP_END;
      {37,YES,0xFF,CMD,R1 ,NO },    // CMD37; UNTAG_ERASE_GROUP;
      {38,YES,0xFF,CMD,R1b,NO },    // CMD38; ERASE: erase all tagged sectors;
      {42,YES,0xFF,CMD,R1b,NO },    // CMD42; LOCK_UNLOCK;
      {58,NO ,0xFF,CMD,R3 ,NO },    // CMD58; READ_OCR: read OCR register;
      {59,YES,0xFF,CMD,R1 ,NO }    // CMD59; CRC_ON_OFF: toggles CRC checking;
                               };

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

xdata LONG Result = {0L};              // ADC0 decimated value

xdata LOG_ENTRY LogRecord;             // Memory space for each log entry
xdata unsigned long uLogCount;         // Current number of table entries
LOG_ENTRY xdata *pLogTable;            // Pointer to buffer for table entries
xdata STATE State = RESET;             // System state variable;  Determines
                                       // how log update function will exec;
xdata unsigned long PHYSICAL_SIZE;     // MMC size variable;  Set during
                                       // initialization;
xdata unsigned long LOG_SIZE;          // Available number of bytes for log
                                       // table;

xdata unsigned long PHYSICAL_BLOCKS;   // MMC block number;  Computed during
                                       // initialization;

xdata char LOCAL_BLOCK[BUFFER_SIZE];
xdata char SCRATCH_BLOCK[PHYSICAL_BLOCK_SIZE];

xdata char error;
//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
```

```
void main (void);

// Support Subroutines
void MENU_ListCommands (void);          // Outputs user menu choices via UART


// Logging Subroutines
void LogUpdate (void);                  // Builds MMC log table
unsigned long LogFindCount();           // Returns current number of log entries
void LogErase (void);                   // Erases entire log table
void LogPrint (void);                   // Prints log through UART
void LogInit (LOG_ENTRY *pEntry);       // Initializes area for building entries

// High Level MMC_FLASH Functions

void MMC_FLASH_Init (void);             // Initializes MMC and configures it to
                                        // accept SPI commands;

                                        // Reads <length> bytes starting at
                                        // <address> and stores them at <pchar>;
unsigned char MMC_FLASH_Read (unsigned long address, unsigned char *pchar,
                          unsigned int length);

                                        // Clears <length> bytes starting at
                                        // <address>; uses memory at <scratch>
                                        // for temporary storage;
unsigned char MMC_FLASH_Clear (unsigned long address, unsigned char *scratch,
                           unsigned int length);

                                        // Writes <length> bytes of data at
                                        // <wdata> to <address> in MMC;
                                        // <scratch> provides temporary storage;
unsigned char MMC_FLASH_Write (unsigned long address, unsigned char *scratch,
                           unsigned char *wdata, unsigned int length);

                                        // Clears <length> bytes of FLASH
                                        // starting at <address1>; Requires that
                                        // desired erase area be sector aligned;
unsigned char MMC_FLASH_MassErase (unsigned long address1,
                                   unsigned long length);

// Low Level MMC_FLASH_ Functions

                                        // Decodes and executes MMC commands;
                                        // <cmd> is an index into the command
                                        // table and <argument> contains a
                                        // 32-bit argument if necessary;  If a
                                        // data operation is taking place, the
                                        // data will be stored to or read from
                                        // the location pointed to by <pchar>;
unsigned int MMC_Command_Exec (unsigned char cmd, unsigned long argument,
                           unsigned char *pchar);


// Initialization Subroutines

void SYSCLK_Init (void);
void PORT_Init (void);
```

```
void UART0_Init (void);
void ADC0_Init (void);
void Soft_Init (void);
void Timer0_Init (int counts);
void Timer2_Init (int counts);
void SPI_Init (void);

// Interrupt Service Routines

void ADC0_ISR (void);
void Soft_ISR (void);

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

void main (void) {
   idata char key_press;               // Input character from UART;
   // Disable Watchdog timer
   PCA0MD &= ~0x40;                    // WDTE = 0 (clear watchdog timer
                                       // enable);

   PORT_Init ();                       // Initialize crossbar and GPIO;
   SYSCLK_Init ();                     // Initialize oscillator;
   UART0_Init ();                      // Initialize UART0;
   SPI_Init ();                        // Initialize SPI0;
   Timer2_Init (SYSCLK/SAMPLE_RATE);   // Init Timer2 for 16-bit autoreload;
   ADC0_Init ();                       // Init ADC0;
   Soft_Init ();                       // Initialize software interrupts;
   MMC_FLASH_Init();                   // Initialize MMC card;
   AD0EN = 1;                          // enable ADC0;

   State = RESET;                      // Set global state machine to reset
                                       // state;

                                       // Initialize log table buffer pointer
   pLogTable = (LOG_ENTRY xdata *)LOCAL_BLOCK;
   uLogCount = LogFindCount();         // Find current number of log table
                                       // entries;

   printf ("\n");                      // Print list of commands;
   MENU_ListCommands ();

   State = STOPPED;                    // Global state is STOPPED; no data
                                       // is being logged;
   EA = 1;                             // Enable global interrupts;

    while (1)                          // Serial port command decoder;
   {
      key_press = getchar();           // Get command character;
      key_press = tolower(key_press);  // Convert to lower case;

      switch (key_press)
      {
         case 'c':                     // Clear log;
           if(State == STOPPED)        // Only execute if not logging;
           {
            printf ("\n Clear Log\n");
            LogErase();                // erase log entries;
```

```
            uLogCount = LogFindCount();// update global log entry count;
           }
           break;
         case 'd':                    // Display log;
           if(State == STOPPED)       // Only execute if not logging;
           {
            printf ("\n Display Log\n");
            LogPrint();               // Print the log entries;
           }
           break;
         case 'i':                    // Init RTC;
           if(State == STOPPED)       // Only execute if not logging;
           {
            printf ("\n Init RTC values\n");
            EA = 0;                   // Disable interrupts;
            LogInit(&LogRecord);      // Clear current time;
            EA = 1;                   // Reenable interrupts;
           }
           break;
         case 'p':                    // Stop logging;
           if(State != STOPPED)       // Only execute if not stopped already;
           {
            State = FINISHED;         // Set state to FINISHED
            printf ("\n Stop Logging\n");
            while(State != STOPPED){} // Wait for State = STOPPED;
           }
           break;
         case 's':                    // Start logging
           if(State == STOPPED)       // Only execute if not logging;
           {
            printf ("\n Start Logging\n");
            State = RUNNING;          // Start logging data
           }
           break;
         case '?':                    // List commands;
           if(State == STOPPED)       // Only execute if not logging;
           {
            printf ("\n List Commands\n");
            MENU_ListCommands();      // List Commands
           }
           break;
         default:                     // Indicate unknown command;
           if(State == STOPPED)       // Only execute if not logging;
           {
            printf ("\n Unknown command: '%x'\n", key_press);
            MENU_ListCommands();      // Print Menu again;
           }
           break;
      } // switch
   } // while
}

//-----------------------------------------------------------------------------
// Support Subroutines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// MENU_ListCommands
//-----------------------------------------------------------------------------
```

```
// This routine prints a list of available commands.
//
void MENU_ListCommands (void)
{
   printf ("\nData logging example version %s\n", VERSION);
   printf ("Copyright 2004 Silicon Laboratories.\n\n");
   printf ("Command List\n");
   printf ("============================================\n");
   printf (" 'c' - Clear Log\n");
   printf (" 'd' - Display Log\n");
   printf (" 'i' - Init RTC\n");
   printf (" 'p' - Stop Logging\n");
   printf (" 's' - Start Logging\n");
   printf (" '?' - List Commands\n");
   printf ("\n");
}

//-----------------------------------------------------------------------------
// Logging Subroutines
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
// LogUpdate()
//-----------------------------------------------------------------------------
// This routine is called by the ADC ISR at ~1Hz if State == RUNNING or
// FINISHED.  Here we read the decimated ADC value, convert it to temperature
// in hundredths of a degree C, and add the log entry to the log table buffer.
// If the buffer is full, or the user has stopped the logger, we must commit
// the buffer to the MMC FLASH.  <State> determines if the system is logging
// normally (State == RUNNING), or if the user has stopped the logging
// process (State == FINISHED).
//
//
void LogUpdate (void)
{
   idata ULONG voltage;                // Long voltage value;
   idata int temp_int, temp_frac;      // Integer and fractional portions of
                                       // Temperature;
                                       // Count variable for number of
                                       // Log entries in local buffer;
   static idata unsigned int lLogCount = 0;

   EA = 0;                             // Disable interrupts (precautionary);
   voltage.l = Result.l;               // Retrieve 32-bit ADC value;
   EA = 1;                             // Re-enable interrupts;
                                       // Calculate voltage in .01 millivolt;
                                       // units;
   voltage.l = voltage.l * ((VREF*PREC_FACTOR) / FULL_SCALE / TEMP_SLOPE);
                                       // Handle temp sensor voltage offset;
   voltage.l = voltage.l - ((V_OFFSET*PREC_FACTOR / TEMP_SLOPE));
   voltage.b[4] = voltage.b[3];        // Scale down by PREC_FACTOR with a
   voltage.b[3] = voltage.b[2];        // 10-bit shift;  <voltage> now contains
   voltage.b[2] = voltage.b[1];        // temperature value;
   voltage.b[1] = voltage.b[0];
   voltage.b[0] = 0;
   voltage.l = voltage.l >> 2;
   LogRecord.wTemp = (int)voltage.l;   // Store temp value in temporary log
                                       // entry;
```

```
if(uLogCount == 0)                  // If the FLASH table has been cleared,
   {                                // The local buffer is reset;
   lLogCount = 0;                   // Reset number of local table entries;
                                    // Reset local buffer pointer;
   pLogTable = (LOG_ENTRY xdata *)LOCAL_BLOCK;
   }
if(State == RUNNING)                // Execute the following if the logger
{                                   // is logging normally;
                                    // Check to see if the log table is
                                    // full;
   if ((uLogCount*LOG_ENTRY_SIZE) < LOG_SIZE)
   {

      *pLogTable = LogRecord;       // Copy temporary log entry to buffer;
      pLogTable++;                  // Increment buffer pointer;
      lLogCount++;                  // Increment local log entry count;
      uLogCount++;                  // Increment global log entry count;
                                    // If the buffer is full, it must be
                                    // written to FLASH;
      if(lLogCount == (unsigned int)(BUFFER_SIZE / LOG_ENTRY_SIZE))
      {
                                    // Call FLASH Write function;  Write to
                                    // address pointed at by the global
                                    // entry count less the local buffer
                                    // count;
         MMC_FLASH_Write((uLogCount -
                         (unsigned long)lLogCount)*LOG_ENTRY_SIZE,
                         (unsigned char xdata * )SCRATCH_BLOCK,
                         (unsigned char xdata *)LOCAL_BLOCK, BUFFER_SIZE);
         lLogCount = 0;             // Reset the local buffer size
                                    // and pointer;
         pLogTable = (LOG_ENTRY xdata *)LOCAL_BLOCK;
      }
                                    // Update display;
      temp_int = LogRecord.wTemp / 100;
      temp_frac = LogRecord.wTemp - ((long) temp_int * 100L);

   printf (" %08lu\t", uLogCount);
      printf ("%02u: ",(unsigned)LogRecord.uDay);
      printf ("%02u:",(unsigned) LogRecord.bHour);
      printf ("%02u:",(unsigned) LogRecord.bMin);
      printf ("%02u ",(unsigned) LogRecord.bSec);
      printf ("%+02d.%02d\n", temp_int, temp_frac);


   }

   else                             // If the FLASH table is full, stop
   {                                // logging data and print the full
      State = STOPPED;              // message;
      printf ("Log is full\n");
   }
}
else if(State == FINISHED)          // If the data logger has been stopped
{                                   // by the user, write the local buffer
                                    // to FLASH;
   MMC_FLASH_Write((uLogCount - (unsigned long)lLogCount)*LOG_ENTRY_SIZE,
                                   (unsigned char xdata * )SCRATCH_BLOCK,
```

```
                                    (unsigned char xdata *)LOCAL_BLOCK,
                                    lLogCount*LOG_ENTRY_SIZE);
      lLogCount = 0;                    // Reset the local buffer size;
                                        // and pointer;
      pLogTable = (LOG_ENTRY xdata *)LOCAL_BLOCK;
      State = STOPPED;                  // Set the state to STOPPED;
   }
}

//-----------------------------------------------------------------------------
// LogFindCount()
//-----------------------------------------------------------------------------
// This function finds the number of entries already stored in the MMC log;
//
unsigned long LogFindCount()
{
   unsigned long Count = 0;             // Count variable, incremented as table
                                        // entries are read;
   unsigned long i = 0;                 // Address variable, used to read table
                                        // table entries from FLASH;
   LOG_ENTRY xdata *TempEntry;          // Temporary log entry space;

                                        // Initialize temp space in
                                        // SCRATCH_BLOCK of external memory;
   TempEntry = (LOG_ENTRY xdata *)SCRATCH_BLOCK;

                                        // Loop through the table looking for a
                                        // blank entry;
   for (i=LOG_ADDR;i<LOG_SIZE;i += LOG_ENTRY_SIZE)
   {
                                        // Read one entry from address i of
                                        // FLASH;
      MMC_FLASH_Read((unsigned long)(i),(unsigned char xdata *) SCRATCH_BLOCK,
         (unsigned int)LOG_ENTRY_SIZE);

                                        // Check if entry is blank;
      if ((TempEntry->bSec == 0x00)&&(TempEntry->bMin == 0x00)
           && (TempEntry->bHour == 0x00))
      {
                                        // If entry is blank, set Count;
         Count = (i/LOG_ENTRY_SIZE) - LOG_ADDR;
         break;                         // Break out of loop;
      }
   }
   return Count;                        // Return entry count;
}

//-----------------------------------------------------------------------------
// LogErase
//-----------------------------------------------------------------------------
// This function clears the log table using the FLASH Mass Erase capability.
//
void LogErase (void)
{
                                        // Call Mass Erase function with start
                                        // of table as address and log size as
                                        // length;
   MMC_FLASH_MassErase(LOG_ADDR, LOG_SIZE);
   uLogCount = 0;                       // Reset global count;
```

```
}

//-----------------------------------------------------------------------------
// LogPrint
//-----------------------------------------------------------------------------
// This function prints the log table.  Entries are read one at a time, temp
// is broken into the integer and fractional portions, and the log entry is
// displayed on the PC through UART.
//
void LogPrint (void)
{
   idata long temp_int, temp_frac;     // Integer and fractional portions of
                                       // temperature;
   idata unsigned long i;              // Log index;
   unsigned char xdata *pchar;         // Pointer to external mem space for
                                       // FLASH Read function;
   LOG_ENTRY xdata *TempEntry;


   printf ("Entry#\tTime\t\tResult\n");// Print display column headers;
                                       // Assign pointers to local block;
                                       // FLASHRead function stores incoming
                                       // data at pchar, and then that data can
                                       // be accessed as log entries through
                                       // TempEntry;
   pchar = (unsigned char xdata *)LOCAL_BLOCK;
   TempEntry = (LOG_ENTRY xdata *)LOCAL_BLOCK;

   for (i = 0; i < uLogCount; i++)     // For each entry in the table,
   {                                   // do the following;
                                       // Read the entry from FLASH;
      MMC_FLASH_Read((unsigned long)(LOG_ADDR + i*LOG_ENTRY_SIZE), pchar,
         (unsigned int)LOG_ENTRY_SIZE);

      // break temperature into integer and fractional components
      temp_int = (long) (TempEntry->wTemp) / 100L;
      temp_frac = (long) (TempEntry->wTemp) - ((long) temp_int * 100L);

      // display log entry
      printf (" %lu\t%03u: %02u:%02u:%02u ", (i + 1),
         TempEntry->uDay, (unsigned) TempEntry->bHour,
         (unsigned) TempEntry->bMin,
         (unsigned) TempEntry->bSec);
      printf ("%+02ld.%02ld\n", temp_int, temp_frac);

   }
}


//-----------------------------------------------------------------------------
// LogInit
//-----------------------------------------------------------------------------
// Initialize the Log Entry space (all zeros);
//
void LogInit (LOG_ENTRY *pEntry)
{
   pEntry->wTemp = 0;
   pEntry->uDay = 0;
   pEntry->bHour = 0;
```

