October 20th

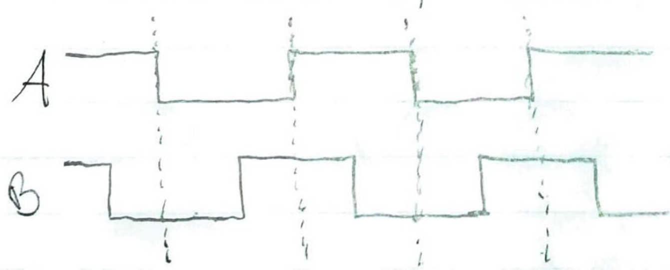
Microcontroller Applications
Serge Haddad
Midterm Review

Rotary Encoder HW

Q1) Rotary encoders can be found in applications such as industrial controls, robotics and rotating radar platforms.

Q2) When using a quadrature encoder, non-volatile memory is needed because the system will not remember the encoder's position when there is a power cycle (ON → OFF → ON). This is not the case with absolute encoders since the system will be able to know the encoder's position when there is a power cycle (ON→OFF→ON).

Q3) Reverse (CCW)



Q4) CPR Figure 9: $CPR = 4 \times PPR = 4 \times 48 = 192$
CPR Figure 10: $CPR = 4 \times PPR = 4 \times 16 = 64$

Interrupt-driven programming HW

Q1) An interrupt-driven program is a program where the operation of the program depends upon the occurrence of interrupts only!

Q2) When the interrupt-driven program is idle (no event to process), the CPU will wait until an interrupt is triggered so that the CPU can handle the interrupt. When idle, the CPU usually loops in a forever loop (while loop) until an interrupt is triggered.

Q3a) This is a simple condition checking to see if pin RB3 on PIC32 is "high" (meaning PORTBbits.RB3 = 1). This is being used as rising-edge detection.

b) This statement takes into account any wrap around from 0xFFFF to 0x0000 between any two readings. This statement fixes the issue with this operation: period = final - start, by adding of * 0x10000 to the operation.

c) The statement "start = final" is used as a timestamp so that the system can perform a period measurement.

d) "of = 0;" refers to the timer interrupt variable being reset to 0. This occurs when the change notification interrupt completes measuring a period on pin RB3 on PIC32.

(overflow variable)

Q4)

```
...
frequency = (100000/timePeriod) / 16;   // Motor frequency.
motor_rpm = frequency * 60;             // Motor RPM.
motor_rps = motor_rpm / 60;             // Motor RPS.
...
```

(motor PPR)

* 1Hz = 60rpm

Q5)

```
...
frequency = (100000/timePeriod) / 16;   // Motor frequency.
motor_rpm = frequency * 60;             // Motor RPM.
gearbox_rpm = motor_rpm / 10;           // Gearbox RPM.
...
```

(motor PPR)

(gearbox ratio)

Q6) #define LED LATBbits.RB7
extern int of;
int frequency;
while (1){
delay_us(ONE_SEC);
if (of >= 8){
LED = 1;
}
else{
frequency = 100000/timePeriod;
}

$$* \frac{5 \text{ seconds}}{(65'537 \times 10\mu s)} \cong 8$$

$$\frac{1}{100kHz}$$

Q7) ...
frequency = 1000000/timePeriod;
...

