Karan Vora (kv2154)
Real-Time Embedded Systems Quiz 2

Problem 1):

```
uint32_t BlockWritePortA(uint16_t PinVals)
{
		uint32_t TempVariable;
		GPIOA -> MODER |= 0x55555555;
		GPIOA -> OTYPER |= 0x0000;
		GPIOA -> PUPDR |= 0x00000000;
		TempVariable = GPIOA -> ODR;
		return(TempVariable);
}
```

==

Problem 2):

Solution a):
An interrupt is not always an urgent, high-priority task. For example, an asychnronous input from a peripheral that doesn't need an immediate response might still be interrupt-driven.

Solution b):
Using interrupts is not always faster than polling. This tradeoff depends on the interrupt latency of the system, the frequency of the event you are polling/interrupting from, and possibly other factors.

Solution c):
System latency is always larger than interrupt latency (It is at least as large as the interrupt latency). System latency is the interrupt latency plus the maximum amount of time that interrupts might be disabled.

Solution d):
Global variables used within an ISR should be declared volatile. This is true; since ISRs are triggered asynchronously, the compiler cannot predict when they will occur (and so might "optimize" out operations involving these variables), and we always want the freshest data in an ISR (declaring as volatile requires the compiler to load the value again every time it is used, instead of caching it).

Solution e):
As interrupts vector is the memory location at an interrupt handler, which prioritizes interrupts and saves them in a queue if more than one interrupts are waiting to be handled. As interrupts is a signal from a device attached to a computer or from a program with in the computer that tells the OS to stop and decide what to do next, when an interrupt is generated the OS saves its execution state my means of a context switch, a procedure that a computer processor follows to change from one task to another while ensuring that the tasks do not conflict. Once the OS has saved the execution state, it starts to execute the interrupts handler at the interrupt vector. Interrupts vector is the memory address at an interrupt handler. Memory is synonym to RAM, to yes interrupts vector is stored in ram if a device wants to register an interrupt handler function, you need to call appropriate OS calls and it would create an entry in the interrupt vector table. This entry would point to whenever the interrupt handler function

resides in memory/RAM. It's the OS that holds the responsibility to manage the interrupts vector table. Thus whenever that specific interrupt occurs, the interrupt handler function will be called.

Solution f):
An ISR, or interrupt service routine, can return a value to indicate the status of the interrupt, and can take arguments to specify the data associated with the interrupt. The ISR can also perform other tasks, such as handling I/O or communication, but these are not required. In most cases, the ISR will simply clear the interrupt flag, return the value from the interrupt service routine, and continue execution at the next instruction. An ISR can return a value to indicate the status of the interrupt, and can take arguments to specify the data associated with the interrupt. The ISR can also perform other tasks, such as handling I/O or communication, but these are not required. In most cases, the ISR will simply clear the interrupt flag, return the value from the interrupt service routine, and continue execution at the next instruction.

Solution g):
Yes, it is okay for an ISR to safely access the SPI bus that has multiple slaves. There are a few things to keep in mind when doing this, however. First, make sure that the SPI bus is properly configured for the slaves that are attached to it. Second, make sure that the ISR is configured to properly handle interrupts the from various the signals slaves. that Finally, will make be sure traveling that on the the ISR bus. is Finally, properly make configured sure to that handle the errors ISR that is may able occur to on handle the any SPI possible bus. data All corruption of that these might issues occur can on be the addressed bus. by Overall, following this proper is protocol a and safe configuration practice guidelines. and Overall, should it not is cause safe any for problems.

Solution h):
n the startup file, we find that all of the interrupt handlers are initially mapped to a default interrupt handler with a "weak alias." A comment in the startup file explains: Provide weak aliases for each Exception handler to the Default_Handler. As they are weak aliases, any function with the same name will override this definition. The default handler just executes an infinite loop preserving the system state for examination by a debugger

Solution i):
From the data sheet, pin 4 and 5 can fire external interrupts.

==

Problem 3):

Solution to problem 3 is attached in the end of the PDF

==

Problem 4):

```
#include <ad5933.h>
#include <stdio.h>
#define BUS_ADDRESS (0x0D)
#define CONTROL_REGISTER (0x80)
#define STATUS_REGISTER (0x8F)
```

```
#define REGISTER1 (0x92)
#define REGISTER2 (0x93)
#define CONTROL_TMEASURE (0b10010000)
#define STATUS_TVALID (0x01)

float getTemperature()
{
    uint8_t *bufferArray, tempArray[2];
    int tempVal_int = 0;
    Start_I2C();
    I2C_Send_Start_Condition(BUS_ADDRESS, 1);
    I2C_Send_Start_Condition(CONTROL_REGISTER, 1);
    I2C_Write_Byte(CONTROL_TMEASURE);
    I2C_Send_Stop_Condition();
    I2C_Send_Start_Condtion(STATUS_REGISTER, 0);
    I2C_RequestRead(*bufferArray, 1);

    if(*bufferArray == STATUS_TVALID)
    {
        I2C_Send_Start_Condition(0x90u, 1);
        I2C_Send_Start_Condition();
        I2C_Send_Start_Condition(BUS_ADDRESS, 0);
        I2C_Send_Start_Condition(REGISTER1, 0);
        I2C_RequestRead(*tempArray[0], 1);
        I2C_Send_Start_Condition(REGISTER2, 0);
        I2C_RequestRead(*tempArray[1], 1);
        I2C_Send_Stop_Condition();
        tempVal_int = (tempArr[0] << 8 | tempArr[1]) & 0x1FFF;

        if((tempArr[0] & (1 << 5)) == 0)
        {
            return tempVal_int / 32.0;
        }
        else
     {
          return (tempVal_int - 16384) / 32.0;
     }
   }

   else
   {
       return 1000000.0;
   }
}

int main()
{
                getTemperature();
                return 0;
```

