Problem 1.

A circular buffer with BUFFER_SIZE elements is defined as

```
volatile unsigned char buffer[BUFFER_SIZE - 1];
volatile unsigned char front = 0;
volatile unsigned char back = 0;
```

Two additional global variables are used as flags to indicate when the buffer is full and empty,

```
volatile unsigned char Buffer_Full;
volatile unsigned char Buffer_Empty;
```

Buffer_Full will be 1 if the buffer is full and 0 otherwise, Buffer_Empty will be 1 if the buffer is empty and 0 otherwise, Code C functions get and put to respectively get elements from and put elements in this buffer. Set and clear the flags as appropriate and avoid overwriting the buffer. If the buffer is full do not allow it to be overwritten. If it is empty return 0xFF. Show the definition of any additional variables needed.

Note that the buffer is actually of size BUFFER_SIZE - 1 not BUFFER_SIZE as a result of the its declaration being volatile unsigned char buffer[BUFFER_SIZE - 1];. With that being said, observe the following get and put functions:

```
#define FAIL 1
#define SUCCESS 0

unsigned char get() {
  if (Buffer_Empty)
    return 0xFF;
  char c = buffer[back++];
  back %= BUFFER_SIZE - 1;
  if (back == front)
    Buffer_Empty = 1;
  return c;
} // get()

int put(char c) {
  if (Buffer_Full)
    return FAIL;
  buffer[front++] = c;
  front %= BUFFER_SIZE - 1;
  if (front == back)
    Buffer_Full = 1;
  return SUCCESS;
} // put()
```

Problem 2.

The following code is used to "bit bang" an SPI on a `PIC24FJ64GA002` microcontroller. Assume the microcontroller has a 16 MHz instruction clock frequency FCY.

```
#define MOSI PORTBbits.RB1
#define MISO PORTBbits.RB2
#define SCK PORTBbits.RB0

unsigned char SPI_transfer(unsigned char data) {
    unsigned char counter;
    for(counter=8; counter; counter--) {
        if(data&0x80)
        MOSI = 1;
        else
        MOSI = 0;
        SCK = 0;
        data <<= 1;
        if(MISO)
        data |= 0x01;
        SCK = 1;
    } // for
    return(data);
} // SPI_transfer()
```

To answer (b) and (c) in the following you will need to use MPLAB X and the XC16 compiler.

(a) Recall that SPI operates in one of 4 modes depending on the clock polarity and the active edge of the clock used to latch the data. Which SPI mode does this function implement?

In this function the SPI clock idles at a high state, and is low when active (i.e. `CKP = 1`). Since the data is latched immediately following the transition from an idle to an active clock state (leading edge of clock cycle), and data is shifted to MOSI immediately after the transition from an active to idle clock state (trailing edge), it follows that the clock phase is 0 (`CKE = 1`). Hence, this function implements mode(1,0).

For more evidence that the clock phase is 0 and not 1 (despite the lecture notes claiming that this function implements mode(1,1)), notice that the out side–MOSI–only holds the data valid until the trailing edge of the current clock cycle (`SCK = 1`), after which it updates MOSI before the leading edge of the next clock cycle.

(b) Write a simple `main` function which uses this function, `SPI_transfer`, to send the ASCII character "M".

```
int main() {
    CLKDIVbits.RCDIV = 0;
    AD1PCFG = 0x9fff;
    TRISB = 0xffff;
    PORTB = 0xffff;
    TRISB &= 0xfffc; // set RB0, RB1 as outputs
    unsigned char response = SPI_transfer('M');
    return 0;
} // main()
```

Continued on next page ⟶

(c) Simulate your code using the MPLAB X simulator. Look at the output from the `SCK` and `MOSI` pins using the Logic Analyzer (Window > Simulator > Logic Analyzer). What do the waveforms look like? Infer the SPI clock frequency from the `SCK` output.

