COMPENG 4DS4: Embedded Systems
Lab 1: Bare-Metal I/O (Sensors and Actuators)
Group 16

Feb 7, 2025

Tim Tang, tangt30, 400325326
Michael Nguyen, nguyem77, 400331180
Yao Yuan Xie, xie5, 400316352
Mohamed El Sayed, elsaym11, 400306074

Declaration Contribution

Name	Contributions
Tim Tang	Completed Experiment 4 Problems 4 and 5
Michael Nguyen	Developed problems 2 and 3
Yao Yuan Xie	Experiment/problem 1
Mohamed El Sayed	Developed experiment/Problem 2, debugged problem 4

Experiment 1: Control DC motor

Experiment Setup

Below is a table of the speed and direction of the motor given the input PWM value. Values far from ~35 are hard to tell the speed of the motor.

Input DC	Observation
0	backwards
10	backwards
20	backwards
30	backwards very slow
40	forward very slow
50	Forward slow
60	forward
70	forward fast
80	forward fast
90	forward fast
100	forward fastest
-10	backwards fast
-20	backwards fast
-30	backwards fast
-40	backwards fast
-50	backwards fast
-60	backwards fast
-70	backwards fast
-80	backwards fast
-90	backwards fast
-100	backwards fastest

It is observed that the DC motor transitions from backwards to forwards rotation somewhere between 30-40 input_DC value. The formula used to calculate duty cycle is:

dutyCycle_DC = input_DC * 0.025f/100.0f + 0.0615;

This corresponds to 6.9-7.15% duty cycle. To make the input prompt more intuitive, the formula should be adjusted so 0 input_DC value corresponds to 7% duty cycle.

dutyCycle_DC = input_DC * 0.025f/100.0f + 0.07;

Problem 1

If the range of input_Servo is -90 to 90 degrees, then the servo motor needs to step up 0.025% duty cycle every time input_Servo value is increased by 1. And by setting 0 degrees to 7.5%, we can have a duty cycle range of 5-10%.

dutyCycle_Servo = input_Servo * 0.05f/180.0f + 0.075;

Experiment 2: UART Communication

Problem 2

For problem 2, UART was implemented to write to our UART serial connection (via Realterm) and read the user input to control the DC and Servo motors. We used the main two functions for UART which was WriteBlocking to write text to the Realterm terminal, which we mainly used to indicate to the user which motor the input is currently going to control, and the other crucial function ReadBlocking, which was used to read the user input. We had a char array (string) that had reserved memory for 8 chars, while reading and storing the user input one char at a time, while incrementing the char array. We had two conditions to exit the reading segment of the char array and that was to receive an "!", which we used to indicate the end of the user input or it would reach 8 characters, which would be much more than the space we need (since the max values for user input would be -100 to 100 for DC motor and -90 to 90 for the servo motor). Then we use atoi to convert the string to a valid number to use in the PWM. If the user input does not end up in the allowed range, we will not set the PWM and ask the user to reenter the inputs again.

```
UART_WriteBlocking(TARGET_UART, txbuff, sizeof(txbuff) - 1);
printf("%s", txbuff);
int i;
for (i = 0; i < 8; i++) {
        // retrieve bytes one at a time
        UART_ReadBlocking(TARGET_UART, &ch, 1);
        printf("%c", ch);
        if (ch != '!') {
                inputbuff[i] = ch;
        }
        else {
                inputbuff[i] = ch;
                break;
        }
}
//convert to integer
input_DC = atoi(inputbuff);
```

