ELEC342 Laboratory Report 1

Arman Chowdhury (43925197)

I. Introduction:

We were given the task of designing and building a simple set of traffic lights using a PIC18F14K22 microcontroller and a MCP23S17 Port Expander in assembly language. We were required to use the PIC controller in its SPI mode of operation and use the Port Expander as a slave device. In the design of our traffic lights special consideration had to be given to the wiring and initializing the two chips, the communication link amongst various other things. We made close references to the datasheet of both chips.

II. Procedure and Discussion:

Throughout the design process there were many key points in which crucial decisions were made in relation to the design of the traffic light system. These are highlighted as follows:

- **Wiring and Design Overview**

From Section 14 (Master Synchronous Serial Port Module) of the PIC18F14K22 datasheet we discovered the different registers and pins required for the PIC to run in SPI (Serial Peripheral Interface) mode. The overview of the design and wiring are shown below.

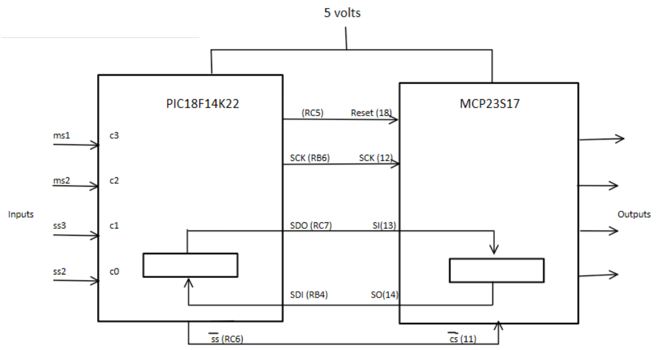

Figure 1: Design Overview

We discovered that when the PIC controller is in SPI mode, it allowed eight bits of data to be synchronously transmitted and received. To establish communication, typically three pins are used, namely – SDO (Serial Data Out), SDI (Serial Data In), SCK (Serial Clock). The Port Expander (MCP23S17) was used to accommodate our output pins for the traffic lights. We allocated four I/O pins of the PIC controller for our sensor inputs.

To use the PIC controller there were four main registers that needed to be considered. These are SSPCON1 (Control Register), SSPSTATUS (Status Register), SSPBUF (Serial Receive/Transmit Buffer) and SSPSR (Shift Register). Each register had a specific set of 8-bits which, when configured would allow us to setup the PIC controller to perform our desired set of instructions.

- **Initializing the PIC18F14K22**

To use the PIC controller various registers needed to be initialized. Since the I/O pins we were using were spread over PORTB and PORTC, we had to initialize TRISB and TRISC registers. Configuring these two registers would set the I/O pins to either function as an input or an output.

For both these registers, we needed the pins to be set to 1 for an input, else the pins were set to 0. This resulted in the following bit sequences for the TRISC and TRISB registers.

TRISC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0
Bit	0	0	0	X	1	1	1	1

TRISB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
Bit	x	0	x	1	x	x	x	x

The next Registers we had to configure were the control register (SSPCON1) and the status register (SSPSTAT).

Through examination of the PIC data sheet we could find certain key pins which needed to be set to control certain characteristics of the system. Bit-7 was the Sample Bit (SMP) which we set to 0. This was because we wanted the input data to be sampled at the middle of data output time. This was also where our data was stable and thus could be read with no errors. Bit-6 was the SPI Clock Select Bit (CKE) which we set to 1. Bit-5 to bit-1 are only required in I^2C mode and we kept it at the default value of 0. Bit-0 was the Buffer Full Status Bit (BF), which we set to 0. However, we later changed this bit per our needs. We ended up with the following 8-bit sequence for this register.