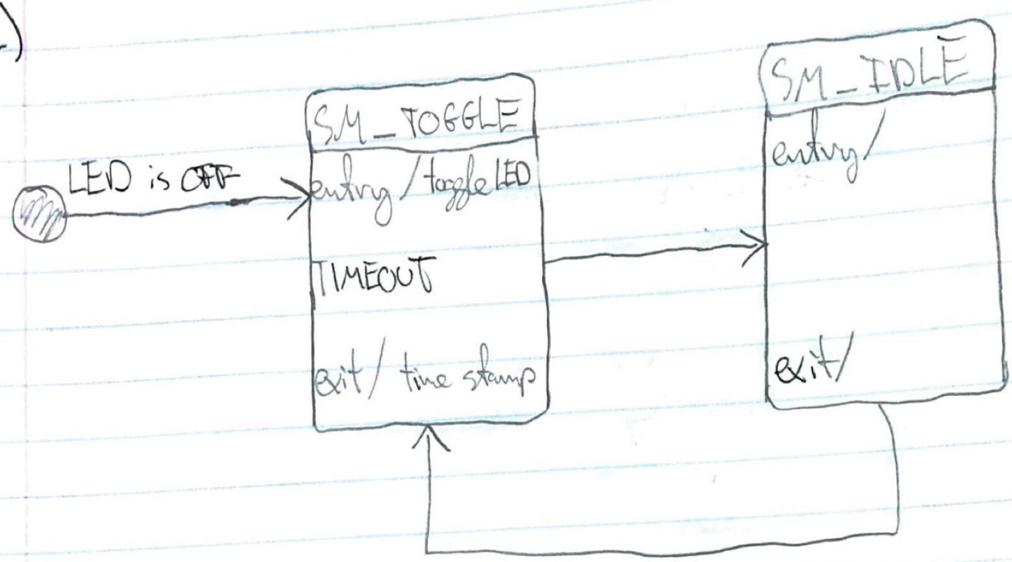
*8MHz / 8

Q8) There is an atomicity issue with the code in example 2. In this code, the global broadcast variable "timePeriod" gets accessed in the while loop, which could lead to data corruption. The solution to this problem is as follows:

```
...
CNIE = 0;                                   // Disables CN interrupt.
frequency = TICK_FREQUENCY/timePeriod;      // Calculates frequency.
CNIE = 1;                                   // Enables CN interrupt.
...
```

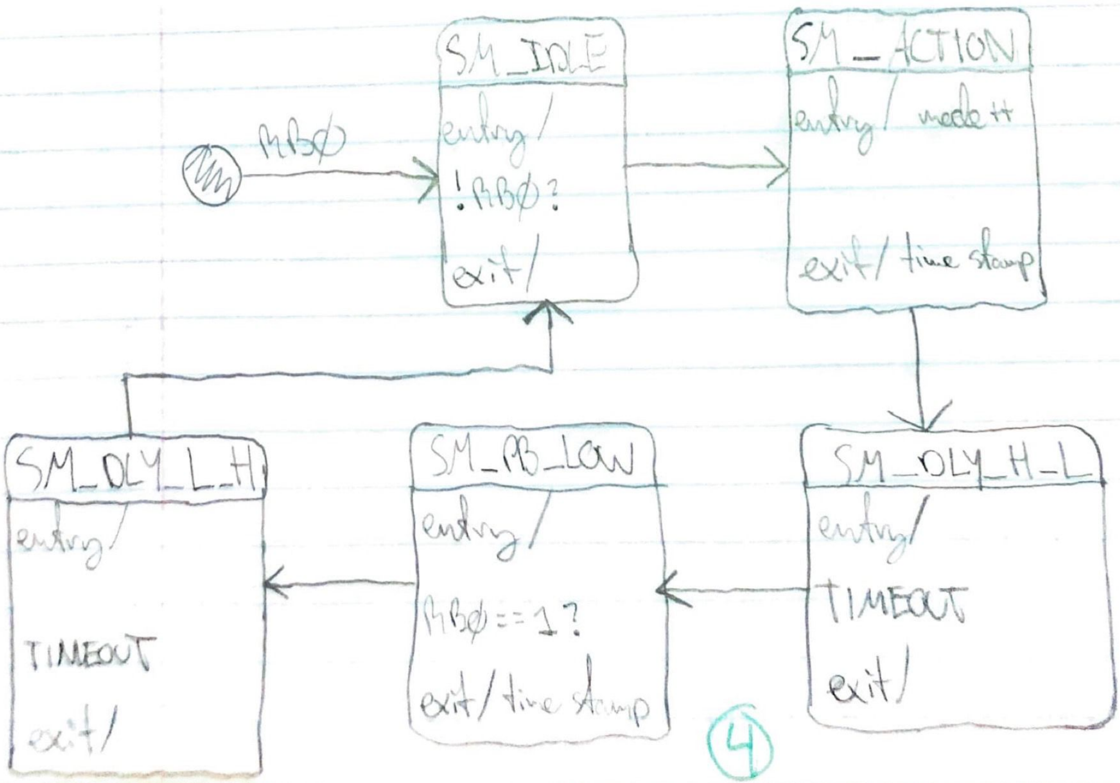
Intro to state machines and UML HW

Q1)



Q2) The line in bold characters refers to the equivalent of a while or for loop waiting for a certain amount of time to elapse in a "non-blocking" manner. While and for loops are "blocking" functions.

