ECE-GY 6843 Real Time Embedded Systems Exam 1

Yogya Sharma

(ys5250@nyu.edu)

1.

C code:

```
#define PORTA_BASE_ADDR             0x40020000
#define PUPDR_REG                   0xC
#define MODER_REG                   0x0
#define ODR_REG                     0x14
#define OTYPER_REG                  0x4

Volatile unsigned uint32_t* PORTA_ PUPDR    = (uint32_t*)( PORTA_BASE_ADDR + PUPDR_REG)
Volatile unsigned uint32_t* PORTA_MODER     = (uint32_t*)( PORTA_BASE_ADDR + MODER_REG)
Volatile unsigned uint32_t* PORTA_ODR       = (uint32_t*)( PORTA_BASE_ADDR + ODR_REG)
Volatile unsigned uint32_t* PORTA_OTYPER    = (uint32_t*)( PORTA_BASE_ADDR + OTYPER_REG)


Uint32_t BlockWritePortA(uint16_t PinVals)  {
*PORTA_MODER = 0xAAAAAAAA;              //Set each pin to general purpose output mode
*PORTA_OTYPER = 0x0;                    //Set each pin to output push-pull
*PORTA_PUPDR = 0x0;                     //Set each pin to 0, no pull up or down resistor
*PORTA_ODR = PinVals;                   //Set ODR bits as per the PinVals value
uint32_t odr_value = *PORTA_ODR;        //Value of the ODR register is stored in variable odr_value
return odr_value
}
```

2.

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.

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. Polling and interrupts let CPU stop what it is currently doing and respond to the more important task. Polling and interrupts are different from each other in many aspects, but the basic point that distinguishes both is that in polling the CPU keeps in checking I/O devices at regular intervals whether it needs CPU service, whereas in interrupts the I/O devices tell the CPU that its services are needed.. Interrupts can be faster than polling, but it depends on the situation. If the device that the CPU is waiting on is ready to send data, then an interrupt can be used to notify the CPU. However, if the device is not ready to send data, then the CPU will have to wait until the device is ready before it can send any data. Polling is also not always faster than interrupts. In some cases, polling can be slower than using an interrupt because the CPU must wait for the poll request to complete before it can start processing other requests. This waiting time can be significant if there are many requests waiting to be processed. In polling the CPU wastes a lot of CPU cycles by repeatedly checking the command-ready bit of every device.

c) System latency is always larger than interrupt latency. System latency is the interrupt latency plus the maximum amount of time that interrupts might be disabled. System latency is at least as large as the interrupt latency. Latency is a networking term to decide the total time it take a data packet to travel from one node to another. Interrupt latency is the amount of time that passes from when an interrupt is generated to when it is first serviced. System latency is the amount of time that passes from when an interrupt is generated to when it is fully serviced. System latency will always be larger than interrupt latency because it includes the time it takes to service the interrupt, in addition to the time it takes for the operating system to process the interrupt. Therefore, interrupts are usually processed much faster than system calls.
Interrupts are typically processed much faster than system calls because they do not require the operating system to process them. System calls, on the other hand, require the operating system to do some processing before they can be executed. This can delay the execution of an application because the operating system may be busy doing other tasks.

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).
Instead the variables shared between ISR functions and normal functions should be. Declaring a variable as volatile tells the compiler to not 'cache' the value of the variable's values into a processor registers and because such variables might change at any time and thus the compiler must reload the variable from the memory whenever it is referenced, rather than relying on a copy it might have in a processor register. A global or static variable used in an interrupt will appear to change unexpectedly at the task level, and hence it is volatile.

e) The interrupt vector table must be placed in a specific location in memory. True; the startup file lays out the interrupt vector table and marks it as such; the linker script describes where in memory the interrupt vector table should go. The processor will look for specific interrupt vectors at the expected locations in memory. Interrupt vector is the memory location at an interrupt handler, which prioritizes interrupts and saves them in a queue when more than one interrupt is waiting to be handled. The interrupt vector table is a table of pointers to the functions that will handle interrupts for a given program. The table must be placed in a specific location in memory so that the processor can find it when an interrupt occurs. If the table is not in the correct location, the processor will not be able to find the correct handler function and the interrupt will not be handled. This can cause problems with the program's execution. Most processors have a specific area of memory reserved for this purpose. It is important to keep this area consistent across different processors so that the table will be in the same location. If the table is not located in this specific area, the processor may not be able to find it and will generate an invalid interrupt request. This can cause unexpected behavior in your program.

