M37515: INTERFACING WITH EEPROM USING I²C

Abstract

The following article will briefly discuss the synchronous serial I²C protocol and the M37515 I²C/SMBus hardware implementation. A discussion on the firmware structure and implementation on the I²C interface to the serial EEprom and M37515 MCU.

Author: Howard Chan

1.0 Introduction

I²C is a simple serial bus for communication among multiple hardware peripherals and devices in an embedded system. The Mitsubishi M37515 is a 8-bit MCU with an I²C/SMBus (System Management Bus) interface. The M37515 is a popular device used in keyboard controller and Smart battery applications. The SMBus uses a subset of the I²C protocol which is normally applied to power management applications like Smart Battery. The SMBus protocol will not be discussed in this article.

1.1 I²C: Description

I²C is a synchronous serial bus developed by Philips to simplify embedded device communication design between different peripherals. Many devices like EEproms, ADCs, LCD drivers, FPGAs, etc support the I²C bus protocol. These devices can communicate through a 2-wire bus, with data transfer rates of 100Kbits/s to 400Kbits/s. The number of devices on the bus is limited by the maximum bus capacitance of 400pF.

Most devices are used as slave devices while MCUs are typically master devices. I²C supports multi-mastering, which means more than one device is allowed to control the bus. I²C has collision detection and arbitration to maintain data integrity. I²C uses two lines, Serial Data Address Line (SDA) and Serial Clock Line (SCL) . These lines are bi-directional and are pulled-up high.

1.2 I²C: Protocol

I²C is a multi-master/slave protocol. All devices connected on the bus must have an open-collector or open-drain output. A transaction begins when the bus is free (i.e. both SCL and SDA is high), a master may initiate a transfer by generating a START condition. Then the master sends an address byte that contains the slave address and transfer direction. The addressed slave device must then acknowledge the master. If the transfer direction is from master to slave, the master would becomes the transmitter and writes to the bus. While the slave becomes the receiver and reads the data and acknowledges the transmitter, and vice versa. When the transfer is complete, the master sends a stop condition and the bus becomes free. In both transfer directions, the master generates the clock SCL and the START/STOP conditions.

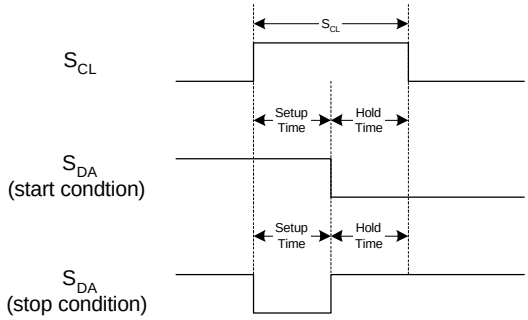

Figure 1. Start/Stop conditions

The start condition is generated by a High to Low transition in the SDA line during the High period of the SCL line as shown in *figure 1*. A stop condition is generated by a Low to High transition in the SDA during the High period of the SCL line also shown in *figure 1*.

The number of bytes transferred per START/STOP frame is unrestricted. Data bytes must be 8-bits long with the most significant bit (MSB) first. Each valid data bit sent to the SDA line must remain high for '1' or low for '0' during the high period of the SCL, otherwise any transition in the SDA line while SCL is high will be read as a START/STOP condition as shown in *figure 2*. Thus, transitions can only be made during the low period of SCL.

An acknowledge bit must follow each byte. After the last bit of the byte is sent, an ACK clock (acknowledgement clock) is generated by the master (9th clock). An ACK (acknowledge bit, low) must be sent by the receiver and remain a stable low during the high period of the ACK clock as shown in *figure 2*.

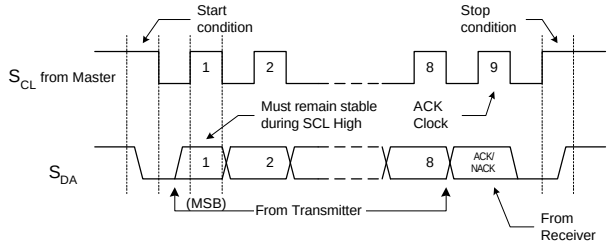

Figure 2. Data format

If the slave-receiver doesn't return an ACK (e.g. an error, or is unable to receive the data), then the slave-receiver device must leave the SDA line high (NACK). The master will abort the transfer by generating a stop condition. If the slave-receiver does return an ACK, but sometime later it is unable to receive any more data. Then the slave must generate a NACK (not acknowledge, high) on the first byte to follow (*see figure*

2). The slave will then need to keep the SDA line high for the master to generate a stop condition.

If the receiver is the master and transfer is coming to an end. Then the master needs to send a NACK after the last byte is sent. The slave-transmitter must release the SDA line to high to allow the master to generate a START/STOP condition.

At the beginning of each transfer, the master generates the start condition and then it sends a slave address. The standard slave address is 7-bits followed by a direction bit (8^{th} bit, (R/#W)) as shown in *figure 3*. When the direction bit is a WRITE (low or zero), the addressed slave device becomes the receiver and the master becomes the transmitter. When the direction bit is a READ (high or one), the addressed slave device becomes the transmitter and the master becomes the receiver.

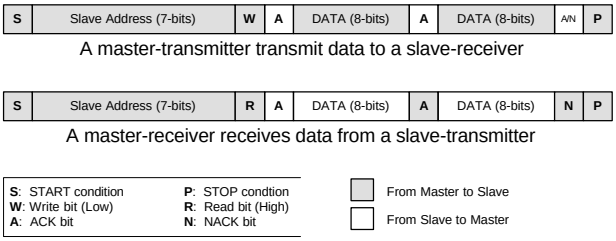

Figure 3. I²C Address/data communication format (S_{DA})

2.0 Serial EEPROM Interface

The Serial EEPROM that will be used in this discussion is the Microchip 24C01B. The EEPROM is an 8-pin device that uses the I^2C serial interface. The slave address assigned to this device by the manufacturer is 1010XXX, where X = Don't Care. The serial EEPROM supports several transfer modes such as: Byte Write, Page Write, Current Address Read, Random Read, and Sequential Read.

