

COMPENG 4DS4 Lab 1 Report

Aaron Pinto
pintoa9

Raeed Hassan
hassam41

Jingming Liu
liuj171

Jeffrey Guo
guoj69

February 10, 2023

Declaration of Contributions

Problem	Contributions
1	Aaron, Jeffrey
2	Aaron, Raed
3	Aaron, Jingming
4	Jingming, Raed
5	Jingming, Raed

Experiment 1: Control DC motor

The observations regarding the speed and direction of the motor at different duty cycles are shown below in Table 1. These observations were made after centering the duty cycle values for the motor.

Table 1: Observations of Speed and Direction at Different Duty Cycles

Duty Cycle	Observation
0	Motor does not move
10	Motor moves forward
20	Forward speed increases
30	Forward speed increases further
40	Forward speed increases further
50	Forward speed increases further
60	Forward speed increases further
70	Forward speed increases further
80	Forward speed increases further
90	Forward speed increases further
100	Forward speed increases further
-10	Motor moves backward
-20	Backward fast
-30	Backward speed increases
-40	Backward speed increases further
-50	Backward very very fast
-60	Backward very very very fast
-70	Backward very very very very fast
-80	Backward very very very very very fast
-90	Backward very very very very very very fast
-100	Backward speed increases further

Problem 1

We control the servo motor through the same method as the DC motor is controlled. The pins and clock for the servo motor are additionally initialized in `BOARD_InitBootPins`, `BOARD_InitBootClocks`, and `setupPWM`. The main function for Problem 1 is shown in Listing 1. The duty cycle equations for the DC motor and servo motor are on lines 19 and 20, centered around a value near 0.075 which we honed in on through trial and error.

```
1 int main(void) {
2     uint8_t ch;
3     int motorInput;
4     int servoInput;
5     float motorDutyCycle;
6     float servoDutyCycle;
7
8     BOARD_InitBootPins();
```

```

9   BOARD_InitBootClocks();
10
11   setupPWM();
12
13   /***** Delay *****/
14   for (volatile int i = 0U; i < 1000000; i++)
15       __asm("NOP");
16
17   scanf("%d, %d", &motorInput, &servoInput);
18
19   motorDutyCycle = motorInput * 0.025f / 100.0f + 0.070745;
20   servoDutyCycle = servoInput * 0.025f / 45.0f + 0.078;
21
22   updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, servoDutyCycle);
23   updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, motorDutyCycle);
24
25   FTM_SetSoftwareTrigger(FTM_MOTOR, true);
26
27   while (1) {
28   }
29 }

```

Listing 1: Problem 1 main

Problem 2

We re-use the functions of Problem 1 and combine them with the UART functions established in Experiment 2. The main function for Problem 2 is shown in Listing 2. We customized the "Hello World" message to distinguish our program from others, making it easier to know if we've established a connection between our telemetry modules.

The input string that we read is in the form "%4d;%3d;", with the motor duty cycle input as 4 characters (ranging from -100 to 100, with positive numbers padded with a 0 in front), and the servo duty cycle input as 3 characters (ranging from -45 to 45, with positive numbers padded with a 0 in front), with semi-colons as separators.

```

1  int main(void) {
2      char buffer[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
3
4      int motorInput;
5      int servoInput;
6      float motorDutyCycle;
7      float servoDutyCycle;
8
9      /* Init board hardware. */
10     BOARD_InitBootPins();
11     BOARD_InitBootClocks();
12     BOARD_InitDebugConsole();
13
14     setupUART();
15     setupPWM();
16
17     /***** Delay *****/
18     for (volatile int i = 0U; i < 1000000; i++)
19         __asm("NOP");
20
21     char txbuff[] = "Hello World aaron\r\n";
22     UART_WriteBlocking(TARGET_UART, txbuff, sizeof(txbuff) - 1);
23
24     while (1) {
25         UART_ReadBlocking(TARGET_UART, &buffer[0], 9);

```

```

26
27     sscanf(buffer, "%4d;%3d;", &motorInput, &servoInput);
28
29     printf("%.9s\n%d %d\n", buffer, motorInput, servoInput);
30
31     motorDutyCycle = motorInput * 0.025f / 100.0f + 0.070745;
32     servoDutyCycle = servoInput * 0.025f / 45.0f + 0.078;
33
34     updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, servoDutyCycle);
35     updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, motorDutyCycle);
36
37     FTM_SetSoftwareTrigger(FTM_MOTOR, true);
38 }
39 }