SSPSTAT	SMP	CKE	D/A	P	S	R/W	UA	BF
Bit	0	1	X	X	X	X	X	0

Lastly, we looked at the SSPCON1 register. Bit-7, the Write Collision Detect Bit (WCOL) and bit-6, the Receive Overflow indicator bit (SSPOV) was left at their default value of 0, since it didn't concern our design and the PIC controller was not operating in slave mode. Bit-5 was the Synchronous Serial Port Enable (SSPEN) which was set to 1. This enabled the serial port and configured SCK, SDO, SDI and SS as serial port pins. Bit-4 was the Clock Polarity Select (CKP) bit and was set to 0. Bit-3 to bit-0 were all set to 0. This made the SPI Master Mode clock run at a speed of $F_{OSC}/4$. We finally ended up with the following bit sequence for this register.

SSPCON1	WCOL	SSPOV	SSPEN	CKP	SSPM			
Bit	X	X	1	0	0	0	0	0

- **Clock**

To implement our desired system research was required to use an appropriate clock.

The Port Expander's maximum operational speed was 10MHz. We found from the PIC controller's datasheet that its maximum operating speed was 16MHz, which was clearly higher than that of the Port Expanders'. We designed our system so that the PIC being the master would supply the clock to the Port Expander. This allowed synchronicity to remain between the two devices. We chose a clock speed of 4MHz and configured the SSPM bits so it divided this value by 4. This resulted in a 1MHz clock speed for the system.

We could have picked 16MHz or even 8MHz as the clock speed and since we divide this value by 4, we would have still been under the maximum limit for the Port Expander. However, the variable delay we designed relied upon the clock speed we chose and any change to this clock speed would have changed the amount the system was delayed for. Also, we wanted to conserve power and this clock choice would allow optimum power usage for the system.

We also looked at the clock specifications for the Port Expander and compared that to the available clock signals of the PIC controller. We wanted to read the data when it was stable and on the rising clock edge. This meant that the clock we needed to use relied on the two bits. This waveform if highlighted in the figure below.

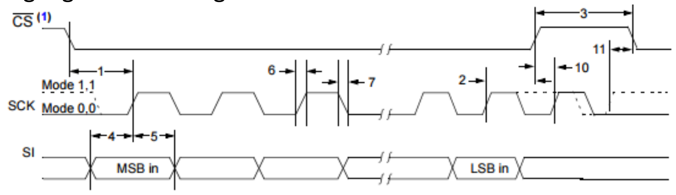

Figure 2: Clock required for Port Expander in SPI Input Mode

The clock we needed was formed using two bits (CKP and CKE). They were set to their respective values which corresponded to the correct clock which the Port Expander would recognize and accept.

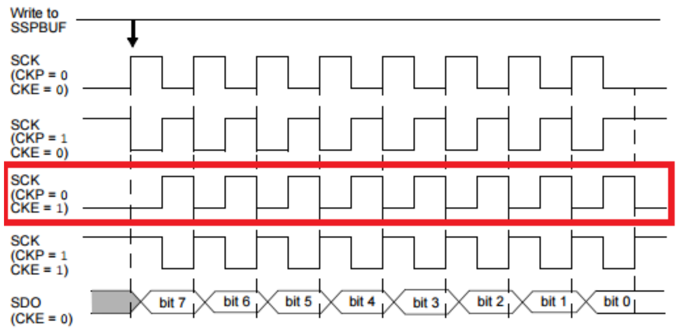

Figure 3: Clock we chose from the options

- **Transmission**

After the PIC initialization process was complete we needed to initialize the Port Expander. To do this a transmission link was required between the two chips. This was so the initialization protocol could be communicated from the PIC to the port expander. We saw that, for the PIC to communicate with the Port Expander, a specific set of waveforms using a data packet arrangement was required.

Firstly, we had to pull down the CS (Chip Select) bit to start transmission. Secondly, we saw that there had to be three packets of 8-bit data sent to the Port Expander. In between each packet, we set the BF (Buffer Full) bit to 1, to let the Port Expander know that a full byte of data had been transmitted.

The three packets of data we needed to send were, the Chip ID, the Register Address then the 8-bit Value we wanted to send. The last bit in the Chip ID byte was responsible for letting the Port Expander know whether we were performing a read or a write operation. The byte for the Register Address changed each time we wanted to write to a different register. So, we set up a memory location called "SPIregadd" which we used to pass register addresses through a subroutine to the Port Expander. We also set up a memory location called "SPIvalue" which we used to pass the byte of data we needed to send.

