Lab 5 Report

PART 1

Report the BRG4 value that you are using and the frequency of the SPI4 clock that you are currently using at this point.

BRG4 = 8

SPI4CLK = 214ns

Verify the SPI clock rate. Report the SYSCLK rate, the PB2CLK rate and the BRG value you used along with the actual SPI clock rate measured using the logic analyzer. *Include these results in your report.*

SYSCLK = 84MHz

PB2CLK = 84MHz

BRG = 8

SPI = 210ns

Give a copy of the segment of C code and its disassembly that demonstrates the **worst case** (fewest instructions between RBF high and CS high) (including the disassembly of called subroutines if appropriate) *in your report*. Based on this, determine and *report* what your worst case CS hold should be. Show how you arrived at your answer. Is the 25LC256 parameter 3 requirement met? Did you have to adjust the code or add nop's to make sure it was met? *Include the code samples and these question answers in your report.*

Lab 5 Report

```
!    while(SPI4STATbits.SPIRBF == 0);        //wait for receive buffer full
0x9D000608: NOP
0x9D00060C: LUI V0, -16510
0x9D000610: LW V0, 5648(V0)
0x9D000614: EXT V0, V0, 0, 1
0x9D000618: ANDI V0, V0, 255
0x9D00061C: BEQ V0, ZERO, 0x9D00060C
0x9D000620: NOP
!
!    SPI4BUF;
0x9D000624: LUI V0, -16510
0x9D000628: LW V0, 5664(V0)
!
!    LATFSET = _LATF_LATF8_MASK;             //negate the CS
0x9D00062C: LUI V0, -16506
0x9D000630: ADDIU V1, ZERO, 256
0x9D000634: SW V1, 1336(V0)
!}
0x9D000638: NOP
```

```
    while(SPI4STATbits.SPIRBF == 0);        //wait for receive buffer full

    SPI4BUF;

    LATFSET = _LATF_LATF8_MASK;             //negate the CS
}
```

The worst case scenario is 8 machine instructions. I found the code that had the fewest instructions and then counted how many to make sure there was enough. Since I already met the requirements of the hold I did not need to add any nops.

Now, look for instances in your Part 1 code where CS is negated then quickly (within 4 C instructions) asserted again. Do at least 5 SYSCLKS elapse between CS negation and CS assertion? Obtain a copy of the C code segment and its disassembly representing the worst case for your report. Based on this, determine and *report* what your worst case CS hold should be. Show how you arrived at your answer. Is the 25LC256 parameter 4 requirement met? Did you have to adjust the code or add nop's to make sure it was met? *Include the code samples and these question answers in your report.*

```
    LATFSET = _LATF_LATF8_MASK;             //negate the CS
/*****************************************************************
    LATFCLR = _LATF_LATF8_MASK;             //assert the CS
```

Lab 5 Report

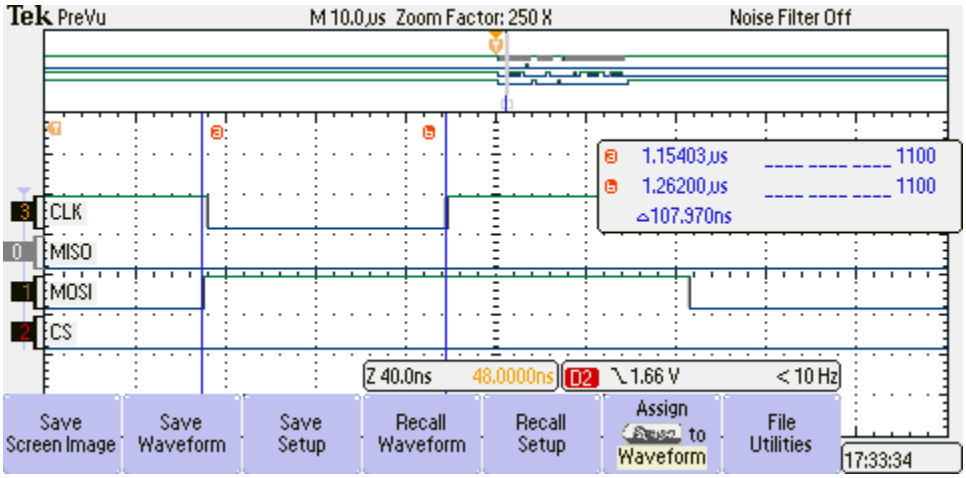
```
    :
!     LATFSET = _LATF_LATF8_MASK;              //negate the CS
0x9D000530: LUI V0, -16506
0x9D000534: ADDIU V1, ZERO, 256
0x9D000538: SW V1, 1336(V0)
!/*****************************************************************************/
!     LATFCLR = _LATF_LATF8_MASK;              //assert the CS
0x9D00053C: LUI V0, -16506
0x9D000540: ADDIU V1, ZERO, 256
0x9D000544: SW V1, 1332(V0)
!
```

The worst case scenario is show above where the write enable ends and immediately goes into the write command. It was the closest that the two commands come to each other. Since there is only 3 lines between them the hold requirement is not met. I needed to add 2 nops to the code to get the hold time.