To perform a Byte Write, the master will generate a start condition and send the slave address with the direction bit set to HIGH as in *Figure 4*. When the slave device matches the address, it will send an ACK to the master during the 9^{th} clock cycle. The next byte sent to the EEPROM will be the word address that moves the EEPROM's internal address pointer. Then the data sent by the master will be written to the memory location pointed by the EEPROM's address pointer. Finally the master will generate a stop condition which will signal the EEPROM to initiate the internal write cycle. At this time the EEPROM will not generate any acknowledge signals till the transaction is complete.

A Page Write is similar to a Byte Write, except the master can transmit up to eight bytes before generating a stop condition. Each byte sent to the device will increment the address pointer for the next byte transaction. The EEPROM stores the data to a eight byte buffer, which is then written to memory after the device has received a stop condtion from the Master (*see figure 4*).

Read operations are initiated the same way as a write operation except the direction bit is set to READ. The EEPROM maintains the address pointer from the last byte accessed incremented by one. In a Current Address Read transaction, the EEPROM acknowledges the master after receiving the slave address and transmits the data byte pointed by it's internal address pointer (*figure 5*). The pointer is incremented by one for the next transaction. Sequential Read behave the same way as a Current Address Read transaction except data is continually transmitted by the slave device till the master generates a stop condition (*figure 5*). For Random Read, the master generates the start condition and sends the slave address with the direction bit set to WRITE (*figure 5*). Then the next byte sent is the word address to be accessed. This operations will change the EEPROM internal address pointer. Then without generating a stop condition, a Current Address Read or Sequential Read transaction will follow. Notice that the Current Address Read and Sequential Read transaction will regenerate another Start condition as shown in *figure 5*.

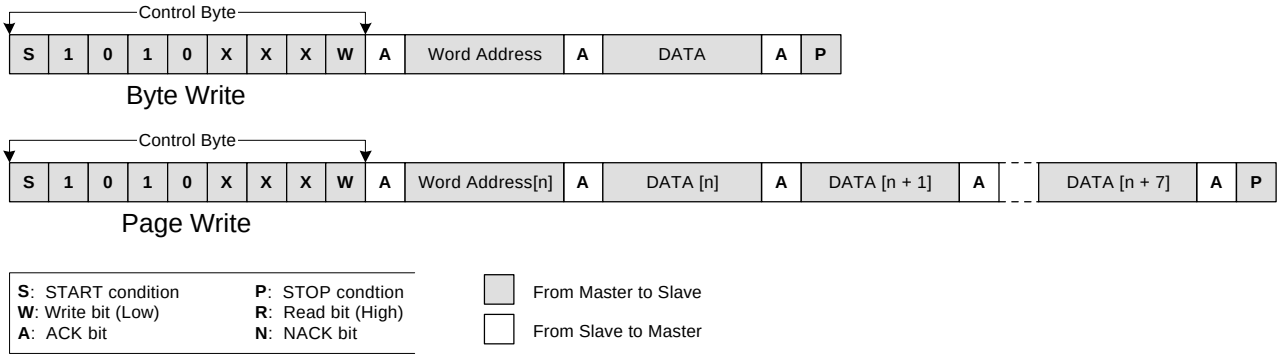

Figure 4. Write transfer mode

Figure 5. Read Transfer Mode

3.1 Implementation: Hardware

The M37515 has a built-in multi-master I²C/SMBus peripheral. This interface conforms to the Philips I²C-BUS data transfer format and SMBus compliant. This multi-master I²C interface consists of the address register, the data shift register, status register and other control circuits. The interface supports 10-bit and 7-bit addressing format, High-speed (400KHz) and Standard (100KHz) clock mode, Master transmission and reception, and Slave Transmission and reception.

To interface the M37515 to the 24C01B serial EEPROM simply requires connecting the SDA & SCL lines together. The rest is done in firmware (see section 3.2). Note that the transfer rate has been set as 100KHz, so a 10K pull-up is connected to the SDA line Additional peripherals can be connected to the I²C bus, as long as the total capacitance doesn't exceed 400pF.

3.2 Implementation: Firmware

The firmware takes advantage of the M37515 I²C/SMBus interface. The sample firmware will exercise the I²C interface by interfacing with an external serial EEPROM. The firmware will write a variable size block of data from ROM to external EEPROM, then read the EEPROM to ram. The main routine calls the routines WriteEEprom, SeekEEprom and ReadEEprom. The routines format the data that will be sent the EEPROM and then calls the function MasterIIC which sets up the I²C hardware and starts the I²C peripheral. An interrupt service routine handles the transmit complete/received data interrupt.

The main routine initializes the MCU, Timers, and the I²C peripherals. Then it breaks the data to be written to the EEPROM in 8 byte blocks, due to the limitations of the EEPROM. Then it calls WriteEEprom and waits for the function to complete. Then SeekEEprom is called to move the EEPROM word address pointer to the address to be read to ram. ReadEEprom is called to read the EEPROM to memory.