```

Listing 2: Problem 2 main

Problem 3

With the introduction of the interrupt, we make sure the UART reads only 7 bytes after the IRQ by a counter. When the inputs are sufficient, the IRQ will be cleared. Note the char array was changed to volatile variable compared to Problem 2. The updated `setupUART` function is shown in Listing 3, and the `UART4_RX_TX_IRQHandler` function is shown in Listing 4. The main function for Problem 3 with interrupt handling is shown in Listing 5.

```

1 void setupUART() {
2     uart_config_t config;
3
4     UART_GetDefaultConfig(&config);
5     config.baudRate_Bps = 57600;
6     config.enableTx = true;
7     config.enableRx = true;
8     config.enableRxRTS = true;
9     config.enableTxCTS = true;
10    UART_Init(TARGET_UART, &config, CLOCK_GetFreq(kCLOCK_BusClk));
11
12    /***** Enable Interrupts *****/
13    UART_EnableInterrupts(TARGET_UART, kUART_RxDataRegFullInterruptEnable);
14    EnableIRQ(UART4_RX_TX_IRQn);
15 }

```

Listing 3: Problem 3 setupUART

```

1 void UART4_RX_TX_IRQHandler() {
2     UART_GetStatusFlags(TARGET_UART);
3     ch = UART_ReadByte(TARGET_UART);
4
5     PRINTF("GOT an interrupt from UART \n");
6
7     if (new_char == 7) {
8         return;
9     }
10
11    PRINTF("new_char = %d \n", new_char);
12    PRINTF("ch = %c \n", ch);
13
14    buffer[new_char] = ch;
15    PRINTF(" inside the interrupt buffer[new_char] = %s \n ", buffer);
16    new_char += 1;
17
18 }

```

Listing 4: Problem 3 UART4_RX_TX_IRQHandler

```

1  int main(void) {
2      char txbuff[] = "Hello World\r\n";
3
4      int motorInput = 0;
5      int servoInput = 0;
6      float motorDutyCycle;
7      float servoDutyCycle;
8      /* Init board hardware. */
9      BOARD_InitBootPins();
10     BOARD_InitBootClocks();
11     BOARD_InitDebugConsole();
12
13     setupUART();
14     setupPWM();
15
16     /***** Delay *****/
17     for (volatile int i = 0U; i < 1000000; i++)
18         __asm("NOP");
19
20     PRINTF("%s", txbuff);
21
22     UART_WriteBlocking(TARGET_UART, txbuff, sizeof(txbuff) - 1);
23
24     while (1) {
25         if (new_char == 7) {
26             new_char = 0;
27
28             PRINTF("outside the interrupt buffer = %s \n ", buffer);
29
30             PRINTF("cleared interrupt\n");
31
32             sscanf(buffer, "%4d%3d", &motorInput, &servoInput);
33             PRINTF("GOT input as motorInput = %d, servoInput = %d", motorInput, servoInput);
34             motorDutyCycle = motorInput * 0.025f / 100.0f + 0.070745;
35             servoDutyCycle = servoInput * 0.025f / 45.0f + 0.078;
36
37             updatePWM_dutyCycle(FTM_CHANNEL_SERVO_MOTOR, servoDutyCycle);
38             updatePWM_dutyCycle(FTM_CHANNEL_DC_MOTOR, motorDutyCycle);
39
40             FTM_SetSoftwareTrigger(FTM_MOTOR, true);
41             FTM_SetSoftwareTrigger(FTM_MOTOR, true);
42         }
43     }
44 }

```

Listing 5: Problem 3 main

Problem 4

The code listing for Problem 4, the `SPI_write` function, is Listing 6. The function is mostly a duplicate of the provided `SPI_read` function. It changes the initialized sizes of `masterTxData` and `masterRxData` to always be 3 bytes as we are only writing 1 byte of data, and two additional bytes are required to select between READ or WRITE and to select the register address.

As outlined in the FXOS8700CQ documentation, we set the value of R/W to 1 for write. The order of bits for a write operation is as follows:

Byte 0: R/W, ADDR[6], ADDR[5], ADDR[4], ADDR[3], ADDR[2], ADDR[1], ADDR[0],

Byte 1: ADDR[7], X, X, X, X, X, X, X,

Byte 2: DATA[7], DATA[6], DATA[5], DATA[4], DATA[3], DATA[2], DATA[1], DATA[0].

```

1 status_t SPI_write(uint8_t regAddress, uint8_t value)
2 {
3     dsp_i_transfer_t masterXfer;
4     uint8_t *masterTxData = (uint8_t *)malloc(3);
5     uint8_t *masterRxData = (uint8_t *)malloc(3);
6
7     masterTxData[0] = regAddress | 0x80; // Sets the most significant bit (enable write)
8     masterTxData[1] = regAddress & 0x80; // Clear the least significant 7 bits
9     masterTxData[2] = value;
10
11     masterXfer.txData = masterTxData;
12     masterXfer.rxData = masterRxData;
13     masterXfer.dataSize = 3;
14     masterXfer.configFlags = kDSPI_MasterCtar0 | kDSPI_MasterPcs0 |
        kDSPI_MasterPcsContinuous;
15     status_t ret = DSPI_MasterTransferBlocking(SPI1, &masterXfer);
16
17     free(masterTxData);
18     free(masterRxData);
19
20     return ret;
21 }

