NYU TANDON SCHOOL OF ENGINEERING Name: \_\_\_\_\_\_\_\_\_\_\_Ankita Gupta\_\_\_\_\_\_\_\_\_\_\_

ECE6483 – Real Time Embedded Systems ID: \_\_\_\_\_\_\_\_\_\_\_ag9135\_\_\_\_\_\_\_\_\_\_\_\_\_\_

EXAM 2 / FALL 2022

Instructions: Answer all questions. You may uses any reference materials from class or from our class websites and/or reference sheets.

**Question 1:**

We spent a significant amount of time in class discussing how to read and write memory using both C and assembly. Since our architecture uses memory mapped I/O, setting up the GPIO registers is simply reading and writing the appropriate memory locations. The following algorithm (steps 1.a, 1.b etc.)can be used for our controller to do a block GPIO write, meaning writing all port pins (15 in total) simultaneously, for any port (say PORTA).

**Use the below algorithm steps (LOCATED ON THE FOLLOWING PAGES) to write a C function:**

Assumption Taken that the following structure exists –

**typedef struct**

{  
**volatile uint32\_t** MODER; *// mode register, offset: 0x00*

**volatile uint32\_t** OTYPER; *// output type register, offset: 0x04*

**volatile uint32\_t** PUPDR; *// pull-up/pull-down register, offset: 0x0C*

**volatile uint32\_t** ODR; *// output data register, offset: 0x14*

} GPIOA;

*#define GPIOA\_BASE (0x40020000)*

*#define GPIOA ((GPIOA \*) GPIOA\_BASE)*

*// we would be able to access the GPIOD MODER register as GPIOD->MODER*

uint32\_t BlockWritePortA(uint16\_t PinVals) {

//Answer 1 in accordance with the struct defined above

GPIOA->MODER = 0x55555555; // general purpose output mode is 0 for all 16 even bits in MODER and 1 for all 16 odd bits in MODER

GPIOA->OTYPER = 0x0000; //output push-pull is 0 for every bit that can be set in OTYPER

GPIOA->PUPDR = 0x00000000; //no pullup-pull-down is 00 for all 16 sets in PUPDR

GPIOA->ODR = PinVals; //individually setting each bit as 0 or 1as per Pin is off (0) or on (1) for lower 16 bits assuming reserved bits are handled while adding input (PinVals)

uint32\_t odr\_val = GPIOA->ODR;

return odr\_val;

}

PinVals is a 16 bit number with 1’s in positions to turn “on” and 0’s in the pin positions to turn “off”. The function should implement algorithms parts 1.a, 1.b, 1.c and 1.d described above and return the value of the ODR register above. Note, the addresses of the registers (memory) are the Base Address + Offset as indicated above.

# PORTA has a base address of 0x40020000

## 1.a Set each pin in the MODER register to general purpose output mode

Table, timeline

Description automatically generated

1.b Set each pin in the OTYPER register to output push-pull

A picture containing application

Description automatically generated

## 1.c Set each pin in the PUPDR register to 0, meaning no pull up or down resistor

Timeline

Description automatically generated

Text

Description automatically generated

***1.d Set the ODR register bits to a 1 or 0, depending on if the pin is turned on or off.***

Table

Description automatically generated with medium confidence

**Question 2:**

Please evaluate and explain the following statements:

1. An interrupt is always a high priority, urgent task.

Answer 2 a) False

Although quick response from peripherals is not required may nevertheless be driven by interrupts.

1. Using interrupts is always faster than polling.

Answer 2 b) False

This trade-off is influenced by the system's interrupt latency, the frequency of the event you are polling or interrupting from, and perhaps other variables as well.

1. System latency is always larger than interrupt latency

Answer 2 c) True

System latency is a superset including the interrupt latency and additionally the max. amount of time the interrupts might be disabled.

1. Global variables used within an ISR should always be declared volatile.

Answer 2 d) True

Interrupt Service Routines are usually triggered asynchronously and might not be predictable for the compiler. Hence, in order to fetch the latest value of ISR (cached value) and not an compiler updated value, we cannot declare ISR to be volatile.

