EEC172 **Midterm** **Winter 2014**

This examination is closed book, closed notes, closed calculator

Show all your work on these pages!

Excerpts from the UC Davis *Code of Academic Conduct*:

1. Each student should act with personal honesty at all times.
2. Each student should act with fairness to others in the class. This means, for example, that when taking an examination, students should not seek an unfair advantage over classmates through cheating or other dishonest behavior.
3. Students should take group as well as individual responsibility for honorable behavior.

I understand the honor code and agree to be bound by it.

Signature _____

NAME (print) SOLUTION

Student ID _____

PROBLEM	MAXIMUM (Points)	SCORE
1	30	
2	20	
TOTAL	50	

I. SysTick, GPIO programming

1. Answer the following questions regarding SysTick

   a) How many bits wide is the SysTick counter? (1) 24 bits

   b) How does the SysTick Reload Value register (STRELOAD) function during SysTick operation? How is this register value set using StellarisWare? (2)

When the SysTick counter reaches 0, it is automatically reloaded with the value in STRELOAD. STRELOAD's value is set using the function SysTickPeriodSet(unsigned long ulPeriod).

   c) What happens when 0xFF is written to the SysTick Current Value (STCURRENT) using the code: NVIC_ST_CURRENT_R = 0xFF; Be specific. (1)

A write of any value to STCURRENT clears the counter to 0. Then the STRELOAD value is automatically loaded into the counter.

   d) Assuming we are not using an external reference clock, how does the SysTick counter clock frequency relate to the Stellaris system clock frequency? (1)

The SysTick counter operates using the system clock directly with no prescaling. Thus, the frequencies are the same.

   e) Write a **complete**, short program that uses SysTick to accurately measure the number of system clock cycles to execute the instruction: UARTprintf("Hello world\n");

   Set the system clock to be 4 MHz. Your program should initialize the system clock and SysTick appropriately, make the measurement and store the result in a variable. (You do not need to show header file #includes. You may **NOT** use interrupts.) (10)

```
main(){
unsigned long dly, X;
SysCtlClockSet (SYSCTL_SYSDIV_50 | SYSCTL_USE_PLL
                | SYSCTL_OSC_MAIN | SYSCTL_XTAL_8MHZ);
SysTickPeriodSet (0xFFFFFF);        // max period
SysTickEnable();
NVIC_ST_CURRENT_R = 0x00;           // force period load
UARTprintf("Hello world\n");
X = SysTickValueGet();              // read counter
dly = 0xFFFFFF - X;                 // delay time
}
```

2. Assume that PE1 (connected to an active low switch) has been initialized as an input and PF2 (connected to an active high LED) has been initialized as an output. Using StellarisWare API functions shown on the last page, write the following code fragments.

   a) Assume that all necessary initializations have been done. Write a code fragment that does just the following.

      - Polls PE1 until it becomes low (when the switch is pressed)
      - Makes PF2 output high, turning on the LED

   Show the declaration of any variables that you use. (5)

```
long status;

do {
  status = GPIOPinRead(GPIO_PORTE_BASE, GPIO_PIN_1) & 0xFF;
} while (status != 0);
GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, GPIO_PIN_2);
```

b) Assume that **all** necessary initializations have been done. Write an interrupt handler for PE1 falling edge interrupts. Whenever the switch is pressed, your interrupt handler will run. The interrupt handler should count the number of interrupts (switch presses) that occur and toggle PF2 each time the handler executes. Show the declaration of any variables that you use. (10)

```
volatile int IntCount;          // global. Assume initialized
                                // to 0.

void SwitchHandler(void) {
  GPIOPinIntClear(GPIO_PORTE_BASE, GPIO_PIN_1); //clear int
  IntCount++;
  if (IntCount & 0x1)           // test bit 0
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, GPIO_PIN_2);
  else
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
}
```