MOSI Setup

Required = 20ns

Recorded = 108ns

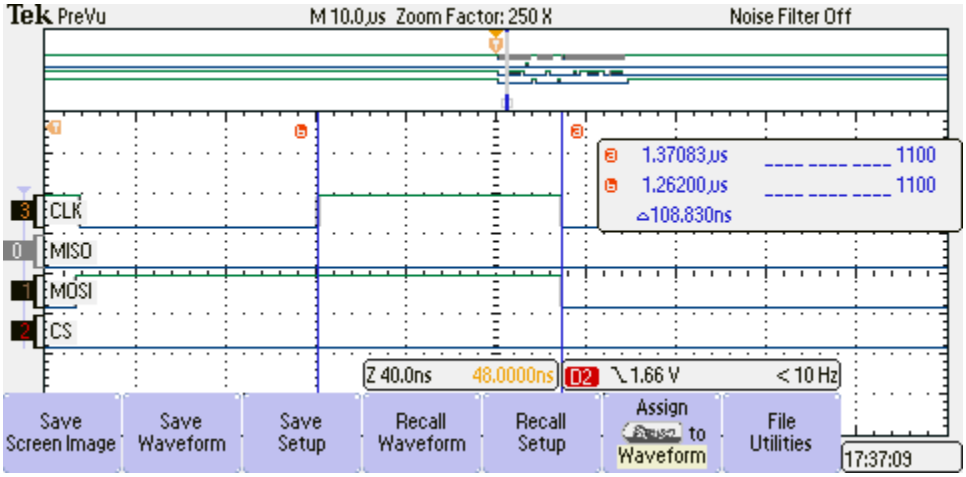


Lab 5 Report

MOSI Hold

Required = 40ns

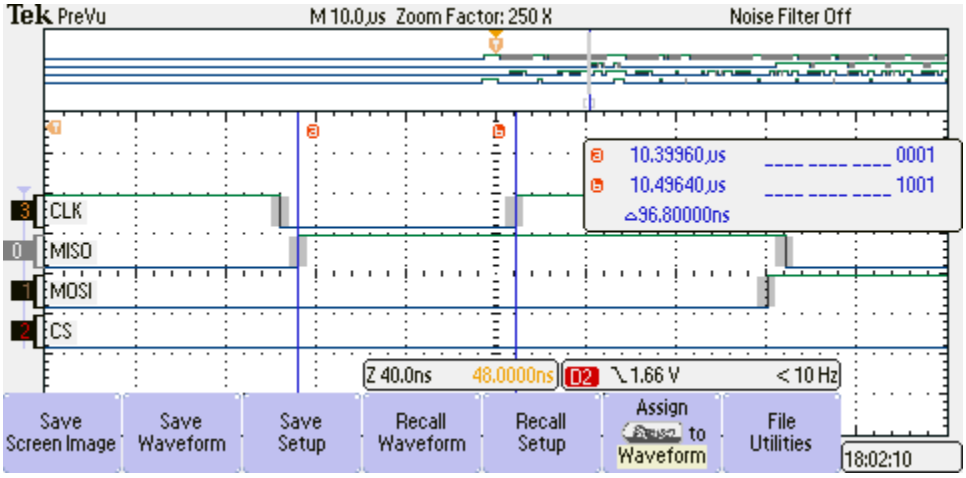
Recorded = 108ns



MISO Setup

Required = 5ns

Recorded = 97ns

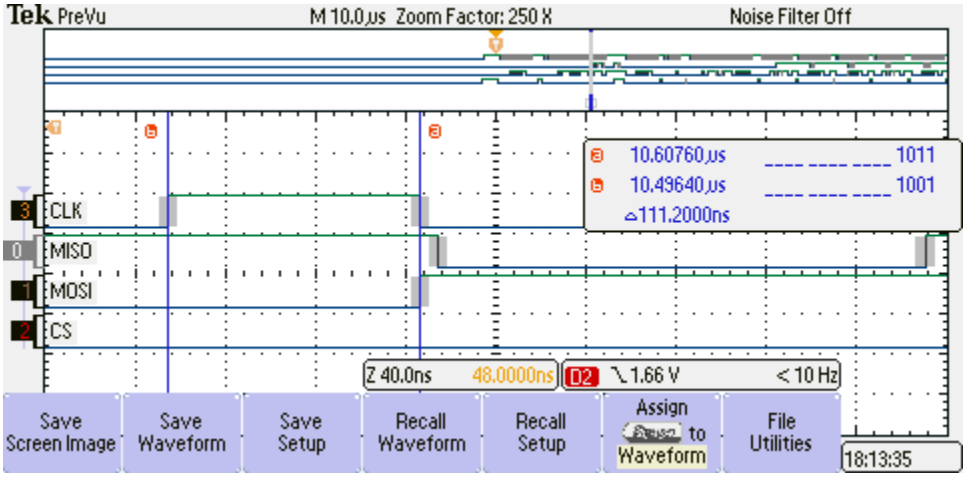


Lab 5 Report

MISO Hold

Required = 5ns

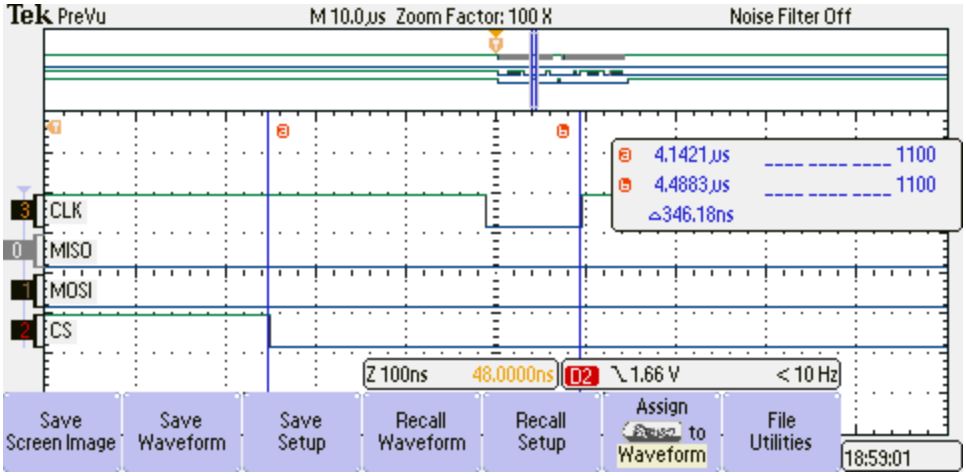
Recorded = 111ns



CS Setup

Required = 100ns

Recorded = 346ns

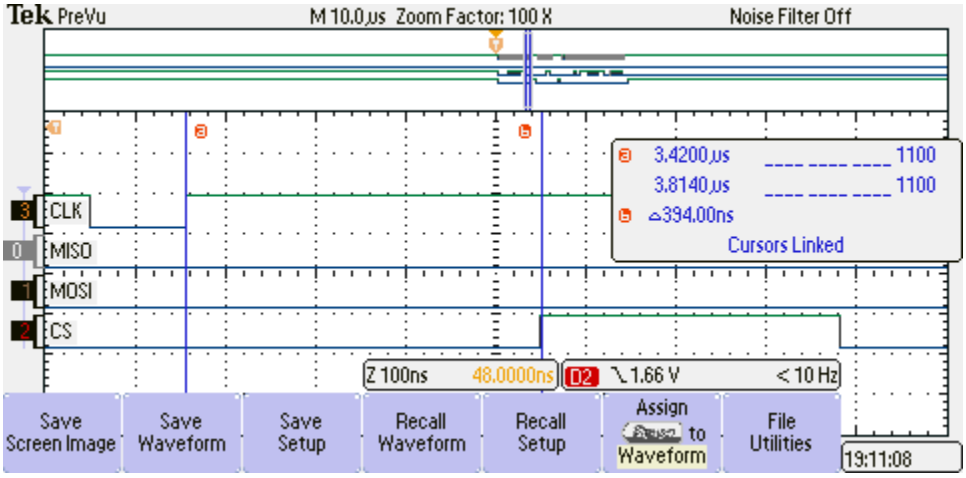


Lab 5 Report

CS Hold Time

Required = 200ns

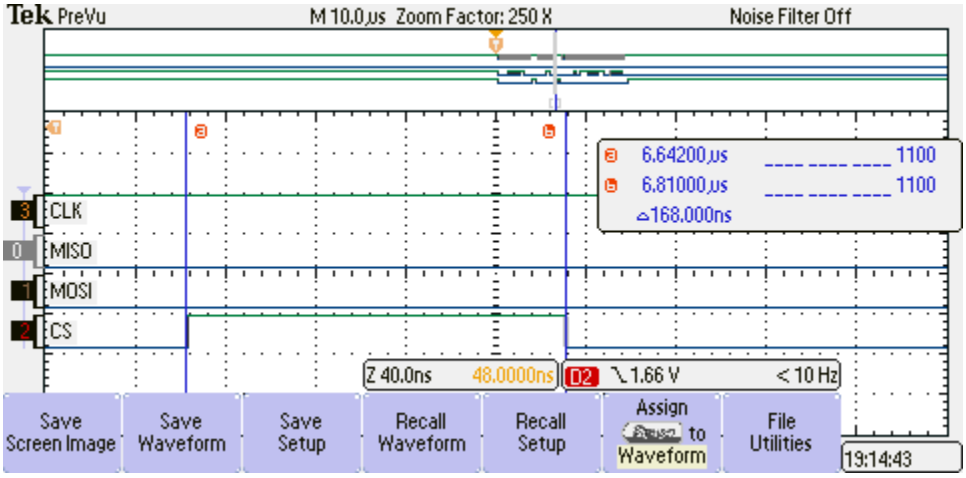
Recorded = 394ns



CS Disable Time

Required = 50ns

Recorded = 168ns



Lab 5 Report

There are 43 clock cycles between writing to an empty SPI4BUF and the TBE bit is set. I used the stopwatch window to record the cycles between when we right to an empty SPI4BUF and when the TBE bit becomes set. This should be fairly accurate but could be slightly off depending on how accurate the stopwatch window is.

Part 2

Code Description

State 0 is the initial state for the which kicks off the read status sequence. It asserts the CS line and then writes the read status value to the 25LC256. After waiting for the TBE it writes the initial dummy and transitions to state 1. State 1 continues the read status by reading the dummy and then switching to state 2. State three finishes the read status by reading the dummy and then negates the CS. It then Asserts the CS to prepare for whatever is next. If a write is still in progress it will continue to do a read status. Once a write is not longer in progress it checks whether a write or a read has been set in the operation variable. If a write was in the operation variable, then we write a 0x06 to the 25LC256 to start the write process and then go to state 3. If it is not a write than it must be read so we write a 0x03 to the 25LC256 and then wait for the TBE. After waiting for the TBE we load the MSA to the 25LC256 and then go to state 4.