Q3)



Q4) When systems undergo a well-defined sequence of transitions, state machines should be considered.

Q5) A very common design that takes advantage of state machines are traffic light systems or alarm systems.

Q6) By implementing a design using state machines, code that is normally complex will be simpler and when issues arrise and debugging is needed, simple changes to the code can be done while debugging.

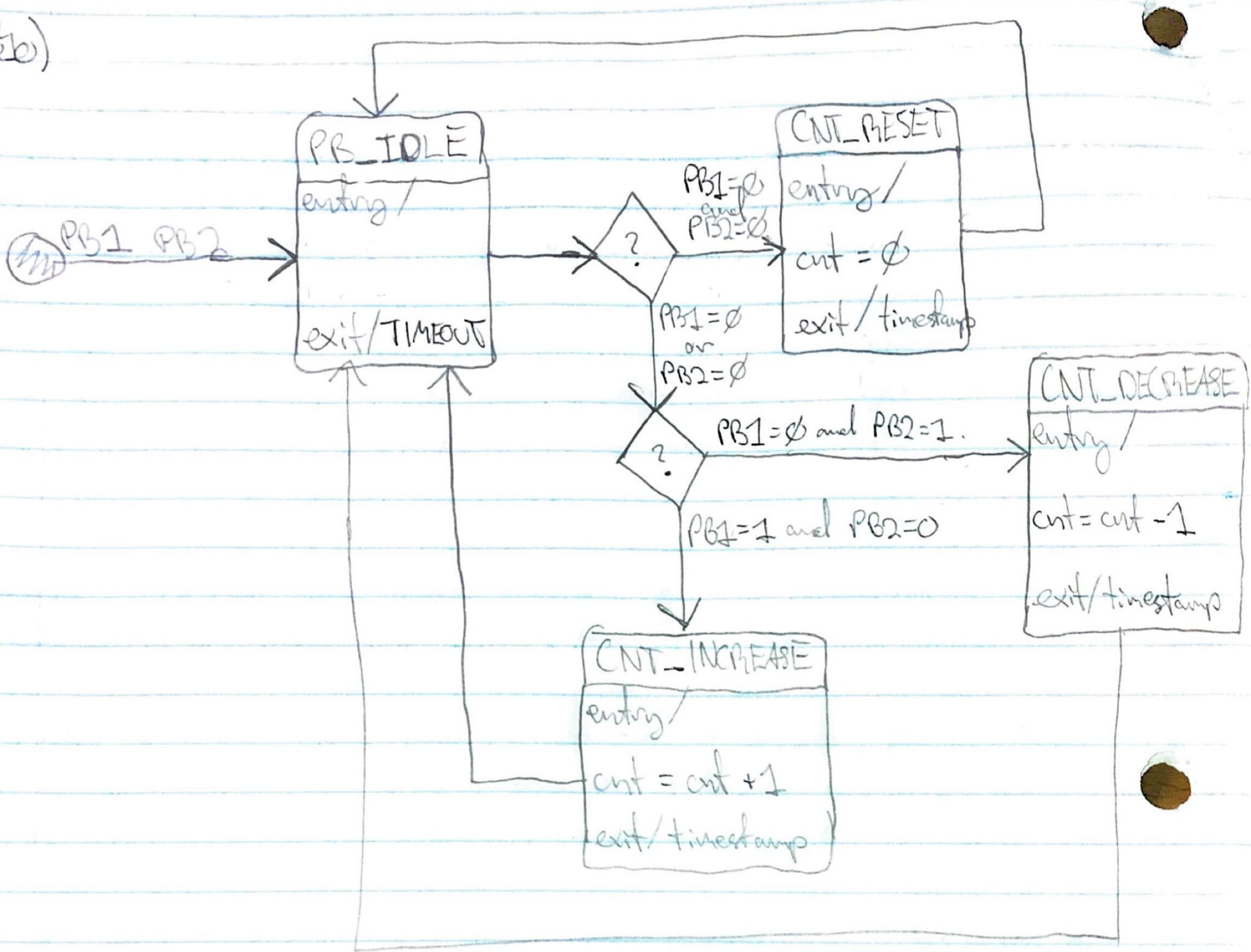
Q7) When "non-blocking" type functions are used, SM responsiveness improves since the microcontroller will be able to run multiple tasks in parallel.

