Cole Desrosiers

NYU Tandon School of Engineering

Take Home Quiz 2

ECE6483 – Real Time Embedded Systems

PROBLEM 1 (20 Points):

Suppose you are required to retrieve the temperature from the Analog Devices AD5933 chip, which is a chip used to make very precise impedance measurements. The interface is I2C and the relevant portions of the datasheet for this chip are attached. Also assume you have access to several I2C functions as follows:

a. `int Start_I2C()`
    i. Initializes all of the I2C hardware pins etc. Returns 1 if successful, 0 otherwise
b. `int I2C_Write_Byte(uint8_t data)`
    i. Writes 'data' on the I2C Bus. Returns 1 if slave ACK, 0 is slave NAK
c. `int I2C_Send_Start_Condition(uint8_t address, int isRead)`
    i. Send a start or restart condition, followed by 7-bit address, followed by 'isRead' bit, which is 0 if write, 1 if read.
    ii. Returns 1 if successful, 0 otherwise
d. `int I2C_Send_Stop_Condition()`
    i. returns 1 if successful, 0 otherwise
e. `int I2C_Request_Read(unint8_t *buffer, int numBytes)`
    i. reads 'numBytes' bytes off the bus, each followed by an master ACK except for the last byte, which if followed by a master NAK indicating an end to the read. Returns 1 if it gets all the bytes, 0 otherwise and buffer holds the bytes.

Using the datasheet provided, write a C code function `float GetTemperature()` required to get the temperature of the AD5933 chip. You can add any additional variables as long as you state what they are for and be sure to state any assumptions made.

Problem 1

Protocol: from ADS933 Datasheet

Block Read

In this operation, the master device reads a block of data from a slave device (see Figure 34). The start address for a block read must previously have been set by setting the address pointer.

1. The master device asserts a start condition on SDA.
2. The master sends the 7-bit slave address followed by the write bit (low).
3. The addressed slave device asserts an acknowledge on SDA.
4. The master sends a command code (1010 0001) that tells the slave device to expect a block read.
5. The slave asserts an acknowledge on SDA.
6. The master sends a byte-count data byte that tells the slave how many data bytes to expect.
7. The slave asserts an acknowledge on SDA.
8. The master asserts a repeat start condition on SDA. This is required to set the read bit high.
9. The master sends the 7-bit slave address followed by the read bit (high).
10. The slave asserts an acknowledge on SDA.
11. The master receives the data bytes.
12. The master asserts an acknowledge on SDA after each data byte.
13. A no acknowledge is generated after the last byte to signal the end of the read.
14. The master asserts a stop condition on SDA to end the transaction.

} 0x92. Registers at addresses 0x92 and 0x93 hold the temperature data. From the datasheet, we know that this value (the start address for the block read) is established prior to the communication. Therefore, I will assume that the address pointer is pointing to 0x92.