State 3 ends the write enable and starts the write. This state does the write up until the MSA is loaded to the 25LC256. After the MSA is sent to the 25LC256 the state will shift to state 6. State 4 is the continuation of the read in which we read the dummy and then load the LSA to the 25LC256 and then change the state to state 5. State 5 reads the dummy and then writes a dummy for the read and proceeds to state 8. State 6 is the continuation of the write we read the dummy and then load the LSA to the 25LC256 and then proceeds to state 7. State 7 is the portion of the code where we actually write the data to the global pointer and then moving to state 10. State 8 reads the dummy for the read process and if we have more than 1 byte left to read we right another dummy. After this it moves to state 9.

State 9 is a loop that continues to loop and reads the data until there is no more data to be read. Once state 9 has gone through all the data that there is to be read it will set EEPromSysBusy to 0 as well as the index and state to 0. It also negates the CS to end the process. State 10 is the loop for the write which continues to write until we have no more data items to write. Once all the data has been written it will move to state 11. State 11 reads the final dummy and sets the EEPromSysBusy, index, and state to 0 as well as negate the CS.

Lab 5 Report

```
        case 3:
            SPI4BUF;                                    //read iDummy
            LATFSET = _LATF_LATF8_MASK;                 //negate the CS
            LATFCLR = _LATF_LATF8_MASK;                 //assert the CS
            SPI4BUF = 0x2;                              //send write command to 25LC256
            while(SPI4STATbits.SPITBE == 0);            //wait for Transmit buffer to empty
            SPI4BUF = MSA;                              //send MSA to 25LC256
```

```
!            SPI4BUF;                                   //read iDummy
0x9D00041C: LUI V0, -16510
0x9D000420: LW V0, 5664(V0)
!
!            LATFSET = _LATF_LATF8_MASK;                //negate the CS
0x9D000424: LUI V0, -16506
0x9D000428: ADDIU V1, ZERO, 256
0x9D00042C: SW V1, 1336(V0)
!            LATFCLR = _LATF_LATF8_MASK;                //assert the CS
0x9D000430: LUI V0, -16506
0x9D000434: ADDIU V1, ZERO, 256
0x9D000438: SW V1, 1332(V0)
!
!            SPI4BUF = 0x2;                             //send write command to 25LC256
0x9D00043C: LUI V0, -16510
0x9D000440: ADDIU V1, ZERO, 2
```

There are only 3 clock cycles between negating the CS and asserting again. This does not meet the timing requirements and it was necessary to at 2 “NOP” to the code to meet the timing requirements.

Rechecking CS timing requirements:

CS Setup

Measured = 285.8ns

Required = 100ns

Requirement met.

Lab 5 Report

CS Hold

Measured = 1.05us

Required = 200ns

Requirement met

CS Disable

Measured = 71.6ns

Required = 50ns

Requirement met

SPICLK = 4.67MHz/214ns

SYSCLK = 84MHz

PBCLK = 84MHz

Latency + Execution Times

1.) 2.452us (+/- 902ns)
2.) 1.592us (+/- 1.57us)
3.) 3.444us (+/- 2.24us)
4.) 1.98us (+/- 1.268us)
5.) 1.692us (+/ 1.58us)
6.) 2.12us (+/- 1.068us)

Lab5Part1 Code

```
// PIC32MZ2048ECG144, EFM144 or or EFG144 based HMZ144 board Configuration Bit Settings

// DEVCFG2

#if defined(__32MZ2048EFG144__) || defined(__32MZ2048EFM144__)

#pragma config FPLLIDIV = DIV_4         // System PLL Input Divider (4x Divider) for 24MHz clock (Rev C1 board w EFG) 24MHz/4 = 6MHz

                                        // also 24MHz clock rev C board w EFM (weird - went back to C. rev D also is EFM but with Osc)

#pragma config UPLLFSEL = FREQ_24MHZ    // USB PLL Input Frequency Selection (USB PLL input is 24 MHz)

#else
```

Lab 5 Report

```
#pragma config FPLLIDIV = DIV_2        // System PLL Input Divider (2x Divider) for 12 MHz crystal (Rev B and C boards w ECG) 12MHz/2 = 6MHz

#pragma config UPLLEN = OFF            // USB PLL Enable (USB PLL is disabled)

#endif

#pragma config FPLLRNG = RANGE_5_10_MHZ // System PLL Input Range (5-10 MHz Input)

#pragma config FPLLICLK = PLL_POSC     // System PLL Input Clock Selection (POSC is input to the System PLL)

#pragma config FPLLMULT = MUL_112      // System PLL Multiplier (PLL Multiply by 112) 6MHz * 112 = 672MHz

#pragma config FPLLODIV = DIV_8        // System PLL Output Clock Divider (8x Divider) 672MHz / 2 = 84MHz


// DEVCFG1

#pragma config FNOSC = SPLL            // Oscillator Selection Bits (Primary Osc (HS,EC))

#pragma config FSOSCEN = OFF           // Secondary Oscillator Enable (Disable SOSC)

#if defined(__32MZ2048EFG144__)

#pragma config POSCMOD = EC            // Primary Oscillator Configuration EC - External clock osc

                        // Rev C1 board w EFG uses an Oscillator (Rev D boards too))

#else

#pragma config POSCMOD = HS            // Primary Oscillator Configuration HS - Crystal osc

                        // Rev B and C (w ECG or EFM) use Crystals

#endif

#pragma config FCKSM = CSDCMD          // Clock Switching and Monitor Selection (Clock Switch Disabled, FSCM Disabled)

#pragma config FWDTEN = OFF            // Watchdog Timer Enable (WDT Disabled)

#pragma config FDMTEN = OFF            // Deadman Timer Enable (Deadman Timer is disabled)

#pragma config DMTINTV = WIN_127_128   // Default DMT Count Window Interval (Window/Interval value is 127/128 counter value)

#pragma config DMTCNT = DMT31          // Max Deadman Timer count = 2^31


// DEVCFG0
```