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.A true ISR can't return a value or have arguments. But a helper routine which implements part of an ISR is just a function, which could have either or both if the API by which it interacts with the overall ISR provides for it.
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.

g) SPI is not safe to interrupt. If a transaction has begun on an SPI bus, the bus may not be used for anything else until the transaction is complete. The driver is not thread safe for performance reasons, if you need to access the SPI bus from multiple threads then use the spiAcquireBus() and spiReleaseBus() APIs to gain exclusive access.
The SPI module supports interfacing with one or more slave devices on the bus. It is important to note that when multiple slave devices are being used, each slave should have an independent Slave Select connection from the master.

h) AVR - by default if you do not specify an ISR, then the interrupt vector in flash will be 0x0000, which means that your application will **jump into reset** whenever this interrupt happens.
If there is no ISR defined, the location for the jump instruction in the interrupt vector will either be null, it may be a jump to an exception routine, it may jump to the beginning of the program, or it may contain a "return from interrupt" (e.g. RTI) instruction.

i) The Pin Change Interrupts share an ISR between all the pins on a port (port B, C, and D). And anytime a pin changes on that port, it calls the port's ISR which must then decide which pin caused the

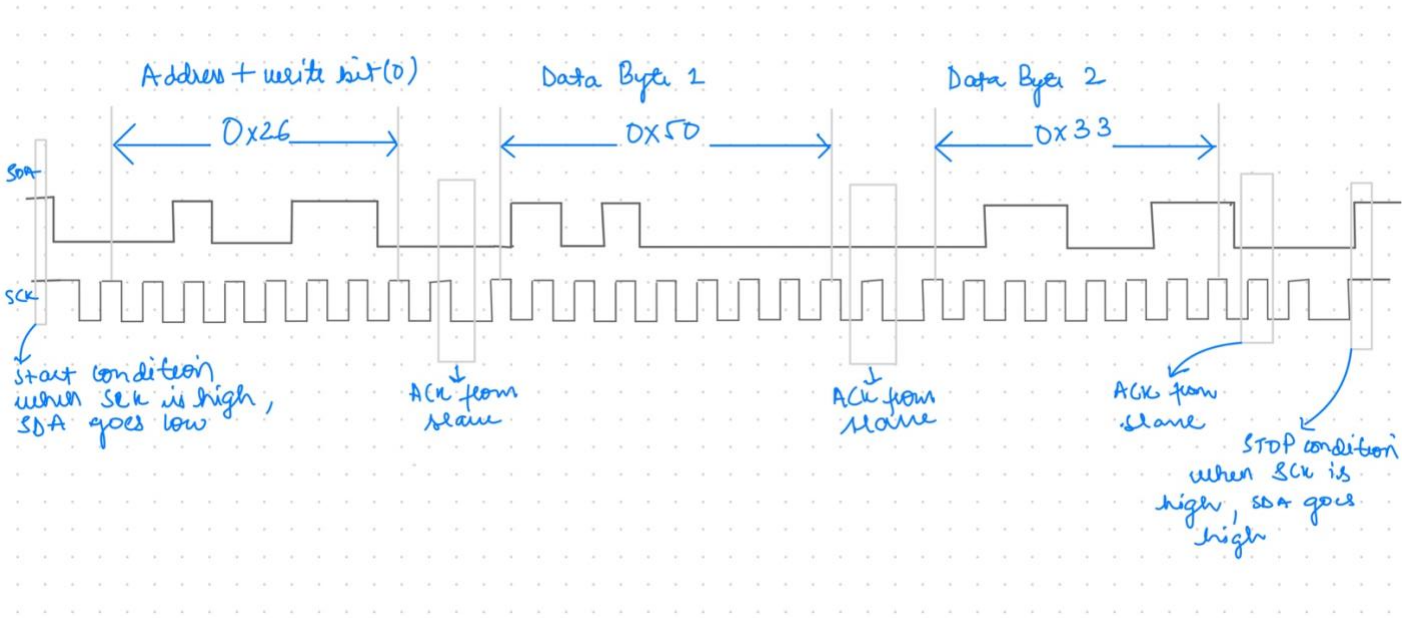
interrupt. So Pin Change Interrupts are harder to use but you get the benefit of being about to use any pin.