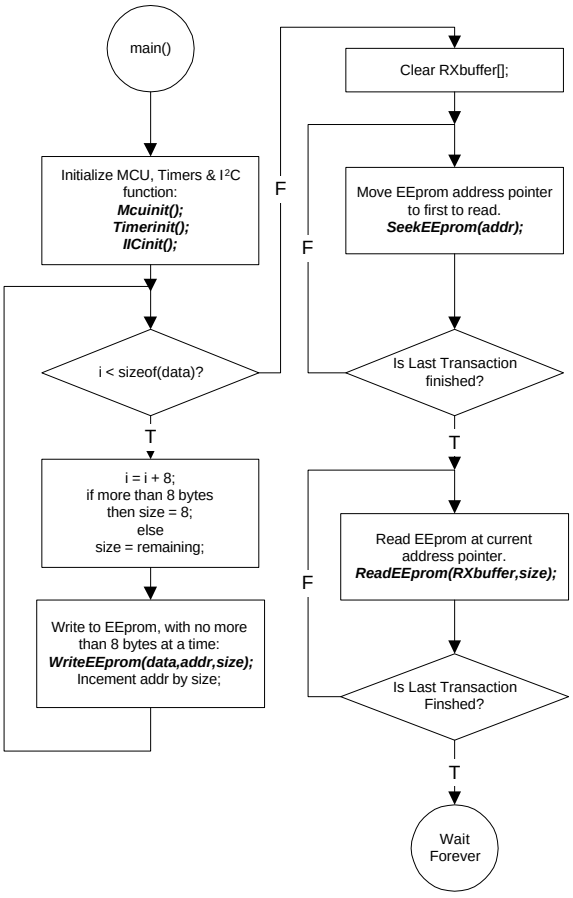

Figure 6. Software Flow Diagram of Main( )

The details of the firmware is outlined in the Software Flow Diagrams shown on *Figures 6, 7, & 8* and commented on the source code listing on section 5.0.