1. The interrupt vector table must be placed in a specific location in memory.

Answer 2 e) True

The interrupt vector table is laid out in startup file and its location in memory is specified in the linker script. The processor will scan the anticipated regions of memory for specific interrupt vectors.

1. An ISR can return a value and take arguments

Answer 2 f) False

The ISRs are not meant to return or input any parameters, like void functions. We cannot call the ISR to be able to pass arguments in it as interrupts can occur anytime. And because there is no calling code to read a return value, an ISR also cannot return a value.

1. It is OK for an ISR to safely access the SPI bus that has multiple slaves.

Answer 2 g) False

It is risky to interrupt SPI. An SPI bus cannot be used again until the transaction is finished if a transaction has started on it. For e.g., the SPI bus might communicate with multiple slaves at the same time and if the interrupt occurs for even 1 of those slaves, the other slaves will also get to know ISR’s data to slave 1 and hence, the security of the data might be compromised.

1. If you enable an interrupt in your C code, but don’t define the ISR, what code executes when the interrupt is triggered?

Answer 2 h)

The interrupt vector table is laid out in startup file. So, if an interrupt is not defined in the code and interrupts gets automatically triggered. It will run into an infinite loop.

The following code gets executed –

*.section .text.Default\_Handler,"ax",%progbits*

*Default\_Handler:*

*Infinite\_Loop:*

*b Infinite\_Loop*

*.size Default\_Handler, .-Default\_Handler*

1. According to the datasheet of our ARM CORTEX F4, which pins can fire pin change interrupts?

Answer 2 i) Pin 4 and pin 5 can fire pin change interrupts.

**Question 3:**

1. Draw the timing diagram for SDA and SCK that shows a master writing two bytes of data to slave at address (7 bit address) 0x13. The first byte is 0x50 and the second byte is 0x33.

*# as per the simplified sample of I2C write from lecture 7 slide 36*

Diagram

Description automatically generated

The slave address would be 0x26 because the 8th bit would define if it was being read/write bit after 0x13.

1. Show the timing diagram like in a., but for a simple single byte “echo” transaction. That is, the master sends one byte, “A” for example, and the slave sends back the same “A” to the master.

Assumptions –

* The data byte sent is ‘0x50’
* The address of the slave is 0x13 which is writing (having 8th bit as 0)
* The address of the master is 0x43 which is writing (having 8th bit as 0)

*#Simplified sample as per the single byte write and single byte read first by master, repeated start and followed by slave in reference to lecture 7 slide 33/34*

Diagram

Description automatically generated

**Question 4:**

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:

* 1. int Start\_I2C()
     1. Initializes all of the I2C hardware pins etc. Returns 1 if successful, 0 otherwise
  2. int I2C\_Write\_Byte(uint8\_t data)
     1. Writes ‘data’ on the I2C Bus. Returns 1 if slave ACKs, or 0 if slave NAKs

1. int I2C\_Send\_Start\_Condition(uint8\_t address, int isRead)
   * 1. Send a start or restart condition, followed by 7-bit address, followed by ‘isRead’ bit, which is 0 if write, 1 if read.
     2. Returns 1 if successful, 0 otherwise
2. int I2C\_Send\_Stop\_Condition()

i. returns 1 if successful, 0 otherwise

1. int I2C\_Request\_Read(unint8\_t \*buffer, int numBytes)

i. reads ‘numBytes’ bytes off the bus, each followed by a master ACK except for the last byte, which is 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.

Answer 4)



Assumptions –

* AD5933 has a 7-bit serial bus slave address
* Default serial bus address on booting timestamp is 0x0D as slave address
* No of digits in temperature is restricted to 2 places
* The initial address of reading the temperature is 92
* The write is 0 and read is 1

The C code will be as follows –

#define SLAVE\_ADDRESS 0x0D

#define TEMPFROMSENSOR\_OUTPUT\_ADDRESS 0x92

#define READ 1

#define WRITE 0

uint8\_t \*buffer; //globally defined

//convert F to C

float tempConvert (unit8\_t \* f)

{

int temp = (\*(f) << 8 | \*(data + 1)) & 0x1FFF;

if ((\*f & (1<<5)) == 0)