II. Short answers

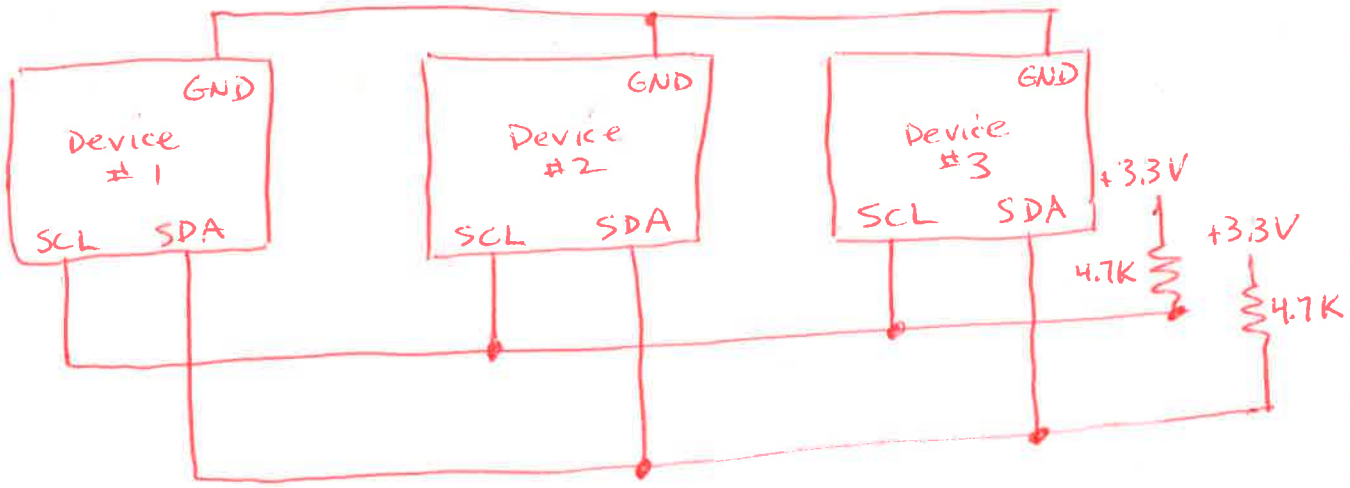
1. Explain how bit-stuffing works in the CAN protocol. Why is it needed? (3)

After sending 5 identical bits, the sender stuffs a dummy complement bit into the data stream. The receiver discards the dummy bit after receiving 5 identical bits. Bit stuffing is needed to enable the sender and receivers to maintain synchronization.

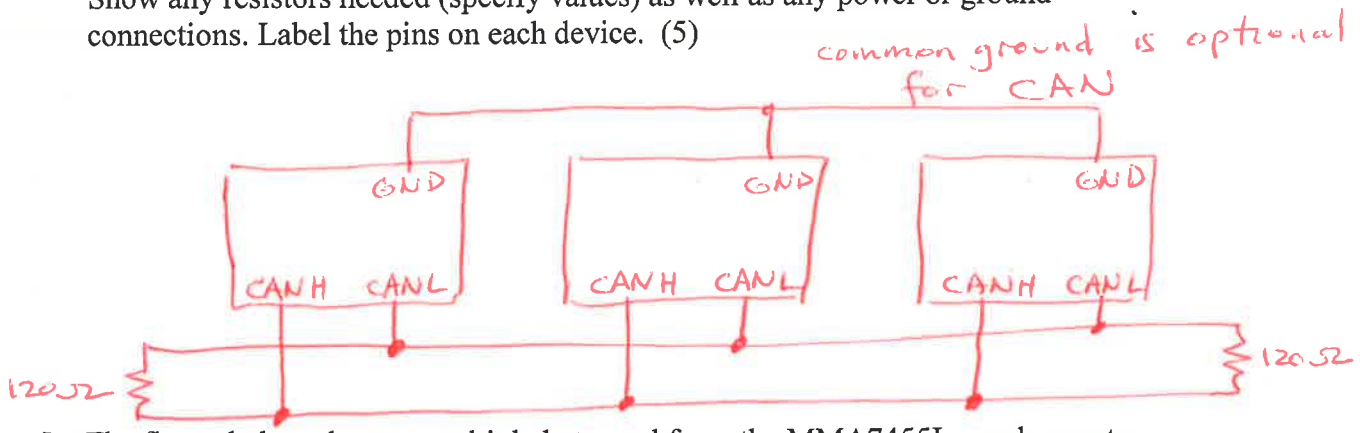
2. For a standard analog servo operating at a 50 Hz update rate, a pulse width of 1.0 ms moves the servo to the neutral (centered) position. If the update rate is changed to 25 Hz, what pulse width is needed to move the servo to the neutral position? Justify your answer. (2)

The pulse width determines the servo position, independent of the update rate. Thus, a pulse width of 1.0 ms is needed to center the servo at the new update rate.