Since the Buffer Registers in both chips worked like a shift register, we used a branch loop and a bit test to determine whether the three bytes of data were completely transmitted. A combination of the subroutines "SPI_SEND" and "FULLSIGNAL_SEND" were used to make this transmission possible.

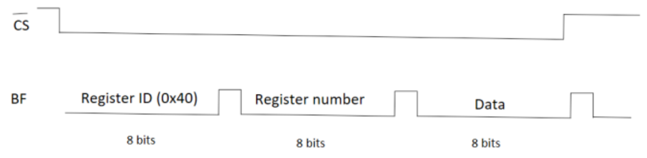

Figure 4: Transmission Waveform

- **Initializing the Port Expander**

After setting up the transmission line we could initialize the Port Expander. Since we were only using the Port Expander as a slave the only registers we needed to initialize were IODIR-A and IODIR-B. These IODIR registers controlled the direction of the data, - either input or output. Since we were only using the I/O pins as outputs, we set all of them to 0.

IODIR-A/B	IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0
Bit	0	0	0	0	0	0	X	X

Since there were twelve lights, we decided to split them in equal portions and have six on either PORT. We also understood that writing to the registers OLATA and OLATB would cause the lights on our board to change.

- **States and Traffic Lights Table**

Light	Hex	Bit Sequence							
MR	0X20	0	0	1	0	0	0	0	0
SR	0X10	0	0	0	1	0	0	0	0
MY	0X01	0	0	0	0	0	0	0	1
SY	0X08	0	0	0	0	1	0	0	0
MG	0X02	0	0	0	0	0	0	1	0
SG	0X04	0	0	0	0	0	1	0	0

There were only six unique lights, to which we assigned a unique 8-bit sequence to and used to form the different combinations required. We then used these bit sequences in an array table containing our 8 states.

We initially started off with 6 states and this covered all possible light combinations for the outcomes of the traffic lights. Shortly afterwards we tried to incorporate the four sensors and realized that we needed separate states where the sensor inputs could be sampled. We arrived at the following state transition diagram.

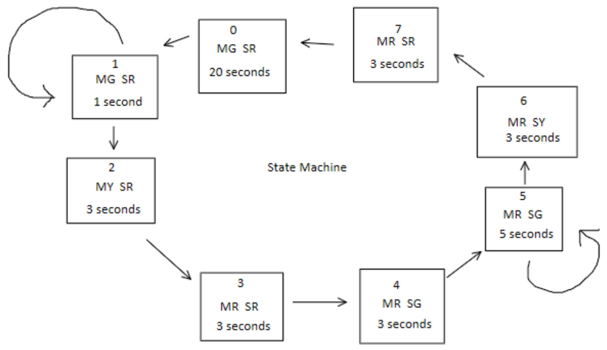

Figure 6: State Transition Diagram with Delays

We performed sensor checks in states one and five every second. This ensured that the lights would be on for the correct times and then would enter a state where a decision could be made based on the sensor inputs. The way in which we would move from one state to another was made easier by the array tables, since we could just increment the index.

- **Array Table for Variable Delay**

We made another array table for the Variable Delay. The table takes the value in the Working Register and uses it to jump to the appropriate index in the table. It does this by loading the value from the Working Register to the Program Counter. This then returns the 8-bit value in that index back to the working register.

Even though we only had four unique delays (twenty seconds, one second, three seconds and five seconds), we still thought it would be very convenient to just have the delays for each state in the same order as that, in the state transition diagram. This was a much simpler design rather than have extra sets of instructions and "if-conditions" for each respective state.

The Variable Delay subroutine was fine-tuned so that in every complete cycle it would have gone through roughly a million instructions. Given that the operating speed was 1MHz, this seemed to work out perfectly to give us roughly a one second delay.

- **Sensor Check**

Once the lights and states were set up and allowed to cycle through, we needed to perform sensor checks. As mentioned earlier, sensor checks were only required in states one and five. Thus, we needed to determine when our current states were one or five.