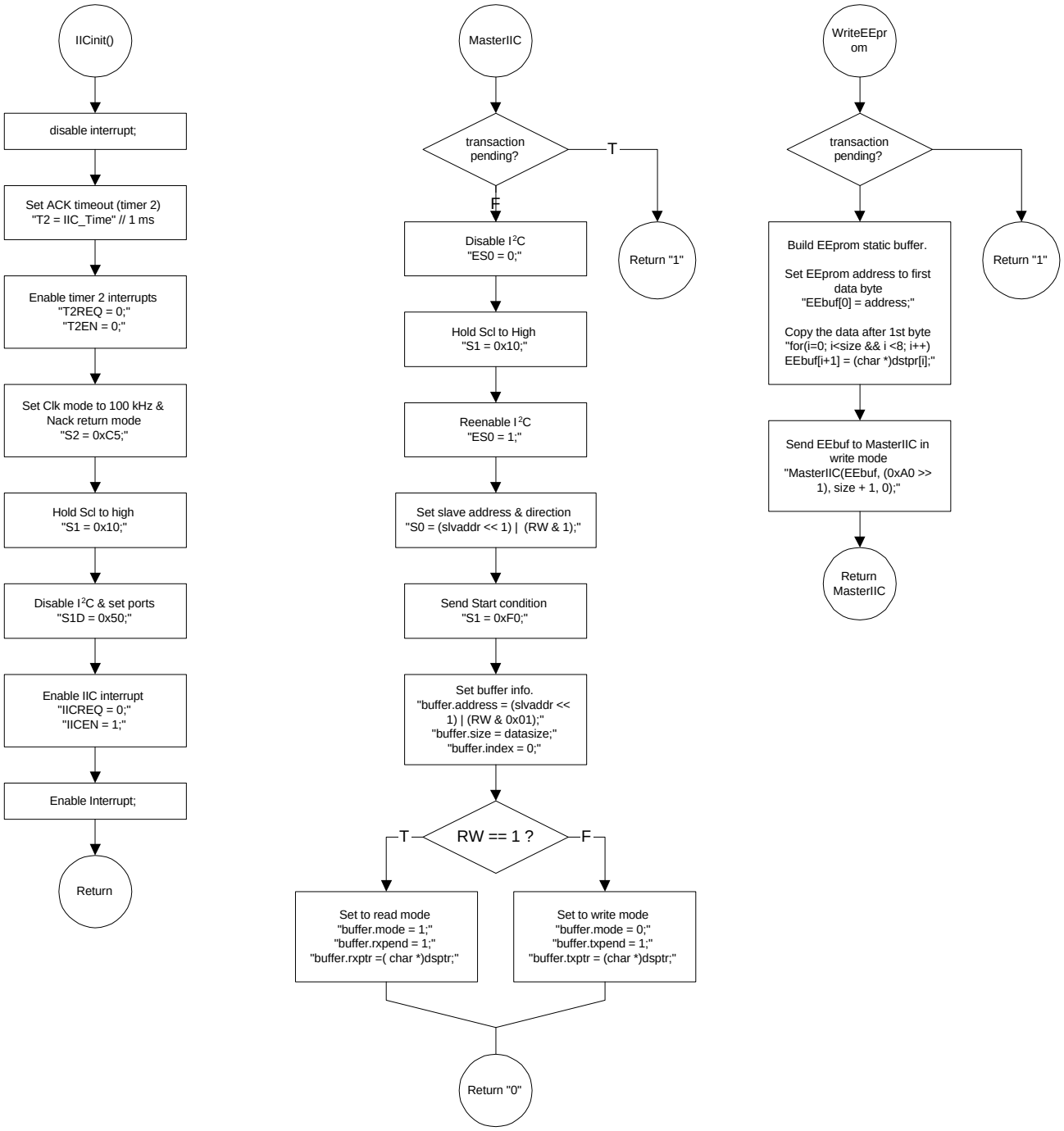

Figure 7. Software Flow Diagram of I²C Functions

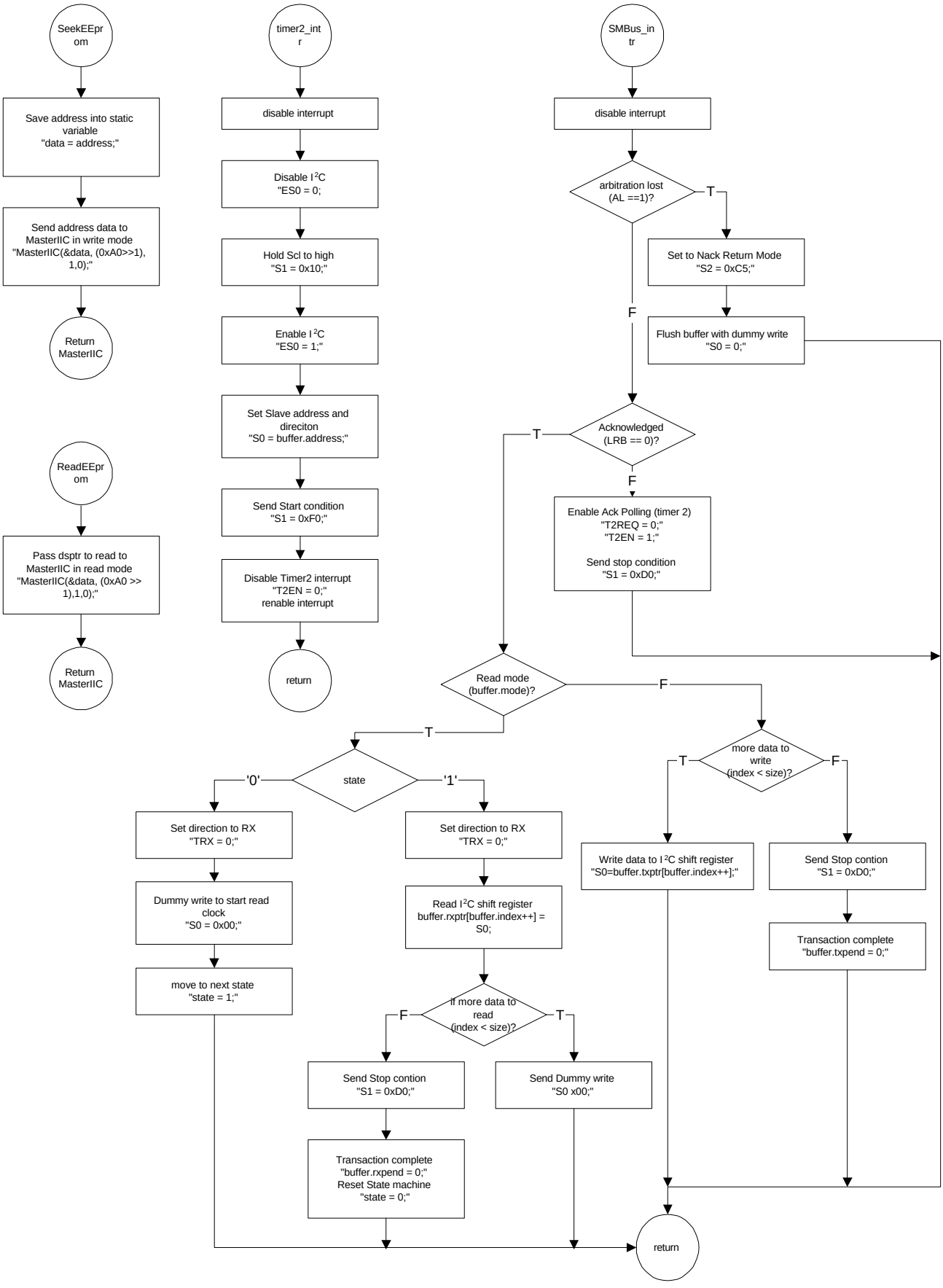

Figure 8. Software Flow Diagram of I²C routines (cont)

5.0 LISTING FOR M37515 IIC FIRMWARE

----------LISTING FOR M37515.C----------

```
// M37515 Demo Board - Mitsubishi Electronics America
// Board Designed by Mark Gould
// Firmware Programmed by Howard Chan
// Version 0.1   3/12/98
// Version 0.5   3/13/98
// Version 0.9   3/14/98
// Version 1.0   3/24/98
// ------------------------------------------------------------
// Mini App Note: The following code is designed for the M37515 board.
// The sample firmware consists of the following files:
//   m37515.h - Declaration of SFR registers.
//   m37515.c - Main Source code. Also sample of Multi EEprom page write and
//       read.
//   iic.h - header file to include to use IIC & EEprom function
//   iic.c - IIC routines for Master Read, Write, & Ack Polling.
//   func.h - header file to include to use AD-functions and timer init.
//
// The code below, simply intializes the MCU, the Timer (AD-sampling) and the IIC system
// Then a loop is used to write a block of more than 8 words to the function
// WriteEEprom which can handle a maximum of 8 bytes at a time (limitation of EEprom
// refer to Microchip 24C01B Serial EEprom Data Sheet.) Then the Target buffer
// is cleared of old data.  The Seek function is called to move the EEprom
// address pointer to the first byte to be read.  Where ReadEEprom is used to
// Read the consecutive block(Read have no limitation).  Finally the main()