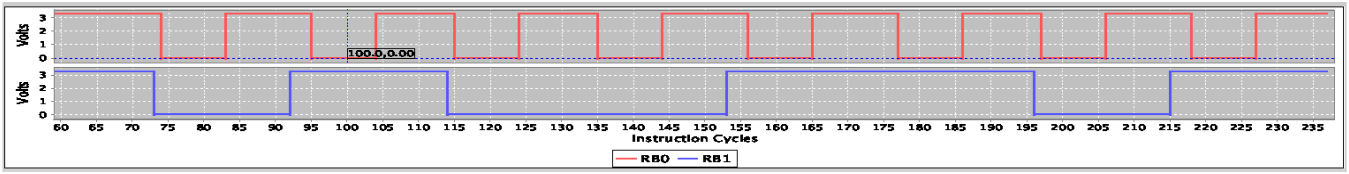

Figure 1: Waveforms from execution of `SPI_transfer('M')`.

The period of a single SPI clock cycle is between 20 and 21 MCU clock cycles depending on the whether `data & 0x80` evaluates to true or not. As such, the clock frequency is between

$(20\ cycles)\frac{62.5\ ns}{1\ cycle}\frac{1\ second}{10^9\ ns} = 1.25 \times 10^{-6}\ sec$ yielding a SPI clock frequency of $\frac{1}{1.25\times10^{-6}} = 800000\ Hz$

and

$(21\ cycles)\frac{62.5\ ns}{1\ cycle}\frac{1\ second}{10^9\ ns} = 1.3125 \times 10^{-6}\ sec$ yielding a SPI clock frequency of $\frac{1}{1.3125\times10^{-6}} = 761904.7619\ Hz$

Therefore, the SPI clock frequency is roughly 800 kHz.

Problem 3.

The Microchip 25LC256 is a 256 Kbit serial EEPROM. The following questions relate to reading data from this device with SPI2 on a PIC24FJ64GA002 . Assume SPI2 operates in 8-bit mode with a 5 MHz clock.

(a) For SPI there are 8 phasing modes which are controlled by the CKP, CKE, and SMP bits in the SPI2CON1 control register. What values should we use for these bits to read the 25LC256?

The 25LC256 EEPROM expects a low idle clock state and a high active clock state. As such, the clock polarity should be `CKP = 0`.

Data on the SI pin of the 25LC256 EEPROM is latched on the rising edge of the clock input. As such, the serial output (MOSI) should change on the transition from an active clock state to an idle clock state to ensure that the correct value is latched at the next rising edge. It follows that `CKE = 1` (i.e. Mode(0,0)).

According to the 25LC256 device documentation (DS01069B), input data should be sampled at the end of the data output time, meaning that `SMP = 1`.

In summary, `CKP = 0, CKE = 1, SMP = 1`.

(b) The 8-bit data is written to and read from the 8-bit buffer SPI2BUF. The Receive Buffer Full status bit, SPIRBF, is located in the SPI2STAT register. The active low chip select, $\overline{\text{CS}}$, for the 25LC256 is connected to pin `_RB14` which is configured as a digital output. Code a C function to read a 4-byte word from the 25LC256. Use the prototype: `int readEEPROM(int address)`.

Note that to return a 4-byte word, the function prototype must be changed to return a 32-bit `long int` instead of a 16-bit `int` type:

```
long int readEEPROM(int address) { // assumes that SPI2 is enabled
    _RB14 = 0; // pull chip select low to select 25LC256 device
    _SPI2IF = 0;
    // transmit 8 bit read instruction
    SPI2BUF = 0b00000011;
    while (_SPI2IF == 0);
    _SPI2IF = 0;
    // transmit 16-bit address to start read with MSB first
    int i;
    for (i = 1; i >= 0; i--) {
      SPI2BUF = address >> (i * 8);
      while (_SPI2IF == 0);
      _SPI2IF = 0;
    } // for
    // read 32 bits
    long int data = 0;
    for (i = 0; i < 4; i++) {
      SPI2BUF = 0;
      while (SPI2STATbits.SPIRBF == 0);
      data |= ((long int) SPI2BUF) << (8 * (3 - i));
    } // for
    // pull chip select high to finish read
    _RB14 = 1;
    return data;
} // readEEPROM()
```

Continued on next page ⟶

(c) With a 5 MHz SPI clock, SCK, how long does it take to read the 4-byte word from the EEPROM?