Q8) Many SM tasks are able to run in parallel thanks to "break" statements. When "break" statements are used, many SM tasks are able to run in parallel.

Q9)

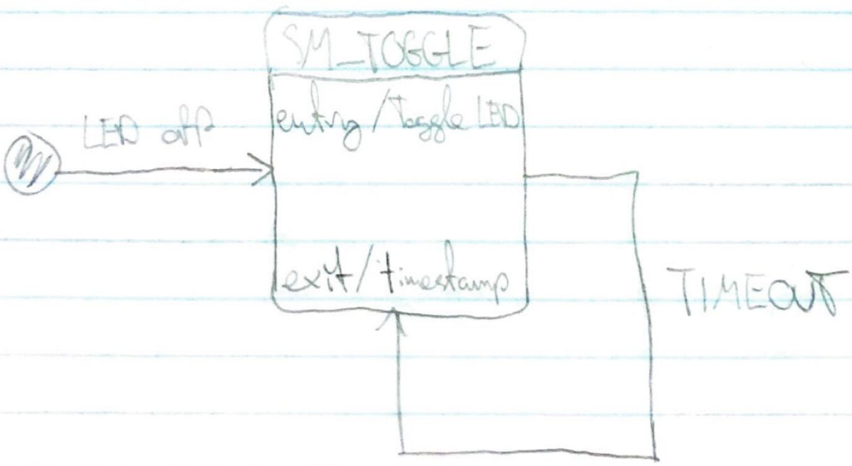
```
...
case 3: i = 7;
        state = 4;
        break;
case 4: if (i > 0) {
            function();
            i--;
        }
        else {
            state = 5;
        }
        break;
case 5:
...
```

Q10)



PB1 PB2
PB_IDLE
entry/
exit/TIMEOUT
?
PB1=Ø and PB2=Ø
CNT_RESET
entry/
cnt = Ø
exit/timestamp
PB1=Ø or PB2=Ø
?
PB1=Ø and PB2=1.
CNT_DECREASE
entry/
cnt = cnt -1
exit/timestamp
PB1=1 and PB2=0
CNT_INCREASE
entry/
cnt = cnt +1
exit/timestamp

Q11)



LED off
SM_TOGGLE
entry/Toggle LED
exit/timestamp
TIMEOUT

Leonardo Fusser (1946995) 18th of November 2021

Control Systems HW

Microcontroller Applications

Serge Hould

Q1)

- For the P-channel MOSFET Q2A, the gate must be driven LOW, and for the N-channel MOSFET Q1B, the gate must be driven LOW. For the P-channel MOSFET Q2B, the gate must be driven HIGH, and for the N-channel MOSFET Q1A, the gate must be driven HIGH. These conditions must be met in order for the motor to be turned ON and spinning in the CW direction. Working further backwards, the En signal should be HIGH and the Dir signal should be HIGH in order to satisfy the above conditions.

Q2.1)

- For the P-channel MOSFET Q2A, the gate must be driven HIGH, and for the N-channel MOSFET Q1B, the gate must be driven HIGH. For the P-channel MOSFET Q2B, the gate must be driven LOW, and for the N-channel MOSFET Q1A, the gate must be driven LOW. These conditions must be met in order for the motor to be turned ON and spinning in the CCW direction. Working further backwards, the En signal should be HIGH and the Dir signal should be LOW in order to satisfy the above conditions.

Q2.2)

- We could obtain a motor speed of 1’100 RPM (half of 2’200 RPM) by changing the En signal to a 50% PWM signal. With this, the amount of time the motor runs is split in half, which in turn reduces the speed of the motor by half.

Q3)

- To stop the motor completely, the En signal should be LOW. The Dir signal level does not matter here because it will never be able to turn the motor ON or OFF.

Q4)

- The effect of too much P-term in a PI controller results in large overshoots and oscillations being created in the output of the control system.

Q5)

- The effect of too much I-term in a PI controller results in an error (SSE) of almost near zero but many oscillations appear in the output of the control system.

Q6)

- The effect of too little I-term in a PI controller results in a large amount of error (SSE) being produced by the control system.