3.

a) 7-bit address = 0x13 = 001 0011; For write = 0010 0110

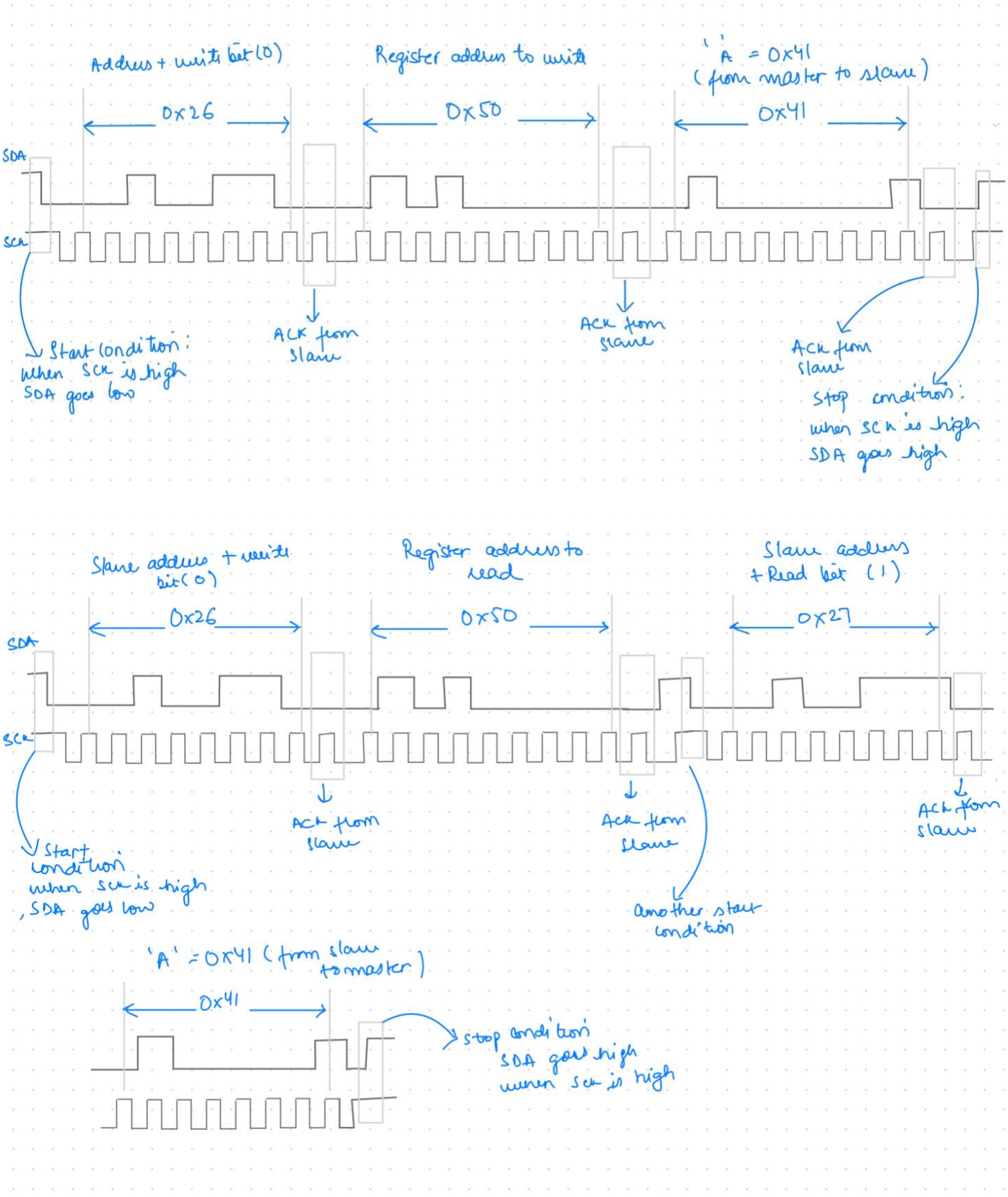
Byte 1 = 0x50 = 010 1000 ; Byte 2 = 0x33 = 0011 0011

As shown in the timing diagram, in I2C the MSB is sent first.