// waits forever, as a timer 1 interrupt occurs to sample the 10-bit AD data.
//
// NOTE: The while loop (EX. while(SeekEEprom(0x20));) is used, because the EEprom
// functions return a '1' if the IIC is busy with a current read or write.  Thus
// it waits till it can perform the transaction before continuing.
//
#pragma language=extended

#include "iic.h"
#include "func.h"

//---------Globals----------------------
const char data[] = {'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E'};

char RXbuffer[sizeof(data)];  // This buffer will accommidate the data above

//----------Main function -----------------

void main(void) {
    char i;         // just a temp storage variable
    char size;

    Mcuinit();      // Initialize MCU registers & ports
    Timerinit();    // Intialize timer1 for AD-sampling
    IICinit();      // Initialize IIC and timer2 (IIC)

    for(i = 0; i < sizeof(data); i += 8) {
        if(i < sizeof(data) - 8)        // If more than 8 bytes then
            size = 8;                   //   Write 8 bytes at a time
        else                            // else
            size = sizeof(data) - i;    //   Write remaining bytes
        while(WriteEEprom(&data[i],0x20 + i,size));  // write data to EEprom
    }
    for(i = 0; i < sizeof(data); i++)   // Clear RXbuffer
        RXbuffer[i] = 0;

    while(SeekEEprom(0x20));            // Move EEprom address pointer to 1st byte to read
    while(ReadEEprom(RXbuffer,sizeof(data))); // Read EEprom data

    while(1);       // Wait here
}
```