```
   pEntry->bMin = 0;
   pEntry->bSec = 0;
}


//-----------------------------------------------------------------------------
// MMC_Command_Exec
//-----------------------------------------------------------------------------
//
// This function generates the necessary SPI traffic for all MMC SPI commands.
// The three parameters are described below:
//
// cmd:      This parameter is used to index into the command table and read
//           the desired command.  The Command Table Index Constants allow the
//           caller to use a meaningful constant name in the cmd parameter
//           instead of a simple index number.  For example, instead of calling
//           MMC_Command_Exec (0, argument, pchar) to send the MMC into idle
//           state, the user can call
//           MMC_Command_Exec (GO_IDLE_STATE, argument, pchar);
//
// argument: This parameter is used for MMC commands that require an argument.
//           MMC arguments are 32-bits long and can be values such as an
//           an address, a block length setting, or register settings for the
//           MMC.
//
// pchar:    This parameter is a pointer to the local data location for MMC
//           data operations.  When a read or write occurs, data will be stored
//           or retrieved from the location pointed to by pchar.
//
// The MMC_Command_Exec function indexes the command table using the cmd
// parameter. It reads the command table entry into memory and uses information
// from that entry to determine how to proceed.  Returns the 16-bit card
// response value;
//

unsigned int MMC_Command_Exec (unsigned char cmd, unsigned long argument,
                           unsigned char *pchar)
{
   idata COMMAND current_command;      // Local space for the command table
                                       // entry;
   idata ULONG long_arg;               // Union variable for easy byte
                                       // transfers of the argument;
                                       // Static variable that holds the
                                       // current data block length;
   static unsigned long current_blklen = 512;
   unsigned long old_blklen = 512;     // Temp variable to preserve data block
                                       // length during temporary changes;
   idata unsigned int counter = 0;     // Byte counter for multi-byte fields;
   idata UINT card_response;           // Variable for storing card response;
   idata unsigned char data_resp;      // Variable for storing data response;
   idata unsigned char dummy_CRC;      // Dummy variable for storing CRC field;


   current_command = commandlist[cmd]; // Retrieve desired command table entry
                                       // from code space;
   SPI0DAT = 0xFF;                     // Send buffer SPI clocks to ensure no
   while(!SPIF){}                      // MMC operations are pending;
   SPIF = 0;
   NSSMD0 = 0;                         // Select MMC by pulling CS low;
```

```
SPI0DAT = 0xFF;                     // Send another byte of SPI clocks;
while(!SPIF){}
SPIF = 0;
                                    // Issue command opcode;
SPI0DAT = (current_command.command_byte | 0x40);
long_arg.l = argument;              // Make argument byte addressable;
                                    // If current command changes block
                                    // length, update block length variable
                                    // to keep track;
                                    // Command byte = 16 means that a set
                                    // block length command is taking place
                                    // and block length variable must be
                                    // set;
if(current_command.command_byte == 16)
{
   current_blklen = argument;
}
                                    // Command byte = 9 or 10 means that a
                                    // 16-byte register value is being read
                                    // from the card, block length must be
                                    // set to 16 bytes, and restored at the
                                    // end of the transfer;
if((current_command.command_byte == 9)||
   (current_command.command_byte == 10))
{
   old_blklen = current_blklen;     // Command is a GET_CSD or GET_CID,
   current_blklen = 16;             // set block length to 16-bytes;
}
while(!SPIF){}                      // Wait for initial SPI transfer to end;
SPIF = 0;                           // Clear SPI Interrupt flag;

                                    // If an argument is required, transmit
                                    // one, otherwise transmit 4 bytes of
                                    // 0x00;
if(current_command.arg_required == YES)
{
   counter = 0;
   while(counter <= 3)
   {
      SPI0DAT = long_arg.b[counter];
      counter++;
      while(!SPIF){}
      SPIF = 0;
   }
}
else
{
   counter = 0;
   while(counter <= 3)
   {
      SPI0DAT = 0x00;
      counter++;
      while(!SPIF){}
      SPIF = 0;
   }
}
SPI0DAT = current_command.CRC;      // Transmit CRC byte;  In all cases
while(!SPIF){}                      // except CMD0, this will be a dummy
SPIF = 0;                           // character;
```

```
                                    // The command table entry will indicate
                                    // what type of response to expect for
                                    // a given command;  The following
                                    // conditional handles the MMC response;
if(current_command.response == R1)  // Read the R1 response from the card;
{
   do
   {
      SPI0DAT = 0xFF;               // Write dummy value to SPI so that
      while(!SPIF){}                // the response byte will be shifted in;
      SPIF = 0;
      card_response.b[0] = SPI0DAT; // Save the response;
   }
   while((card_response.b[0] & BUSY_BIT));
}
                                    // Read the R1b response;
else if(current_command.response == R1b)
{
   do
   {
      SPI0DAT = 0xFF;               // Start SPI transfer;
      while(!SPIF){}
      SPIF = 0;
      card_response.b[0] = SPI0DAT; // Save card response
   }
   while((card_response.b[0] & BUSY_BIT));
   do                               // Wait for busy signal to end;
   {
      SPI0DAT = 0xFF;
      while(!SPIF){}
      SPIF = 0;
   }
   while(SPI0DAT == 0x00);          // When byte from card is non-zero,
}                                   // card is no longer busy;
                                    // Read R2 response
else if(current_command.response == R2)
{
   do
   {
      SPI0DAT = 0xFF;               // Start SPI transfer;
      while(!SPIF){}
      SPIF = 0;
      card_response.b[0] = SPI0DAT; // Read first byte of response;
   }
   while((card_response.b[0] & BUSY_BIT));
   SPI0DAT = 0xFF;
   while(!SPIF){}
   SPIF = 0;
   card_response.b[1] = SPI0DAT;    // Read second byte of response;
}
else                                // Read R3 response;
{
   do
   {
      SPI0DAT = 0xFF;               // Start SPI transfer;
      while(!SPIF){}
      SPIF = 0;
      card_response.b[0] = SPI0DAT; // Read first byte of response;
```

```
      }
      while((card_response.b[0] & BUSY_BIT));
      counter = 0;
      while(counter <= 3)              // Read next three bytes and store them
      {                                // in local memory;  These bytes make up
         counter++;                    // the Operating Conditions Register
         SPI0DAT = 0xFF;               // (OCR);
         while(!SPIF){}
         SPIF = 0;
         *pchar++ = SPI0DAT;
      }
   }
   switch(current_command.trans_type)  // This conditional handles all data
   {                                   // operations;  The command entry
                                       // determines what type, if any, data
                                       // operations need to occur;
      case RD:                         // Read data from the MMC;
         do                            // Wait for a start read token from
         {                             // the MMC;
            SPI0DAT = 0xFF;            // Start a SPI transfer;
            while(!SPIF){}
            SPIF = 0;
         }
         while(SPI0DAT != START_SBR);  // Check for a start read token;
         counter = 0;                  // Reset byte counter;
                                       // Read <current_blklen> bytes;
         while(counter < (unsigned int)current_blklen)
         {
            SPI0DAT = 0x00;            // Start SPI transfer;
            while(!SPIF){}
            SPIF = 0;
            *pchar++ = SPI0DAT;        // Store data byte in local memory;
            counter++;                 // Increment data byte counter;
         }
         SPI0DAT = 0x00;               // After all data is read, read the two
         while(!SPIF){}                // CRC bytes;  These bytes are not used
         SPIF = 0;                     // in this mode, but the placeholders
         dummy_CRC = SPI0DAT;          // must be read anyway;
         SPI0DAT = 0x00;
         while(!SPIF){}
         SPIF = 0;
         dummy_CRC = SPI0DAT;
         break;
      case WR:                         // Write data to the MMC;
         SPI0DAT = 0xFF;               // Start by sending 8 SPI clocks so
         while(!SPIF){}                // the MMC can prepare for the write;
         SPIF = 0;
         SPI0DAT = START_SBW;          // Send the start write block token;
         while(!SPIF){}
         SPIF = 0;
         counter = 0;                  // Reset byte counter;
                                       // Write <current_blklen> bytes to MMC;
         while(counter < (unsigned int)current_blklen)
         {
            SPI0DAT = *pchar++;        // Write data byte out through SPI;
            while(!SPIF){}
            SPIF = 0;
            counter++;                 // Increment byte counter;
         }
```

```
         SPI0DAT = 0xFF;               // Write CRC bytes (don't cares);
         while(!SPIF){}
         SPIF = 0;
         SPI0DAT = 0xFF;
         while(!SPIF){}
         SPIF = 0;

         do                            // Read Data Response from card;
         {
            SPI0DAT = 0xFF;
            while(!SPIF){}
            SPIF = 0;
            data_resp = SPI0DAT;
         }                             // When bit 0 of the MMC response
                                       // is clear, a valid data response
                                       // has been received;
         while((data_resp & DATA_RESP_MASK) != 0x01);

         do                            // Wait for end of busy signal;
         {
            SPI0DAT = 0xFF;            // Start SPI transfer to receive
            while(!SPIF){}             // busy tokens;
            SPIF = 0;
         }
         while(SPI0DAT == 0x00);       // When a non-zero token is returned,
                                       // card is no longer busy;
         SPI0DAT = 0xFF;               // Issue 8 SPI clocks so that all card
         while(!SPIF){}                // operations can complete;
         SPIF = 0;
         break;
      default: break;
   }
   SPI0DAT = 0xFF;
   while(!SPIF){}
   SPIF = 0;


   NSSMD0 = 1;                         // Deselect memory card;
   SPI0DAT = 0xFF;                     // Send 8 more SPI clocks to ensure
   while(!SPIF){}                      // the card has finished all necessary
   SPIF = 0;                           // operations;
                                       // Restore old block length if needed;
   if((current_command.command_byte == 9)||
      (current_command.command_byte == 10))
   {
      current_blklen = old_blklen;
   }
   return card_response.i;
}


//-----------------------------------------------------------------------------
// MMC_FLASH_Init
//-----------------------------------------------------------------------------
//
// This function initializes the flash card, configures it to operate in SPI
// mode, and reads the operating conditions register to ensure that the device
// has initialized correctly.  It also determines the size of the card by
```

```
// reading the Card Specific Data Register (CSD).

void MMC_FLASH_Init (void)
{
   idata UINT card_status;             // Stores card status returned from
                                       // MMC function calls(MMC_Command_Exec);
   idata unsigned char counter = 0;    // SPI byte counter;
   idata unsigned int size;            // Stores size variable from card;
   unsigned char xdata *pchar;         // Xdata pointer for storing MMC
                                       // register values;
                                       // Transmit at least 64 SPI clocks
                                       // before any bus comm occurs.
   pchar = (unsigned char xdata*)LOCAL_BLOCK;
   for(counter = 0; counter < 8; counter++)
   {
      SPI0DAT = 0xFF;
      while(!SPIF){}
      SPIF = 0;
   }
   NSSMD0 = 0;                         // Select the MMC with the CS pin;
                                       // Send 16 more SPI clocks to
                                       // ensure proper startup;
   for(counter = 0; counter < 2; counter++)
   {
      SPI0DAT = 0xFF;
      while(!SPIF){}
      SPIF = 0;
   }
                                       // Send the GO_IDLE_STATE command with
                                       // CS driven low;  This causes the MMC
                                       // to enter SPI mode;
   card_status.i = MMC_Command_Exec(GO_IDLE_STATE,EMPTY,EMPTY);
                                       // Send the SEND_OP_COND command
   do                                  // until the MMC indicates that it is
   {                                   // no longer busy (ready for commands);
      SPI0DAT = 0xFF;
      while(!SPIF){}
      SPIF = 0;
      card_status.i = MMC_Command_Exec(SEND_OP_COND,EMPTY,EMPTY);
   }
   while ((card_status.b[0] & 0x01));
   SPI0DAT = 0xFF;                     // Send 8 more SPI clocks to complete
   while(!SPIF){}                      // the initialization sequence;
   SPIF = 0;
   do                                  // Read the Operating Conditions
   {                                   // Register (OCR);
      card_status.i = MMC_Command_Exec(READ_OCR,EMPTY,pchar);
   }
   while(!(*pchar&0x80));              // Check the card busy bit of the OCR;

   card_status.i = MMC_Command_Exec(SEND_STATUS,EMPTY,EMPTY);
                                       // Get the Card Specific Data (CSD)
                                       // register to determine the size of the
                                       // MMC;
   card_status.i = MMC_Command_Exec(SEND_CSD,EMPTY,pchar);
   pchar += 9;                         // Size indicator is in the 9th byte of
                                       // CSD register;
                                       // Extract size indicator bits;
   size = (unsigned int)((((*pchar) & 0x03) << 1) |
```

```
                          (((*(pchar+1)) & 0x80) >> 7));
   switch(size)                          // Assign PHYSICAL_SIZE variable to
   {                                     // appropriate size constant;
      case 1: PHYSICAL_SIZE = PS_8MB; break;
      case 2: PHYSICAL_SIZE = PS_16MB; break;
      case 3: PHYSICAL_SIZE = PS_32MB; break;
      case 4: PHYSICAL_SIZE = PS_64MB; break;
      case 5: PHYSICAL_SIZE = PS_128MB; break;
      default: break;
   }
                                         // Determine the number of MMC sectors;
   PHYSICAL_BLOCKS = PHYSICAL_SIZE / PHYSICAL_BLOCK_SIZE;
   LOG_SIZE = PHYSICAL_SIZE - LOG_ADDR;
}
//-----------------------------------------------------------------------------
// MMC_FLASH_Read
//-----------------------------------------------------------------------------
//
// This function reads <length> bytes of FLASH from MMC address <address>, and
// stores them in external RAM at the location pointed to by <pchar>.
// There are two cases that must be considered when performing a read.  If the
// requested data is located entirely in a single FLASH block, the function
// sets the read length appropriately and issues a read command.  If requested
// data crosses a FLASH block boundary, the read operation is broken into two
// parts.  The first part reads data from the starting address to the end of
// the starting block, and then reads from the start of the next block to the
// end of the requested data.  Before each read, the read length must be set
// to the proper value.
unsigned char MMC_FLASH_Read (unsigned long address, unsigned char *pchar,
                          unsigned int length)
{
   idata unsigned long flash_page_1;   // Stores address of first FLASH page;
   idata unsigned long flash_page_2;   // Stores address of second FLASH page;
   idata unsigned int card_status;     // Stores MMC status after each MMC
                                       // command;

   if(length > 512) return 0;          // Test for valid data length;  Length
                                       // must be less than 512 bytes;
                                       // Find address of first FLASH block;
   flash_page_1 = address & ~(PHYSICAL_BLOCK_SIZE-1);
                                       // Find address of second FLASH block;
   flash_page_2 = (address+length-1) & ~(PHYSICAL_BLOCK_SIZE-1);
   if(flash_page_1 == flash_page_2)    // Execute the following if data is
   {                                   // located within one FLASH block;
                                       // Set read length to requested data
                                       // length;
      card_status = MMC_Command_Exec(SET_BLOCKLEN,(unsigned long)length,
                                  EMPTY);
                                       // Issue read command;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,address,pchar);
   }
   else                                // Execute the following if data crosses
   {                                   // MMC block boundary;
                                       // Set the read length to the length
                                       // from the starting address to the
                                       // end of the first FLASH page;
      card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                 (unsigned long)(flash_page_2 - address),
                                  EMPTY);
```

```
                                    // Issue read command;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,address,pchar);
                                    // Set read length to the length from
                                    // the start of the second FLASH page
                                    // to the end of the data;
      card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                (unsigned long)length -
                                (flash_page_2 - address),
                                 EMPTY);
                                    // Issue second read command;  Notice
                                    // that the incoming data stored in
                                    // external RAM must be offset from the
                                    // original pointer value by the length
                                    // of data stored during the first read
                                    // operation;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_2,
                                 pchar + (flash_page_2 - address));
   }
}

//-----------------------------------------------------------------------------
// MMC_FLASH_Clear
//-----------------------------------------------------------------------------
//
// This function erases <length> bytes of flash starting at address <address>.
// The <scratch> pointer points to a 512 byte area of XRAM that can
// be used as temporary storage space.  The flow of this function is similar
// to the FLASH_Read function in that there are two possible cases.  If the
// space to be cleared is contained within one MMC block, the block can be
// stored locally and erased from the MMC.  Then the desired area can be
// cleared in the local copy and the block can be written back to the MMC.  If
// the desired clear area crosses a FLASH block boundary, the previous steps
// must be executed seperately for both blocks.
unsigned char MMC_FLASH_Clear (unsigned long address, unsigned char *scratch,
                          unsigned int length)
{
   idata unsigned long flash_page_1;   // Stores address of first FLASH page;
   idata unsigned long flash_page_2;   // Stores address of second FLASH page;
   idata unsigned int card_status;     // Stores MMC status after each MMC
                                       // command;
   idata unsigned int counter;         // Counter for clearing bytes in local
                                       // block copy;
   unsigned char xdata *index;         // Index into local block used for
                                       // clearing desired data;
   if(length > 512) return 0;          // Test desired clear length;  If
                                       // length > 512, break out and return
                                       // zero;
                                       // Calculate first FLASH page address;
   flash_page_1 = address & ~(PHYSICAL_BLOCK_SIZE-1);
                                       // Calculate second FLASH page address;
   flash_page_2 = (address+length-1) & ~(PHYSICAL_BLOCK_SIZE-1);
   if(flash_page_1 == flash_page_2)    // Clear space all in one FLASH block
   {                                   // condition;
                                       // Read first FLASH block;
      card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                 (unsigned long)PHYSICAL_BLOCK_SIZE,
                                 EMPTY);
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_1,scratch);
                                       // Set index to address of area to clear
```