Experiment 3: UART Communication with Interrupts

Problem 3

Problem 3 is made up of two components: the while loop in the main function and the IRQ handler. The code is functionally similar to problem 2, where we are using array inputbuff[] to store characters until the entered integer is submitted using "!". The interrupt is triggered every time some character is received via UART transmission. A boolean flag is used to keep track of whether the system is collecting the PWM or servo value. After the interrupt, the new_char flag is set (similar to the experiment) which allows the main function to store the received character in inputbuff[]. Once "!" is received (i.e. value is submitted), the inputbuff[] is converted into an integer using atoi() and the value is stored in either input_DC or input_Servo respectively. Once the servo value is submitted, both of the input values are submitted to the motors and the cycle restarts.

```
// retrieve bytes one at a time
void UART4_RX_TX_IRQHandler()
{
        printf("\nInterrupt triggered!\n");

        UART_GetStatusFlags(TARGET_UART);
        ch = UART_ReadByte(TARGET_UART);
        printf("%c", ch);
        new_char = 1;
}
```

```
if(new_char) {
        new_char = 0;
        inputbuff[i] = ch;
        i++;
        //signifies end of string
        if ((ch == '!') || (i >= 8)) {
                //flag == 0 means DC input
                if (!flag) {
                        input_DC = atoi(inputbuff);
                        //swap to fetching servo input
                        flag = 1;
                        UART_WriteBlocking(TARGET_UART, msg_servo, sizeof(msg_servo) - 1);
                        printf("%s", msg_servo);
                }
                //flag == 1 means Servo input
                else {
                        input_Servo = atoi(inputbuff);
                        //check for valid input and submit values after servo input is complete
                        if ((input_DC >= -100) && (input_DC <= 100) && (input_Servo >= -90) && (input_Servo <= 90)) {
                                printf("\nInput is valid! (%d DC and %d Servo)\n", input_DC, input_Servo);
                                //from part 1
                                dutyCycle_DC = input_DC * 0.025f/100.0f + 0.0615;
                                dutyCycle_Servo = input_Servo * 0.05f/180.0f + 0.075;
                                updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, dutyCycle_DC);
                                updatePWM_dutyCycle(FTM_CHANNEL_Servo_MOTOR, dutyCycle_Servo);

                                FTM_SetSoftwareTrigger(FTM_MOTOR, true);
                        }
                        //restart cycle
                        flag = 0;
                        UART_WriteBlocking(TARGET_UART, msg_pwm, sizeof(msg_pwm) - 1);
                        printf("%s", msg_pwm);
                }
                //reset counter
                i = 0;
        }
}
```

Experiment 4: Interface with Accelerometer Sensor

Problem 4

The problem includes creating the SPI write function given the prototype below. The function was based on the provided SPI_read function. The details for how to use SPI read and write were obtained from “accelerometer&magnetometer.pdf” Sections 10.2.1 and 10.2.

Since only 1 byte of data is defined in the prototype, its assumed the function will only send one byte when called. To achieve this, the tx buffer is composed of 3 single-byte registers. The first bit of the first register is set to 1 using the or bitwise operator. The next 7 bits are the address location starting from the second bit to the last. The second register has the first bit of the address in the first bit of the register. The rest is ignored so it is filled with zeros. The last register is set to the data parameter.

The data size is set to a constant 3, and there is no use of the rx buffer or registers. The flags were kept the same as in SPI read.

```
status_t SPI_write(uint8_t regAddress, uint8_t value);
```

```
status_t SPI_write(uint8_t regAddress, uint8_t value) {
    dspi_transfer_t masterXfer;
    uint8_t *masterTxData = (uint8_t*)malloc(3);

    masterTxData[0] = regAddress | 0x80; //Set the most significant bit
    masterTxData[1] = regAddress & 0x80; //Clear the least significant 7 bits
    masterTxData[2] = value;

    masterXfer.txData = masterTxData;
    //masterXfer.rxData = masterRxData;

    masterXfer.dataSize = 3;
    masterXfer.configFlags = kDSPI_MasterCtar0 | kDSPI_MasterPcs0 | kDSPI_MasterPcsContinuous;
    status_t ret = DSPI_MasterTransferBlocking(SPI1, &masterXfer);

    free(masterTxData);

    return ret;
}
```

Problem 5

The steps in this problem to port the IO devices and set up the SPI communication protocol were taken from experiment 4 parts A and B. This includes changes to the initiation of the clock, pins, and fsl_fxos files. The function for SPI write was taken from Problem 4 and added to the main source file. The main functionality was provided by the demo app.