----------LISTING FOR IIC.H----------

```
#ifndef IIC
```

```
#define IIC 0

// Function Declarations for EEprom
char WriteEEprom(void *dsptr, unsigned char address, unsigned char size);
char SeekEEprom(unsigned char address);
char ReadEEprom(void *dsptr, unsigned char size);
void Delay(int time);

// Function Declarations for IIC
void IICinit(void);
char MasterIIC(void *dsptr, char slvaddr, char datasize, char RW);
interrupt [0xF0 - 0xDC] void SMBus_intr(void);
interrupt [0xE8 - 0xDC] void timer2_intr(void);

#endif
```

----------LISTING FOR IIC.C----------

```
// M37515 Demo Board - Mitsubishi Electronics America
// Firmware Programmed by Howard Chan
// Version 0.1   3/12/98
// Version 0.5   3/13/98
// Version 0.9    3/14/98
// Version 1.0   3/24/98
// ------------------------------------------------------------
// Mini App Note: The following routines are used for interfacing with
// the M37515 IIC hardware.  The main engine is in the MasterIIC() function
// and the SMBus_intr() function.  The others are just application functions for
// EEprom.  NOTE: Slave mode hasn't been implemented yet.
//
//

#include "m37515.h"
#include <intr740.h>

//-------Definitions-------------
#define WRITE 0         // Write
#define READ 1          // Read
#define NULL 0          // Null
#define IIC_TIME 0x01   // IIC polling time = f(Xin)

//-------Buffer Global-----------

struct {
    char *rxptr;      // Rx Data pointer  NOTE: should have seperate pointers
    char *txptr;      // Tx Data pointer            to prevent accidental overwrite.
    char index;       // index of data
    char size;        // size of data
    char address;     // Slave Address
    char mode:1;      // '1' - tx mode, '0' - rx mode
    char txpend:1;    // Are more TX transactions pending?
    char rxpend:1;    // Are more RX transactions pending?
} buffer;           // IIC bufferbuffer;

char test5;         // Debug Only

struct {
    char clkmode;
} IICparameters;

//--------IIC Functions-----------
//  FUNC: void IICinit(void)
//  DESC: Initialize IIC system to single master bus.
void IICinit(void) {
    disable_interrupt();
    T2 = IIC_TIME;      // Set IIC Time
    T2REQ = 0;          // Reset Timer 2 Interrupt Request
    T2EN = 0;           // Disable Timer 2 Interrupt (Called when Polling required)

    S2 = 0xC5;          //  11000101b - Set Clock mode & NACK return mode
                        //  ||||||||
                        //  |||+++++--  100 @ Standard Clock Mode
                        //  ||+-------  FAST: 0 - Standard Clock Mode
                        //  |+--------  ACK bit: 1 - ACK non-return mode
                        //  +---------  ACK: 1 - No ACK clock sent
    S1 = 0x10;          //  00010000b - Hold Scl to high
                        //  ||||||||
```

```
                        //  ||||++++--  Don't Care
                        //  |||+------  PIN: 1 - Clr IRQ for next INT
                        //  ||+-------  BB: 0 - Send Stop Condition
                        //  |+--------  TRX: 0 - Recieve Mode
                        //  +---------  MST: 0 - Slave Mode (Don't generate start or stop)
    S1D = 0x50;         //  01010000b - Disable communications and set ports
                        //  ||||||||
                        //  |||||+++--  Bit counter - Don't Use
                        //  ||||+-----  ES0: 0 - Disable S0 IIC
                        //  |||+------  ALS: 1 - Free form (EEprom)
                        //  ||+-------  SAD: 0 - 7-bit address
                        //  |+--------  TSEL: 1 - P24 & P25
                        //  +---------  TISS: 0 - CMOS input
    // ICON & IREQ Setting
    IICREQ = 0;         //  Clear interrupt request
    IICEN = 1;          // '1'- Enable, '0' Disable IIC interrupt

    enable_interrupt();
}

//  FUNC: void Delay(int time)
//  DESC: Add delay
//  time: Lenth of delay: (Actual time hasn't been calculated.
void Delay(int time) {
    int i;
    for(i = 0; i<time; i++);
}

//  FUNC: char MasterIIC(void *dsptr, char slvaddr, char datasize, char RW)
//  DESC: Access IIC bus as Master transaction
//  return: "0" - successful transaction, "1" - Read or Write Pending, Transaction denied
//  dsptr: pointer to a data structure to write to IIC.
//  slvaddr: slave address of device on bus.
//  size: total size of data transcation
//  RW: "0" - Write mode, "1" - Read mode;
char MasterIIC(void *dsptr, char slvaddr, char datasize, char RW) {
    if (buffer.rxpend || buffer.txpend)
        return 1;   // '1' - Read or Write is still pending

    ES0 = 0;        // Disable IIC
    ACKbit = !RW;   // '1' =  ACK non-return mode, '0' = ACK return mode
    S1 = 0x10;      //  00010000b - Hold Scl to high
                    //  ||||||||
                    //  ||||++++--  Don't Care
                    //  |||+------  PIN: 1 - Clr IRQ for next INT
                    //  ||+-------  BB: 0 - Send Stop Condition
                    //  |+--------  TRX: 0 - Recieve Mode
                    //  +---------  MST: 0 - Slave Mode (Don't generate start or stop)
    ES0 = 1;        // ES0: 1 - Enable IIC

    S0 = (slvaddr << 1) | (RW & 0x01);   // Set Slave address and transfer direction
                    //  SSSSSSSR
                    //  ||||||||
                    //  |||||||+--  RW#: 0 - Write mode, '1' - Read mode
                    //  +++++++---  Slave Address
    S1 = 0xF0;      //  11110000b - Send Start Condition
                    //  ||||||||
                    //  ||||++++--  Don't Care
                    //  |||+------  PIN: 1 - Clr IRQ for next INT
                    //  ||+-------  BB: 1 - Send Start Condition
                    //  |+--------  TRX: 1 - Transmit Mode
                    //  +---------  MST: 1 - Master Mode (generate start or stop)

    buffer.address = (slvaddr << 1) | (RW & 0x01);  // Set Slave address and direction
    buffer.size = datasize;     // Set total size of ds
    buffer.index = 0x00;        // Set index to point to first data
    if(RW) {                    // if RW = '1' then read
        buffer.mode = 1;        // '1' Read mode
        buffer.rxpend = 1;      // '1' Read pending
        buffer.rxptr = (char *)dsptr;   // Set Rx pointer to dsptr
    } else {
        buffer.mode = 0;        // '0' Write mode
        buffer.txpend = 1;      // '1' Write pending
        buffer.txptr = (char *)dsptr;   // Set Tx pointer to dsptr
    }
    return 0;                   // '0' - Transaction complete
}
```

```
//  FUNC: char WriteEEprom(void *dsptr, unsigned char address, unsigned char size)
//  DESC: Write data to IIC bus via Buffer.
//  NOTE: The Microchip 24C01B Serial EEprom has only a 8-byte write page.