Q7)

- The effect of too little P-term in a PI controller also results in a large amount of error (SSE) being produced by the control system.

Q8)

- An ON/OFF controller is a special case of a P controller only when the Kp variable is set to a very high value.

Q9)

- A condition that could make a PID controller oscillate is when there is too much P-term or I-term that exists in the control system.

Q10)

- Some disadvantages of a pure P controller are that the deviation between the SP variable and PV variable cannot be large and that a pure P controller cannot handle sudden deviations between these two variables very well.

Q12)

- One of the main advantages of a PI controller is that it is able to reduce the amount of error (SSE) in its control system better than most other controllers do. This is also why it is one of the most widely used controllers. It also typically has faster response and settling times than what most other controllers have.

Q13)

- In a PID or PI control system, anti-windup acts as a “capping” feature where it is able to limit large overshoots from occurring in the control system’s output.

Q14)

a) Absolute overshoot value of around 9.

b) Settling time of around ~675mS.

c) SSE value of around 2.

Q15)

$$\frac{120\ PPR*10}{360} = \frac{1{,}200}{360} = \sim 3.33\ tics\ per\ degree$$

Q16)

```
...
if(PORTGbits.RG7){
      if(PORTGbits.RG6) {pos++;}
      else {pos--;}
else{
      if(PORTGbits.RG6) {pos--;}
      else {pos++;}
...
```

Q17)

$F = 250Hz$

$PWM = 35\%$

$T_{ON} = 35\%\ of\ F$

$T_{OFF} = 65\%\ of\ F$

$$T = \frac{1}{F} = \frac{1}{250Hz} = 4mS$$

$$T_{ON} = \frac{(PWM\ in\ \%)}{100} * T = \frac{35}{100} * 4mS = 1.4mS$$

$$T_{OFF} = T - T_{ON} = 4mS - 1.4mS = 2.6mS$$

- The ON time should be set to 3500 to make the Timer ISR output a 35% PWM signal. Refer to above calculations for exact timing values.

Q18)

- If the ON time was set to 2230 to make the motor spin in the CW direction, then the new ON time should be set to -2230 to make the motor spin in the CCW direction.

SPI Protocol HW

Microcontroller Applications

Serge Hould

Q1)

- Five wires are required in total for a SPI master. Two wires are for the MOSI and MISO pins between the SPI master and the individual SPI slaves and two wires are for the chip-select pins on the two individual SPI slaves. The fifth wire is for the SCK pins between the SPI master and the individual SPI slaves.

Q2)

- The SPI serial link is synchronous because SPI communication usually relies on a clock and chip-select signal.

Q3)

- In a SPI master-slave communication, the SPI master is the one who provides the clock signal to the SPI slaves connected to it.

Q4)

- Another name for the SDO master pin is MOSI (SDO on master connected to SDI on slave = MOSI).

Q5)

- Another name for the SDI slave pin is MOSI (SDO on master connected to SDI on slave = MOSI).

Q6)

- In a SPI master-salve communication, the SS (slave select or otherwise know as chip-select) line needs to be asserted low when the SPI master writes any data or reads any data from the connected individual SPI slave.

Q7)

- When the internal SPI module in PIC32 is configured as a SPI and has a connected SPI slave to the SDIx, SDOx, SS and SCKx pins, it behaves like a circular buffer. If writing data, the SPIxBUF gets loaded with data and is automatically transmitted via SPIxTXB to SPIxSR where the data (from MSB to LSB) gets clocked out on the SDOx pin (this is a shift register, typically 8-bits). When reading data, the SPIxSR gets clocked in with data from the SPI slave on the SDIx pin and gets automatically received by SPIxBUF through SPIxRXB. Note, when writing or reading data to and from SPI slaves, the SSx usually needs to be asserted low (depends on chip manufacturer).

Q8)

- If the clock line is low when the SS line is not asserted (in other words, when clock is low at idle), the clock polarity is 0.

Q9)

- When the clock polarity (CPOL) and the clock phase (CPHA) is 1, the data is sampled on the second clock edge (rising edge). If the clock polarity changed and the clock phase remained the same, the data would still be sampled on the second clock edge (but on falling edge).

Q10)

- The manufacturer decides the mode for SPI communication. This information is given in manufacturer datasheets.