3. Draw a schematic to show how three devices can be connected over the I2C bus. Show any resistors needed (specify values) as well as any power or ground connections. Label the pins on each device. (5)



4. Draw a schematic to show how three devices can be connected over the CAN bus. Show any resistors needed (specify values) as well as any power or ground connections. Label the pins on each device. (5)



5. The figure below shows a multiple byte read from the MMA7455L accelerometer over the I2C bus. To implement this transfer using StellarisWare, you would need to use the API function I2CMasterControl(I2C0_MASTER_BASE, *ulCMD*), where the various *ulCMD* options are listed on the last page.

   A sequence of I2CMasterControl function calls is required to implement this transfer. List the *ulCMD* options needed for each step in the sequence and show which part of the transfer (in the figure below) corresponds to each I2CMaster Control function call. {Note: other function calls would also be required to fully implement this transfer, but you do not need to show the complete code. Just indicate which I2CMasterControl() calls are needed.} (5)

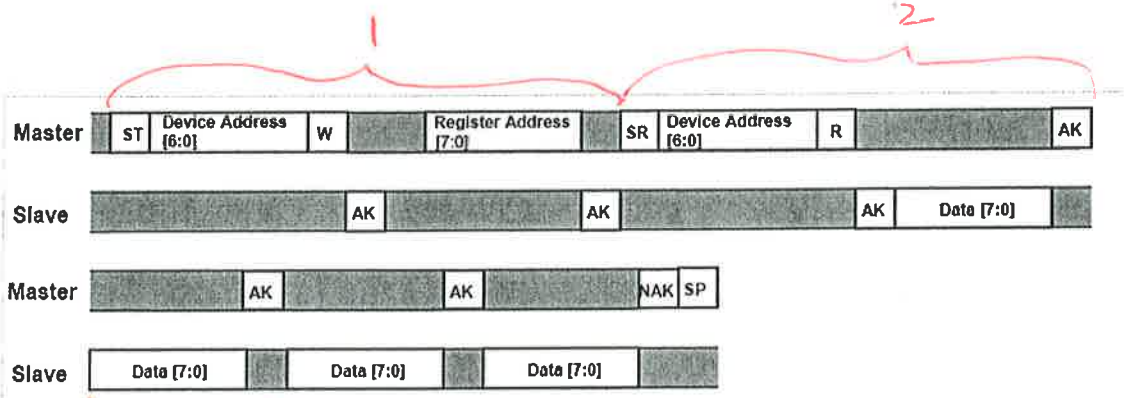


Figure 8. Multiple Bytes Read - The Master is reading multiple sequential registers from the MMA7455L

1. I2C_MASTER_CMD_BURST_SEND_START
2. I2C_MASTER_CMD_BURST_RECEIVE_START
3. I2C_MASTER_CMD_BURST_RECEIVE_CONT
4. I2C_MASTER_CMD_BURST_RECEIVE_CONT
5. I2C_MASTER_CMD_BURST_RECEIVE_FINISH

Useful StellarisWare API Function Prototypes

void SysCtlClockSet(unsigned long ulConfig)

ulConfig options: SYSCTL_SYSDIV_*n* (n=1 to 64),
SYSCTL_USE_PLL or SYSCTL_USE_OSC
SYSCTL_OSC_MAIN,
SYSCTL_XTAL_*x*MHZ

unsigned long SysCtlClockGet(void)

---

void SysTickEnable(void)

void SysTickIntEnable(void)

void SysTickPeriodSet(unsigned long ulPeriod)

unsigned long SysTickValueGet(void)

---

long GPIOPinRead(unsigned long ulPort, unsigned char ucPins)

void GPIOPinWrite(unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)

void GPIOPinIntClear(unsigned long ulPort, unsigned char ucPins)

ulPort options: GPIO_PORT*x*_BASE (x = port name)
GPIO_PIN_*n* (n = 0 to 7)

---

void I2CMasterControl(unsigned long ulBase, unsigned long ulCmd)

ulBase options: I2C0_MASTER_BASE

ulCmd options: I2C_MASTER_CMD_SINGLE_SEND
I2C_MASTER_CMD_SINGLE_RECEIVE
I2C_MASTER_CMD_BURST_SEND_START
I2C_MASTER_CMD_BURST_SEND_CONT
I2C_MASTER_CMD_BURST_SEND_FINISH
I2C_MASTER_CMD_BURST_RECEIVE_START
I2C_MASTER_CMD_BURST_RECEIVE_CONT
I2C_MASTER_CMD_BURST_RECEIVE_FINISH