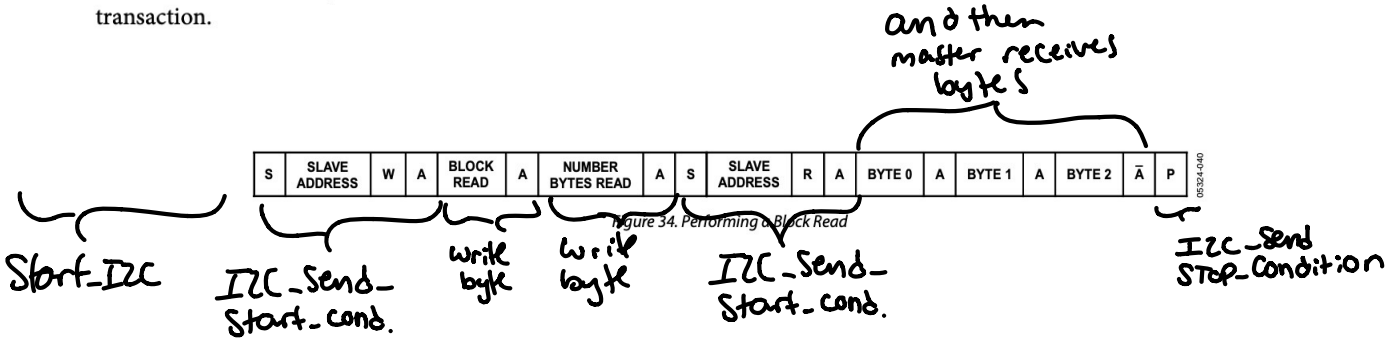


Figure 34. Performing a Block Read

```
float *buffer = 0x92
```
— set prior to communicaton

```
Float getTemperature()
{
  int Start_I2C()
  if(R0==0)
  {
    return -1;
  }else{
    int I2C_Send_Start_Condition(0001101, 0)
    if(R0==0)
    {
      return -1;
    }else{
      int I2C_Write_Byte(10100001)
      if(R0==0)
      {
        return -1;
      }else{
        int I2C_Write_Byte(00000010)
        if(R0==0)
        {
          return -1;
        }else{
          int I2C_Send_StartCondition(0001101, 1)
          if(R0==0)
          {
            return -1;
          }else{
```

Check for success, if return address (R0) holds '0', there is an error and the function ends. If R0 holds '1', continue transaction.

Check

write

0x0D (Slave Address)

Check

Command code

Check

# of bytes we want to read (2)

Check

Read

0x0D (Slave Address)

Here, the master should read two bytes, sending an ACK after the first and a NACK after the last byte has been transmitted. Because the number of bytes and *buffer have already been established in the protocol, the "int I2C_Request_Read()" seems a little redundant. Either way, I will use it if it's the only read function we have to use.

```
Int I2C_Request_Read(*buffer, 00000010)
if(R0==0)
{
  return -1;
}else{
    int I2C_Send_Stop_Condition()
    if(R0==0)
    {
      return -1;
    }
```

two bytes

established prior to transaction

check

check

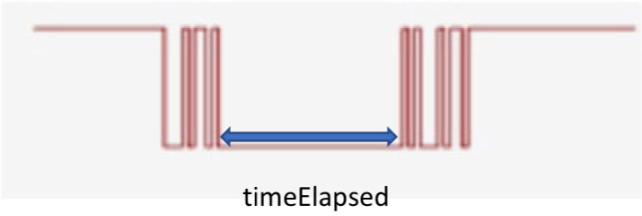
```
} //else1
  } //else2
    } //else3
      } //else4
        } //else5
          } //else6

return *buffer

} //end function
```

PROBLEM 2 (20 Points):

The requirement for this question is to use interrupts to measure the time elapsed while a single button (GPIO) is being pressed. The GPIO pin is pulled to a high state when the button is not depressed. In this application, the button is mechanical and experiences significant bouncing. The goal is to measure the contiguous time (in ms) between when the button is pressed and when the button is released, excluding the bouncing. See diagram below.



timeElapsed

You have already setup 3 interrupt handlers. One handler is the ISR that handles Timer0's overflow. That timer is set up with a prescaler of 32, a counter TOP value of 250, and is set up to reset to 0 and count up to TOP. The CLK is running at 16MHz. Here is the handler definition:

reset ms

```
void  Timer0_OV_Handler()
```

- does not return anything

The other 2 handlers are for the pin that is connected to the button. One handles the RISING transition, and the other handles the FALLING transition. Here are the definitions:

```
void pinRising_Handler()
```

```
void pinFalling_Handler()
```

Write the required code snippits (in each handler) that assigns the variable `timeElapsed` to the specified time in the timing diagram. You may add any additional variables, but **be sure to state any assumptions.**

- GPIO port an pin were unspecified, so I just used 'x' for port and 'n' for pin

Pseudo Code

The plan is to take note of the first falling edge and when it is triggered again. If the time difference between the two falling edges is greater than _____, then we can assume that this is 'stable' and we can start the timer.

void Timer0_OV

If falling edge is detected and overflow detected (a delay), start timer. Stop when rising edge detected

PinRising()

- detect rising edge (when GPIO is pulled high)