Lab 5 Report

```
#pragma config JTAGEN = OFF           // JTAG Enable (JTAG Disabled)
#pragma config ICESEL = ICS_PGx2      // ICD/ICE is on PGEC2/PGED2 pins (not default)

#include <xc.h>
#include <sys/attribs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>



//Functions
int ReadStatus();
void Write(int address, int data);
int Read(int address);

int main()
{
   int stat;
   int read;
   SYSKEY = 0; // Ensure lock
   SYSKEY = 0xAA996655; // Write Key 1
   SYSKEY = 0x556699AA; // Write Key 2
   PB2DIV = _PB2DIV_ON_MASK | 0 & _PB2DIV_PBDIV_MASK; // 0 = div by 1, 1 = div by 2, 2 = div by 3
etc up to 128
   SYSKEY = 0; // Re lock
   PRECON = (1 & _PRECON_PFMWS_MASK) | ((2 << _PRECON_PREFEN_POSITION) &
_PRECON_PREFEN_MASK);

   SPI4BRG = 8;                          //set baud rate
```

```
    ANSELBCLR = _ANSELB_ANSB3_MASK;
    SDI4R = 0x8;
    RPF2R = 0x8;

    LATFSET = _LATF_LATF8_MASK;
    TRISFCLR = _TRISF_TRISF8_MASK;

    SPI4CON = (_SPI4CON_MSTEN_MASK | _SPI4CON_CKP_MASK);    //Turn on master and ckp

    SPI4CONSET = _SPI4CON_ON_MASK;                    //turn on SPI4CON

    ReadStatus();
    Read(0xF800);
    Write(0xF800, 0xAA);
    Read(0xF800);

    return 0;
}

int Read(int address)
{
    int readItem;
    while(ReadStatus() & 1);

    LATFCLR = _LATF_LATF8_MASK;          //assert the CS

    SPI4BUF = 0x3;                    //write a read

    while(SPI4STATbits.SPITBE == 0);       //wait for Transmit buffer to empty
```

Lab 5 Report

```
    SPI4BUF = (address >> 8);             //write MSA

    while(SPI4STATbits.SPIRBF == 0);      //wait for receive buffer full

    SPI4BUF;                              //read SPI4BUF

    SPI4BUF = address;                    //write LSA

    while(SPI4STATbits.SPIRBF == 0);      //wait for receive buffer full

    SPI4BUF;                              //read SPI4BUF

    SPI4BUF = 'x';                        //write dummy

    while(SPI4STATbits.SPIRBF == 0);      //wait for receive buffer full

    SPI4BUF;                              //read SPI4BUF

    while(SPI4STATbits.SPIRBF == 0);      //wait for receive buffer full

    readItem = SPI4BUF;                   //read data

    LATFSET = _LATF_LATF8_MASK;           //negate the CS

    return readItem;
}

void Write(int address, int data)
```

Lab 5 Report

```
{

   while(ReadStatus() & 1);
/********Write Enable*************************************************************/
   LATFCLR = _LATF_LATF8_MASK;          //assert the CS

   SPI4BUF = 0x6;                       //write a write enable

   while(SPI4STATbits.SPIRBF == 0);     //wait for receive buffer full

   SPI4BUF;                             //read SPI4BUF

   LATFSET = _LATF_LATF8_MASK;          //negate the CS
/*****************************************************************************/
   asm("NOP");                          //added NOP for hold time
   asm("NOP");

   LATFCLR = _LATF_LATF8_MASK;          //assert the CS

   SPI4BUF = 0x2;                       //write a write

   while(SPI4STATbits.SPITBE == 0);     //wait for Transmit buffer to empty

   SPI4BUF = (address >> 8);            //write MSA

   while(SPI4STATbits.SPIRBF == 0);     //wait for receive buffer full

   SPI4BUF;
```

```
    SPI4BUF = address;                  //write LSA

    while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full

    SPI4BUF;

    SPI4BUF = data;

    while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full

    SPI4BUF;

    while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full

    SPI4BUF;

    LATFSET = _LATF_LATF8_MASK;         //negate the CS
}

int ReadStatus()
{
    int status;

    LATFCLR = _LATF_LATF8_MASK;     //assert the CS

    SPI4BUF = 0x5;                  //write a read status

    while(SPI4STATbits.SPITBE == 0);    //wait for Transmit buffer to empty
```