The first 7-bit address + R/W it identifies the slave which is being communicated to, the next byte of data tells the address of the register of that slave. Thus, 0x26 identifies the slave and identifies the slave and indicates that the next transaction will be a WRITE, that is, data will flow from master to slave. After the correct slave gives acknowledgement (Ack) , master sends 0x50. 0x50 is the address of the register at which master will write data. 0x33 id the written to the register at address 0x50 for slave address 0x13.

b) Echo transaction:
As shown in the timing diagram, in I2C the MSB is sent first.



The timing diagram above is the complete one, not two separate diagrams.

Following steps happen for an Echo transaction:

i) Master sends slave address + write (0x26). This is followed by the register address to write to (0x50).
ii) Master sends 0x41 (‘A’) to slave, this gets written to address 0x50 in the slave, stopping the communication.
iii) Master sends start condition again, followed by slave address + write (0x26). Slave acknowledges, master then sends register address which will be read from.
iv) Master then sends slave + read (0x27) and then releases data bus, slave takes over and sends 0x41 (‘A’) back.
v) Master can either choose to send Ack/NAck. It sends stop condition to send communication.

4.

REGISTER MAP

Table 8.

Register	Name	Register Data	Function
0x80	Control	D15 to D8	Read/write
0x81		D7 to D0	Read/write
0x82	Start frequency	D23 to D16	Read/write
0x83		D15 to D8	Read/write
0x84		D7 to D0	Read/write
0x85	Frequency increment	D23 to D16	Read/write
0x86		D15 to D8	Read/write
0x87		D7 to D0	Read/write
0x88	Number of increments	D15 to D8	Read/write
0x89		D7 to D0	Read/write
0x8A	Number of settling time cycles	D15 to D8	Read/write
0x8B		D7 to D0	Read/write
0x8F	Status	D7 to D0	Read only
0x92	Temperature data	D15 to D8	Read only
0x93		D7 to D0	Read only
0x94	Real data	D15 to D8	Read only
0x95		D7 to D0	Read only
0x96	Imaginary data	D15 to D8	Read only
0x97		D7 to D0	Read only

Control register map for measuring temperature:

Function	D15	D14	D13	D12
Measure Temperature	1	0	0	1

TEMPERATURE CONVERSION FORMULA

Positive Temperature = ADC Code (D)/32
Negative Temperature = (ADC Code (D) - 16384)/32
where ADC Code uses all 14 bits of the data byte, including the sign bit.
Negative Temperature = (ADC Code (D) - 8192)/32
where ADC Code (D) is D13, the sign bit, and is removed from the ADC code.)

CONTROL FLOW:

i) Send Start command on the serial interface followed by slave address(0x0d), indicating WRITE.
ii) Write byte 0x90 to Control register to specify slave to measure temperature.
iii) Poll the status register to check if valid temperature measurement is available.
iv) After temperature reading is valid, read the temperature data from registers 0x92 and 0x93.
v) Convert signed value to float value as per formula given in the data sheet. Valid temperature range is -40°C to +150°C.
vi) Return temperature value.

At any step, if any communication is unsuccessful, return a value greater than 150 to identify which part caused the error.