The minimum number of SPI clock cycles needed is $8 + 16 + 32 = 56$ cycles to first transmit the 8-bit read instruction, followed by the 16-bit address, and then finally to read the 32-bits of the 4-byte word from the EEPROM. Assuming a 5 MHz SPI clock, this will take:

$$56 \texttt{ cycles}\left(\frac{1 \texttt{ sec}}{5,000,000 \texttt{ cycles}}\right) = 1.12 \times 10^{-5} \texttt{ sec} = 11.2\mu\texttt{s}$$

Problem 4.

I2C is to be used to interface the microcontroller with a Microchip 24LC04 serial EEPROM. Assume a 16 MHz instruction clock.

(a) The I2C module is to operate with 7-bit address in master mode with a 400 KHz serial clock and no interrupts. Use IC1 and show the code to initialize this module in the PIC24FJ64GA002.

First, observe the following calculation used to find the desired value of `I2C1BRG`:

$$\texttt{I2C1BRG} = \left(\frac{\texttt{FCY}}{\texttt{FSCL}} - \frac{\texttt{FCY}}{10,000,000}\right) - 1 = \left(\frac{16,000,000}{400,000} - \frac{16,000,000}{10,000,000}\right) - 1 = 37.4 \approx 37$$

Therefore, the code needed to initialize IC1 is:

```
I2C1CON = 0x0000;
I2C1BRG = 37; // FSCL = 400 kHz
IFS1bits.MI2C1IF = 0;
I2C1CONbits.I2CEN = 1;
```

(b) Sketch the timing diagram for a read operation. How long does this take?

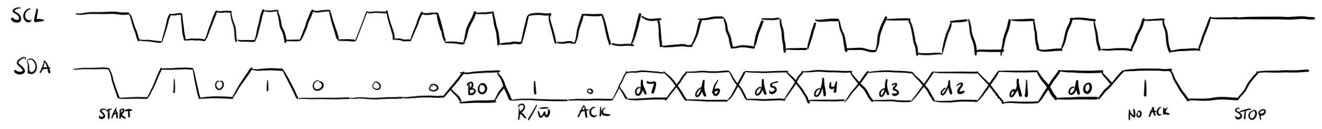

Figure 2: I2C Timing diagram for a Microchip 24LC04 serial EEPROM current address read operation.

In total, 20 bit times are required for the current address read operation. This amounts to $20 \cdot \frac{1}{400,000} = 0.00005\ \texttt{sec} = 50\ \mu s$

Note that for a random read, an additional $1 + 9 + 9 = 19$ bit times are needed for the start, control + ACK, and address + ACK to update the EEPROM's current internal address before starting the current address read operation. As such, a random read takes $(20 + 19) \cdot \frac{1}{400,000} = 0.0000975\ \texttt{sec} = 97.5\ \mu s$

(c) If we only read one byte at a time in the 24LC04 serial EEPROM how long would it take to read the entire EEPROM?

Assuming that we perform 1 random read to set the EEPROM's current internal address pointer to the first byte of memory, and performed all subsequent reads as separate current address read operations, it would take $97.5\ \mu s + 511 \times 50\ \mu s = 25647.5\ \mu s = 25.6475\ \texttt{ms}$ to read all 512 bytes of data from the 24LC04 serial EEPROM.

With a sequential read, however, only $19+1+9+512(9)+1 = 4638$ bit times are required to set the EEPROM's current address pointer to the first byte of memory, then to transfer the start, control + ACK, 512 bytes of (data + (N)ACK), and stop bit(s). This amounts to $4638 \cdot \frac{1}{400,000} = 0.011595\ \texttt{sec} = 11.595\ \texttt{ms}$