The way we determined if we were in state one was, we decremented the current state the lights were in and saw what the resulting state was. If the resulting state was zero, this would set the Z, Status bit to 1 and thus would mean that the current state the lights were in was one. Based on this result we would either perform sensor checks or check if the current state was five.

The way we checked if our current state was five was, we took the current state value and exclusive-or-ed it with the light output for state five. This meant that if we were in state five, the Z Status bit would be set to 1. Based on this result we either performed sensor checks or incremented the state. The "decrement method" used earlier wouldn't work with state five since, a decrement would cause the state to be four and not zero. This wouldn't set the Z Status Bit to 1 and thus wouldn't work. However, the "exclusive-or method" could have been implemented for the checking of state one.

- **Sensor Check for State One**
  When in state one we sampled the current PORTC values and performed an and-operation with the number 0x03. Since our least two significant bits were the Side Sensors (SS1 and SS2), this and operation highlighted only those two bits and got rid of the rest. Due to the and-operation, the Z Status bit would have been set to 1 only if both SS1 and SS2 are off. We performed a bit test on this Status bit and if it was set to 1 then we stayed in state one, else we incremented the current state.
- **Sensor Check for State Five**
  We drew up a truth table for this and found that we stay in state five for three specific conditions. We performed this check in two parts, -first we checked the Main Sensors and then we checked the Side Sensors.
  Since, the main road is prioritized over the side road, if either Main Sensors (MS1 and MS2) are triggered, we immediately would have to increment to state six. So, we checked for this first. We performed an and-operation with the number 0x0C which got rid of all the bits we weren't concerned with. We then checked for the Z Status Bit again. Only this time, if the bit was set to 1 we would perform further checks, otherwise we would immediately increment to state six. The Z Status Bit being set to 0 meant that either or both main sensors were triggered.
  We then performed the exact same check we did earlier for state 1. However, this time if the Z Status bit was set to 1 then we would increment and go to state six otherwise we would continue in state five.

To perform these comparisons, we made copies of the current state in memory locations, so the actual lights wouldn't be affected.

III. Alternative Methods

An alternate method would be to have the sensors incorporated into the Port Expander, since there were already enough I/O pins to do this. The initialization for IODIR-A and IODIR-B would have changed.

When we write to OLAT we automatically get a read as well since the two Buffer Registers in the chips work like a shift register. So, when we are in state one or state five we could capture the sensor values at every second and store this value in another register for later use. However, to get a read from the register, this design would require the PIC also simultaneously write to the port expander. Our system would constantly have to write to the lights, even when sampling the sensor values in state 1 and 5.

IV. Conclusion

We used the MCP23S17 and the PIC18FK22 throughout the course of this experiment. It is through this process that we understood how these two chips communicated with each other. We also understood the different modes of operation of both chips. The design choices made for this experiment were made meticulously and were optimum in achieving our objective.