Pin falling

- detect falling edge (when GPIO is pulled low)

```
#include <mbed.h>
```

Global variables

```
volatile int timeElapsed = 0 ;
volatile int pressed = 0;                    // not pressed = 0, pressed = 1
```

```
void Timer_ov_Handler()
{
   if (Pressed == 1 && TIM0->SR & 0x1)       // If overflow bit detected
   {
      TIM0.Start();
   }
   if (pressed == 0)
   {
      TIM0.Stop();
   }
   timeElapsed = TIM0.Read();
}
```

```
void Pinfalling_Handler()
{
   if(Pressed == 0)     // Not pressed
   {
     if(GPIOx->IDR(0<<n)   //pulled low(pressed)
     {
       Pressed = 1;
     }
   }
}
```

```
void PinRising_Handler()
{
   if(Pressed == 1)     // button is pressed
   {
     if(GPIOx->IDR(1<<n)   //pulled high (not pressed)
     {
       Pressed = 0;
     }
   }
}
```

PROBLEM 3 (20 Points):

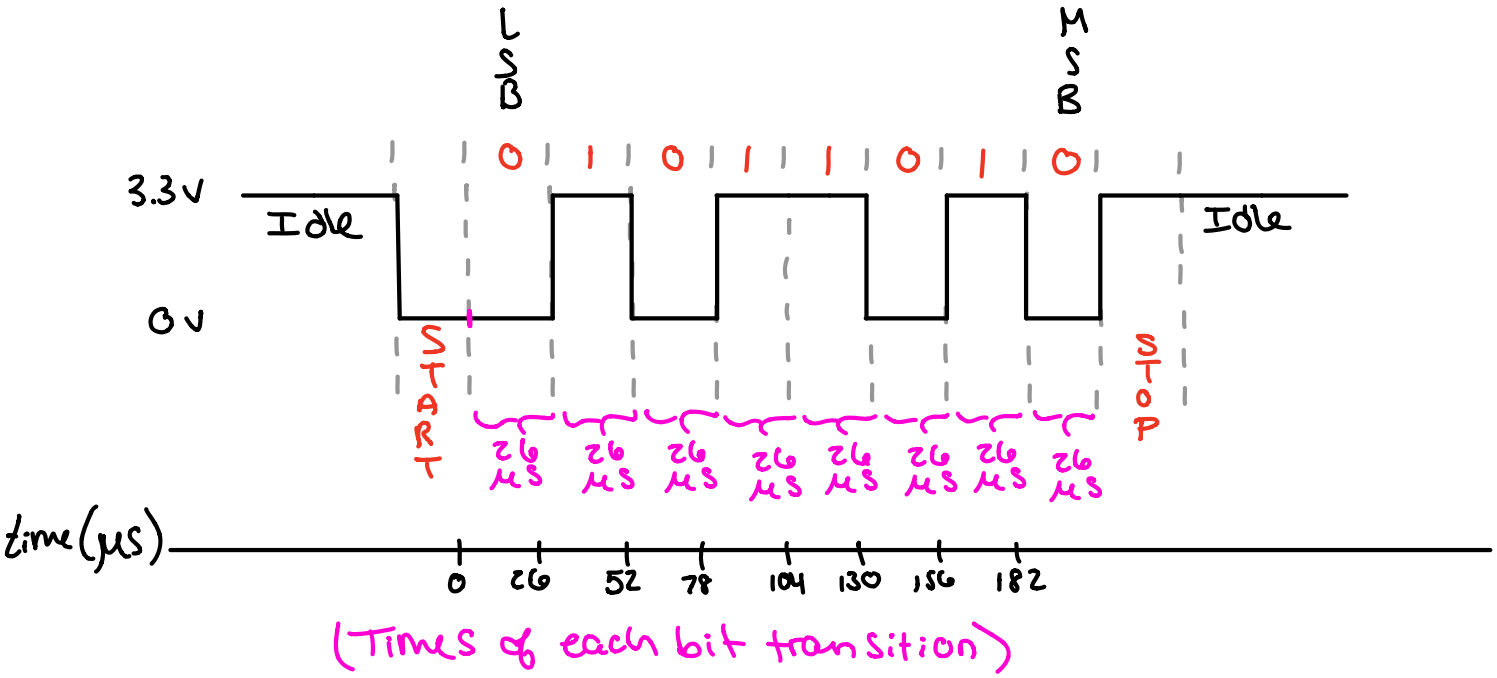
Suppose you use UART to send an ASCII `Z' (0b01011010) using 8N1 (one start bit, 8 data bits, no parity bit, one stop bit), at 38400 baud.