```
                                    // in local block;
      index = (unsigned int)(address % PHYSICAL_BLOCK_SIZE) + scratch;
      counter = 0;
      while(counter<length)         // Clear desired area in local block;
      {
         *index++ = 0x00;
         counter++;
      }
                                    // Tag first FLASH page for erase;
      card_status = MMC_Command_Exec(TAG_SECTOR_START,flash_page_1,EMPTY);
      card_status = MMC_Command_Exec(TAG_SECTOR_END,flash_page_1,EMPTY);
                                    // Erase first FLASH page;
      card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
                                    // Write local copy of block back out
                                    // to MMC;
      card_status = MMC_Command_Exec(WRITE_BLOCK,flash_page_1,scratch);
   }
   else                             // Clear space crosses FLASH block
   {                                // boundaries condition;
                                    // Follow same procedure as for single
                                    // block case above;  Read first block
                                    // clear data from start address to end
                                    // of block;  Erase block in FLASH;
                                    // Write local copy back out;
      card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                (unsigned long)PHYSICAL_BLOCK_SIZE,
                                EMPTY);
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_1,scratch);
      index = (unsigned int)(address % PHYSICAL_BLOCK_SIZE) + scratch;
      counter = (unsigned int)(flash_page_2 - address);
      while(counter > 0)
      {
         *index++ = 0xFF;
         counter--;
      }
      card_status = MMC_Command_Exec(TAG_SECTOR_END,flash_page_1,EMPTY);
      card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
      card_status = MMC_Command_Exec(WRITE_BLOCK,flash_page_1,scratch);
                                    // Same process as above, but using
                                    // second FLASH block;  Area to be
                                    // cleared extends from beginning of
                                    // second FLASH block to end of desired
                                    // clear area;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_2,scratch);
      index = scratch;
      counter = (unsigned int)(length - (flash_page_2 - address));
      while(counter > 0)
      {
         *index++ = 0xFF;
         counter--;
      }
      card_status = MMC_Command_Exec(TAG_SECTOR_END,flash_page_2,EMPTY);
      card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
      card_status = MMC_Command_Exec(WRITE_BLOCK,flash_page_2,scratch);
   }
}

//-----------------------------------------------------------------------------
// MMC_FLASH_Write
```

```
//-----------------------------------------------------------------------------
//
// This function operates much like the MMC_FLASH_Clear and MMC_FLASH_Read
// functions.  As with the others, if the desired write space crosses a FLASH
// block boundary, the operation must be broken into two pieces.
// MMC_FLASH_Write uses the MMC_FLASH_Clear function to clear the write space
// before issuing any writes.  The desired write space is cleared using
// MMC_FLASH_Clear, then the data is read in, the previously cleared write
// space is modified, and the data is written back out.
//
// While it would be more efficient to avoid the MMC_FLASH_Clear and simply
// perform a read-modify-write operation, using MMC_FLASH_Clear helps make the
// process easier to understand.
unsigned char MMC_FLASH_Write (unsigned long address, unsigned char *scratch,
                           unsigned char *wdata, unsigned int length)
{
   idata unsigned long flash_page_1;   // First FLASH page address;
   idata unsigned long flash_page_2;   // Second FLASH page address;
   idata unsigned int card_status;     // Stores status returned from MMC;
   idata unsigned int counter;         // Byte counter used for writes to
                                       // local copy of data block;
   unsigned char xdata *index;         // Pointer into local copy of data
                                       // block, used during modification;
   MMC_FLASH_Clear(address,scratch,length); // Clear desired write space;
   if(length > 512) return 0;          // Check for valid data length;
                                       // Calculate first FLASH page address;
   flash_page_1 = address & ~(PHYSICAL_BLOCK_SIZE-1);
                                       // Calculate second FLASH page address;
   flash_page_2 = (address+length-1) & ~(PHYSICAL_BLOCK_SIZE-1);
   if(flash_page_1 == flash_page_2)    // Handle single FLASH block condition;
   {
                                       // Set block length to default block
                                       // size (512 bytes);
      card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                  (unsigned long)PHYSICAL_BLOCK_SIZE,
                                  EMPTY);
                                       // Read data block;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_1,scratch);
      index = (unsigned int)(address % PHYSICAL_BLOCK_SIZE) + scratch;
      counter = 0;
      while(counter<length)            // Modify write space in local copy;
      {
         *index++ = *wdata++;
         counter++;
      }
                                       // Write modified block back to MMC;
      card_status = MMC_Command_Exec(WRITE_BLOCK,flash_page_1,scratch);
   }
   else                                // Handle multiple FLASH block
   {                                   // condition;
                                       // Set block length to default block
                                       // size (512 bytes);
      card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                  (unsigned long)PHYSICAL_BLOCK_SIZE,
                                  EMPTY);
                                       // Read first data block;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_1,scratch);
      index = (unsigned int)(address % PHYSICAL_BLOCK_SIZE) + scratch;
      counter = (unsigned int)(flash_page_2 - address);
```

```
      while(counter > 0)               // Modify data in local copy of first
      {                                // block;
         *index++ = *wdata++;
         counter--;
      }
                                       // Write local copy back to MMC;
      card_status = MMC_Command_Exec(WRITE_BLOCK,flash_page_1,scratch);
                                       // Read second data block;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_2,scratch);
      index = scratch;
      counter = (unsigned int)(length - (flash_page_2 - address));
      while(counter > 0)               // Modify data in local copy of second
      {                                // block;
         *index++ = *wdata++;
         counter--;
      }
                                       // Write local copy back to MMC;
      card_status = MMC_Command_Exec(WRITE_BLOCK,flash_page_2,scratch);
   }
}

//-----------------------------------------------------------------------------
// MMC_FLASH_MassErase
//-----------------------------------------------------------------------------
//
// This function erases <length> bytes of flash starting with the block
// indicated by <address1>.  This function only handles sector-sized erases
// or larger.  Function should be called with sector-aligned erase addresses.
unsigned char MMC_FLASH_MassErase (unsigned long address1,
                                   unsigned long length)
{
   idata unsigned char card_status;    // Stores card status returned from MMC;
                                       // Store start and end sectors to be
                                       // to be erased;
   idata unsigned long flash_page_1, flash_page_2;
                                       // Store start and end groups to be
                                       // erased;
   idata unsigned long flash_group_1, flash_group_2;
                                       // Compute first sector address for
                                       // erase;
   flash_page_1 = address1 & ~(PHYSICAL_BLOCK_SIZE-1);
                                       // Compute first group address for
                                       // erase;
   flash_group_1 = flash_page_1 &~(PHYSICAL_GROUP_SIZE-1);
                                       // Compute last sector address for
                                       // erase;
   flash_page_2 = (address1 + length) & ~(PHYSICAL_BLOCK_SIZE-1);
                                       // Compute last group address for erase;
   flash_group_2 = flash_page_2 &~(PHYSICAL_GROUP_SIZE-1);

   if(flash_group_1 == flash_group_2)  // Handle condition where entire erase
   {                                   // space is in one erase group;
                                       // Tag first sector;
      card_status = MMC_Command_Exec(TAG_SECTOR_START,flash_page_1,EMPTY);
                                       // Tag last sector;
      card_status = MMC_Command_Exec(TAG_SECTOR_END,flash_page_2,EMPTY);
                                       // Issue erase command;
      card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
   }
```

```
   else                                 // Handle condition where erase space
   {                                    // crosses an erase group boundary;
                                        // Tag first erase sector;
      card_status = MMC_Command_Exec(TAG_SECTOR_START,flash_page_1,EMPTY);
                                        // Tag last sector of first group;
      card_status = MMC_Command_Exec(TAG_SECTOR_END,
                                (flash_group_1 +
                                (unsigned long)(PHYSICAL_GROUP_SIZE
                                - PHYSICAL_BLOCK_SIZE)),EMPTY);
                                        // Issue erase command;
      card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
                                        // Tag first sector of last erase group;
      card_status = MMC_Command_Exec(TAG_SECTOR_START,flash_group_2,EMPTY);
                                        // Tag last erase sector;
      card_status = MMC_Command_Exec(TAG_SECTOR_END,flash_page_2,EMPTY);
                                        // Issue erase;
      card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
                                        // Conditional that erases all groups
                                        // between first and last group;
      if(flash_group_2 > (flash_group_1 + PHYSICAL_GROUP_SIZE))
      {
                                        // Tag first whole group to be erased;
         card_status = MMC_Command_Exec(TAG_ERASE_GROUP_START,
                                   (flash_group_1 +
                                   (unsigned long)PHYSICAL_GROUP_SIZE),EMPTY);
                                        // Tag last whole group to be erased;
         card_status = MMC_Command_Exec(TAG_ERASE_GROUP_END,
                                   (flash_page_2 -
                                   (unsigned long)PHYSICAL_GROUP_SIZE),EMPTY);
                                        // Issue erase command;
         card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
      }
   }

   return card_status;
}

//-----------------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// ADC0_ISR
//-----------------------------------------------------------------------------
//
// ADC0 end-of-conversion ISR
// Here we take the ADC0 sample, add it to a running total <accumulator>, and
// decrement our local decimation counter <int_dec>.  When <int_dec> reaches
// zero, we post the decimated result in the global variable <result>.
//
// In addition, this ISR is used to keep track of time.  Every 4096 samples,
// approximately once a second, we update the seconds, minutes, hours, and days
// for the temperature timestamp.  If the global state is RUNNING or FINISHED,
// a low priority software interrupt56 is generated and the log is updated.
// Using the low priority interrupt allows the MMC communication to execute
// without disrupting the temperature sampling process.  The ADC end-of-conv
// interrupt is set to high priority, so it will be executed even if a low
// priority interrupt is already in progress.
void ADC0_ISR (void) interrupt 10
```

```
{
   static unsigned int_dec=INT_DEC;    // integrate/decimate counter
                                       // we post a new result when
                                       // int_dec = 0
   static LONG accumulator={0L};       // here's where we integrate the
                                       // ADC samples

   AD0INT = 0;                         // clear ADC conversion complete
                                       // indicator

    accumulator.l += ADC0;             // read ADC value and add to running
                                       // total
   int_dec--;                          // update decimation counter

   if (int_dec == 0) {                 // if zero, then post result
      int_dec = INT_DEC;               // reset counter

      // Result = accumulator >> 6
      // Perform my shifting left 2, then byte-swapping
      accumulator.l <<= 2;             // accumulator = accumulator << 2
      Result.b[0] = 0;                 // Result = accumulator >> 8
      Result.b[3] = accumulator.b[2];
      Result.b[2] = accumulator.b[1];
      Result.b[1] = accumulator.b[0];

      accumulator.l = 0L;              // reset accumulator
      LogRecord.bSec++;                // update seconds counter
      if (LogRecord.bSec == 60)
      {
         LogRecord.bSec = 0;
         LogRecord.bMin++;             // update minutes counter
         if (LogRecord.bMin == 60)
         {
            LogRecord.bMin = 0;
            LogRecord.bHour++;         // update hours counter
            if (LogRecord.bHour == 24)
            {
               LogRecord.bHour = 0;
               LogRecord.uDay++;       // update days counter
            }
         }
      }

      if ((State == RUNNING)||(State == FINISHED))
      {
         AD0WINT = 1;
      }
   }
}

//-----------------------------------------------------------------------------
// Soft_ISR
//-----------------------------------------------------------------------------
//
// This ISR executes whenever a log update is needed.  It simply clears the
// interrupt flag and executes the LogUpdate function.  This is a low priority
// ISR, so the ADC end-of-conversion ISR will interrupt it if necessary.  This
// prevents the long MMC communication process from disrupting temperature
// sampling.
```

```
//
void Soft_ISR (void) interrupt 9
{
   AD0WINT = 0;                        // Clear software interrupt flag;
   LogUpdate();
}

//-----------------------------------------------------------------------------
// Initialization Subroutines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use the internal 24.5MHz
// oscillator as its clock source.  Enables missing clock detector reset. Also
// configures and enables the external crystal oscillator.
//
void SYSCLK_Init (void)
{
   OSCICN |= 0x06;                     // Configure internal oscillator for
                                       // its maximum frequency;
   VDM0CN  = 0x80;                     // Enable VDD Monitor
   RSTSRC |= 0x06;                     // Enable missing clock detector and
                                       // VDD Monitor as reset sources;

   CLKMUL = 0x00;                      // Reset multiplier; Internal Osc is
                                       // multiplier source;
   CLKMUL |= 0x80;                     // Enable Clock Multiplier;

// Wait 5 us for multiplier to be enabled
   TMR2CN = 0x00;                      // STOP Timer2; Clear TF2H and TF2L;
                                       // disable low-byte interrupt; disable
                                       // split mode; select internal timebase
   CKCON |= 0x10;                      // Timer2 uses SYSCLK as its timebase

   TMR2RL  = START_SYSCLK / 200000;    // Init reload values 12 MHz / (5E^-6)
   TMR2    = TMR2RL;                   // Init Timer2 with reload value
   ET2 = 0;                            // disable Timer2 interrupts
   TF2H = 0;
   TR2 = 1;                            // start Timer2
   while (!TF2H);                      // wait for overflow
   TF2H = 0;                           // clear overflow indicator
   TR2 = 0;                            // Stop Timer2;

   CLKMUL |= 0xC0;                     // Initialize Clock Multiplier
   while(!(CLKMUL & 0x20))             // Wait for MULRDY
   CLKSEL = 0x02;                      // Select SYSCLK * 4 / 2 as clock source
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.

// P0.0 - SCK
// P0.1 - MISO
```

```
// P0.2 - XTAL1 (skipped by Crossbar)
// P0.3 - XTAL2 (skipped by Crossbar)

// P0.4 - UART TX (push-pull)
// P0.5 - UART RX
// P0.6 - MOSI
// P0.7 - VREF
// P1.7 - NSS

// P2.2 - LED (push-pull)
// P2.1 - SW2

//
void PORT_Init (void)
{
   P0SKIP  = 0x80;                     // skip VREF in crossbar
                                       // assignments
   XBR0    = 0x03;                     // UART0 TX and RX pins enabled, SPI
                                       // enabled
   XBR1    = 0x40;                     // Enable crossbar and weak pull-ups

   P0MDIN &= ~0x80;                    // configure VREF as analog input

   P0MDOUT |= 0x1D;                    // enable TX0,SCK,MOSI as a push-pull
   P2MDOUT |= 0x04;                    // enable LED as a push-pull output
}

//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------
//
// Configure the UART0 using Timer1, for <BAUDRATE> and 8-N-1.
//
void UART0_Init (void)
{
   SCON0 = 0x10;                       // SCON0: 8-bit variable bit rate
                                       //        level of STOP bit is ignored
                                       //        RX enabled
                                       //        ninth bits are zeros
                                       //        clear RI0 and TI0 bits
   if (SYSCLK/BAUDRATE/2/256 < 1)
   {
      TH1 = -(SYSCLK/BAUDRATE/2);
      CKCON |=  0x08;                  // T1M = 1; SCA1:0 = xx
   }
   else if (SYSCLK/BAUDRATE/2/256 < 4)
   {
      TH1 = -(SYSCLK/BAUDRATE/2/4);
      CKCON &= ~0x0B;
      CKCON |=  0x01;                  // T1M = 0; SCA1:0 = 01
   }
   else if (SYSCLK/BAUDRATE/2/256 < 12)
   {
      TH1 = -(SYSCLK/BAUDRATE/2/12);
      CKCON &= ~0x0B;                  // T1M = 0; SCA1:0 = 00
   }
   else
   {
      TH1 = -(SYSCLK/BAUDRATE/2/48);
```

```
      CKCON &= ~0x0B;                   // T1M = 0; SCA1:0 = 10
      CKCON |=  0x02;
   }

   TL1 = TH1;                           // init Timer1
   TMOD &= ~0xf0;                       // TMOD: timer 1 in 8-bit autoreload
   TMOD |=  0x20;
   TR1 = 1;                             // START Timer1
   TI0 = 1;                             // Indicate TX0 ready
}

//-----------------------------------------------------------------------------
// SPI0_Init
//-----------------------------------------------------------------------------
//
// Configure SPI0 for 8-bit, 2MHz SCK, Master mode, polled operation, data
// sampled on 1st SCK rising edge.
//
void SPI_Init (void)
{
   SPI0CFG = 0x70;                      // data sampled on rising edge, clk
                                        // active low,
                                        // 8-bit data words, master mode;