V. References

[1] "ELEC342 report", Sandy Mason, May 7th, 2017

[2] "MCP23017/MCP23S17", 16-Bit I/O Expander with Serial Interface, Microchip

[3] "PIC18(L)F1XK22",2—Pin Flash Microcontrollers with XLP Technology, Microchip

VI. Appendix

```
;*****************************************************************************
;                                                 *
;   This file is a basic code template for code generation on the            *
;   PIC18F14K22. This file contains the basic code building blocks to build   *
;   upon.                                          *
;*****************************************************************************


;------------------------------------------------------------------------------
; PROCESSOR DECLARATION
;------------------------------------------------------------------------------


    LIST     P=PIC18F14K22          ; list directive to define processor
    #INCLUDE <P18F14K22.INC>        ; processor specific variable definitions


;------------------------------------------------------------------------------
;
; CONFIGURATION WORD SETUP
;
; The 'CONFIG' directive is used to embed the configuration word within the
; .asm file. The lables following the directive are located in the respective
; .inc file.  See the data sheet for additional information on configuration
; word settings.
;
;------------------------------------------------------------------------------


    ;Setup CONFIG11H
    CONFIG FOSC = IRC, PLLEN = OFF, PCLKEN = OFF, FCMEN = OFF, IESO = OFF
    ;Setup CONFIG2L
    CONFIG PWRTEN = ON, BOREN = OFF, BORV = 19
    ;Setup CONFIG2H
    CONFIG WDTEN = OFF, WDTPS = 1
    ;Setup CONFIG3H
    CONFIG MCLRE = OFF, HFOFST = OFF
    ;Setup CONFIG4L
    CONFIG STVREN = OFF, LVP = OFF, BBSIZ = OFF, XINST = OFF
    ;Setup CONFIG5L
    CONFIG CP0 = OFF, CP1 = OFF
    ;Setup CONFIG5H
    CONFIG CPB = OFF, CPD = OFF
    ;Setup CONFIG6L
    CONFIG WRT0 = OFF, WRT1 = OFF
    ;Setup CONFIG6H
    CONFIG WRTB = OFF, WRTC = OFF, WRTD = OFF
    ;Setup CONFIG7L
    CONFIG EBTR0 = OFF, EBTR1 = OFF
    ;Setup CONFIG7H
    CONFIG EBTRB = OFF

;------------------------------------------------------------------------------
;
; VARIABLE DEFINITIONS
;
; Refer to datasheet for available data memory (RAM) organization
;
;------------------------------------------------------------------------------

   CBLOCK 0x60 ; Sample GPR variable register allocations
      MYVAR1  ; user variable at address 0x60
      MYVAR2  ; user variable at address 0x61
      MYVAR3  ; user variable at address 0x62
   ENDC

   CBLOCK  0x00       ; Access RAM
   ENDC

;------------------------------------------------------------------------------
; EEPROM INITIALIZATION
; The 18F14K22 has 256 bytes of non-volatile EEPROM starting at 0xF00000
;------------------------------------------------------------------------------

;DATAEE    ORG  0xF00000 ; Starting address for EEPROM for 18F14K22
;    DE    "MCHP"        ; Place 'M' 'C' 'H' 'P' at address 0,1,2,3
```

```
;-----------------------------------------------------------------------------
; RESET VECTOR
;-----------------------------------------------------------------------------

RES_VECT  ORG     0x0000            ; processor reset vector
          GOTO    START             ; go to beginning of program

;-----------------------------------------------------------------------------
; HIGH PRIORITY INTERRUPT VECTOR
;-----------------------------------------------------------------------------

ISRH      ORG     0x0008

          ; Run the High Priority Interrupt Service Routine
          GOTO    HIGH_ISR

;-----------------------------------------------------------------------------
; LOW PRIORITY INTERRUPT VECTOR
;-----------------------------------------------------------------------------

ISRL      ORG     0x0018

          ; Run the High Priority Interrupt Service Routine
          GOTO    LOW_ISR

;-----------------------------------------------------------------------------
; HIGH PRIORITY INTERRUPT SERVICE ROUTINE
;-----------------------------------------------------------------------------

HIGH_ISR

          ; Insert High Priority ISR Here

          RETFIE  FAST

;-----------------------------------------------------------------------------
; LOW PRIORITY INTERRUPT SERVICE ROUTINE
```

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

LOW_ISR
        RETFIE

;----------------------------------------------------------------------------
; MAIN PROGRAM
;----------------------------------------------------------------------------

SPIregadd       equu    0x20
SPIvalue        equ     0x21
state           equ     0x22
temp1           equ     0x23
temp            equ     0x24


; Variable delay in seconds

DELAY
   movwf   0x70
DELAYLOOP0
   movlw   0x19
   movwf   0x71
DELAYLOOP1
   movlw   0x20
   movwf   0x72
DELAYLOOP2
   movlw   0x00
   movwf   0x73
DELAYLOOP3
   nop
   decfsz  0x73
   goto    DELAYLOOP3
   decfsz  0x72
   goto    DELAYLOOP2
   decfsz  0x71
   goto    DELAYLOOP1
```