a. Draw a timing diagram for this communication. Assume bit transitions are instantaneous. Make sure to label the time axis (label the time of each bit transition).
b. Suppose the receiver is operating at 50000 baud, and is sampling bits at the middle of each bit period (i.e. at 10 s, 30 s, 50 s, etc.). On your timing diagram from the previous part, mark the times at which the receiver would sample the incoming waveform.

rate mismatch in this example?

a.)

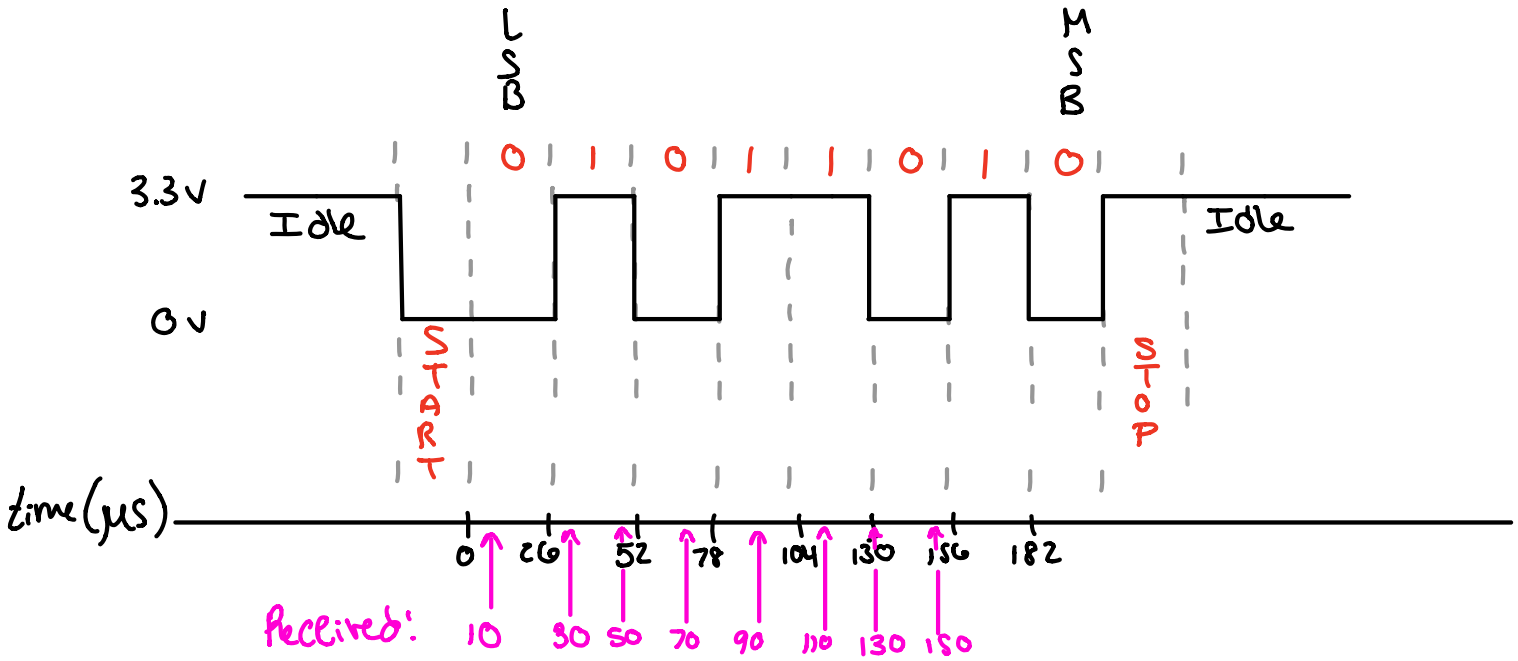
$$\text{Bit period} = \frac{1}{\text{Baud rate}} = \frac{1}{38400} = 0.000026\ \text{s} = 26\ \mu\text{s}$$



(Times of each bit transition)

b.)