   SPI0CN = 0x0F;                       // 4-wire mode; SPI enabled; flags
                                        // cleared
   SPI0CKR = SYSCLK/2/10000000;         // SPI clock <= 10MHz
}

//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------
//
// Configure ADC0 to use Timer2 overflows as conversion source, to
// generate an interrupt on conversion complete, and to sense the output of
// the temp sensor with a gain of 2 (we want the white noise).  Enables ADC
// end of conversion interrupt.  Leaves ADC disabled.
//
void ADC0_Init (void)
{
   ADC0CN = 0x02;                       // ADC0 disabled; normal tracking
                                        // mode; ADC0 conversions are initiated
                                        // on overflow of Timer2;
   AMX0P = 0x1E;                        // Select temp sensor as positive input;
   AMX0N = 0x1F;                        // Select GND as negative input;
   ADC0CF = (SYSCLK/2000000) << 3;      // ADC conversion clock <= 2.0MHz
   REF0CN = 0x07;                       // Enable temp sensor, bias generator,
                                        // and internal VREF;
   EIE1 |= 0x08;                        // Enable ADC0 EOC interrupt;
   EIP1 |= 0x08;                        // ADC EOC interrupt is high priority;
}


//-----------------------------------------------------------------------------
// Soft_Init
//-----------------------------------------------------------------------------
//
// This function enables ADC Window Compare interrupts and clears the interrupt
// flag.  Since the window compare interrupt is not being used in this example,
```

```
// we can use it as a low priority software interrupt.  This interrupt can be
// used to execute log updates without disturbing the ADC sampling process.
//
void Soft_Init (void)
{
   AD0WINT = 0;                        // Clear ADC0 window compare interrupt
                                       // flag;
   EIE1 |= 0x04;                       // Enable ADC0 window compare
                                       // interrupts;
}

//-----------------------------------------------------------------------------
// Timer2_Init
//-----------------------------------------------------------------------------
//
// This routine initializes Timer2 to use SYSCLK as its timebase and to
// generate an overflow at <SAMPLE_RATE> Hz.
//
void Timer2_Init (int counts)
{
   TMR2CN = 0x01;                      // Clear TF2H, TF2L; disable TF2L
                                       // interrupts; T2 in 16-bit mode;
                                       // Timer2 stopped;
   CKCON |= 0x30;                      // Timer 2 uses SYSCLK as clock
                                       // source
   TMR2RL = -counts;                   // reload once per second
   TMR2 = TMR2RL;                      // init Timer2
   ET2 = 0;                            // Disable Timer2 interrupts
   TR2 = 1;                            // Start Timer2
}
```

MMC_DataLogger_ EEPROM.c

```
//-----------------------------------------------------------------------------
// MMC_DataLogger_EEPROM.c
//-----------------------------------------------------------------------------
// Copyright 2003 Silicon Laboratories
//
// AUTH: BW / JS / GV
// DATE: 8 MAR 04
//
// This software shows an example of a data logging application that maintains
// the log on an MMC card.  In addition, this software uses an external EEPROM
// to buffer MMC data.  This removes the need to buffer large amounts of data in
// on-chip external memory.
//
// Control Function:
//
// The system is controlled via the hardware UART, operating at a baud rate
// determined by the constant <BAUDRATE>, using Timer1 overflows as the baud
// rate source.  The commands are as follows (not case sensitive):
//  'c' - Clear Log
//  'd' - Display Log
//  'i' - Init RTC
//  'p' - Stop Logging
//  's' - Start Logging
```

```
//  '?' - List Commands
//
// Sampling Function:
//
// The ADC is configured to sample the on-chip temperature sensor at 4.096kHz,
// using Timer0 (in 8-bit auto-reload mode) as the start-of-conversion source.
// The ADC result is accumulated and decimated by a factor of 4096, yielding
// a 16-bit resolution quantitity from the original 10-bit sample at an
// effective output word rate of about 1Hz.  This decimated value is
// stored in the global variable <result>.
//
// A note about oversampling and averaging as it applies to this temp
// sensor example:  The transfer characteristic of the temp sensor on the
// 'F320 family of devices is 2.86mV/C.  The LSB size of the ADC using the
// internal VREF (2.43V) as its voltage reference is 2.3mV/code.
// This means that adjacent ADC codes are about ~1 degrees C apart.
//
// If we desire a higher temperature resolution, we can achieve it by
// oversampling and averaging (See AN118 on the Silicon labs website).  For
// each additional bit of resolution required, we must oversample by a power
// of 4.  For example, increasing the resolution by 4 bits requires
// oversampling by a factor of 4^4, or 256.
//
// By what factor must we oversample to achieve a temperature resolution to
// the nearest hundredth of a degree C?  In other words, "How may bits of
// resolution do we need to add?"  The difference between 1 degrees C and
// 0.01 degrees C is a factor of 100 (100 is between 2^6 and 2^7, so we need
// somewhere between 6 and 7 more bits of resolution).  Choosing '6 bits',
// we calculate our oversampling ratio to be 4^6, or 4096.
//
// A note about accuracy:  oversampling and averaging provides a method to
// increase the 'resolution' of a measurement.  The technique does nothing
// to improve a measurement's 'accuracy'.  Just because we can measure a
// 0.01 degree change in temperature does not mean that the measurements
// are accurate to 0.01 degrees.  Some possible sources of inaccuracies in
// this system are:
//  1. manufacturing tolerances in the temperature sensor itself (transfer
//     characteristic variation)
//  2. VDD or VREF tolerance
//  3. ADC offset, gain, and linearity variations
//  4. Device self-heating
//
// Temperature Clock Function:
//
// The temperature clock maintains a record of days, hours, minutes, and
// seconds.  The current time record is stored with the temperature value
// in each log entry.  Clock updates are performed in the ADC end-of-conversion
// ISR at approximately once every second.
//
// Storage Function:
//
// MMC FLASH is used for storing the log entries.  Each entry contains
// the temperature in hundredths of a degree C, the day, hour, minute, and
// second that the reading was taken.  The LogUpdate function stores log
// entries in an EEPROM buffer and then writes that buffer out to the
// MMC when it is full.  Communication with the MMC is performed through the
// MMC access functions.  These functions provide transparent MMC access to
// the higher level functions (logging functions).  The MMC interface is broken
// into two pieces.  The high level piece consists of the user callable MMC
```

```
// access functions (MMC_FLASH_Read, MMC_FLASH_Write, MMC_FLASH_Clear,
// MMC_FLASH_MassErase).  These functions are called by the user to execute
// data operations on the MMC.  They break down the data operations into MMC
// commands.  The low level piece consists of a single command execution
// function (MMC_Command_Exec) which is called by the MMC data manipulation
// functions.  This function is called every time a command must be sent to the
// MMC.  It handles all of the required SPI traffic between the Cygnal device
// and the MMC.  Communication between the EEPROM and the Cygnal device is
// performed using SMBus.
//
// Target: C8051F32x
// Tool chain: KEIL C51 6.03 / KEIL EVAL C51
//


//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------


#include <c8051f320.h>                 // SFR declarations
#include <stdio.h>                     // printf() and getchar()
#include <ctype.h>                     // tolower()

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F31x, 'F32x, 'F33x
//-----------------------------------------------------------------------------

sfr16 DP       = 0x82;                 // data pointer
sfr16 TMR2RL   = 0xca;                 // Timer2 reload value
sfr16 TMR2     = 0xcc;                 // Timer2 counter
sfr16 ADC0     = 0xbd;                 // ADC0 Data

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
#define VERSION      "1.0"            // version identifier
#define TRUE         1
#define FALSE        0

#define START_SYSCLK 12000000
#define SYSCLK       START_SYSCLK * 2  // SYSCLK frequency in Hz
#define BAUDRATE     115200            // Baud rate of UART in bps
#define SAMPLE_RATE  4096              // Sample frequency in Hz
#define INT_DEC      4096              // integrate and decimate ratio

#define TEMP_OFFSET  50900000L         // Temp sensor offset constant used
                                       // in conversion of ADC sample to temp
                                       // value;
#define TEMP_SLOPE   187433L           // Temp sensor slope constant used
                                       // in conversion of ADC sample to temp
                                       // value;
#define TEMP_VREF    2430              // VREF offset constant used
                                       // in conversion of ADC sample to temp
                                       // value;

#define EEPROM_SIZE 32768
#define EEPROM_PAGE_SIZE 64
```

```
// Constants that define available card sizes, 8MB through 128MB
#define PS_8MB       8388608L
#define PS_16MB      16777216L
#define PS_32MB      33554432L
#define PS_64MB      67108864L
#define PS_128MB     134217728L

// Physical size in bytes of one MMC FLASH sector
#define PHYSICAL_BLOCK_SIZE     512

// Erase group size = 16 MMC FLASH sectors
#define PHYSICAL_GROUP_SIZE     PHYSICAL_BLOCK_SIZE * 16

// Log table start address in MMC FLASH
#define LOG_ADDR     0x0000


// Size in bytes for each log entry
#define LOG_ENTRY_SIZE sizeof(LOG_ENTRY)

// Size of EEPROM buffer that stores table entries
// before they are written to MMC.
#define BUFFER_SIZE  LOG_ENTRY_SIZE * 10

// Command table value definitions
// Used in the MMC_Command_Exec function to
// decode and execute MMC command requests
#define     EMPTY  0
#define     YES   1
#define     NO    0
#define     CMD   0
#define     RD    1
#define     WR    2
#define     R1    0
#define     R1b   1
#define     R2    2
#define     R3    3

// Start and stop data tokens for single and multiple
// block MMC data operations
#define     START_SBR      0xFE
#define     START_MBR      0xFE
#define     START_SBW      0xFE
#define     START_MBW      0xFC
#define     STOP_MBW       0xFD

// Mask for data response token after an MMC write
#define     DATA_RESP_MASK 0x11

// Mask for busy token in R1b response
#define     BUSY_BIT       0x80

// Command Table Index Constants:
// Definitions for each table entry in the command table.
// These allow the MMC_Command_Exec function to be called with a
// meaningful parameter rather than a number.
#define     GO_IDLE_STATE            0
#define     SEND_OP_COND             1
#define     SEND_CSD                 2
```

```
#define     SEND_CID                 3
#define     STOP_TRANSMISSION        4
#define     SEND_STATUS              5
#define     SET_BLOCKLEN             6
#define     READ_SINGLE_BLOCK        7
#define     READ_MULTIPLE_BLOCK      8
#define     WRITE_BLOCK              9
#define     WRITE_MULTIPLE_BLOCK    10
#define     PROGRAM_CSD             11
#define     SET_WRITE_PROT          12
#define     CLR_WRITE_PROT          13
#define     SEND_WRITE_PROT         14
#define     TAG_SECTOR_START        15
#define     TAG_SECTOR_END          16
#define     UNTAG_SECTOR            17
#define     TAG_ERASE_GROUP_START   18
#define     TAG_ERASE_GROUP_END     19
#define     UNTAG_ERASE_GROUP       20
#define     ERASE                   21
#define     LOCK_UNLOCK             22
#define     READ_OCR                23
#define     CRC_ON_OFF              24

// LOCAL_BLOCK is the start of an EEPROM buffer for incoming temperature data;
// When this buffer is full, the page is written out to the MMC and erased for
// new data;
#define     LOCAL_BLOCK             0x1000
// SCRATCH_BLOCK is used by the high level MMC functions as temporary storage;
#define     SCRATCH_BLOCK           0x0000

// SMBus Definitions

#define  SMB_FREQUENCY  300000         // Target SCL clock rate

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

// Device addresses (7 bits, lsb is a don't care)
#define  EEPROM_ADDR    0xA0           // Device address for slave target
                                       // Note: This address is specified
                                       // in the Microchip 24LC02B
                                       // datasheet.
// SMBus Buffer Size
#define  SMB_BUFF_SIZE  0x08           // Defines the maximum number of bytes
                                       // that can be sent or received in a
                                       // single transfer

// Status vector - top 4 bits only
#define  SMB_MTSTA      0xE0           // (MT) start transmitted
#define  SMB_MTDB       0xC0           // (MT) data byte transmitted
#define  SMB_MRDB       0x80           // (MR) data byte received
// End status vector definition

//-----------------------------------------------------------------------------
// UNIONs, STRUCTUREs, and ENUMs
//-----------------------------------------------------------------------------
typedef union LONG                     // byte-addressable LONG
{
   long l;
```

```
   unsigned char b[4];
} LONG;

typedef union INT                       // byte-addressable INT
{
   int i;
   unsigned char b[2];
} INT;

typedef union                           // byte-addressable unsigned long
{
      unsigned long l;
      unsigned char b[4];

} ULONG;

typedef union UINT                      // byte-addressable unsigned int
{
      unsigned int i;
      unsigned char b[2];
} UINT;

typedef struct LOG_ENTRY                // (7 bytes per entry)
{
   int wTemp;                           // temperature in hundredths of a
                                        // degree
   unsigned int uDay;                   // day of entry
   unsigned char bHour;                 // hour of entry
   unsigned char bMin;                  // minute of entry
   unsigned char bSec;                  // second of entry
} LOG_ENTRY;

// The states listed below represent various phases of
// operation;
typedef enum STATE
{
   RESET,                               // Device reset has occurred;
   RUNNING,                             // Data is being logged normally;
   FINISHED,                            // Logging stopped, store buffer;
   STOPPED                              // Logging completed, buffer stored;
} STATE;

// This structure defines entries into the command table;
typedef struct
{
      unsigned char command_byte;       // OpCode;
      unsigned char arg_required;       // Indicates argument requirement;
      unsigned char CRC;                // Holds CRC for command if necessary;
      unsigned char trans_type;         // Indicates command transfer type;
      unsigned char response;           // Indicates expected response;
      unsigned char var_length;         // Indicates varialble length transfer;
} command;

// Command table for MMC.  This table contains all commands available in SPI
// mode;  Format of command entries is described above in command structure
// definition;
code command commandlist[25] = {
      { 0,NO ,0x95,CMD,R1 ,NO },    // CMD0;  GO_IDLE_STATE: reset card;
      { 1,NO ,0xFF,CMD,R1 ,NO },    // CMD1;  SEND_OP_COND: initialize card;
```

```
      { 9,NO ,0xFF,RD ,R1 ,NO },    // CMD9;  SEND_CSD: get card specific data;
      {10,NO ,0xFF,RD ,R1 ,NO },    // CMD10; SEND_CID: get card identifier;
      {12,NO ,0xFF,CMD,R1 ,NO },    // CMD12; STOP_TRANSMISSION: end read;
      {13,NO ,0xFF,CMD,R2 ,NO },    // CMD13; SEND_STATUS: read card status;
      {16,YES,0xFF,CMD,R1 ,NO },    // CMD16; SET_BLOCKLEN: set block size;
      {17,YES,0xFF,RD ,R1 ,NO },    // CMD17; READ_SINGLE_BLOCK: read 1 block;
      {18,YES,0xFF,RD ,R1 ,YES},    // CMD18; READ_MULTIPLE_BLOCK: read > 1;
      {24,YES,0xFF,WR ,R1 ,NO },    // CMD24; WRITE_BLOCK: write 1 block;
      {25,YES,0xFF,WR ,R1 ,YES},    // CMD25; WRITE_MULTIPLE_BLOCK: write > 1;
      {27,NO ,0xFF,CMD,R1 ,NO },    // CMD27; PROGRAM_CSD: program CSD;
      {28,YES,0xFF,CMD,R1b,NO },    // CMD28; SET_WRITE_PROT: set wp for group;
      {29,YES,0xFF,CMD,R1b,NO },    // CMD29; CLR_WRITE_PROT: clear group wp;
      {30,YES,0xFF,CMD,R1 ,NO },    // CMD30; SEND_WRITE_PROT: check wp status;
      {32,YES,0xFF,CMD,R1 ,NO },    // CMD32; TAG_SECTOR_START: tag 1st erase;
      {33,YES,0xFF,CMD,R1 ,NO },    // CMD33; TAG_SECTOR_END: tag end(single);
      {34,YES,0xFF,CMD,R1 ,NO },    // CMD34; UNTAG_SECTOR: deselect for erase;
      {35,YES,0xFF,CMD,R1 ,NO },    // CMD35; TAG_ERASE_GROUP_START;
      {36,YES,0xFF,CMD,R1 ,NO },    // CMD36; TAG_ERASE_GROUP_END;
      {37,YES,0xFF,CMD,R1 ,NO },    // CMD37; UNTAG_ERASE_GROUP;
      {38,YES,0xFF,CMD,R1b,NO },    // CMD38; ERASE: erase all tagged sectors;
      {42,YES,0xFF,CMD,R1b,NO },    // CMD42; LOCK_UNLOCK;
      {58,NO ,0xFF,CMD,R3 ,NO },    // CMD58; READ_OCR: read OCR register;
      {59,YES,0xFF,CMD,R1 ,NO },    // CMD59; CRC_ON_OFF: toggles CRC checking;
                                };