//      So the maximum write size per block is 8.  For data that is larger than
//      8 bytes, Make subsequent calls to this function with the address incremented.
//  return: "0" - successful transaction, "1" - Read or Write Pending, Transaction denied
//  dsptr: pointer to a data structure to write to IIC.
//  address: address of device or memory to write.
//  size: total size of data. (Size limit to 8-bytes at a time)
char WriteEEprom(void *dsptr, unsigned char address, unsigned char size) {
    static char EEbuf[9];       // 8 data bytes + 1 EEprom address byte
    char i;

    if (buffer.rxpend || buffer.txpend)
        return 1;               // '1' - Read or Write is still pending
    EEbuf[0] = address;         // Set EEprom address

    for(i = 0; (i < size) && (i < 8); i++)  // Copy data to write buffer
        EEbuf[i + 1] = ((char *)dsptr)[i];

    return MasterIIC(EEbuf, (0xA0 >> 1), size + 1, WRITE);
}

//  FUNC: char SeekEEprom(unsigned char address)
//  DESC: Move EEprom pointer to correct address
//  return: "0" - successful transaction, "1" - Read or Write Pending, Transaction denied
//  address: address of device or memory to read.
char SeekEEprom(unsigned char address) {
    static unsigned char data;
    data = address;     // Ensure that data has an address during interrupts
    return MasterIIC(&data, (0xA0 >> 1),1,WRITE);
}

//  FUNC: char ReadEEprom(void *dsptr, unsigned char size)
//  DESC: Read data to IIC bus via Buffer
//  return: "0" - successful transaction, "1" - Read or Write Pending, Transaction denied
//  dsptr: pointer to a data structure to read to IIC.
//  address: address of device or memory to read.
//  size: total size of data. (No size limit)
char ReadEEprom(void *dsptr, unsigned char size) {
    return MasterIIC(dsptr, (0xA0 >> 1), size, READ);
}

//  FUNC: interrupt [0xE8 - 0xDC] void timer2_intr(void)
//  DESC: This interrupt is used for ACK polling.
interrupt [0xE8 - 0xDC] void timer2_intr(void) {
    disable_interrupt();   // Initiate Acknowledge Polling
    ES0 = 0;    // ES0: 0 - Enable IIC
    S1 = 0x10;  //  00010000b - Hold Scl to high
                //  ||||||||
                //  ||||++++--  Don't Care
                //  |||+------  PIN: 1 - Clr IRQ for next INT
                //  ||+-------  BB: 0 - Send Stop Condition
                //  |+--------  TRX: 0 - Recieve Mode
                //  +---------  MST: 0 - Slave Mode (Don't generate start or stop)
    //  Delay(1);
    ES0 = 1;    // ES0: 1 - Enable IIC

    S0 = buffer.address;// Set Slave address and transfer direction
                        //  SSSSSSSR
                        //  ||||||||
                        //  |||||||+--  RW#: 0 - Write mode, '1' - Read mode
                        //  +++++++---  Slave Address
    S1 = 0xF0;  //  11110000b - Send Start Condition
                //  ||||||||
                //  ||||++++--  Don't Care
                //  |||+------  PIN: 1 - Clr IRQ for next INT
                //  ||+-------  BB: 1 - Send Start Condition
                //  |+--------  TRX: 1 - Transmit Mode
                //  +---------  MST: 1 - Master Mode (generate start or stop)
    T2EN = 0;   // Disable Timer 2 Interrupt
    enable_interrupt();
}

//  FUNC: interrupt [0xF0 - 0xDC] void SMBus_intr(void)
//  DESC: Interrupt call services the IIC bus as a Master.
interrupt [0xF0 - 0xDC] void SMBus_intr(void) {
    static char state;  // Internal State counter
```

```
    disable_interrupt();
    if(AL == 1)  {  // If bus arbitration is lost
        S2 = 0xC5;  //  11000101b - return NACK return mode
                    //  ||||||||
                    //  |||+++++--  100 @ Standard Clock Mode
                    //  ||+-------  FAST: 0 - Standard Clock Mode
                    //  |+--------  ACK bit: 1 - ACK non-return mode
                    //  +---------  ACK: 1 - ACK clock sent
        S0 = 0x00;  // Flush buffer with dummy data(like a read)
    }

    if(LRB) {       // If no ACK then Poll again.
        test5++;    // No Ack flag
        T2REQ = 0;  // Reset Timer 2 Interrupt Request
        T2EN = 1;   // Enable Ack Polling (Timer 2)
        S1 = 0xD0;  //  11010000b - Send Stop condition
    } else
        if(buffer.mode) {
            switch(state) {
                case 0:
                    TRX = 0;    // Set direction to Rx
                    S0 = 0x00;  // Dummy write to start next read
                    state = 1;  // move to next state
                    break;
                case 1:
                    TRX = 0;    // Set direction to Rx
                    buffer.rxptr[buffer.index++] = S0; // Read IIC shift reg.
                    if (buffer.index < buffer.size) {  // If more data then
                        S0 = 0x00;  // Dummy write to start next read
                        state = 1;  // Move to next state
                    } else {
                        S1 = 0xD0;  //  11010000b - Send Stop condition
                                    //  ||||||||
                                    //  ||||++++--  Don't Care
                                    //  |||+------  PIN: 1 - Clr IRQ for next INT
                                    //  ||+-------  BB: 0 - Send Stop Condition
                                    //  |+--------  TRX: 1 - Transmit Mode
                                    //  +---------  MST: 1 - Master Mode
                        buffer.rxpend = 0;  // Transaction complete
                        state = 0;  // Reset State Machine
                    }
                    break;
                default:
                    state = 0;      // Reset State Machine
                    S1 = 0xD0;      // Send Stop condition
                    buffer.rxpend = 0;  // RX transaction is complete
            }
        } else {
            //--------------Write Mode---------------
            if(buffer.index < buffer.size) {        // If no more data to flush
                S0 = buffer.txptr[buffer.index++];  // Write data to IIC shift reg.
            } else {
                S1 = 0xD0;  //  11010000b - Send Stop condition
                            //  ||||||||
                            //  ||||++++--  Don't Care
                            //  |||+------  PIN: 1 - Clr IRQ for next INT
                            //  ||+-------  BB: 0 - Send Stop Condition
                            //  |+--------  TRX: 1 - Transmit Mode
                            //  +---------  MST: 1 - Master Mode
                buffer.txpend = 0;  // Transcation complete
                state = 0;  // Reset State Machine
            }
        }
    enable_interrupt();
}
```