Receiver Bit period $= \frac{1}{\text{Baud rate}} = \frac{1}{50000} = 0.000020\ s = 20\ \mu s$



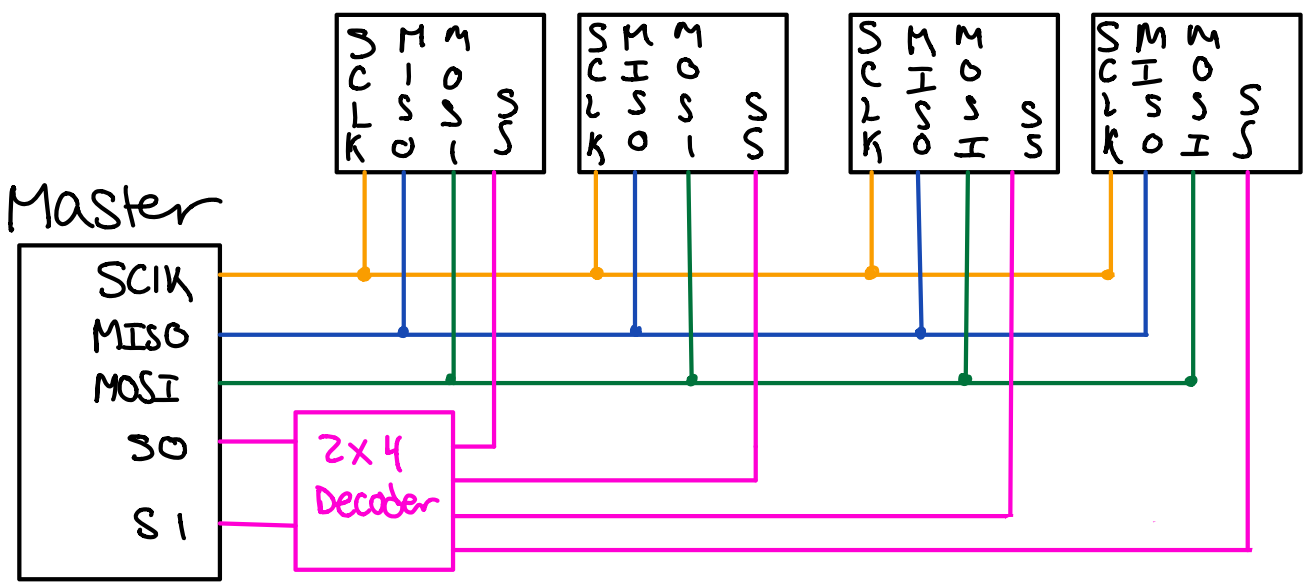
c.) The receiver receives '01101100'

There is no indication of a baud rate mismatch because we did not make use of a parity bit(s) in this example

PROBLEM 4 (10 Points):

Suppose you would like to communicate using 1 master and 4 slaves using SPI. Propose a design that will allow complete duplex communication between the master and all 4 slaves using only 5 GPIO lines. You may propose additional hardware if necessary.

The Clk line is necessary for any Synchronous Communications so we will have a clk. To support full duplex, we need 2 communication wires (MISO & MOSI). These are 3 necessary GPIO Pins. So now, if we only have access to 5 GPIO pins, we only have 2 left for the slave select. If we use a 2x4 binary decoder, we only need 2 GPIO pins to select from the 4 slaves:



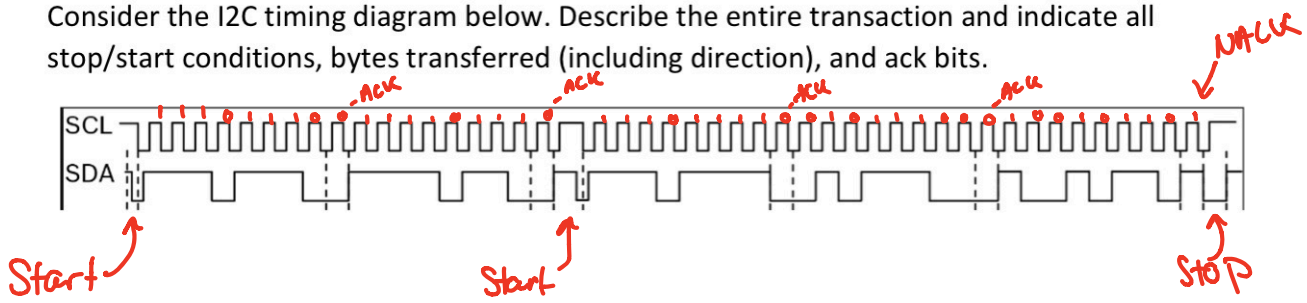
S0	S1	S0	S1	S2	S3
0	0	✓	X	X	X
0	1	X	✓	X	X
1	0	X	X	✓	X
1	1	X	X	X	✓

✓ = Active

X = Not Active

PROBLEM 5 (20 Points): (bits loaded on rising edge, read on falling)

Consider the I2C timing diagram below. Describe the entire transaction and indicate all stop/start conditions, bytes transferred (including direction), and ack bits.



We start with a start condition, where the master leaves SCL high and pulls SDA low. Next, the master sends the 7-bit address the slave (1110111) followed by a 0, indicating a write condition. Then, the slave with that address sends an ACK. Next, the master writes a byte of data (11110111) - this is typically the 8-bit address of the register the master wants to read/write from. Then, the slave sends an ACK. Next, there is another Start condition, where the Clk is high and the data line is pulled low. The master then restates the 7-bit address of the slave it wants to communicate with (1110111), followed by a '1', indicating the master wants to read data from the slave at that address. The slave then sends an ACK. Next, the master reads the byte (01011100) from the slave and the master sends an ACK indicating it recieved the byte. There is no Stop Condition, so the master will read another byte from the slave, which is (10010110). This byte is followed by a '1', indicating a NACK from the Master. This means here, the NACK signifies the end of the read. Finally, we receive a stop condition, where SCL line is high and the SDA line is pulled high. This indicates the end of transaction.

PROBLEM 6 (20 Points):

Please evaluate (T or F) and explain the following statements:

F a. An interrupt is always a high priority, urgent task.
T b. Using interrupts is always faster than polling.
T c. System latency is always larger than interrupt latency
T d. Global variables used within an ISR should always be declared volatile.
T e. The interrupt vector table must be placed in a specific location in memory.
F f. An ISR can return a value and take arguments
T g. It is OK for an ISR to safely access the SPI bus that has multiple slaves.

a.) False, there are multiple priority levels for interrupts and interrupts can also be set to a low priority, not requiring urgent accomodation.

b.) True, in the polling method, the microcontroller constantly checks the status of whatever register it needs to, waiting for an update so it can perform a given task. Performance wise, this hardware implementation is not very efficient. In comparison, interrupts dont continuously check on the status of a register, they wait for a certain event to occur and then the microcontroller stops executing and jumps to run the interrupt code. This is much better in performance compared to polling.

c.) True, System latency includes the interrupt latency. Interrupt latency is defined as the time from the inital IRQ to the start of the ISR. This includes the time it takes to finish the intrustion that was being executed when the interrupt was requested, the interrupt acknowledgement, and the interrupt sequence. The system latency includes all of this in addition to the time it takes to execute the ISR, return from the interrupt, and fetch and execute the next instruction.

d.) True, we should always declare global variables used in an ISR as volatile to make sure they are updated correctly and make sure that any changes made to these variables by the processor are accounted for.

e.) True, the interrupt vector table needs to be placed in a specific memory location because all of the interrupts are sequencially enumerated. This is important because the vectors are addresses that communicate to a given handler where to locate the ISR for that interrupt. The enumerations identify which interrupt occured.

f.) False, ISR's cannot return anything (they are void) nor can they take in any parameters. It cannot return anything because there is no caller in the code to read the return value from the ISR. In addition, we don't call the ISR and therefore the ISR has no parameters to take in.

g.) True, the ISR can safely access the SPI bus that has multiple slaves. The ISR is implemented independently from the SPI communication.