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

xdata LONG Result = {0L};               // ADC0 decimated value

xdata LOG_ENTRY LogRecord;              // Memory space for each log entry
xdata unsigned long uLogCount;          // Current number of table entries
xdata unsigned int pLogTable;           // Pointer to buffer for table entries
xdata STATE State = RESET;              // System state variable;  Determines
                                        // how log update function will exec;
xdata unsigned long PHYSICAL_SIZE;      // MMC size variable;  Set during
                                        // initialization;
xdata unsigned long PHYSICAL_BLOCKS;    // MMC block number;  Computed during
                                        // initialization;
xdata unsigned long LOG_SIZE;           // Available number of bytes for log
                                        // table

xdata unsigned char EEPROM_PageBuffer[64];

xdata unsigned char* pSMB_DATA_IN;      // Global pointer for SMBus data
                                        // All receive data is written here

xdata unsigned char SMB_SINGLEBYTE_OUT;// Global holder for single byte writes.

xdata unsigned char* pSMB_DATA_OUT;     // Global pointer for SMBus data.
                                        // All transmit data is read from here

xdata unsigned char SMB_DATA_LEN;       // Global holder for number of bytes
                                        // to send or receive in the current
                                        // SMBus transfer.

xdata UINT WORD_ADDR;                   // Global holder for the EEPROM word
                                        // address that will be accessed in
```

```
                                    // the next transfer

xdata unsigned char TARGET;         // Target SMBus slave address

bit SMB_BUSY = 0;                   // Software flag to indicate when the
                                    // EEPROM_ReadByte() or
                                    // EEPROM_WriteByte()
                                    // functions have claimed the SMBus

bit SMB_RW;                         // Software flag to indicate the
                                    // direction of the current transfer

bit SMB_SENDWORDADDR;               // When set, this flag causes the ISR
                                    // to send the 8-bit <WORD_ADDR>
                                    // after sending the slave address.

bit SMB_HIGHBYTENOTSENT;


bit SMB_RANDOMREAD;                 // When set, this flag causes the ISR
                                    // to send a START signal after sending
                                    // the word address.

bit SMB_ACKPOLL;                    // When set, this flag causes the ISR
                                    // to send a repeated START until the
                                    // slave has acknowledged its address

bit update;
//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------

void main (void);

// Support Subroutines
void MENU_ListCommands (void);      // Outputs user menu choices via UART


// Logging Subroutines
void LogUpdate (void);              // Builds MMC log table
unsigned long LogFindCount();       // Returns current number of log entries
void LogErase (void);               // Erases entire log table
void LogPrint (void);               // Prints log through UART
void LogInit (LOG_ENTRY *pEntry);   // Initializes area for building entries

// High Level MMC_FLASH Functions

void MMC_FLASH_Init (void);         // Initializes MMC and configures it to
                                    // accept SPI commands;

                                    // Reads <length> bytes starting at
                                    // <address> and stores them at <pchar>;
unsigned char MMC_FLASH_Read (unsigned long address, unsigned int pchar,
                        unsigned int length);

                                    // Writes <length> bytes of data at
                                    // <wdata> to <address> in MMC;
                                    // <scratch> provides temporary storage;
unsigned char MMC_FLASH_Write (unsigned long address, unsigned int scratch,
```

```
                          unsigned int wdata, unsigned int length);

                                        // Clears <length> bytes of FLASH
                                        // starting at <address1>; Requires that
                                        // desired erase area be sector aligned;
unsigned char MMC_FLASH_MassErase (unsigned long address1,
                                   unsigned long length);

// Low Level MMC_FLASH_ Functions

                                        // Decodes and executes MMC commands;
                                        // <cmd> is an index into the command
                                        // table and <argument> contains a
                                        // 32-bit argument if necessary;  If a
                                        // data operation is taking place, the
                                        // data will be stored to or read from
                                        // the location pointed to by <pchar>;
unsigned int MMC_Command_Exec (unsigned char cmd, unsigned long argument,
                               unsigned int pchar);

// SMBus Routines
                                        // Send a block of data of size
                                        // <len> bytes to address <addr> in the
                                        // EEPROM;
void EEPROM_WriteArray(unsigned int addr, char* SrcAddr,unsigned char len);
                                        // Read a block of data of size <len>
                                        // from address <src_addr> in the EEPROM
                                        // and store it at <dest_addr> in local
                                        // memory;
void EEPROM_ReadArray (unsigned char* dest_addr, unsigned int src_addr,
                       unsigned char len);
                                        // Writes <dat> to address <addr> in the
                                        // EEPROM;
void EEPROM_WriteByte( unsigned int addr, unsigned char dat);
                                        // Returns the value at address <addr>
                                        // in the EEPROM;
unsigned char EEPROM_ReadByte( unsigned int addr);
                                        // Clear <length> bytes of data at
                                        // address <addr> in the EEPROM;
void EEPROM_Clear (unsigned int addr, unsigned int length);
// Initialization Subroutines

void SYSCLK_Init (void);
void PORT_Init (void);
void UART0_Init (void);
void ADC0_Init (void);
void Timer0_Init (void);
void Timer2_Init (int counts);
void SPI_Init (void);
void SMBus_Init (void);

// Interrupt Service Routines

void ADC0_ISR (void);
void Soft_ISR (void);