{return temp / 32.0;}

else

{return (temp – 8192) / 32.0;} //14th bit will be the sign bit

}

float getTemperature()

{

// Initializing I2C hardware

start\_I2C();

// write to slave address location

int success = I2C\_Send\_Start\_Condition(SLAVE\_ADDRESS, WRITE);

if (success == 0) {

return;

}

//communicated to expect a block read to slave

success = I2C\_Write\_Byte(0xA1);

if (success == 0) {

return;

}

//communicated the number of bytes to be read by the slave

success = I2C\_Write\_Byte(2);

if (success == 0) {

return;

}

// slave to read the data written into it

success = I2C\_Send\_Start\_Condition(TEMPFROMSENSOR\_OUTPUT\_ADDRESS, READ);

if (success == 0) {

return;

}

//master to ack after each data byte

success = I2C\_Request\_Read(buffer,2);

if (success == 0) {

return;

}

// send stop condition

success = I2C\_Send\_Stop\_Condition()

if (success == 0) {

return;

}

return tempConvert(buffer); //returning the temperature in Celsius.

}

**Question 5:**

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 249, and is set up to reset to 0 and count up to TOP. The CLK is running at 8MHz. Here is the handler definition:

void Timer0\_OV\_Handler()

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.**

Answer 5)

Assumptions –

GPIO Port – x

GPIO Pin – n

To calculate the time elapsed, we will start calculating the time difference between two falling edges when it is greater than 0.25 ms.

C code is as follows –

//Defining global variables

bool state = true; // avoid timer to get triggered inside external interrupt handler

bool rising\_edge = false; // no rising edge by default

bool falling\_edge = false; // no falling edge by default

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)

// Get value from CYCCNT register

uint32\_t t\_start = 0;

bool start\_flag = false;

uint32\_t t\_stop = 0;

bool stop\_flag = false;

uint32\_t time\_elapsed = 0;

//Handler function to initiate timer

void Timer0\_OV\_Handler()

{

if(timer\_count > 0)

timer\_count--;

else

{

//Reload timer\_count with value 10

timer\_count = 10;

//External interrupt introduced to make the state variable as true, 0.25 ms elapsed and stop the timer state = true;

HAL\_TIM\_Base\_Stop\_IT(&htim1);

//If falling\_edge variable is true and GPIO is low -> Valid input after debouncing

if( (falling\_edge == true) && GPIO\_ReadPin(Push\_Button\_GPIO\_Port, Push\_Button\_Pin) == 0)

{

if(!start\_flag)

{

t\_start = get\_cycle\_count();

start\_flag = true;

stop\_flag = false;

}

else if(!stop\_flag)

{

t\_stop = get\_cycle\_count();

Stop\_flag = true;

Start\_flag = true;

time\_elapsed = ((t\_start – t\_stop)/(8000000)) – 0.00025; //reducing the threshold time of 0.25ms to sec

}

}

//If falling\_edge variable is true and GPIO is low -> Valid input after debouncing

else if((rising\_edge == true) && GPIO\_ReadPin(Push\_Button\_GPIO\_Port, Push\_Button\_Pin)==1)

{

if(!start\_flag)

{

t\_start=get\_cycle\_count();

start\_flag=true;

stop\_flag=false;

}

else if(!stop\_flag)

{

t\_stop = get\_cycle\_count();

stop\_flag = true;

start\_flag = false;

time\_elapsed = ((t\_start – t\_stop)/(8000000)) – 0.00025; //reducing the threshold time of 0.25ms to sec

}

}

falling\_edge = false;

rising\_edge = false;

}

}

//Handler function called during Rising Edge Transition of button

void pinRising\_Handler()

{

if(state==true)

{

//Start the timer

HAL\_TIM\_Base\_Start\_IT(&htim1);

state = false;

rising\_edge = true;

}

else

{

//No operation (will consume only one CPU cycle)

\_\_NOP();

}

}

//Handler function called during the Falling Edge Transition of button

void pinFalling\_Handler()

{

if(state==true)

{

//Start the timer

HAL\_TIM\_Base\_Start\_IT(&htim1);

state=false;

falling\_edge=true;

}

else

{

//No operation (will consume only one CPU cycle)

\_\_NOP();

}

}