Q11)

- The SPIRBF flag gets set automatically after receiving 8-bits (1-byte) of data.

Q12)

- The method of clearing the SPIRBF flag can be done by reading the data in SPIxBUF.

Q13)

```
...
/*Sets SPI1 baud rate to 1MegaBAUD (1MHz clock)*/
SPI1BRG = (SYS_REG / (2*1000000)) - 1;
...
/*Sets SPI1 to mode 3*/
SPI1CONbits.CKP = 1;  //Sets clock polarity (CPOL) to 1.
SPI1CONbits.CKE = 0;  //Sets clock edge (CPHA) to 1.
...
SS = 0;    //Enable transmission.

SPI1BUF = 0x83;       //Tells DS3234 to get ready to write day.
while(!SPI1STATbits.SPIRBF);     //Wait for TX to complete.
dummy = SPI1BUF;      //Clears SPIRBF flag.

SPI1BUF = 0x06;       //Tells DS3234 to set the day to 6.
while(!SPI1STATbits.SPIRBF);     //Wait for TX to complete.
dummy = SPI1BUF;      //Clears SPIRBF flag.

SS = 1;    //Disable transmission.
...
```

Q14)

Clock Signal	CPOL	CPHA	Mode
CLK1	0	0	0
CLK2	0	1	1
CLK3	1	0	2
CLK4	1	1	3

Embedded Systems Software Layers HW

Microcontroller Applications

Serge Hould

Q1)

Location	Layer Name
Top (high-level)	Application/Tasks
Middle	Driver
Bottom (low-level)	HAL

Q2)

- When the hardware is modified, the layer that is mostly affected is the HAL layer. This is because it contains all the registers specific to the hardware being used. In some cases, the driver layer might also get affected if the design is not well done.

Q3)

```
/*Super loop*/
while(1){
	VendingMachineTask(); -> 1
	MotorTask();          -> 2
}
```

1: application-task. This is because this function does not require low level programming.

2: application-task. This is because this function does not require low level programming.

Q4)

```
/****************** NVM library begins ***************************/

#define SELECT        0

#define DESELECT      1


//////////////////////////Driver & HAL layer/////////////////////// -> 2*

/* Writes a 16 bit value to a EEPROM */

void WriteNVM(int address, int data) {

      // Waits until any work in progress is completed

      CheckWIP();                   // checks WIP
      WriteEnable();                // sets the write enable latch

      // perform a 16 bit write sequence
      CS1(SELECT);                  // selects the Serial EEPROM
      WriteSPI2(SEE_WRITE);         // writes command
      WriteSPI2(address >> 8);      // address MSB first
      WriteSPI2(address & 0xfe);    // address LSB (word
      WriteSPI2(data & 0xff);       // sends byte
      CS1(DESELECT);                // deselects the Serial EEPROM

}



/* sets the write enable latch */

void WriteEnable(void) {

      CS1(SELECT);            // select the Serial EEPROM pin
      WriteSPI2(SEE_WEN);     // sets the latch
      CS1(DESELECT);          // deselect Serial EEPROM pin

}



/* Waits until any work in progress is completed */
void CheckWIP(void){
      while (ReadSR() & 0x1);
}
```

```
/* write 8-bit data to SPI */

int WriteSPI2(int data) {

      SPI2BUF = data;                  // writes buffer register

      while(!SPI2STATbits.SPIRBF);     // polls SPI buffer full bit

      return SPI2BUF;                  // returns buffer register

}




/* Chip Select Serial EEPROM#1 */
#define CSEE_LATD3    // cspin
void CS1(intsel){
      CSEE = sel;     // select/deselect cspin
}



/* Check the Serial EEPROM status register */
intReadSR(void) {
      int   i;
      CS1(SELECT);          // select the Serial EEPROM
      WriteSPI2(SEE_STAT);  // send Read Status command
      i= WriteSPI2(0);      // send dummy, read status
      CS1(DESELECT);        // deselect Serial EEPROM
      return i;             // return status
}
//////////////////////////////////////////////////////////////////////

/***************** NVM library ends ******************************/
```

2*: Driver and HAL. This is because the NVM library requires low level programming.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


int main(void){
      while(1){
            Temperature_log_task();     //Invokes the NVM library -> 1*
      }
}
```

*1: application-task. This is because this function does not require low level programming.