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

```
void main (void)
{
   idata char key_press;               // Input character from UART;
   // Disable Watchdog timer
   PCA0MD &= ~0x40;                    // WDTE = 0 (clear watchdog timer
                                       // enable);

   PORT_Init ();                       // Initialize crossbar and GPIO;
   SYSCLK_Init ();                     // Initialize oscillator;
   UART0_Init ();                      // Initialize UART0;
   SPI_Init ();                        // Initialize SPI0;
   Timer0_Init ();
   Timer2_Init (SYSCLK/SAMPLE_RATE);   // Init Timer2 for 16-bit autoreload;
   ADC0_Init ();                       // Init ADC0;

   AD0EN = 1;                          // enable ADC0;

   State = RESET;                      // Set global state machine to reset
                                       // state;
   SMBus_Init();                       // Configure and enable SMBus

   State = STOPPED;                    // Global state is STOPPED; no data
                                       // is being logged;
   EA = 1;                             // Enable global interrupts;
                                       // Clear space in EEPROM for buffers
   EEPROM_Clear((unsigned int)LOCAL_BLOCK,
                (unsigned int)PHYSICAL_BLOCK_SIZE);
   EEPROM_Clear((unsigned int)SCRATCH_BLOCK,
                (unsigned int)PHYSICAL_BLOCK_SIZE);
   MMC_FLASH_Init();                   // Initialize MMC card;
                                       // Initialize log table buffer pointer
   pLogTable = LOCAL_BLOCK;
   uLogCount = LogFindCount();         // Find current number of log table
                                       // entries;
   update = 0;
   printf ("\n");                      // Print list of commands;
   MENU_ListCommands ();
    while (1)                          // Serial port command decoder;
   {
      if(RI0 == 1)
      {
         key_press = getchar();        // Get command character;
                                       // Convert to lower case;
         key_press = tolower(key_press);

         switch (key_press) {
            case 'c':                  // Clear log;
              if(State == STOPPED)     // Only execute if not logging;
              {
               printf ("\n Clear Log\n");
               LogErase();             // erase log entries;
                                       // update global log entry count;
               uLogCount = LogFindCount();
              }
               break;
            case 'd':                  // Display log;
              if(State == STOPPED)     // Only execute if not logging;
              {
```

```
                printf ("\n Display Log\n");
                LogPrint();             // Print the log entries;
               }
               break;
             case 'i':                  // Init RTC;
               if(State == STOPPED)     // Only execute if not logging;
               {
                printf ("\n Init RTC values\n");
                EA = 0;                 // Disable interrupts;
                LogInit(&LogRecord);    // Clear current time;
                EA = 1;                 // Reenable interrupts;
               }
               break;
             case 'p':                  // Stop logging;
               if(State != STOPPED)     // Only execute if not stopped already;
               {
                State = FINISHED;       // Set state to FINISHED
                printf ("\n Stop Logging\n");
                update = 1;             // Update one more time to
                                        // clean up results;
               }
               break;
             case 's':                  // Start logging
               if(State == STOPPED)     // Only execute if not logging;
               {
                printf ("\n Start Logging\n");
                State = RUNNING;        // Start logging data
               }
               break;
             case '?':                  // List commands;
               if(State == STOPPED)     // Only execute if not logging;
               {
                printf ("\n List Commands\n");
                MENU_ListCommands();    // List Commands
               }
               break;
             default:                   // Indicate unknown command;
               if(State == STOPPED)     // Only execute if not logging;
               {
                printf ("\n Unknown command: '%x'\n", key_press);
                MENU_ListCommands();    // Print Menu again;
               }
               break;
          } // switch
       } // if
       if(update)
       {
          update = 0;
          LogUpdate();
       }
    } // while
}

//-------------------------------------------------------------------------
// Support Subroutines
//-------------------------------------------------------------------------

//-------------------------------------------------------------------------
// MENU_ListCommands
```

```
//-----------------------------------------------------------------------------
// This routine prints a list of available commands.
//
void MENU_ListCommands (void)
{
   printf ("\nData logging example version %s\n", VERSION);
   printf ("Copyright 2004 Silicon Laboratories.\n\n");
   printf ("Command List\n");
   printf ("=============================================\n");
   printf (" 'c' - Clear Log\n");
   printf (" 'd' - Display Log\n");
   printf (" 'i' - Init RTC\n");
   printf (" 'p' - Stop Logging\n");
   printf (" 's' - Start Logging\n");
   printf (" '?' - List Commands\n");
   printf ("\n");
}

//-----------------------------------------------------------------------------
// Logging Subroutines
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
// LogUpdate()
//-----------------------------------------------------------------------------
// This routine is called by the ADC ISR at ~1Hz if State == RUNNING or
// FINISHED.  Here we read the decimated ADC value, convert it to temperature
// in hundredths of a degree C, and add the log entry to the log table buffer.
// If the buffer is full, or the user has stopped the logger, we must commit
// the buffer to the MMC FLASH.  <State> determines if the system is logging
// normally (State == RUNNING), or if the user has stopped the logging
// process (State == FINISHED).
//
//
void LogUpdate (void)
{
   static idata long temperature;      // long temperature value;

   idata int temp_int, temp_frac;      // Integer and fractional portions of
                                       // Temperature;
                                       // Count variable for number of
                                       // Log entries in local buffer;
   static idata unsigned int lLogCount = 0;
   unsigned char* l2_pointer;
   LOG_ENTRY record_buffer;
   LOG_ENTRY* l_pointer;
   EA = 0;                             // Disable interrupts (precautionary);
   temperature = Result.l;             // Retrieve 32-bit ADC value;
   record_buffer = LogRecord;
   EA = 1;                             // Re-enable interrupts;

                                       // Calculate temperature in hundredths
                                       // of a degree (16-bit full scale);
   temperature = (temperature * TEMP_VREF) - TEMP_OFFSET;
   temperature = 100 * temperature / TEMP_SLOPE;

   record_buffer.wTemp = (int)temperature; // Store temp value in temporary log
                                       // entry;
```

```
if(uLogCount == 0)                  // If the FLASH table has been cleared,
{                                   // The local buffer is reset;
   lLogCount = 0;                   // Reset number of local table entries;
                                    // Reset local buffer pointer;
   pLogTable = LOCAL_BLOCK;

}
if(State == RUNNING)                // Execute the following if the logger
{                                   // is logging normally;
                                    // Check to see if the log table is
                                    // full;
   if ((uLogCount*LOG_ENTRY_SIZE) < LOG_SIZE)
   {
      l_pointer = &record_buffer;
      l2_pointer = (unsigned char*)(l_pointer);
                                    // Write the current log entry to the
                                    // EEPROM buffer;
      EEPROM_WriteArray(pLogTable, l2_pointer,
                        LOG_ENTRY_SIZE);
      pLogTable+=LOG_ENTRY_SIZE;    // Increment buffer pointer;
      lLogCount++;                  // Increment local log entry count;
      uLogCount++;                  // Increment global log entry count;
                                    // If the buffer is full, it must be
                                    // written to FLASH;
      if(lLogCount == (unsigned int)(BUFFER_SIZE / LOG_ENTRY_SIZE))
      {
                                    // Call FLASH Write function;  Write to
                                    // address pointed at by the global
                                    // entry count less the EEPROM buffer
                                    // count;
         MMC_FLASH_Write((uLogCount -
                          (unsigned long)lLogCount)*LOG_ENTRY_SIZE,
                          SCRATCH_BLOCK, LOCAL_BLOCK, BUFFER_SIZE);
         lLogCount = 0;             // Reset the EEPROM buffer size
                                    // and pointer;
         pLogTable = LOCAL_BLOCK;
      }
                                    // Update display;
      temp_int = record_buffer.wTemp / 100;
      temp_frac = record_buffer.wTemp - ((long) temp_int * 100L);

   printf (" %08lu\t%04u: %02u:%02u:%02u ", uLogCount,
         (unsigned)record_buffer.uDay, (unsigned) record_buffer.bHour,
         (unsigned) record_buffer.bMin, (unsigned) record_buffer.bSec);

      printf ("%+02d.%02d\n", temp_int, temp_frac);

   }

   else                             // If the FLASH table is full, stop
   {                                // logging data and print the full
      State = STOPPED;              // message;
      printf ("Log is full\n");
   }
}
else if(State == FINISHED)          // If the data logger has been stopped
{                                   // by the user, write the local buffer
                                    // to FLASH;
```

```
      MMC_FLASH_Write((uLogCount - (unsigned long)lLogCount)*LOG_ENTRY_SIZE,
                                  SCRATCH_BLOCK, LOCAL_BLOCK,
                                  lLogCount*LOG_ENTRY_SIZE);
      lLogCount = 0;                  // Reset the local buffer size;
                                      // and pointer;
      pLogTable = LOCAL_BLOCK;
      State = STOPPED;                // Set the state to STOPPED;
   }
}

//-----------------------------------------------------------------------------
// LogFindCount()
//-----------------------------------------------------------------------------
// This function finds the number of entries already stored in the MMC log;
//
unsigned long LogFindCount()
{
   unsigned long Count = 0;           // Count variable, incremented as table
                                      // entries are read;
   unsigned long i = 0;               // Address variable, used to read table
                                      // table entries from FLASH;
   LOG_ENTRY Entry;
   LOG_ENTRY *TempEntry;              // Temporary log entry space;

                                      // Initialize temp space in
                                      // SCRATCH_BLOCK of external memory;

                                      // Loop through the table looking for a
                                      // blank entry;
   TempEntry = &Entry;
   for (i=LOG_ADDR;i<LOG_SIZE;i += LOG_ENTRY_SIZE)
   {
                                      // Read one entry from address i of
                                      // FLASH;
      MMC_FLASH_Read((unsigned long)(i), SCRATCH_BLOCK,
         (unsigned int)LOG_ENTRY_SIZE);
                                      // Move the entry from EEPROM space to
                                      // local memory for testing;
      EEPROM_ReadArray((unsigned char*)TempEntry, SCRATCH_BLOCK, LOG_ENTRY_SIZE);
                                      // Check if entry is blank;
      if ((TempEntry->bSec == 0x00)&&(TempEntry->bMin == 0x00)
           && (TempEntry->bHour == 0x00))
      {
                                      // If entry is blank, set Count;
         Count = (i/LOG_ENTRY_SIZE) - LOG_ADDR;
         break;                       // Break out of loop;
      }
   }
   return Count;                      // Return entry count;
}

//-----------------------------------------------------------------------------
// LogErase
//-----------------------------------------------------------------------------
// This function clears the log table using the FLASH Mass Erase capability.
//
void LogErase (void)
{
                                      // Call Mass Erase function with start
```

```
                                    // of table as address and log size as
                                    // length;
   MMC_FLASH_MassErase(LOG_ADDR, LOG_SIZE);
   uLogCount = 0;                   // Reset global count;
}

//-----------------------------------------------------------------------------
// LogPrint
//-----------------------------------------------------------------------------
// This function prints the log table.  Entries are read one at a time, temp
// is broken into the integer and fractional portions, and the log entry is
// displayed on the PC through UART.
//
void LogPrint (void)
{
   idata long temp_int, temp_frac;  // Integer and fractional portions of
                                    // temperature;
   idata unsigned long i;           // Log index;
   LOG_ENTRY Entry;
   LOG_ENTRY *TempEntry;

   TempEntry = &Entry;
   printf ("Entry#\tTime\t\tResult\n");// Print display column headers;
                                    // Assign pointers to local block;
                                    // FLASHRead function stores incoming
                                    // data at pchar, and then that data can
                                    // be accessed as log entries through
                                    // TempEntry;

   for (i = 0; i < uLogCount; i++)  // For each entry in the table,
   {                                // do the following;
                                    // Read the entry from FLASH;
      MMC_FLASH_Read((unsigned long)(LOG_ADDR + i*LOG_ENTRY_SIZE), SCRATCH_BLOCK,
         (unsigned int)LOG_ENTRY_SIZE);
                                    // Move entry from EEPROM space to local
                                    // memory;
      EEPROM_ReadArray((unsigned char*)TempEntry, SCRATCH_BLOCK, LOG_ENTRY_SIZE);
      // break temperature into integer and fractional components
      temp_int = (long) (TempEntry->wTemp) / 100L;
      temp_frac = (long) (TempEntry->wTemp) - ((long) temp_int * 100L);

      // display log entry
      printf (" %lu\t%03u: %02u:%02u:%02u ", (i + 1),
         TempEntry->uDay, (unsigned) TempEntry->bHour,
         (unsigned) TempEntry->bMin,
         (unsigned) TempEntry->bSec);
      printf ("%+02ld.%02ld\n", temp_int, temp_frac);

   }
}


//-----------------------------------------------------------------------------
// LogInit
//-----------------------------------------------------------------------------
// Initialize the Log Entry space (all zeros);
//
void LogInit (LOG_ENTRY *pEntry)
{
```

```
   pEntry->wTemp = 0;
   pEntry->uDay = 0;
   pEntry->bHour = 0;
   pEntry->bMin = 0;
   pEntry->bSec = 0;
}


//-----------------------------------------------------------------------------
// MMC_Command_Exec
//-----------------------------------------------------------------------------
//
// This function generates the necessary SPI traffic for all MMC SPI commands.
// The three parameters are described below:
//
// cmd:      This parameter is used to index into the command table and read
//           the desired command.  The Command Table Index Constants allow the
//           caller to use a meaningful constant name in the cmd parameter
//           instead of a simple index number.  For example, instead of calling
//           MMC_Command_Exec (0, argument, pchar) to send the MMC into idle
//           state, the user can call
//           MMC_Command_Exec (GO_IDLE_STATE, argument, pchar);
//
// argument: This parameter is used for MMC commands that require an argument.
//           MMC arguments are 32-bits long and can be values such as an
//           an address, a block length setting, or register settings for the
//           MMC.
//
// pchar:    This parameter is a pointer to the EEPROM data location for MMC
//           data operations.  When a read or write occurs, data will be stored
//           or retrieved from the location pointed to by pchar.
//
// The MMC_Command_Exec function indexes the command table using the cmd
// parameter. It reads the command table entry into memory and uses information
// from that entry to determine how to proceed.  Returns 16-bit card response
// value.
//

unsigned int MMC_Command_Exec (unsigned char cmd, unsigned long argument,
                               unsigned int pchar)
{
   idata command current_command;      // Local space for the command table
                                       // entry;
   idata ULONG long_arg;               // Union variable for easy byte
                                       // transfers of the argument;
                                       // Static variable that holds the
                                       // current data block length;
   static unsigned long current_blklen = 512;
                                       // Temp variable to preserve data block
   idata unsigned long old_blklen = 512;
                                       // length during temporary changes;
   idata unsigned int counter = 0;     // Byte counter for multi-byte fields;
   idata UINT card_response;           // Variable for storing card response;
   idata unsigned char data_resp;      // Variable for storing data response;
   idata unsigned char dummy_CRC;      // Dummy variable for storing CRC field;
                                       // Index variable for keeping track of
                                       // where you are in the EEPROM page;
   idata unsigned char EEPROM_BufferIndex;
```

```
current_command = commandlist[cmd]; // Retrieve desired command table entry
                                    // from code space;
SPI0DAT = 0xFF;                     // Send buffer SPI clocks to ensure no
while(!SPIF){}                      // MMC operations are pending;
SPIF = 0;
NSSMD0 = 0;                         // Select MMC by pulling CS low;
SPI0DAT = 0xFF;                     // Send another byte of SPI clocks;
while(!SPIF){}
SPIF = 0;
                                    // Issue command opcode;
SPI0DAT = (current_command.command_byte | 0x40);
long_arg.l = argument;              // Make argument byte addressable;
                                    // If current command changes block
                                    // length, update block length variable
                                    // to keep track;
if(current_command.command_byte == 16)
{
   current_blklen = argument;       // Command is a read, set local block
}                                   // length;
if((current_command.command_byte == 9)||
   (current_command.command_byte == 10))
{
   old_blklen = current_blklen;     // Command is a GET_CSD or GET_CID,
   current_blklen = 16;             // set block length to 16-bytes;
}
while(!SPIF){}                      // Wait for initial SPI transfer to end;
SPIF = 0;                           // Clear SPI Interrupt flag;

                                    // If an argument is required, transmit
                                    // one, otherwise transmit 4 bytes of
                                    // 0xFF;
if(current_command.arg_required == YES)
{
   counter = 0;
   while(counter <= 3)
   {
      SPI0DAT = long_arg.b[counter];
      counter++;
      while(!SPIF){}
      SPIF = 0;
   }
}
else
{
   counter = 0;
   while(counter <= 3)
   {
      SPI0DAT = 0x00;
      counter++;
      while(!SPIF){}
      SPIF = 0;
   }
}
SPI0DAT = current_command.CRC;      // Transmit CRC byte;  In all cases
while(!SPIF){}                      // except CMD0, this will be a dummy
SPIF = 0;                           // character;

                                    // The command table entry will indicate
```

```
                                    // what type of response to expect for
                                    // a given command;  The following
                                    // conditional handles the MMC response;
if(current_command.response == R1)  // Read the R1 response from the card;
{
   do
   {
      SPI0DAT = 0xFF;               // Write dummy value to SPI so that
      while(!SPIF){}                // the response byte will be shifted in;
      SPIF = 0;
      card_response.b[0] = SPI0DAT; // Save the response;
   }
   while((card_response.b[0] & BUSY_BIT));
}
                                    // Read the R1b response;
else if(current_command.response == R1b)
{
   do
   {
      SPI0DAT = 0xFF;               // Start SPI transfer;
      while(!SPIF){}
      SPIF = 0;
      card_response.b[0] = SPI0DAT; // Save card response
   }
   while((card_response.b[0] & BUSY_BIT));
   do                               // Wait for busy signal to end;
   {
      SPI0DAT = 0xFF;
      while(!SPIF){}
      SPIF = 0;
   }
   while(SPI0DAT == 0x00);          // When byte from card is non-zero,
}                                   // card is no longer busy;
                                    // Read R2 response
else if(current_command.response == R2)
{
   do
   {
      SPI0DAT = 0xFF;               // Start SPI transfer;
      while(!SPIF){}
      SPIF = 0;
      card_response.b[0] = SPI0DAT; // Read first byte of response;
   }
   while((card_response.b[0] & BUSY_BIT));
   SPI0DAT = 0xFF;
   while(!SPIF){}
   SPIF = 0;
   card_response.b[1] = SPI0DAT;    // Read second byte of response;
}
else                                // Read R3 response;
{
   do
   {
      SPI0DAT = 0xFF;               // Start SPI transfer;
      while(!SPIF){}
      SPIF = 0;
      card_response.b[0] = SPI0DAT; // Read first byte of response;
   }
   while((card_response.b[0] & BUSY_BIT));
```

```
      counter = 0;
      while(counter <= 3)              // Read next three bytes and store them
      {                                // in local memory;  These bytes make up
         counter++;                    // the Operating Conditions Register
         SPI0DAT = 0xFF;               // (OCR);
         while(!SPIF){}
         SPIF = 0;
         data_resp = SPI0DAT;
                                       // Send data read from MMC to EEPROM
                                       // buffer;
         EEPROM_WriteByte(pchar,data_resp);
         pchar++;
      }
   }
   switch(current_command.trans_type)  // This conditional handles all data
   {                                   // operations;  The command entry
                                       // determines what type, if any, data
                                       // operations need to occur;
      case RD:                         // Read data from the MMC;
         do                            // Wait for a start read token from
         {                             // the MMC;
            SPI0DAT = 0xFF;            // Start a SPI transfer;
            while(!SPIF){}
            SPIF = 0;
         }
         while(SPI0DAT != START_SBR);  // Check for a start read token;
         counter = 0;                  // Reset byte counter;
                                       // Read <current_blklen> bytes;
         EEPROM_BufferIndex = 0;
         while(counter < (unsigned int)current_blklen)
         {
            SPI0DAT = 0x00;            // Start SPI transfer;
            while(!SPIF){}
            SPIF = 0;
                                       // Build a small buffer of data