Lab 5 Report

```
    SPI4BUF = 'X';                  //write dummy byte

    while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full

    SPI4BUF;                        //read SPI4BUFF

    while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full

    status = SPI4BUF;

    LATFSET = _LATF_LATF8_MASK;         //negate the CS

    return status;
}
```

Lab5Part2 code

```
// PIC32MZ2048ECG144, EFM144 or or EFG144 based HMZ144 board Configuration Bit Settings
// DEVCFG2
#if defined(__32MZ2048EFG144__) || defined(__32MZ2048EFM144__)
#pragma config FPLLIDIV = DIV_4         // System PLL Input Divider (4x Divider) for 24MHz clock (Rev C1 board w EFG) 24MHz/4 = 6MHz
                                        // also 24MHz clock rev C board w EFM (weird - went back to C. rev D also is EFM but with Osc)
#pragma config UPLLFSEL = FREQ_24MHZ    // USB PLL Input Frequency Selection (USB PLL input is 24 MHz)
#else
#pragma config FPLLIDIV = DIV_2         // System PLL Input Divider (2x Divider) for 12 MHz crystal (Rev B and C boards w ECG) 12MHz/2 = 6MHz
#pragma config UPLLEN = OFF             // USB PLL Enable (USB PLL is disabled)
#endif
```

Lab 5 Report

```
#pragma config FPLLRNG = RANGE_5_10_MHZ // System PLL Input Range (5-10 MHz Input)

#pragma config FPLLICLK = PLL_POSC      // System PLL Input Clock Selection (POSC is input to the System PLL)

#pragma config FPLLMULT = MUL_112       // System PLL Multiplier (PLL Multiply by 112) 6MHz * 112 = 672MHz

#pragma config FPLLODIV = DIV_8         // System PLL Output Clock Divider (8x Divider) 672MHz / 2 = 84MHz


// DEVCFG1

#pragma config FNOSC = SPLL             // Oscillator Selection Bits (Primary Osc (HS,EC))

#pragma config FSOSCEN = OFF            // Secondary Oscillator Enable (Disable SOSC)

#if defined(__32MZ2048EFG144__)

#pragma config POSCMOD = EC             // Primary Oscillator Configuration EC - External clock osc

                        // Rev C1 board w EFG uses an Oscillator (Rev D boards too))

#else

#pragma config POSCMOD = HS             // Primary Oscillator Configuration HS - Crystal osc

                        // Rev B and C (w ECG or EFM) use Crystals

#endif

#pragma config FCKSM = CSDCMD           // Clock Switching and Monitor Selection (Clock Switch Disabled, FSCM Disabled)

#pragma config FWDTEN = OFF             // Watchdog Timer Enable (WDT Disabled)

#pragma config FDMTEN = OFF             // Deadman Timer Enable (Deadman Timer is disabled)

#pragma config DMTINTV = WIN_127_128    // Default DMT Count Window Interval (Window/Interval value is 127/128 counter value)

#pragma config DMTCNT = DMT31           // Max Deadman Timer count = 2^31


// DEVCFG0

#pragma config JTAGEN = OFF             // JTAG Enable (JTAG Disabled)

#pragma config ICESEL = ICS_PGx2        // ICD/ICE is on PGEC2/PGED2 pins (not default)
```

Lab 5 Report

```
#include <xc.h>
#include <sys/attribs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ADDRS 0xF800

//Functions
int ReadStatus();
void Write(int address, int data);
int Read(int address);
void ReadEEProm(int nbytes, unsigned int eeprom_address, unsigned char* readbuffer);
void WriteEEProm(int nbytes, unsigned int eeprom_address, unsigned char* writebuffer);

//Global Variables
int EEPromSysBusy = 0;
int State = 0;
int operation;
int NBytes;
int indexnum = 0;
unsigned int GlobalAddress;
unsigned char* GlobalPtr;
unsigned char writebuffer[65];
unsigned char reabuffer[65];

int main()
{
   int stat;
```

Lab 5 Report

```
int read;
SYSKEY = 0; // Ensure lock
SYSKEY = 0xAA996655; // Write Key 1
SYSKEY = 0x556699AA; // Write Key 2
PB2DIV = _PB2DIV_ON_MASK | 0 & _PB2DIV_PBDIV_MASK; // 0 = div by 1, 1 = div by 2, 2 = div by 3 etc up to 128
SYSKEY = 0; // Re lock
PRECON = (1 & _PRECON_PFMWS_MASK) | ((2 << _PRECON_PREFEN_POSITION) & _PRECON_PREFEN_MASK);


//Interrupt Setup
PRISSSET = 7 << _PRISS_PRI7SS_POSITION;             //Give Shadow Set 7 to Interrupt Group Priority 7
IFS5CLR = _IFS5_SPI4RXIF_MASK;                      //Clear SPI4RXIF
INTCONSET =  _INTCON_MVEC_MASK;                     //Set MVEC to 1 for multi-vector mode
IPC41CLR = _IPC41_SPI4RXIP_MASK;                    //Clear SPI4RX priority and subpriority
IPC41SET = 7 << _IPC41_SPI4RXIP_POSITION;           //Set Interrupt Group Priority to 7
IEC5SET = _IEC5_SPI4RXIE_MASK;                      //Enable SPI4RXIE



SPI4BRG = 16;                                       //set baud rate
ANSELBCLR = _ANSELB_ANSB3_MASK;
SDI4R = 0x8;
RPF2R = 0x8;

LATHSET = _LATH_LATH8_MASK;
TRISHCLR = _TRISH_TRISH8_MASK;

LATFSET = _LATF_LATF8_MASK;
TRISFCLR = _TRISF_TRISF8_MASK;
```