----------LISTING FOR FUNC.H----------

```
#ifndef FUNC
#define FUNC 0

// Function Declarations
void Mcuinit(void);
void Timerinit(void);

void ShowLED(unsigned int value);
unsigned int Analog(unsigned char channel);
```

```
// Interrupts
interrupt [0xEA - 0xDC] void timer1_intr(void);

#endif
```

----------LISTING FOR FUNC.C----------

```
// M37515 Demo Board - Mitsubishi Electronics America
// Firmware Programmed by Howard Chan
// Version 0.1   3/12/98
// Version 0.5   3/13/98
// Version 0.9  3/14/98
// Version 1.0   3/24/98
// ------------------------------------------------------------
// Mini App Note: The following routines are used for interfacing with
// the M37515 10-bit A-D, timers, and LEDs.
//
// The 10-bit A-D routine is very simple.  It simply sets the channel bits
// to read and clears the start conversion bit.  Then the bit is polled for
// A-D complete status.  Then the AD-Hi must be read first for an 10-bit
// conversion, else read the AD-lo first and discard the AD-Hi for an 8-bit read
//
// For timers, just set the timer1 registers for the value to be loaded after
// every underflow, and set the interrupt vector for the appropriate ISR.
// Below is the formula used to calculate the time.
// f(Xin)/16 * (TIME) = PRE12 * <T1 or T2>;
//
// where f(Xin) = Input Xtal Frequency (Hz)
// TIME = Time of overflow in (secs)
// PRE12 = Prescaler value to load that will affect timer T1 and T2
// T1 or T2 = Timer value to be loaded on timer underflow.
//
// The LED ports (pins P13-P17) can sink up to 15 mA, so an external driver
// is not necessary.  Just tie the LED to the port, and write a '0' to the
// P1 register to light the LED.


#include "m37515.h"
#include <intr740.h>

//----------Definitions ----------------
#define AD10BIT 1           // 1: 10-bit A-D, 0: 8-bit A-D
#define ADSEL_MSK 0x07      // Mask for Analog inputs xxxxxAAA
#define ADCNV_MSK 0x10      // Mask for AD conversion xxxAxxxx
#define SAMPLE_TIME 0x08    // Sample time for AD input -  about 8 ms.
#define LEDRANGE 0x03ff     // Full scale range of LED output
#define RANGE80 0x333       // 0.8 * LEDRANGE
#define RANGE60 0x266      // 0.6 * LEDRANGE
#define RANGE40 0x199      // 0.4 * LEDRANGE
#define RANGE20 0xCC       // 0.2 * LEDRANGE
#define RANGE00 0x10       // 0.01 * LEDRANGE

unsigned int test;         // Global test variables
int analog1, analog2;      // Watch Variables, for Analog 1 & 2

//  FUNC: void ShowLED(unsigned int value)
//  DESC: Output values to LEDs as a percentage 20, 40, 60, 80, 100
//  value: value to output to LEDs
void ShowLED(unsigned int value) {
       // Output LEDs via trickle method
       // '0' - on, '1' - off
       LED100 = (value >= RANGE80) ? 0 : 1;    // 80 - 100 percent
       LED80 =  (value >= RANGE60) ? 0 : 1;    // 60 - 79 percent
       LED60 =  (value >= RANGE40) ? 0 : 1;    // 40 - 59 percent
       LED40 =  (value >= RANGE20) ? 0 : 1;    // 20 - 39 percent
       LED20 =  (value > RANGE00) ? 0 : 1;     // 00 - 19 percent
}

//  FUNC: unsigned int Analog(unsigned char channel)
//  DESC: Read Channel of A/D port  NOTE: Possible bit manipulation due to high bits.
//  return: Results of 10-bit A-D conversion
//  channel: select AD channel to read from 0 - 7
unsigned int Analog(unsigned char channel) {
    union {
```

```
        unsigned int word;
        unsigned char byte[2];
    } adword;
    ADCON = ADSEL_MSK & channel;    // Mask correct channel and start conversion
    //  ADCON &= 0xEF;              // Start Conversion (zero bit 4)
    while(!(ADCON & ADCNV_MSK));    // wait till conversion complete (bit 4)
#if AD10BIT
    adword.byte[1] = ADH;           // convert ADH & ADL to an integer (10-bit)
#endif
    adword.byte[0] = ADL;             // NOTE: ADL
    return adword.word;             // Return 10-bit A/D value
}

//      FUNC: void battery(void)
//      DESC: A function that simply samples AD channel 0 & 1 and outputs
//             the first 8-bit to P0 and last 2-bits + channel to P1;
char port;
void battery(void) {
    //        char port;
    char temp0,temp1;
    union {
        unsigned int word;
        unsigned char byte[2];
    } ad[2];

    ad[0].word = Analog(0);         // Evaluate AD #0
    ad[1].word = Analog(1);         // Evaluate AD #1

    // DEBUG VARIABLES
    analog1 = ad[0].word;
    analog2 = ad[1].word;

    port = (ad[1].word >= ad[0].word) ? 1:0;

    temp0 = ad[port].byte[0];           // Set low byte
    temp1 = (0xf8 & P1) | (0x03 & ad[port].byte[1]);   // Set high byte last 2 bits.
    temp1 |= (port) ? 0x04: 0;          // Set bit for port. 1 - port 1, 0 - port 0

    P0 = temp0;                         // Place data on the bus (low)
    P1 = temp1;                         //                       (high)
}

//  FUNC: interrupt [0xEA - 0xDC] void timer1_intr(void)
//  DESC: Interrupt call samples AD port #0 and displays data to LED
interrupt [0xEA - 0xDC] void timer1_intr(void) {
    static char samp[2];      // Static variable assignments.
    static char i;
    char button;
    disable_interrupt();

    // Debounce routine
    samp[i^=0x01] = P4.1;     // store sample & index to next sample.
    if(samp[0]==samp[1])
        button = samp[0];     // Update button status if same.

    // Battery interrupt
    if(button == 0)
        battery();

    // Sample Pot
    test = Analog(2);      // For Debug Use
    ShowLED(test);          // Display Analog results to LED
    enable_interrupt();
}

//-------------------Initialization functions ------------

//  FUNC: void Timerinit(void)
//  DESC: Initialize Timer 1 interrupt for A-D use.
void Timerinit(void) {
    disable_interrupt();
    PRE12 = 0xff;      // SET Pre-scaler for timer 1 & 2
    T1 = SAMPLE_TIME; // Set AD sample Time
    T1REQ = 0;         // Reset Timer 1 Interrupt Request
    T1EN = 1;          // Enable Timer 1 Interrupt
    enable_interrupt();
}
```

```
//  FUNC: void Mcuinit(void)
//  DESC: Initialize Mcu ports, cpu mode and AD-mode
void Mcuinit(void) {
    disable_interrupt();

    CPUM = 0x04;
    /* Set CPU mode register                        */
    /*    00000100B                                 */
    /*    ||||||||                                  */
    /*    ||||||++--------------- PROCESSOR MODE BIT            */
    /*    ||||||                    00 : SINGLE CHIP MODE       */
    /*    |||||+----------------- STACK PAGE IN PAGE 1           */
    /*    |||+------------------- PORT Xc : I/O PORT FUNC.       */
    /*    ||+-------------------- MAIN CLOCK Xin-Xout : EXECTE   */
    /*    ++--------------------- COUNTER SOURCE : F(Xin)/2      */

    P0D = 0xFF; /*    11111111B (1:OUTPUT, 0:INPUT)     */
                /*    ||||||||                          */
                /*    |||||||+---------------- DATA0     */
                /*    ||||||+----------------- DATA1     */
                /*    |||||+------------------ DATA2     */
                /*    ||||+------------------- DATA3     */
                /*    |||+-------------------- DATA4     */
                /*    ||+--------------------- DATA5     */
                /*    |+---------------------- DATA6      */
                /*    +----------------------- DATA7     */
  P0  = 0x00;

  P1D = 0xFF;   /*    11111111B (1:OUTPUT, 0:INPUT)     */
                /*    ||||||||                          */
                /*    |||||+++---------------- No Use    */
                /*    ||||+------------------- LED0      */
                /*    |||+-------------------- LED1      */
                /*    ||+--------------------- LED2      */
                /*    |+---------------------- LED3      */
                /*    +----------------------- LED4      */
  P1.0 = 0x00;

  P2D = 0xFF;   /*    11111111B (1:OUTPUT, 0:INPUT)             */
                /*    ||||||||                                  */
                /*    ||||||++---------------- No Use           */
                /*    |||||+------------------ SMBus Clock       */
                /*    ||||+------------------- SMBus Data        */
                /*    |||+-------------------- EEprom IIC Data   */
                /*    ||+--------------------- EEprom IIC Clk    */
                /*    |+---------------------- No Use           */
                /*    +----------------------- No Use           */

  P3D = 0x00;   /*    00000000B (1:OUTPUT, 0:INPUT)             */
                /*    ||||||||                                  */
                /*    |||||||+---------------- AN0(Voltage +)    */
                /*    ||||||+----------------- AN1(Voltage -)    */
                /*    |||||+------------------ AN2(POT)          */
                /*    +++++------------------- No Use           */

  ADCON  = 0x00;
                /* Set A-D ctrl reg.                          */
                /*    00000000B,ADCON                         */
                /*    ||||||||                                */
                /*    |||||+++---------------- ANALOG INPUT PIN SELECT   */
                /*    |||||                                   */
                /*    |||||                                   */
                /*    ||||+------------------  NO USE         */
                /*    |||+-------------------  A/D CONVERSION COMPLETION BIT    */
                /*    +++--------------------  NO USE         */

  enable_interrupt();
}
```