                                       // in local memory;
            EEPROM_PageBuffer[EEPROM_BufferIndex] = SPI0DAT;
            EEPROM_BufferIndex++;
                                       // If the buffer reaches the size of a
                                       // physical EEPROM page, write buffered
                                       // data out to EEPROM and reset buffer;
            if(EEPROM_BufferIndex >= EEPROM_PAGE_SIZE)
            {
                                       // Send data to EEPROM;
               EEPROM_WriteArray(pchar, &EEPROM_PageBuffer[0], EEPROM_PAGE_SIZE);
                                       // Increment EEPROM page;
               pchar += EEPROM_PAGE_SIZE;
               EEPROM_BufferIndex = 0; // Reset local buffer index;
            }
            counter++;                 // Increment data byte counter;
         }
                                       // Write any remaining buffered data to
         if(EEPROM_BufferIndex != 0)   // EEPROM;
            EEPROM_WriteArray(pchar, &EEPROM_PageBuffer[0], EEPROM_BufferIndex);

         SPI0DAT = 0x00;               // After all data is read, read the two
         while(!SPIF){}                // CRC bytes;  These bytes are not used
         SPIF = 0;                     // in this mode, but the placeholders
         dummy_CRC = SPI0DAT;          // must be read anyway;
```

```
   SPI0DAT = 0x00;
   while(!SPIF){}
   SPIF = 0;
   dummy_CRC = SPI0DAT;
   break;
case WR:                          // Write data to the MMC;
   SPI0DAT = 0xFF;                // Start by sending 8 SPI clocks so
   while(!SPIF){}                 // the MMC can prepare for the write;
   SPIF = 0;
   SPI0DAT = START_SBW;           // Send the start write block token;
   counter = 0;                   // Reset byte counter;
   while(!SPIF){}
   SPIF = 0;
                                  // Prime Buffer Index
   EEPROM_BufferIndex = EEPROM_PAGE_SIZE;
                                  // Write <current_blklen> bytes to MMC;
   while(counter < (unsigned int)current_blklen)
   {                              // When EEPROM page is full, reset local
                                  // buffer and move to next EEPROM page;
      if(EEPROM_BufferIndex > EEPROM_PAGE_SIZE-1)
      {
                                  // Send data to EEPROM;
         EEPROM_ReadArray(&EEPROM_PageBuffer[0],pchar,EEPROM_PAGE_SIZE);
                                  // Reset local buffer index;
         EEPROM_BufferIndex = 0;
                                  // Update EEPROM page;
         pchar += EEPROM_PAGE_SIZE;
      }
                                  // Write data byte out through SPI;
      SPI0DAT = EEPROM_PageBuffer[EEPROM_BufferIndex];
      EEPROM_BufferIndex++;
      counter++;                  // Increment byte counter;
      while(!SPIF){}
      SPIF = 0;
   }

   SPI0DAT = 0xFF;                // Write CRC bytes (don't cares);
   while(!SPIF){}
   SPIF = 0;
   SPI0DAT = 0xFF;
   while(!SPIF){}
   SPIF = 0;

   do                             // Read Data Response from card;
   {                              //
      SPI0DAT = 0xFF;
      while(!SPIF){}
      SPIF = 0;
      data_resp = SPI0DAT;
   }                              // When bit 0 of the MMC response
                                  // is clear, a valid data response
                                  // has been received;
   while((data_resp & DATA_RESP_MASK) != 0x01);

   do                             // Wait for end of busy signal;
   {
      SPI0DAT = 0xFF;             // Start SPI transfer to receive
      while(!SPIF){}              // busy tokens;
      SPIF = 0;
```

```
         }
         while(SPI0DAT == 0x00);       // When a non-zero token is returned,
                                       // card is no longer busy;
         SPI0DAT = 0xFF;               // Issue 8 SPI clocks so that all card
         while(!SPIF){}                // operations can complete;
         SPIF = 0;
         break;
      default: break;
   }
   SPI0DAT = 0xFF;
   while(!SPIF){}
   SPIF = 0;


   NSSMD0 = 1;                         // Deselect memory card;
   SPI0DAT = 0xFF;                     // Send 8 more SPI clocks to ensure
   while(!SPIF){}                      // the card has finished all necessary
   SPIF = 0;                           // operations;
                                       // Restore old block length if needed;
   if((current_command.command_byte == 9)||
      (current_command.command_byte == 10))
   {
      current_blklen = old_blklen;
   }
   return card_response.i;
}


//-----------------------------------------------------------------------------
// MMC_FLASH_Init
//-----------------------------------------------------------------------------
//
// This function initializes the flash card, configures it to operate in SPI
// mode, and reads the operating conditions register to ensure that the device
// has initialized correctly.  It also determines the size of the card by
// reading the Card Specific Data Register (CSD).

void MMC_FLASH_Init (void)
{
   idata UINT card_status;             // Stores card status returned from
                                       // MMC function calls(MMC_Command_Exec);
   idata unsigned char counter = 0;    // SPI byte counter;
   idata unsigned int size;            // Stores size variable from card;
   unsigned int pchar;                 // Pointer int EEPROM for storing MMC
                                       // register values;
   unsigned char space[4];             // 4 bytes of memory for use in storing
                                       // temporary register values from MMC;
   unsigned char *lpchar;              // Local storage for data from MMC;
                                       // Transmit at least 64 SPI clocks
                                       // before any bus comm occurs.

   pchar = LOCAL_BLOCK;
   lpchar = &space[0];
   for(counter = 0; counter < 10; counter++)
   {
      SPI0DAT = 0xFF;
      while(!SPIF){}
      SPIF = 0;
   }
```

```
                                    // Send the GO_IDLE_STATE command with
                                    // CS driven low;  This causes the MMC
                                    // to enter SPI mode;
   card_status.i = MMC_Command_Exec(GO_IDLE_STATE,EMPTY,EMPTY);
                                    // Send the SEND_OP_COND command
   do                               // until the MMC indicates that it is
   {                                // no longer busy (ready for commands);
      SPI0DAT = 0xFF;
      while(!SPIF){}
      SPIF = 0;
      card_status.i = MMC_Command_Exec(SEND_OP_COND,EMPTY,EMPTY);
   }
   while ((card_status.b[0] & 0x01));
   SPI0DAT = 0xFF;                  // Send 8 more SPI clocks to complete
   while(!SPIF){}                   // the initialization sequence;
   SPIF = 0;
   do                               // Read the Operating Conditions
   {                                // Register (OCR);
      card_status.i = MMC_Command_Exec(READ_OCR,EMPTY,pchar);
                                    // Get OCR from EEPROM;
      EEPROM_ReadArray(lpchar,pchar,4);
   }
   while(!(*lpchar&0x80));          // Test for card ready;

   card_status.i = MMC_Command_Exec(SEND_STATUS,EMPTY,EMPTY);
                                    // Get the Card Specific Data (CSD)
                                    // register to determine the size of the
                                    // MMC;
   card_status.i = MMC_Command_Exec(SEND_CSD,EMPTY,pchar);
                                    // Get CSD from EEPROM;
   *lpchar = EEPROM_ReadByte(pchar+9);
                                    // Size indicator is in the 9th byte of
                                    // CSD register;
                                    // Extract size indicator bits;
   size = (unsigned int)((((*lpchar) & 0x03) << 1) |
                         (((*(lpchar+1)) & 0x80) >> 7));
   switch(size)                     // Assign PHYSICAL_SIZE variable to
   {                                // appropriate size constant;
      case 1: PHYSICAL_SIZE = PS_8MB;   break;
      case 2: PHYSICAL_SIZE = PS_16MB;  break;
      case 3: PHYSICAL_SIZE = PS_32MB;  break;
      case 4: PHYSICAL_SIZE = PS_64MB;  break;
      case 5: PHYSICAL_SIZE = PS_128MB; break;
      default: break;
   }
                                    // Determine the number of MMC sectors;
   PHYSICAL_BLOCKS = PHYSICAL_SIZE / PHYSICAL_BLOCK_SIZE;
   LOG_SIZE = PHYSICAL_SIZE - LOG_ADDR;
}
//-----------------------------------------------------------------------------
// MMC_FLASH_Read
//-----------------------------------------------------------------------------
//
// This function reads <length> bytes of MMC data from address <address>, and
// stores them in EEPROM space at the location pointed to by <pchar>.
// There are two cases that must be considered when performing a read.  If the
// requested data is located entirely in a single FLASH block, the function
// sets the read length appropriately and issues a read command.  If requested
// data crosses a FLASH block boundary, the read operation is broken into two
```

```
// parts.  The first part reads data from the starting address to the end of
// the starting block, and then reads from the start of the next block to the
// end of the requested data.  Before each read, the read length must be set
// to the proper value.
unsigned char MMC_FLASH_Read (unsigned long address, unsigned int pchar,
                        unsigned int length)
   {
   idata unsigned long flash_page_1;   // Stores address of first FLASH page;
   idata unsigned long flash_page_2;   // Stores address of second FLASH page;
   idata unsigned int card_status;     // Stores MMC status after each MMC
                                       // command;

   if(length > 512) return 0;          // Test for valid data length;  Length
                                       // must be less than 512 bytes;
                                       // Find address of first FLASH block;
   flash_page_1 = address & ~(PHYSICAL_BLOCK_SIZE-1);
                                       // Find address of second FLASH block;
   flash_page_2 = (address+length-1) & ~(PHYSICAL_BLOCK_SIZE-1);
   if(flash_page_1 == flash_page_2)    // Execute the following if data is
   {                                   // located within one FLASH block;
                                       // Set read length to requested data
                                       // length;
      card_status = MMC_Command_Exec(SET_BLOCKLEN,(unsigned long)length,
                                  EMPTY);
                                       // Issue read command;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,address,pchar);
   }
   else                                // Execute the following if data crosses
   {                                   // MMC block boundary;
                                       // Set the read length to the length
                                       // from the starting address to the
                                       // end of the first FLASH page;
      card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                  (unsigned long)(flash_page_2 - address),
                                   EMPTY);
                                       // Issue read command;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,address,pchar);
                                       // Set read length to the length from
                                       // the start of the second FLASH page
                                       // to the end of the data;
      card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                  (unsigned long)length -
                                  (flash_page_2 - address),
                                   EMPTY);
                                       // Issue second read command;  Notice
                                       // that the incoming data stored in
                                       // external RAM must be offset from the
                                       // original pointer value by the length
                                       // of data stored during the first read
                                       // operation;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_2,
                                   pchar + (flash_page_2 - address));
   }
}


//-----------------------------------------------------------------------------
// MMC_FLASH_Write
//-----------------------------------------------------------------------------
```

```
//
// This function operates much like the MMC_FLASH_Read function.  As
// with the MMC_FLASH_Read function, if the desired write space crosses a FLASH
// block boundary, the operation must be broken into two pieces.
// MMC_FLASH_Write first reads the addressed block of MMC data into the EEPROM,
// then modifies the contents of that block in EEPROM space;  Finally, the MMC
// block is erased in the MMC and replaced by the updated block from EEPROM
// space;
//
unsigned char MMC_FLASH_Write (unsigned long address, unsigned int scratch,
                          unsigned int wdata, unsigned int length)
{
   idata unsigned long flash_page_1;   // First FLASH page address;
   idata unsigned long flash_page_2;   // Second FLASH page address;
   idata unsigned int card_status;     // Stores status returned from MMC;
   idata unsigned int counter;         // Byte counter used for writes to
                                       // local copy of data block;
   idata unsigned int temp_length;
   unsigned int index;                 // Pointer into local copy of data
                                       // block, used during modification;
   if(length > 512) return 0;          // Check for valid data length;
                                       // Calculate first FLASH page address;
   flash_page_1 = address & ~(PHYSICAL_BLOCK_SIZE-1);
                                       // Calculate second FLASH page address;
   flash_page_2 = (address+length-1) & ~(PHYSICAL_BLOCK_SIZE-1);
   if(flash_page_1 == flash_page_2)    // Handle single FLASH block condition;
   {
                                       // Set block length to default block
                                       // size (512 bytes);
      card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                (unsigned long)PHYSICAL_BLOCK_SIZE,
                                EMPTY);
                                       // Read data block into EEPROM;
      card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_1,scratch);
      index = (unsigned int)(address % PHYSICAL_BLOCK_SIZE) + scratch;
      counter = 0;
      while(length>0)                  // This loop updates the temporary
      {                                // MMC Page(in EEPROM) with the contents
         if(length<=64)                // of the current temperature buffer;
         {
            counter = length;
            length = 0;
         }
         else
         {
            counter = 64;
            length -= 64;
         }
                                       // Read temperature data
         EEPROM_ReadArray(&EEPROM_PageBuffer[0],wdata,counter);
                                       // Store in temp MMC page(in EEPROM);
         EEPROM_WriteArray(index, &EEPROM_PageBuffer[0], counter);
         wdata+=EEPROM_PAGE_SIZE;      // Update temperature buffer pointer;
         index+=EEPROM_PAGE_SIZE;      // Update temp MMC page (in EEPROM)
                                       // pointer;
         counter = 0;
      }
                                       // Write modified block back to MMC;
      card_status = MMC_Command_Exec(WRITE_BLOCK,flash_page_1,scratch);
```

```
}
else                                    // Handle multiple FLASH block
{                                       // condition;
                                        // Set block length to default block
                                        // size (512 bytes);
   card_status = MMC_Command_Exec(SET_BLOCKLEN,
                                (unsigned long)PHYSICAL_BLOCK_SIZE,
                                EMPTY);
                                        // Read first data block into EEPROM;
   card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_1,scratch);
   index = (unsigned int)(address % PHYSICAL_BLOCK_SIZE) + scratch;
   temp_length = length;
   length = (unsigned int)(flash_page_2 - address);
   while(length>0)                      // Modify write space in EEPROM copy;
   {
      if(length<=64)
      {
         counter = length;
         length = 0;
      }
      else
      {
         counter = 64;
         length -= 64;
      }
                                        // Read temperature data from buffer;
      EEPROM_ReadArray(&EEPROM_PageBuffer[0],wdata,counter);
                                        // Write temperature data to copied
                                        // MMC page;
      EEPROM_WriteArray(index, &EEPROM_PageBuffer[0], counter);
      wdata+=counter;
      index+=counter;
      counter = 0;
   }
                                        // Write EEPROM copy back to MMC;
   card_status = MMC_Command_Exec(WRITE_BLOCK,flash_page_1,scratch);
                                        // Read second data block;
   card_status = MMC_Command_Exec(READ_SINGLE_BLOCK,flash_page_2,scratch);
   index = scratch;
   length = (unsigned int)(temp_length - (flash_page_2 - address));
   while(length>0)                      // Modify write space in EEPROM copy;
   {
      if(length<=64)
      {
         counter = length;
         length = 0;
      }
      else
      {
         counter = 64;
         length -= 64;
      }
                                        // Read temperature data from buffer;
      EEPROM_ReadArray(&EEPROM_PageBuffer[0],wdata,counter);
                                        // Write data to MMC page copy in
                                        // EEPROM;
      EEPROM_WriteArray(index, &EEPROM_PageBuffer[0], counter);
      wdata+=counter;
      index+=counter;
```

```
         counter = 0;
      }
                                       // Write EEPROM copy back to MMC;
      card_status = MMC_Command_Exec(WRITE_BLOCK,flash_page_2,scratch);
   }
}

//-----------------------------------------------------------------------------
// MMC_FLASH_MassErase
//-----------------------------------------------------------------------------
//
// This function erases <length> bytes of flash starting with the block
// indicated by <address1>.  This function only handles sector-sized erases
// or larger.  Function should be called with sector-aligned erase addresses.
unsigned char MMC_FLASH_MassErase (unsigned long address1,
                                   unsigned long length)
{
   idata unsigned char card_status;    // Stores card status returned from MMC;
                                       // Store start and end sectors to be
                                       // to be erased;
   idata unsigned long flash_page_1, flash_page_2;
                                       // Store start and end groups to be
                                       // erased;
   idata unsigned long flash_group_1, flash_group_2;
                                       // Compute first sector address for
                                       // erase;
   flash_page_1 = address1 & ~(PHYSICAL_BLOCK_SIZE-1);
                                       // Compute first group address for
                                       // erase;
   flash_group_1 = flash_page_1 &~(PHYSICAL_GROUP_SIZE-1);
                                       // Compute last sector address for
                                       // erase;
   flash_page_2 = (address1 + length) & ~(PHYSICAL_BLOCK_SIZE-1);
                                       // Compute last group address for erase;
   flash_group_2 = flash_page_2 &~(PHYSICAL_GROUP_SIZE-1);

   if(flash_group_1 == flash_group_2)  // Handle condition where entire erase
   {                                   // space is in one erase group;
                                       // Tag first sector;
      card_status = MMC_Command_Exec(TAG_SECTOR_START,flash_page_1,EMPTY);
                                       // Tag last sector;
      card_status = MMC_Command_Exec(TAG_SECTOR_END,flash_page_2,EMPTY);
                                       // Issue erase command;
      card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
   }
   else                                // Handle condition where erase space
   {                                   // crosses an erase group boundary;
                                       // Tag first erase sector;
      card_status = MMC_Command_Exec(TAG_SECTOR_START,flash_page_1,EMPTY);
                                       // Tag last sector of first group;
      card_status = MMC_Command_Exec(TAG_SECTOR_END,
                                (flash_group_1 +
                                (unsigned long)(PHYSICAL_GROUP_SIZE
                                - PHYSICAL_BLOCK_SIZE)),EMPTY);
                                       // Issue erase command;
      card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
                                       // Tag first sector of last erase group;
      card_status = MMC_Command_Exec(TAG_SECTOR_START,flash_group_2,EMPTY);
                                       // Tag last erase sector;
```

```
      card_status = MMC_Command_Exec(TAG_SECTOR_END,flash_page_2,EMPTY);
                                       // Issue erase;
      card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
                                       // Conditional that erases all groups
                                       // between first and last group;
      if(flash_group_2 > (flash_group_1 + PHYSICAL_GROUP_SIZE))
      {
                                       // Tag first whole group to be erased;
         card_status = MMC_Command_Exec(TAG_ERASE_GROUP_START,
                                    (flash_group_1 +
                                    (unsigned long)PHYSICAL_GROUP_SIZE),EMPTY);
                                       // Tag last whole group to be erased;
         card_status = MMC_Command_Exec(TAG_ERASE_GROUP_END,
                                    (flash_page_2 -
                                    (unsigned long)PHYSICAL_GROUP_SIZE),EMPTY);
                                       // Issue erase command;
         card_status = MMC_Command_Exec(ERASE,EMPTY,EMPTY);
      }
   }

   return card_status;
}
// SMBus Subroutines

//-----------------------------------------------------------------------------
// EEPROM_WriteArray()
//
// This function sends a block of data to an EEPROM using SMBus.  EEPROM writes
// must be performed on only one EEPROM page at a time, so writes that span
// more than one page must be broken into multiple smaller writes.  Write
// length is limited to 64 bytes (the length of one EEPROM page).
//-----------------------------------------------------------------------------
//
//

void EEPROM_WriteArray(unsigned int addr, unsigned char* SrcAddr,
                       unsigned char len)
{
   unsigned int EEPROM_Page1;
   unsigned int EEPROM_Page2;
   unsigned char section_count;
   unsigned int temp_len;
                                       // Compute the EEPROM pages involved in
                                       // the data transmission;
   EEPROM_Page1 = addr & ~(EEPROM_PAGE_SIZE - 1);
   EEPROM_Page2 = (addr+(unsigned int)len - 1) & ~(EEPROM_PAGE_SIZE - 1);
                                       // If more than one EEPROM page is
                                       // involved, break the write into
                                       // two separate writes.
   if(EEPROM_Page1 != EEPROM_Page2)
   {
                                       // Calculate the length of the second
                                       // write;
      temp_len = (addr + len) - EEPROM_Page2;
                                       // Calculate the length of the first
                                       // write;