```

Listing 6: Problem 4 SPI_write

The main function is shown below in Listing 7. We test our function with the PL_COUNT register since it has 8 bits available for both read and write. The register is also available for both read and write during active mode. After the write, we read the value of the register by SPI and the register value changed as we wanted.

```

1 int main(void)
2 {
3     uint8_t byte;
4     uint8_t write_test_byte = 0xCC;
5     uint8_t read_test_byte;
6
7     /* Init board hardware. */
8     BOARD_InitBootPins();
9     BOARD_InitBootClocks();
10
11     voltageRegulatorEnable();
12     accelerometerEnable();
13
14     setupSPI();
15
16     /****** Delay *****/
17     for (volatile int i = 0U; i < 1000000; i++)
18         __asm("NOP");
19
20     SPI_read(0x0D, &byte, 1);
21     printf("The expected value is 0xC7 and the read value 0x%X\n", byte);
22
23     printf("reading to the PL_COUNT register before writing \n");
24     SPI_read(0x12, &read_test_byte, 1);
25     printf("The expected value for PL_COUNT register is 0x0 and the read value 0x%X\n",
        read_test_byte);
26
27     SPI_write(0x12, write_test_byte);
28     printf("writing to the PL_COUNT register \n");
29
30     SPI_read(0x12, &read_test_byte, 1);
31     printf("The expected value for the PL_COUNT register is 0xCC and the read value 0x%X\n",
        read_test_byte);
32
33     while (1)
34     {

```

```
35 }
36 }
```

Listing 7: Problem 4 main

Problem 5

The functions from Problem 4 and Experiment 4: Part B were used to convert the e-compass SDK example to use SPI instead of the default I2C. The following steps were taken to convert the project to SPI:

1. Add dsp_i driver to the project.
2. Use the same code of the functions BOARD_InitBootPins and BOARD_InitBootClocks from the project in Experiment 4: Part A.
3. Remove the following functions from ecompass_peripheral.c
 - i2c_release_bus_delay
 - BOARD_I2C_ReleaseBus
4. Add the following functions
 - setupSPI
 - voltageRegulatorEnable
 - accelerometerEnable
 - SPI_read
 - SPI_write
5. Update fsl_fxos.h with the "typedef"s for SPI_WriteFunc_t and SPI_ReadFunc_t, and update the fxos_handle_t and fxos_config_t structures to point to the new SPI functions.
6. Update fsl_fxos.c to replace any references to I2C_SendFunc and I2C_ReceiveFunc with SPI_writeFunc and SPI_readFunc. Make sure to update the function arguments for when they appear in FXOS_ReadReg and FXOS_WriteReg.
7. Update the main function similarly to the updates made in Experiment 4: Part B to configure SPI functions instead of I2C. The code listing for the main function is shown below in Listing 8.

```
1 int main(void)
2 {
3     fxos_config_t config = {0};
4
5     uint16_t i           = 0;
6     uint16_t loopCounter = 0;
7     double sinAngle      = 0;
8     double cosAngle      = 0;
9     double Bx            = 0;
10    double By            = 0;
```



```

11     uint8_t array_addr_size = 0;
12
13     status_t result;
14
15     /* Board pin, clock, debug console init */
16     BOARD_InitBootPins();
17     BOARD_InitBootClocks();
18
19     voltageRegulatorEnable();
20     accelerometerEnable();
21
22     setupSPI();
23
24     HW_Timer_init();
25
26     /****** Delay *****/
27     for (volatile uint32_t i = 0; i < 4000000; i++)
28         __asm("NOP");
29
30
31     /* Configure the SPI function */
32     config.SPI_writeFunc = SPI_write;
33     config.SPI_readFunc = SPI_read;
34
35     result = FXOS_Init(&g_fxosHandle, &config);
36     if (kStatus_Success != result)
37     {
38         PRINTF("\r\nSensor device initialize failed!\r\n");
39     }
40
41     /* Get sensor range */
42     if (kStatus_Success != FXOS_ReadReg(&g_fxosHandle, XYZ_DATA_CFG_REG, &
43         g_sensorRange, 1))
44     {
45         PRINTF("\r\nGet sensor range failed!\r\n");
46     }
47
48     switch (g_sensorRange)
49     {
50     case 0x00:
51         g_dataScale = 2U;
52         break;
53     case 0x01:
54         g_dataScale = 4U;
55         break;
56     case 0x10:
57         g_dataScale = 8U;
58         break;
59     default:
60         break;
61     }
62
63     PRINTF("\r\nTo calibrate Magnetometer, roll the board on all orientations to get
64         max and min values\r\n");
65     PRINTF("\r\nPress any key to start calibrating...\r\n");
66     GETCHAR();
67     Magnetometer_Calibrate();
68
69     /* Infinite loops */
70     for (;;)
71     {
72         if (SampleEventFlag == 1) /* Fix loop */
73         {
74             SampleEventFlag = 0;
75             g_Ax_Raw = 0;
76             g_Ay_Raw = 0;
77             g_Az_Raw = 0;
78             g_Ax = 0;