Lab 5 Report

```
SPI4CON = (_SPI4CON_MSTEN_MASK | _SPI4CON_CKP_MASK);    //Turn on master and ckp

SPI4CONSET = _SPI4CON_ON_MASK;                    //turn on SPI4CON

char rbuf[64], wbuf[64];

memset(rbuf, '\0', (size_t)64);
wbuf[64] = '\0';

int  n = 1, count = 0, data = 0x20;

while(count < 64)                              // initialize rbuf and wbuf
{
   wbuf[count] = data++;
   rbuf[count] = 0x00;
   count++;
}
asm("ei");

ReadEEProm(1,ADDRS,rbuf);            // initial read to see what is in the EEPROM at ADDRS

while(EEPromSysBusy)                 // wait for read to be done
{
   LATHINV = _LATH_LATH8_MASK;
}

while(n <= 64)                       // loop to write 1,2,3 and 64 unique items
{
```

```
      WriteEEProm(n,ADDRS,wbuf+64-n);
      ReadEEProm(n,ADDRS,rbuf);
      while(EEPromSysBusy);            // wait for read to be done
      if(n < 3) n++;                   // Go to the next n (can also breakpoint here) SPI4STAT
      else n = 64;
   }

   return 0;
}

int Read(int address)
{
   int readItem;
   while(ReadStatus() & 1);

   LATFCLR = _LATF_LATF8_MASK;          //assert the CS

   SPI4BUF = 0x3;                       //write a read

   while(SPI4STATbits.SPITBE == 0);      //wait for Transmit buffer to empty

   SPI4BUF = (address >> 8);            //write MSA

   while(SPI4STATbits.SPIRBF == 0);      //wait for receive buffer full

   SPI4BUF;                             //read SPI4BUF

   SPI4BUF = address;                   //write LSA
```

```
   while(SPI4STATbits.SPIRBF == 0);       //wait for receive buffer full

   SPI4BUF;                          //read SPI4BUF

   SPI4BUF = 'x';                     //write dummy

   while(SPI4STATbits.SPIRBF == 0);       //wait for receive buffer full

   SPI4BUF;                          //read SPI4BUF

   while(SPI4STATbits.SPIRBF == 0);       //wait for receive buffer full

   readItem = SPI4BUF;                  //read data

   LATFSET = _LATF_LATF8_MASK;           //negate the CS

   return readItem;
}

void Write(int address, int data)
{

   while(ReadStatus() & 1);
/********Write Enable*******************************************************/
   LATFCLR = _LATF_LATF8_MASK;           //assert the CS

   SPI4BUF = 0x6;                     //write a write enable

   while(SPI4STATbits.SPIRBF == 0);       //wait for receive buffer full
```

Lab 5 Report

```
  SPI4BUF;                    //read SPI4BUF

  LATFSET = _LATF_LATF8_MASK;       //negate the CS
/*****************************************************************************/
  asm("NOP");                 //added NOP for hold time
  asm("NOP");

  LATFCLR = _LATF_LATF8_MASK;       //assert the CS

  SPI4BUF = 0x2;                //write a write

  while(SPI4STATbits.SPITBE == 0);    //wait for Transmit buffer to empty

  SPI4BUF = (address >> 8);         //write MSA

  while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full

  SPI4BUF;

  SPI4BUF = address;              //write LSA

  while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full

  SPI4BUF;

  SPI4BUF = data;

  while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full
```

```
    SPI4BUF;

    while(SPI4STATbits.SPIRBF == 0);       //wait for receive buffer full

    SPI4BUF;

    LATFSET = _LATF_LATF8_MASK;           //negate the CS
}

int ReadStatus()
{
    int status;

    LATFCLR = _LATF_LATF8_MASK;        //assert the CS

    SPI4BUF = 0x5;                  //write a read status

    while(SPI4STATbits.SPITBE == 0);    //wait for Transmit buffer to empty

    SPI4BUF = 'X';                  //write dummy byte

    while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full

    SPI4BUF;                        //read SPI4BUFF

    while(SPI4STATbits.SPIRBF == 0);    //wait for receive buffer full

    status = SPI4BUF;
```

```
    LATFSET = _LATF_LATF8_MASK;        //negate the CS

    return status;
}

void ReadEEProm(int nbytes, unsigned int eeprom_address, unsigned char* readbuffer)
{
    while(EEPromSysBusy);              //wait for EEPromSysBusy to be low
    if(State != 0)                     //if not in state 0 throw error
    {
        exit(-1);
    }
    EEPromSysBusy = 1;                 //set EEPromSysBusy to 1
    operation = 0x03;                  //set operation for read
    GlobalAddress = eeprom_address;    //set the global address to given address
    GlobalPtr = readbuffer;            //set the global pointer to writebuffer
    NBytes = nbytes;                   //set global NBytes to given nbytes

    IFS5SET = _IFS5_SPI4RXIF_MASK;     //set the SPI4RXIF flag to start things
}

void WriteEEProm(int nbytes, unsigned int eeprom_address, unsigned char* writebuffer)
{
    while(EEPromSysBusy);              //wait for EEPromSysBusy to be low
    if(State != 0)                     //if not in state 0 throw error
    {
        exit(-1);
    }
```

```
    EEPromSysBusy = 1;              //set EEPromSysBusy to 1
    operation = 0x02;               //set operation for write
    GlobalAddress = eeprom_address;     //set the global address to given address
    GlobalPtr = writebuffer;          //set the global pointer to writebuffer
    NBytes = nbytes;                //set global NBytes to given nbytes

    IFS5SET = _IFS5_SPI4RXIF_MASK;      //set the SPI4RXIF flag to start things
}

void __ISR_AT_VECTOR(_SPI4_RX_VECTOR, IPL7SRS) SPI4ISR(void)
{
    unsigned char status;
    unsigned char MSA = GlobalAddress >> 8;     //set MSA to most significant 8 bits
    unsigned char LSA = GlobalAddress;         //set LSA to rest of the bits
    switch(State)
    {
       case 0:

          LATFCLR = _LATF_LATF8_MASK;        //assert the CS

          SPI4BUF = 0x5;                  //write a read status

          while(SPI4STATbits.SPITBE == 0);    //wait for Transmit buffer to empty

          SPI4BUF = 'X';                  //write dummy byte

          State = 1;

          break;
```