```
    decfsz  0x70
    goto    DELAYLOOP0
    return



; Bits sequence for each individual light

MR  equ 0x20
SR  equ 0x10
MY  equ 0x01
SY  equ 0x08
MG  equ 0x02
SG  equ 0x04



display
    call    stateW      ; get LED pattern in W from state
    movwf   PORTC
    return
stateW
    clrf    PCLATH
    movf    state,W
    andlw   0x07   ; only states 0-7
    addlw   LOW table
    movwf   PCL
    nop
    nop
table                  ; the table must not extend over a 256 byte boundary
    retlw   MG|SR
    retlw   MG|SR
    retlw   MY|SR
    retlw   MR|SR
    retlw   MR|SG
    retlw   MR|SG
    retlw   MR|SY
```

```
   retlw   MR|SR



; Calculate the delay in seconds according to the state


DELAYV
   call    stateDW     ; get LED pattern in W from state
   return
stateDW
   clrf    PCLATH
   movf    state,W
   andlw   0x07   ; only states 0-7
   addlw   LOW tableD
   movwf   PCL
   nop
   nop
tableD              ; the table must not extend over a 256 byte boundary
   retlw   20
   retlw   1
   retlw   3
   retlw   3
   retlw   5
   retlw   1
   retlw   3
   retlw   3



FLASHING_LIGHTS
                              clrf    state   ; state of the traffic lights
LOOP2
                              call    display ; display the current state
                              call    DELAYV  ; calcualte the delay in this state
                              call    DELAY   ; and delay that many seconds

; The lights have been displayed long enough. Now, we recalculate and make a decision based on inputs


                              movf    state,W
```

```
                movwf   temp1
                decfsz  temp1
                goto    notState1

; state 1 - check MR
                movf    PORTC,W
                andlw   0xC0
                btfss   STATUS,2    ; if neither Side is set we stay here
                incf    state       ; not Zero means side road triggered
                goto    LOOP2       ; end of state 1

notState1
                xorlw   0x05        ; are we in state 5 (side check after 5 seconds)
                btfss   STATUS,2    ; if Z flag is set we are in state 5
                goto    notState5

; state 5 - check side road, or check main road
                movf    PORTC,W
                andlw   0xc0        ; side road set?
                btfsc   STATUS,2    ; if Z is set it means we move on
                goto    incState    ; no side road triggered, move to next state
                movf    PORTC,W
                andlw   0x30        ; check main road triggered
                btfss   STATUS,2
                incf    state       ; main road triggered move to next state
                goto    LOOP2
notState5                           ; not in state 1 or 5 - move to the next state

incState
                incf    state
                movlw   0x07
                andwf   state
                goto    LOOP2
```

```
SPI_SEND
                movwf  SSPBUF
LOOP
                btfss  SSPSTAT, BF
                goto   LOOP
                movf   SSPBUF, W
return



; Writing to the Port Ecpander

FULLSIGNAL_SEND
                bcf             PORTC, 6
                movlw  0x40
                call   SPI_SEND
                movf   SPIregadd,W
                call   SPI_SEND
                movf   SPIvalue,W
                call   SPI_SEND
                bsf             PORTC,6
return



START           ; Insert User Program Here

                CLRF    ANSEL
            CLRF    ANSELH
                clrf   PORTB
                movlw  0x10
                movwf  TRISB
                movlw  0x0f
                movwf  PORTC
                clrf   TRISC
                movlw  0x20
                movwf  SSPCON1
```

```
                    movlw   0x40
                    movwf   SSPSTAT


                    ;Initialization for port expander

                    movlw   0x01
                    movwf   SPIregadd
                    movlw   0x00
                    movwf   SPIvalue
                    call    FULLSIGNAL_SEND

                    movlw   0x00
                    movwf   SPIregadd
                    movlw   0x00
                    movwf   SPIvalue
                    call    FULLSIGNAL_SEND

main


                    movlw   0x15
                    movwf   SPIregadd
                    movlw   0xff
                    movwf   SPIvalue
                    call    FULLSIGNAL_SEND

                    movlw   0x14
                    movwf   SPIregadd
                    movlw   0xff
                    movwf   SPIvalue
                    call    FULLSIGNAL_SEND
                    goto    main

STOP      GOTO    STOP                    ; loop program counter

    END
```