```

```

77     g_Ay          = 0;
78     g_Az          = 0;
79     g_Mx_Raw      = 0;
80     g_My_Raw      = 0;
81     g_Mz_Raw      = 0;
82     g_Mx          = 0;
83     g_My          = 0;
84     g_Mz          = 0;
85
86     /* Read sensor data */
87     Sensor_ReadData(&g_Ax_Raw, &g_Ay_Raw, &g_Az_Raw, &g_Mx_Raw, &g_My_Raw, &
        g_Mz_Raw);
88
89     /* Average accel value */
90     for (i = 1; i < MAX_ACCEL_AVG_COUNT; i++)
91     {
92         g_Ax_buff[i] = g_Ax_buff[i - 1];
93         g_Ay_buff[i] = g_Ay_buff[i - 1];
94         g_Az_buff[i] = g_Az_buff[i - 1];
95     }
96
97     g_Ax_buff[0] = g_Ax_Raw;
98     g_Ay_buff[0] = g_Ay_Raw;
99     g_Az_buff[0] = g_Az_Raw;
100
101     for (i = 0; i < MAX_ACCEL_AVG_COUNT; i++)
102     {
103         g_Ax += (double)g_Ax_buff[i];
104         g_Ay += (double)g_Ay_buff[i];
105         g_Az += (double)g_Az_buff[i];
106     }
107
108     g_Ax /= MAX_ACCEL_AVG_COUNT;
109     g_Ay /= MAX_ACCEL_AVG_COUNT;
110     g_Az /= MAX_ACCEL_AVG_COUNT;
111
112     if (g_FirstRun)
113     {
114         g_Mx_LP = g_Mx_Raw;
115         g_My_LP = g_My_Raw;
116         g_Mz_LP = g_Mz_Raw;
117     }
118
119     g_Mx_LP += ((double)g_Mx_Raw - g_Mx_LP) * 0.01;
120     g_My_LP += ((double)g_My_Raw - g_My_LP) * 0.01;
121     g_Mz_LP += ((double)g_Mz_Raw - g_Mz_LP) * 0.01;
122
123     /* Calculate magnetometer values */
124     g_Mx = g_Mx_LP - g_Mx_Offset;
125     g_My = g_My_LP - g_My_Offset;
126     g_Mz = g_Mz_LP - g_Mz_Offset;
127
128     /* Calculate roll angle g_Roll (-180deg, 180deg) and sin, cos */
129     g_Roll = atan2(g_Ay, g_Az) * RadToDeg;
130     sinAngle = sin(g_Roll * DegToRad);
131     cosAngle = cos(g_Roll * DegToRad);
132
133     /* De-rotate by roll angle g_Roll */
134     By = g_My * cosAngle - g_Mz * sinAngle;
135     g_Mz = g_Mz * cosAngle + g_My * sinAngle;
136     g_Az = g_Ay * sinAngle + g_Az * cosAngle;
137
138     /* Calculate pitch angle g_Pitch (-90deg, 90deg) and sin, cos*/
139     g_Pitch = atan2(-g_Ax, g_Az) * RadToDeg;
140     sinAngle = sin(g_Pitch * DegToRad);
141     cosAngle = cos(g_Pitch * DegToRad);
142
143     /* De-rotate by pitch angle g_Pitch */

```

```

144     Bx = g_Mx * cosAngle + g_Mz * sinAngle;
145
146     /* Calculate yaw = ecompass angle psi (-180deg, 180deg) */
147     g_Yaw = atan2(-By, Bx) * RadToDeg;
148     if (g_FirstRun)
149     {
150         g_Yaw_LP = g_Yaw;
151         g_FirstRun = false;
152     }
153
154     g_Yaw_LP += (g_Yaw - g_Yaw_LP) * 0.01;
155
156     if (++loopCounter > 10)
157     {
158         PRINTF("\r\nCompass Angle: %3.1lf", g_Yaw_LP);
159         loopCounter = 0;
160     }
161 }
162 } /* End infinite loops */
163 }

```

Listing 8: Problem 5 main