Lab 5 Report

```
case 1:

   SPI4BUF;                    //read SPI4BUFF

   State = 2;

   break;
case 2:

   status = SPI4BUF;

   LATFSET = _LATF_LATF8_MASK;          //negate the CS

   LATFCLR = _LATF_LATF8_MASK;          //assert the CS

   if((status & 0x01) > 0)              //check for write still in progress
   {
      SPI4BUF = 0x05;                  //send read status command to 25LC256

      while(SPI4STATbits.SPITBE == 0);    //wait for Transmit buffer to empty

      SPI4BUF = 'X';                   //write dummy byte

      State = 1;
   }
   else if(operation == 0x02)           //check for write operation
   {
      SPI4BUF = 0x06;                  //write write enable to 25LC256
```

```
    State = 3;
  }
  else
  {
    SPI4BUF = 0x03;                  //write a read

    while(SPI4STATbits.SPITBE == 0);       //wait for Transmit buffer to empty

    SPI4BUF = MSA;                   //send MSA to 25LC256

    State = 4;
  }

  break;

case 3:

  SPI4BUF;                           //read iDummy

  LATFSET = _LATF_LATF8_MASK;              //negate the CS

  asm("nop");
  asm("nop");

  LATFCLR = _LATF_LATF8_MASK;              //assert the CS

  SPI4BUF = 0x2;                     //send write command to 25LC256

  while(SPI4STATbits.SPITBE == 0);         //wait for Transmit buffer to empty
```

```
    SPI4BUF = MSA;                  //send MSA to 25LC256

    State = 6;

    break;

case 4:

    SPI4BUF;                    //read SPI4BUF

    SPI4BUF = LSA;                  //write LSA

    State = 5;

    break;

case 5:

    SPI4BUF;                    //read SPI4BUF

    SPI4BUF = 'x';                  //write iDummy

    State = 8;

    break;

case 6:
```

```
    SPI4BUF;                    //read SPI4BUF

    SPI4BUF = LSA;              //write LSA

    State = 7;

    break;

case 7:

    SPI4BUF;                    //read Dummy

    SPI4BUF = *GlobalPtr;       //write data

    State = 10;

    break;

case 8:

    asm("nop");

    SPI4BUF;                    //read dummy

    if(NBytes > 1)
      SPI4BUF = 'x';            //write dummy if needed

    State = 9;
```

```
    break;

case 9:

    *GlobalPtr = SPI4BUF;                //read data

    if(indexnum < (NBytes - 1))          //traverse readbuffer until end
    {
        if(indexnum < (NBytes - 2))      //check if there is going to be at least 2 more reads
        {
            SPI4BUF = 'x';               //write dummy if there are at least 2 more reads
        }
        State = 9;                       //do state 9 again
        indexnum++;
        GlobalPtr++;
    }
    else
    {
        EEPromSysBusy = 0;               //if at end set EEPromSysBusy to 0
        indexnum = 0;                    //reset index
        LATFSET = _LATF_LATF8_MASK;      //negate the CS
        State = 0;
    }
    break;

case 10:

    SPI4BUF;                             //read dummy
```

```
        if(indexnum < (NBytes - 1))
        {
          indexnum++;
          GlobalPtr++;
          SPI4BUF = *GlobalPtr;              //write data
          State = 10;
        }
        else
        {
          State = 11;
        }
        break;

    case 11:

        SPI4BUF;                        //read dummy

        EEPromSysBusy = 0;                //if at end set EEPromSysBusy to 0
        indexnum = 0;                     //reset index
        LATFSET = _LATF_LATF8_MASK;       //negate the CS
        State = 0;
        break;

    default:

        exit(-1);

        break;
}
```

```
    IFS5CLR = _IFS5_SPI4RXIF_MASK;      //clear interrupt flag
}
```