}

==

Problem 5):

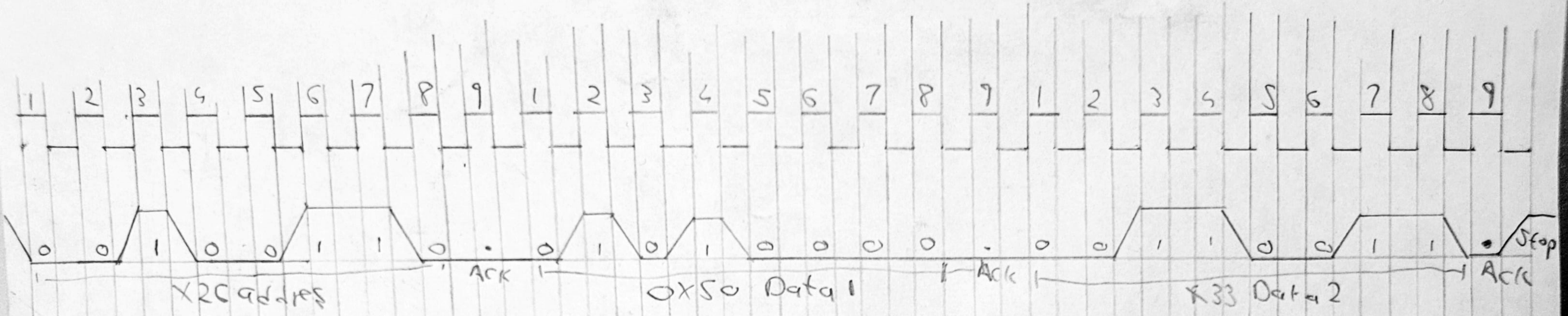
Timer prescaler = 32
Counter TOP Value = 249
CLK = 8Mhz
Periodic interval = 0.125 microseconds = 0.000125 milliseconds

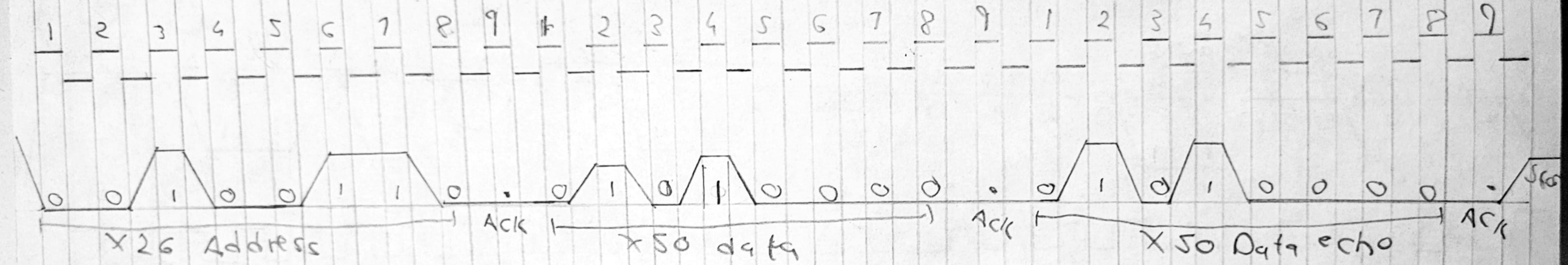
Snippet

```
#define get_cycle_count() *((volatile uint32_t*)0xE0001004)

bool State = true;
bool Rising_Edge = false;
bool Falling_Edge = false;
uint8_t Timer = 10;

uint32_t Start_Time = 0;
bool Start_Flag = false;
uint32_t Stop_Time = 0;
bool Stop_Flag =  false;
uint32_t Time_Elapsed = 0;

void Timer0_OV_Handler()
{
		if(Timer > 0)
		{
			Timer--;
		}
		else
		{
			Timer = 10;
			State = true;
			TIM_Base_Stop_IT(&htim1);

		If( (falling_edge == true) && GPIO_ReadPin(Push_Button_GPIO_Port, Push_Button_Pin)
== 0)
		{
			if(!Start_Flag)
			{
				Start_Time = Get_Cycle_Count();
				Start_Flag = true;
				Stop_Flag = False;
			}

			else if(!Stop_Flag)
```

```
                        {
                                    Stop_Time = Get_Cycle_Count();
                                    Stop_Flag = true;
                                    Start_Flag = true;
                                    Time_Elapsed = ((Start_Time – Stop_Time) / 8000000) –
0.000125;
                        }
            }

            else if((rising_edge == true) && GPIO_ReadPin(Push_Button_GPIO_Port,
Push_Button_Pin) == 1)
            {
                        if(!Start_Flag)
                        {
                                    Start_Time = Get_Cycle_Count();
                                    Start_Flag = true;
                                    Stop_Flag = false;
                        }
                        else if(!Stop_Flag)
                        {
                                    Stop_Time = Get_Cycle_Count;
                                    Stop_Flag = true;
                                    Start_Flag = false;
                                    Time_Elapsed = ((Start_Time – Stop_Time) / 8000000) –
0.000125;
                        }
            }
            Falling_Edge = false;
            Rising_Edge = false;
            }
}

void pinRising_Handler()
{
            if(State == True)
            {
                        TIM_Base_Start_IT(&htim1);
                        State = false;
                        Rising_Edge = True;
            }
            else
            {
                        __NOP();
            }
}

void pinFalling_Handler()
{
            if(State == true)
```

```
		{
			TIM_Base_Start_IT(&htim1);
			State = false;
			Falling_Edge = true;
		}
		else
		{
			__NOP();
		}
}
```





SOlution 3

1st plot is for 3.A

2nd plot is for 3.B