      len = EEPROM_Page2 - addr;
                                       // Setup loop for two seperate writes;
      section_count = 2;
```

```
   }
   else
      section_count = 1;               // Otherwise only a single write is
                                       // necessary;

   while(section_count >= 1)
   {
      while(SMB_BUSY);                 // okay to spin here because
                                       // SMBus Rate >> Temp sample rate
                                       // Set up SMBus for EEPROM write;
      SMB_BUSY = 1;
      TARGET = EEPROM_ADDR;            // Target EEPROM device;
      SMB_RW = WRITE;                  // Indicate write operation;
      SMB_SENDWORDADDR = 1;            // Send word addess after slave address;
      SMB_RANDOMREAD = 0;              // Send a START after the word address;
      SMB_ACKPOLL = 1;                 // Indicate poll for acknoledgement;
      WORD_ADDR.i = addr;              // Indicate data address;
      pSMB_DATA_OUT = SrcAddr;         // Pointer to data;
      SMB_DATA_LEN = len;              // Data length;

      STA = 1;                         // Start SMBus transaction
      while(SMB_BUSY);                 // Poll for SMBus not busy;
      section_count--;
      if(section_count==1)             // Update length and address and
      {                                // perform additional write if needed;
         SrcAddr+=len;
         len = temp_len;
         addr = EEPROM_Page2;
      }
   }
}

//-----------------------------------------------------------------------------
// EEPROM_ReadArray ()
//
// Reads up to 64 data bytes from the EEPROM slave specified by the
// <EEPROM_ADDR> constant.
//-----------------------------------------------------------------------------
void EEPROM_ReadArray (unsigned char* dest_addr, unsigned int src_addr,
                       unsigned char len)
{
   while (SMB_BUSY);                   // Wait for SMBus to be free;
   SMB_BUSY = 1;                       // Claim SMBus (set to busy);

   // Set SMBus ISR parameters
   TARGET = EEPROM_ADDR;               // Set target slave address;
   SMB_RW = WRITE;                     // A random read starts as a
                                       // write then changes to a read
                                       // after the repeated start is
                                       // sent; The ISR handles this
                                       // switchover if the ;
                                       // <SMB_RANDOMREAD> bit is set;
   SMB_SENDWORDADDR = 1;               // Send Word Address after Slave
                                       // Address;
   SMB_RANDOMREAD = 1;                 // Send a START after the word
                                       // address;
   SMB_ACKPOLL = 1;                    // Enable Acknowledge Polling;

   // Specify the Incoming Data
```

```
   WORD_ADDR.i = src_addr;             // Set the target address in the
                                       // EEPROM's internal memory space;
                                       // Set the the incoming data pointer;
   pSMB_DATA_IN = (unsigned char*) dest_addr;

   SMB_DATA_LEN = len;                 // Specify to ISR that the next
                                       // transfer will contain <len> data
                                       // bytes;

   // Initiate SMBus Transfer
   STA = 1;
   while(SMB_BUSY);                    // Wait until data is read;

}


//-----------------------------------------------------------------------------
// EEPROM_WriteByte ()
//
// This function writes the value in <dat> to location <addr> in the EEPROM
// then polls the EEPROM until the write is complete.
//-----------------------------------------------------------------------------
void EEPROM_WriteByte( unsigned int addr, unsigned char dat )
{
   while (SMB_BUSY);                   // Wait for SMBus to be free;
   SMB_BUSY = 1;                       // Claim SMBus (set to busy);

   // Set SMBus ISR parameters
   TARGET = EEPROM_ADDR;               // Set target slave address;
   SMB_RW = WRITE;                     // Mark next transfer as a write;
   SMB_SENDWORDADDR = 1;               // Send Word Address after Slave
                                       // Address;
   SMB_RANDOMREAD = 0;                 // Do not send a START signal after
                                       // the word address;
   SMB_ACKPOLL = 1;                    // Enable Acknowledge Polling (The ISR
                                       // will automatically restart the
                                       // transfer if the slave does not
                                       // acknoledge its address;

   // Specify the Outgoing Data
   WORD_ADDR.i = addr;                 // Set the target address in the
                                       // EEPROM's internal memory space;

   SMB_SINGLEBYTE_OUT = dat;           // Store dat (local variable) in a
                                       // global variable so the ISR can read
                                       // it after this function exits;

   pSMB_DATA_OUT = &SMB_SINGLEBYTE_OUT;// The outgoing data pointer points to
                                       // the <dat> variable;

   SMB_DATA_LEN = 1;                   // Specify to ISR that the next transfer
                                       // will contain one data byte;

   // Initiate SMBus Transfer
   STA = 1;
   while(SMB_BUSY);
}
```

```
//-----------------------------------------------------------------------------
// EEPROM_ReadByte ()
//
// This function returns a single byte from location <addr> in the EEPROM then
// polls the <SMB_BUSY> flag until the read is complete.
//-----------------------------------------------------------------------------
unsigned char EEPROM_ReadByte( unsigned int addr)
{
   unsigned char retval;               // Holds the return value;

   while (SMB_BUSY);                   // Wait for SMBus to be free;
   SMB_BUSY = 1;                       // Claim SMBus (set to busy);

   // Set SMBus ISR parameters
   TARGET = EEPROM_ADDR;               // Set target slave address;
   SMB_RW = WRITE;                     // A random read starts as a write
                                       // then changes to a read after
                                       // the repeated start is sent; The
                                       // ISR handles this switchover if
                                       // the <SMB_RANDOMREAD> bit is set;
   SMB_SENDWORDADDR = 1;               // Send Word Address after Slave
                                       // Address;
   SMB_RANDOMREAD = 1;                 // Send a START after the word address;
   SMB_ACKPOLL = 1;                    // Enable Acknowledge Polling;


   // Specify the Incoming Data
   WORD_ADDR.i = addr;                 // Set the target address in the EEPROM
                                       // internal memory space;

   pSMB_DATA_IN = &retval;             // The incoming data pointer points to
                                       // the <retval> variable;

   SMB_DATA_LEN = 1;                   // Specify to ISR that the next transfer
                                       // will contain one data byte;

   // Initiate SMBus Transfer
   STA = 1;
   while(SMB_BUSY);                    // Wait until data is read;

   return retval;

}

void EEPROM_Clear (unsigned int addr, unsigned int length)
{
   unsigned int i;
   for(i = addr; i < (addr+length); i++)
      EEPROM_WriteByte(i,EMPTY);
}
//-----------------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// ADC0_ISR
//-----------------------------------------------------------------------------
//
// ADC0 end-of-conversion ISR
```

```
// Here we take the ADC0 sample, add it to a running total <accumulator>, and
// decrement our local decimation counter <int_dec>.  When <int_dec> reaches
// zero, we post the decimated result in the global variable <result>.
//
// In addition, this ISR is used to keep track of time.  Every 4096 samples,
// approximately once a second, we update the seconds, minutes, hours, and days
// for the temperature timestamp.  If the global state is RUNNING or FINISHED,
// a low priority software interrupt is generated and the log is updated.
// Using the low priority interrupt allows the MMC communication to execute
// without disrupting the temperature sampling process.  The ADC end-of-conv
// interrupt is set to high priority, so it will be executed even if a low
// priority interrupt is already in progress.
void ADC0_ISR (void) interrupt 10 using 3
{
   static unsigned int_dec=INT_DEC;    // integrate/decimate counter
                                       // we post a new result when
                                       // int_dec = 0
   static LONG accumulator={0L};       // here's where we integrate the
                                       // ADC samples
   xdata unsigned int buffer[256];

   unsigned char count;

   AD0INT = 0;                         // clear ADC conversion complete
                                       // indicator
   if(int_dec % 16 == 0)
   {
      buffer[count] = ADC0;
      count++;
   }
    accumulator.l += ADC0;              // read ADC value and add to running
                                       // total
   int_dec--;                          // update decimation counter

   if (int_dec == 0)                   // if zero, then post result
   {
      int_dec = INT_DEC;               // reset counter

      // Result = accumulator >> 6
      // Perform my shifting left 2, then byte-swapping
      accumulator.l <<= 2;             // accumulator = accumulator << 2
      Result.b[0] = 0;                 // Result = accumulator >> 8
      Result.b[1] = accumulator.b[0];
      Result.b[2] = accumulator.b[1];
      Result.b[3] = accumulator.b[2];
      accumulator.l = 0L;              // reset accumulator
      LogRecord.bSec++;                // update seconds counter
      if (LogRecord.bSec == 60)
      {
         LogRecord.bSec = 0;
         LogRecord.bMin++;             // update minutes counter
         if (LogRecord.bMin == 60)
         {
            LogRecord.bMin = 0;
            LogRecord.bHour++;         // update hours counter
            if (LogRecord.bHour == 24)
            {
               LogRecord.bHour = 0;
               LogRecord.uDay++;       // update days counter
```

```
            }
         }
      }

      if ((State == RUNNING)||(State == FINISHED))
      {
         update = 1;
      }
   }
}



//-----------------------------------------------------------------------------
// SMBus Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// SMBus ISR state machine
// - Master only implementation - no slave or arbitration states defined
// - All incoming data is written starting at the global pointer <pSMB_DATA_IN>
// - All outgoing data is read from the global pointer <pSMB_DATA_OUT>
//
void SMBus_ISR (void) interrupt 7 using 2
{
   bit FAIL = 0;                       // Used by the ISR to flag failed
                                       // transfers;

   static char i;                      // Used by the ISR to count the
                                       // number of data bytes sent or
                                       // received;

   static bit SEND_START = 0;          // Send a start;

   switch (SMB0CN & 0xF0)              // Status vector;
   {
      // Master Transmitter/Receiver: START condition transmitted;
      case SMB_MTSTA:
         if(!SMB_BUSY)
         {
            break;
         }
         SMB0DAT = TARGET;             // Load address of the target;
         SMB0DAT |= SMB_RW;            // Load R/W bit;
         STA = 0;                      // Manually clear START bit;
         i = 0;                        // Reset data byte counter;
         SMB_HIGHBYTENOTSENT = 1;
         break;

      // Master Transmitter: Data byte (or Slave Address) transmitted;
      case SMB_MTDB:
         if (ACK)                      // Slave Address or Data Byte
         {                             // Acknowledged?
            if (SEND_START)
            {
               STA = 1;
               SEND_START = 0;
               break;
            }
            if(SMB_SENDWORDADDR)       // Are we sending the word address?
```

```
   {

      if(SMB_HIGHBYTENOTSENT)
      {
                              // Send word address;
         SMB0DAT = WORD_ADDR.b[0];
                              // Clear flag;
         SMB_HIGHBYTENOTSENT = 0;
         break;
      }
      else
      {
         SMB0DAT = WORD_ADDR.b[1];
                              // Clear flag;
         SMB_SENDWORDADDR = 0;

      }

      if (SMB_RANDOMREAD)
      {
         SEND_START = 1;      // Send START after the next ACK cycle;
         SMB_RW = READ;
      }

      break;
   }

   if (SMB_RW==WRITE)         // Is this transfer a WRITE?
   {

      if (i < SMB_DATA_LEN)   // Is there data to send?
      {                       // Send data byte;
         SMB0DAT = (unsigned char)*pSMB_DATA_OUT;
         pSMB_DATA_OUT++;     // Increment data out pointer;
         i++;                 // Increment number of bytes sent;
      }
      else
      {
        STO = 1;              // Set STO to terminate transfer;
        SMB_BUSY = 0;         // Clear software busy flag;
      }
   }
   else {}                    // If this transfer is a READ,
                              // then take no action; Slave
                              // address was transmitted; A
                              // separate 'case' is defined
                              // for data byte recieved;
}
else                          // If slave NACK;
{
   if(SMB_ACKPOLL)
   {
      STA = 1;                // Restart transfer;
   }
   else
   {
      FAIL = 1;               // Indicate failed transfer
   }                          // and handle at end of ISR;
```

```
         }
         break;

      // Master Receiver: byte received
      case SMB_MRDB:
         if ( i < SMB_DATA_LEN )       // Is there any data remaining?
         {
            *pSMB_DATA_IN = SMB0DAT;   // Store received byte;
            pSMB_DATA_IN++;            // Increment data in pointer;
            i++;                       // Increment number of bytes received;
            ACK = 1;                   // Set ACK bit (may be cleared later
                                       // in the code);

         }

         if (i == SMB_DATA_LEN)        // This is the last byte;
         {
            SMB_BUSY = 0;              // Free SMBus interface;
            ACK = 0;                   // Send NACK to indicate last byte
                                       // of this transfer;
            STO = 1;                   // Send STOP to terminate transfer;
         }

         break;

      default:
         FAIL = 1;                     // Indicate failed transfer
                                       // and handle at end of ISR;
         break;
   }

   if (FAIL)                           // If the transfer failed,
   {                                   // reset communication;
      SMB0CF &= ~0x80;
      SMB0CF |= 0x80;
      SMB_BUSY = 0;                    // Free SMBus;
   }

   SI=0;                               // Clear interrupt flag;
}


//-----------------------------------------------------------------------------
// Initialization Subroutines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use the internal
// oscillator as its clock source.  Enables missing clock detector reset.

void SYSCLK_Init (void)
{
   OSCICN |= 0x03;                     // Configure internal oscillator for
                                       // its maximum frequency;
   VDM0CN  = 0x08;                     // Enable VDD Monitor;
   RSTSRC |= 0x06;                     // Enable missing clock detector and
```

```
                                    // VDD Monitor as reset sources;
   CLKMUL = 0x00;                   // Reset multiplier; Internal Osc is
                                    // multiplier source;
   CLKMUL |= 0x80;                  // Enable Clock Multiplier;

// Wait 5 us for multiplier to be enabled
   TMR2CN = 0x00;                   // STOP Timer2; Clear TF2H and TF2L;
                                    // disable low-byte interrupt; disable
                                    // split mode; select internal timebase
   CKCON |= 0x10;                   // Timer2 uses SYSCLK as its timebase

   TMR2RL  = START_SYSCLK / 200000; // Init reload values 12 MHz / (5E^-6)
   TMR2    = TMR2RL;                // Init Timer2 with reload value
   ET2 = 0;                         // disable Timer2 interrupts
   TF2H = 0;
   TR2 = 1;                         // start Timer2
   while (!TF2H);                   // wait for overflow
   TF2H = 0;                        // clear overflow indicator
   TR2 = 0;                         // Stop Timer2;

   CLKMUL |= 0xC0;                  // Initialize Clock Multiplier
   while(!(CLKMUL & 0x20))          // Wait for MULRDY
   CLKSEL = 0x02;                   // Select SYSCLK * 4 / 2 as clock source
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.

// P0.0 - SCK (push-pull)
// P0.1 - MISO
// P0.2 - MOSI (push-pull)
// P0.3 - NSS (push-pull)

// P0.4 - UART TX (push-pull)
// P0.5 - UART RX
// P0.6 - SDA
// P0.7 - VREF (Skipped by crossbar)
// P1.0 - SCL
// P1.1 - SDA
// P1.2 - SCL


void PORT_Init (void)
{
   P0SKIP  = 0xC0;                  // Skip VREF in crossbar
                                    // assignments; Skip P1.6;
   P1SKIP  = 0x01;                  // Skip P1.0;
   XBR0    = 0x07;                  // UART0 TX and RX pins enabled, SPI
                                    // enabled, SMBus enabled;
   XBR1    = 0x40;                  // Enable crossbar and weak pull-ups;

   P0MDIN &= ~0x80;                 // Configure VREF as analog input;
   P0MDOUT |= 0x1D;                 // Enable TX0,SCK,MOSI as a push-pull;
   P0 &= ~0x40;
   P1 &= ~0x01;
}
```

```
//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------
//
// Configure the UART0 using Timer1, for <BAUDRATE> and 8-N-1.
//
void UART0_Init (void)
{
   SCON0 = 0x10;                       // SCON0: 8-bit variable bit rate
                                       //        level of STOP bit is ignored
                                       //        RX enabled
                                       //        ninth bits are zeros
                                       //        clear RI0 and TI0 bits
   if (SYSCLK/BAUDRATE/2/256 < 1)
   {
      TH1 = -(SYSCLK/BAUDRATE/2);
      CKCON |=  0x08;                  // T1M = 1; SCA1:0 = xx
   }
   else if (SYSCLK/BAUDRATE/2/256 < 4)
   {
      TH1 = -(SYSCLK/BAUDRATE/2/4);
      CKCON &= ~0x0B;
      CKCON |=  0x01;                  // T1M = 0; SCA1:0 = 01
   }
   else if (SYSCLK/BAUDRATE/2/256 < 12)
   {
      TH1 = -(SYSCLK/BAUDRATE/2/12);
      CKCON &= ~0x0B;                  // T1M = 0; SCA1:0 = 00
   }
   else
   {
      TH1 = -(SYSCLK/BAUDRATE/2/48);
      CKCON &= ~0x0B;
      CKCON |=  0x02;                  // T1M = 0; SCA1:0 = 10
   }

   TL1 = TH1;                          // init Timer1
   TMOD &= ~0xf0;                      // TMOD: timer 1 in 8-bit autoreload
   TMOD |=  0x20;
   TR1 = 1;                            // START Timer1
   TI0 = 1;                            // Indicate TX0 ready
}

//-----------------------------------------------------------------------------
// SPI0_Init
//-----------------------------------------------------------------------------
//
// Configure SPI0 for 8-bit, 2MHz SCK, Master mode, polled operation, data
// sampled on 1st SCK rising edge.
//
void SPI_Init (void)
{
   SPI0CFG = 0x70;                     // Data sampled on rising edge, clk
                                       // active low,
                                       // 8-bit data words, master mode;

   SPI0CN = 0x0F;                      // 4-wire mode; SPI enabled; flags
                                       // cleared
```

```
   SPI0CKR = SYSCLK/2/2000000;         // SPI clock <= 2MHz
}

//-----------------------------------------------------------------------------
// SMBus_Init()
//
// SMBus configured as follows:
// - SMBus enabled
// - Slave mode disabled
// - Timer1 used as clock source. The maximum SCL frequency will be
//   approximately 1/3 the Timer1 overflow rate
// - Setup and hold time extensions enabled
// - Free and SCL low timeout detection enabled
//-----------------------------------------------------------------------------
void SMBus_Init (void)
{
   SMB0CF = 0x54;                      // Use Timer0 overflows as SMBus clock
                                       // source;
                                       // Disable slave mode;
                                       // Enable setup & hold time extensions;
                                       // Enable SMBus Free timeout detect;

   SMB0CF |= 0x80;                     // Enable SMBus;
   EIE1 |= 0x01;                       // Enable SMBus interrupts;
}



//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------
//
// Configure ADC0 to use Timer2 overflows as conversion source, to
// generate an interrupt on conversion complete, and to sense the output of
// the temp sensor with a gain of 2 (we want the white noise).  Enables ADC
// end of conversion interrupt.  Leaves ADC disabled.
//
void ADC0_Init (void)
{
   ADC0CN = 0x02;                      // ADC0 disabled; normal tracking
                                       // mode; ADC0 conversions are initiated
                                       // on overflow of Timer2;
   AMX0P = 0x1E;                       // Select temp sensor as positive input;
   AMX0N = 0x1F;                       // Select GND as negative input;
   ADC0CF = (SYSCLK/2000000) << 3;     // ADC conversion clock <= 6.0MHz
   REF0CN = 0x07;                      // Enable temp sensor, bias generator,
                                       // and internal VREF;
   EIE1 |= 0x08;                       // Enable ADC0 EOC interrupt;
   EIP1 |= 0x08;                       // ADC EOC interrupt is high priority;
}


//-----------------------------------------------------------------------------
// Timer0_Init()
//
// Timer0 configured as the SMBus clock source as follows:
// - Timer0 in 8-bit auto-reload mode
// - SYSCLK / 12 as Timer0 clock source
// - Timer0 overflow rate => 3 * SMB_FREQUENCY
```

```
// - The maximum SCL clock rate will be ~1/3 the Timer0 overflow rate
// - Timer0 enabled
//-------------------------------------------------------------------------------
void Timer0_Init (void)
{
   CKCON &= ~0x07;                     // Timer0 clock source = SYSCLK / 12;
   TMOD |= 0x02;                       // Timer0 in 8-bit auto-reload mode;
   TH0 = -(SYSCLK/SMB_FREQUENCY/12/3); // Timer0 configured to overflow at 1/3
                                       // the rate defined by SMB_FREQUENCY;
   TL0 = TH0;                          // Init Timer0;
   TR0 = 1;                            // Timer0 enabled;
}


//-------------------------------------------------------------------------------
// Timer2_Init
//-------------------------------------------------------------------------------
//
// This routine initializes Timer2 to use SYSCLK as its timebase and to
// generate an overflow at <SAMPLE_RATE> Hz.
//
void Timer2_Init (int counts)
{
   TMR2CN = 0x01;                      // Clear TF2H, TF2L; disable TF2L
                                       // interrupts; T2 in 16-bit mode;
                                       // Timer2 stopped;
   CKCON |= 0x30;                      // Timer 2 uses SYSCLK as clock
                                       // source
   TMR2RL = -counts;                   // reload once per second
   TMR2 = TMR2RL;                      // init Timer2
   ET2 = 0;                            // Disable Timer2 interrupts
   TR2 = 1;                            // Start Timer2
}
```

NOTES:

CONTACT INFORMATION

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Email: productinfo@silabs.com
Internet: www.silabs.com

SILICON LABORATORIES