CODE:

```
1. Float GetTemperature(){
Volatile uint8_t *temperature_status;        //Value of status register to check if temp is valid
*temperature_status = 0;
Volatile uint8_t *temp_value_buffer;             //Buffer to read temp data
float temperature_final = 0;
If (Start_I2C() == 0)   {
return 201; }                                 //Return 201 if I2C is unsuccessful
If (I2C_Send_Start_Condition(0x0d, 0)==0) {       //Slave address with 0x0d identified, WRITE
return 202; }                                 //Return 202 if Start Condition (WR) is unsuccessful
if(I2C_Write_Byte(0x80) == 0)   {                 //Inform slave to write to Control Register
return 203; }                                 //Return 203 if Write_Byte() is unsuccessful
if(I2C_Write_Byte(0x90) == 0)   {                 //Write 0x90 Control Register to measure temp.
return 203; }                                 //Return 203 if Write_Byte() is unsuccessful
If (I2C_Send_Start_Condition(0x0d, 0)==0) {       //Slave address with 0x0d identified, WRITE
return 202; }                                 //Return 202 if Start Condition (WR) is unsuccessful
if(I2C_Write_Byte(0x8F) == 0)   {                 //Inform slave to send Status on next Read
return 203; }                                 //Return 203 if Write_Byte() is unsuccessful

while(*temperature_status != 1){
If (I2C_Send_Start_Condition(0x0d, 1)==0) {       //Slave address with 0x0d identified, READ
return 204; }                                 //Return 204 if Start Condition (RD) is unsuccessful
if(I2C_Request_Read(temperature_status, 1) !=1) {   //Request to read Status Register for temp validity
return 205  }                                 // Return 205 if Read Request is unsuccessful
}
                                        //Now temp value is ready to be read from register 0x92 and 0x93
If (I2C_Send_Start_Condition(0x0d, 0)==0) {       //Slave address with 0x0d identified, WRITE
return 202; }                                 //Return 202 if Start Condition (WR) is unsuccessful
if(I2C_Write_Byte(0x92) == 0)   {                 //Inform slave to send Temp value on next Read
return 203; }                                 //Return 203 if Write_Byte() is unsuccessful
If (I2C_Send_Start_Condition(0x0d, 1)==0) {       //Slave address with 0x0d identified, READ
return 204; }                                 //Return 204 if Start Condition (RD) is unsuccessful
if(I2C_Request_Read(temp_value_buffer, 2) !=1) {    //Request to read Temp value (2 bytes)
return 205  }                                 // Return 205 if Read Request is unsuccessful
*temp_value_buffer = *temp_value_buffer & 63;       //AND with 0x3f to make 2 MSB zero.
uint16_t temp_reg = *temp_value_buffer<<8 + *(temp_value_buffer+1);
if(*temp_value_buffer & (1<<5) ==1) {             //check if signed bit is 1 (negative temp)
temperature_final = (temp_reg – 16384)/32;
```

```
}
Else{
temperature_final = temp_reg/32;
}
Return temperature_final
}
```

5. The Timer() with prescaler of 32, counter TOP value of 250 and CLK running at 16MHz, gives an periodic timer after every 1 ms.

```
1.  bool state = True;                  //State variable to ensure that we don't start timer inside external interrupt handler */
bool rise_edge = False;               //This will be set by rising edge interrupt handler */
bool fall_edge = False;              //This will be set by falling edge interrupt handler */
uint8_t timer_count = 10; //This variable will be 0 after 5 ms

//Get cycle count will fetch the clock ticks since the microcontroller was booted from CYCCNT register
#define get_cycle_count() *((volatile uint32_t*)0xE0001004)

uint32_t start_time = 0;
bool start_flag = False;
uint32_t stop_time = 0;
bool stop_flag = False;
uint32_t elapsed = 0;

// Get value from CYCCNT register
//Handler function which is called when Timer 0 of periodic time interval 0.5 expires
void Timer0_OV_Handler()
  {
  If(timer_count > 0) {
     timer_count--;
  }

  else {
     //Reload the timer_count with value 10
     timer_count = 10;
     //Make the state variable as true, 5 ms elapsed and stop the timer state = true;
     TIM_Base_Stop_IT(&htim1);
```

```
    if ((fall_edge == True) && GPIO_ReadPin(Push_Button_GPIO_Port, Push_Button_Pin) == 0)
    {
        if (!start_flag){
            start_time = get_cycle_count();
            start_flag = True;
            stop_falg = False;
        }
        else if(!stop_flag) {
            stop_time = get_cycle_count();
            stop_flag = True;
            start_flag = True;
            elapsed = ((start_time – stop_time)/(16000000)) – 0.0005;
        }
    }

    else if( (rise_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;
            elapsed = ((start_time – stop_time)/(16000000)) – 0.005; }

    }

    fall_edge = False;
    rise_edge = False;

  }

  }

//function called during rise edge transition of button
```

```
void pinRising_Handler()
{
   if ( state == True){
     //Start Timer
     TIM_Base_Start_IT(&htim1);
     state = False;
     rising_edge = True;
   }

   else{
     __NOP();

   }

}


//function called during the fall edge transition of button

void pinFalling_Handler()
{
   if ( state == True) {
       //Start the Timer
       TIM_Base_Start_IT(&htim1);
       state = False;
       fall_edge = True;
       }

   else {
       __NOP(); }
}
```